Speeding Up the Convergence of Online Heuristic Search

and

Scaling Up Offline Heuristic Search

A Thesis
Presented to
The Academic Faculty

by

David A. Furcy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2004

Speeding Up the Convergence of Online Heuristic Search

and

Scaling Up Offline Heuristic Search

ACKNOWLEDGEMENT

I would like to thank my advisor, Sven Koenig, for his help with this research. I thank my committee members for their time and guidance. In particular, it has been a great pleasure to collaborate with Rob Holte. I am grateful to him, Jonathan Schaeffer and everybody at the University of Alberta for their warm welcome during my stay in Edmonton. I also enjoyed our joint work with Ariel Felner.

Over the years, I have benefited from the help of many people. Rich Korf was always willing to share his source code and he provided me with Thorpe's thesis, while Stefan Edelkamp was the one who first introduced me to Thorpe's work. Vadim Bulitko and I talked a lot about real-time search, and he was kind enough to read drafts of some of these chapters. Rong Zhou and Matthew McNaughton helped me get acquainted with the MSA domain. I have also had fruitful discussions with several search experts including Blai Bonet, Hector Geffner, Eric Hansen, István Hernádvölgyi, and Wheeler Ruml. At Georgia Tech, I enjoyed the company and help of Jim Davies, Maxim Likhachev, Yaxin Liu, Patrawadee Prasangsit, and Alex Stoychev.

Last but not least, I am immensely grateful for the love and support of my wife, Elizabeth, and for little Abigail, who kept me awake during the last two weeks of writing and energized me for the last stretch.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Heuristic search algorithms are popular Artificial Intelligence methods for solving the shortest-path problem. This research contributes new heuristic search algorithms that are either faster or scale up to larger problems than existing algorithms. Our contributions apply to both online and offline tasks.

For online tasks, existing real-time heuristic search algorithms learn better informed heuristic values and sometimes eventually converge to a shortest path by repeatedly executing the action leading to a successor state with a minimum cost-to-goal estimate. In contrast, we claim that real-time heuristic search converges faster to a shortest path when it always selects an action leading to a state with a minimum f-value (i.e., a minimum estimate of the cost of a shortest path from start to goal via the state), just like in the offline A* search algorithm. We support this claim by implementing this new non-trivial action-selection rule in FALCONS and by showing empirically that FALCONS significantly reduces the number of actions to convergence of a state-of-the-art real-time search algorithm.

For offline tasks, we scale up two best-first search approaches. First, a greedy variant of A* called WA* is known 1) to consume less memory to find solutions of equal cost when it is diversified (i.e., when it performs expansions in parallel), as in KWA*; and 2) to solve larger problems when it is committed (i.e., when it chooses the state to expand next among a fixed-size subset of the set of generated but unexpanded states), as in MSC-WA*. We claim that WA* solves even larger problems when it is enhanced with both diversity and commitment. We support this claim with our MSC-KWA* algorithm. Second, it is known that breadth-first search solves larger problems when it prunes unpromising states, resulting in the beam search algorithm. We claim that beam search quickly solves even larger problems when it is enhanced with backtracking based on limited discrepancy search. We support this claim with our BULB algorithm. We demonstrate the improved scaling of MSC-KWA* and BULB empirically in three standard benchmark domains. Finally, we apply anytime variants of BULB to the multiple sequence alignment problem in biology.

CHAPTER I

OVERVIEW OF THE DISSERTATION

1.1 Introduction

The most popular methods for solving the shortest-path problem in Artificial Intelligence (AI) are heuristic search algorithms. In particular, best-first search algorithms always expand next a node with the smallest f-value, where the f-value of a node estimates the cost of a shortest path from the start to a goal via the node. In breadth-first (or uniform-cost) search [29], the f-value is equal to the g-value of the node, which is the cost of the shortest path found so far from the start to the node. In the A* algorithm [59], the f-value is the sum of the g-value and the h-value of the node, which is an estimate of the cost of a shortest path from the node to the goal. A* and breadth-first search are offline search algorithms since they find a complete path to the goal before they terminate. In contrast, online (and more specifically real-time) search algorithms interleave searching for a partial path from the current node and traversing this path in the environment. Such algorithms are useful for tasks that have tight time constraints on each action execution. We now discuss in turn our hypotheses pertaining to real-time and offline heuristic search.

Real-time search. Existing real-time heuristic search methods, such as LRTA* [98], repeatedly select and execute the action leading to a successor with minimum h-value. Before each execution, they also update the h-value of the current node so that they learn better informed h-values over time. When the goal is reached (we say that the current trial is over), the agent is reset to the start (and the next trial begins). Their learning component enables real-time search methods to eventually converge to a shortest path. However, we claim that minimizing h-values is not the best action-selection rule for fast convergence. We propose the following hypothesis:

Hypothesis 1 (Real-time search hypothesis) *Real-time heuristic search converges faster to a shortest path when it selects actions leading to nodes with a minimum estimated cost of a shortest path going from the start through the node and to the goal.*

In Chapter 2, we will support this hypothesis with FALCONS, a new real-time search algorithm that converges to a shortest path with significantly fewer actions and trials than LRTA*. We will show that the correct design of our action-selection rule in FALCONS is not trivial. Nevertheless, Appendix A will prove that FALCONS shares the same theoretical properties as LRTA*. We will show empirically that FALCONS converges with fewer actions and trials than LRTA* in all of our thirteen different empirical conditions (corresponding to six standard benchmark domains with two or more heuristic functions per domain). Convergence with fewer actions and trials means that the overall learning time is shorter since both the total time spent executing actions and the total pre-trial setup time are smaller. This speedup is important in domains from real-time control. The main limitations of FALCONS are that 1) the duration of the first trial is sometimes larger because more exploration is performed at the beginning, 2) FALCONS may not perform well in directed domains because its action-selection rule is based exclusively on the f-value of the successor node and does not take into account the edge cost to reach it, and 3) FALCONS only applies to deterministic domains.

Offline search. The main drawback of both breadth-first search and A* is that they store all generated nodes in memory. Therefore, they quickly run out of memory on large graphs. To remedy this problem and scale up heuristic search to larger problems, one common approach is to sacrifice solution quality (breadth-first search and A* are *admissible* algorithms, that is, they always find a shortest path, provided they have enough memory). One typically reduces memory consumption by making the search greedy (but still storing all generated nodes) or by pruning some nodes (that is, not storing some of the generated nodes). We summarize in turn our contributions to each class of approaches.

First, it is known that WA* makes A* search greedy by weighing the h-value more than the g-value when adding them up to compute each f-value. WA* can solve larger problems than A* [132, 52]. It is also known that WA* with diversity (that is, the parallel expansion of several nodes at each iteration, like in KWA* [37]) uses less memory than WA* to find solutions of equal cost. Furthermore, it is known that WA* with commitment (that is, the focus on a sub-set of the candidate nodes for expansion, like in MSC-WA* [88]) scales up to larger problems than WA*. We propose the following hypothesis:

Hypothesis 2 (Offline search hypothesis #1) *WA* solves larger problems when it is enhanced with both diversity and commitment.*

In Chapter 3, we will support this hypothesis with MSC-KWA*, a new offline search algorithm that can solve larger problems than WA*, KWA* and MSC-WA* in three benchmark domains. In our empirical setup, MSC-KWA* is the only considered variant of WA* that can solve all of our random instances in the 48-Puzzle and the 4-peg Towers of Hanoi domain. Furthermore, MSC-KWA* solves the largest percentage of random instances in the Rubik's Cube domain. However, MSC-KWA* shares with WA*, MSC-WA* and KWA* the limitation that it is not memory-bounded. For example, none of these algorithms can solve all of our random instances in the Rubik's Cube. Another limitation of MSC-KWA* is that it takes three parameters as inputs. While the best value of the W parameter is often very close to one, finding the best values for the C and K parameters currently requires trial and error and typically leads to different values for C and K. In general, the behavior of MSC-KWA* is quite sensitive to the values chosen for C and K.

Second, it is known that beam search scales up breadth-first search by limiting the number of nodes at each level of the search to a constant, maximum value and by pruning additional nodes [7, 170]. We propose the following hypothesis:

Hypothesis 3 (Offline search hypothesis #2) *Beam search quickly solves larger problems when it is enhanced with backtracking based on limited discrepancy search.*

In Chapter 4, we will support this hypothesis with BULB, a new offline search algorithm that can solve larger problems than beam search while keeping its runtime reasonably small. In our empirical setup, BULB can solve all of our random instances in the 48-Puzzle, 63-Puzzle, 80-Puzzle, Rubik's Cube and Towers of Hanoi domains in a matter of seconds or minutes and finds solutions that are reasonably close to optimal since their cost is always within an order of magnitude of the optimal cost and in most cases they are approximately within a factor two of optimal. The main drawback of BULB is the need to determine the value of its beam width parameter B that gives the best performance in terms of solution cost and runtime. Too small a value may lead to incompleteness since the search tree is narrow and all of its leaves may have been visited already (thus ending the search without a goal). Too large a value reduces the solution cost but may slow

BULB down significantly and may even lead to incompleteness if the maximum searchable depth becomes smaller than the depth of the shallowest goal. In our empirical setup, the best trade-off between solution cost and runtime is obtained for relatively large values of B (on the order of a few thousands). Therefore, the main limitation of BULB is that its behavior is sensitive to the value of B.

In Chapter 5, we will discuss different ways of transforming BULB into an anytime algorithm called ABULB. In Chapter 6, we will apply ABULB to the multiple sequence alignment problem in biology.

This chapter is organized as follows. Section 1.2 motivates and defines the shortest-path problem. Section 1.3 describes the structure of the dissertation. Finally, Sections 1.4 and 1.5 summarize our research on real-time and offline heuristic search, respectively.

1.2 The shortest-path problem

Many real-world tasks are equivalent to finding a shortest path in a graph, including robot navigation tasks, network routing in transportation tasks, symbolic planning tasks, and sequence alignment tasks in biology. Because of its practical relevance, the shortest-path problem has been of interest to computer scientists in general and AI researchers in particular.

Even though there exist algorithms that solve this problem in time that is at most quadratic in the number of nodes in the graph [29], this low-polynomial complexity is misleading because the number of nodes is often exponential in the solution length (that is, the number of edges in the solution path). Many real-world tasks (including planning tasks and sequence alignment tasks) do translate into exponentially large graphs. Since it is often not possible to find optimal solutions in a reasonable amount of time and without running out of memory, different ways of trading off the solution cost, runtime and memory consumption have been studied. Usually, memory is the most limiting factor and it gets filled up rather quickly. Memory-bounded algorithms have been introduced to address this limitation [96, 143, 177]. However, the price to pay for being able to control the memory consumption is a large runtime overhead due to node re-generations. Such algorithms may take days or weeks to terminate [101, 105], which is not acceptable in many practical situations. Long runtimes remain problematic for inadmissible algorithms as well [99]. In Chapters 3 through 7, we will address the issues of 1) how to scale up offline search to larger problems and 2) how to trade off solution cost and runtime in memory-bounded offline search.

We now formally define the shortest-path problem. The reader weary of formalism can safely skip the following sub-section.

1.2.1 Problem statement

A graph $G = (S, E)$ is defined by a finite set S of nodes[1] and a finite set E of directed edges $e = (n_1, n_2)$ between pairs of nodes n_1, $n_2 \in S$. Let $succ(n) \subseteq S$ denote the set of successors of any node $n \in S$, that is, the set of nodes $n' \in S$ such that $(n, n') \in E$. A path in G from node m to node n is a sequence $\{n_0 = m, n_1, \ldots, n_p = n\}$ of nodes in S such that $\forall i \in [1..p]$: $n_i \in succ(n_{i-1})$. Thus a path is also a sequence of edges $\{(n_0, n_1), (n_1, n_2), \ldots, (n_{p-1}, n_p)\}$. If each edge $e \in E$ is associated with a cost $c(e)$, then the cost of any path $\{n_i\}_{i=0,\ldots,p}$ is equal to $\sum_{i=1}^{p} c(n_{i-1}, n_i)$.

This research is concerned with the *single-source, single-destination shortest-path problem*, which is defined as follows. Given:

- a graph $G = (S, E)$,
- a cost function c defined on E such that $\forall e \in E$: $0 < c(e) < \infty$, and
- two distinguished nodes s_{start} and s_{goal} in S,

find a shortest (or minimum-cost) path from s_{start} to s_{goal} in G.

1.3 Structure of the dissertation

This dissertation contains two parts, one each for real-time search and offline search. This high-level decomposition, as well as the internal structure of the second part, mirror the taxonomy of tasks (and associated methods) that we now describe. This taxonomy of heuristic search algorithms is built upon the task constraints under which the problem may be solved (see Figure 1).

[1]In AI search, nodes are often identified with *states*. A state is a particular configuration of the objects in the representation of the domain. A node is an object manipulated by the search algorithm. Nodes are similar to states since a node

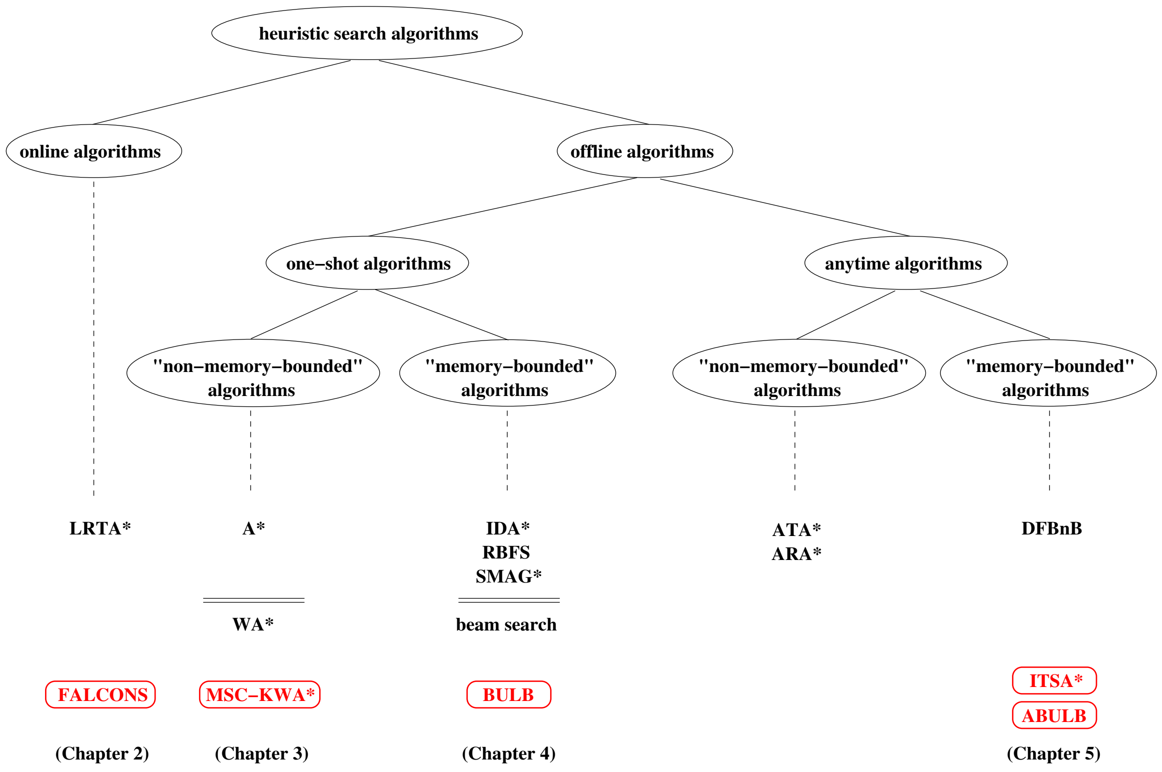


Figure 1: A taxonomy of heuristic search algorithms (with our contributions in red)

First, the taxonomy distinguishes between online and offline tasks (or algorithms). For the former tasks, the agent interleaves searching and acting in the environment. For the latter tasks, the agent performs a complete search to the goal and then executes the sequence of actions corresponding to the edges in the solution path.

Second, the taxonomy distinguishes between one-shot and anytime tasks. For the former tasks, only one solution is produced, namely when the algorithm terminates. For the latter tasks, the algorithm outputs several solutions of increasing quality (that is, of decreasing costs).

Third, the taxonomy distinguishes between tasks for which the available memory can be considered unlimited and tasks for which memory has tight constraints. Of course, internal computer memory is always limited. But as memory becomes cheaper and thus larger, this limit may be higher than the maximum amount of memory consumed by the algorithm. A common example in this class

contains a state description (as well as additional information needed during search, such as g-values, h-values, etc.). In this dissertation, we use the words *state* and *node* interchangeably.

of tasks is robot-navigation in gridworld-like domains, in which the environment is typically represented as a grid that fits in memory. In contrast, many hard shortest-path problems have huge search spaces (or associated graphs). Common examples include combinatorial puzzles (such as the N-Puzzle, the Rubik's Cube, the Towers of Hanoi puzzle, etc.) and the multiple sequence alignment problem. Ensuring completeness in such problems requires that the algorithm be memory-bounded.

In Figure 1, ellipses represent classes of algorithms. Solid lines represent sub-class relations. Dashed lines represent membership relations. Each leaf of the tree is a representative algorithm (or a list of representative algorithms). Red (boxed) algorithms are the new algorithms introduced in this dissertation (and the corresponding chapters). In the case of offline, one-shot algorithms, a double horizontal line separates admissible algorithms (on top) from inadmissible ones.

Following this chapter, the dissertation is split onto two parts. Chapter 2 and Appendix A will discuss our research on real-time search. All remaining chapters (including Appendix B) discuss our research on offline search. This second part is itself split into two sub-parts. Chapters 3 and 4 will introduce two new one-shot heuristic search algorithms. Chapters 5 and 6 will introduce a new family of anytime heuristic search algorithms and will describe their application to the multiple sequence alignment problem in biology, respectively. Finally, Chapter 7 will summarize our contributions and elaborate on some directions for future work on offline search. The mapping between chapters and tasks (and associated algorithms) is depicted at the bottom of Figure 1.

1.4 Overview of our contributions to real-time search

Real-time search methods, such as LRTA* [98], interleave planning (via local search around the current node) and execution of partial paths [79]. Even when task constraints require that actions be chosen in constant time, these methods guarantee that the goal will be reached. Furthermore, they learn better informed h-values during successive trials and eventually converge to a shortest path. This learning capability is quite useful for real-world tasks, including project scheduling [154] and routing for ad-hoc networks [149]. Recently, researchers have attempted to speed up the convergence of LRTA* while maintaining its advantage over traditional search methods, that is, without increasing its lookahead (or the depth of the local search around the current node, typically equal to one). Shimbo and Ishida, for example, achieved a significant speedup by sacrificing the optimality of the resulting path [83, 79]. We, on the other hand, show how to achieve a significant speedup without sacrificing the optimality of the resulting path. This will be our goal in Chapter 2.

We claim that convergence to a shortest path can be sped up by consistently maintaining the focus of the search upon its long-term objective, namely that of finding a shortest path from the start to a goal, as opposed to the short-term objective of reaching a goal as fast as possible from the current node. We thus advocate a radically different way of focusing the search. If the objective is fast convergence to a shortest path, then the search should be focused around what is believed to be a shortest path. In Section 2.4, we will make this intuitive search strategy operational and will motivate 1) the need for a new action-selection rule and 2) our choice of the action-selection rule that leads to nodes with minimum f-values.

To summarize our contributions, we propose a new search strategy that selects actions leading to a node believed to be close to a shortest path from the start to a goal. The question becomes how to estimate the distance from a node to a shortest path, the answer to which is not obvious because 1) a shortest path is what we are looking for, and 2) real-time search methods do not store any path in memory. We propose to estimate the distance from a node to a shortest path using f-values. Since f-values are smallest on a shortest path and larger for nodes off a shortest path, our new action-selection rule chooses an action leading to a node with minimum f-value. Our main contribution in Chapter 2 will be to extend the applicability of A*'s search strategy (namely, guiding the search with smallest f-values) to the real-time search setting. This extension is not trivial for two reasons. First, real-time search methods do not have f-values available, only h-values. We will solve this problem in Section 2.4. Second, the convergence of real-time search methods is facilitated by the fact that they always update the h-value of the current node based on the h-value of the successor node they move to next. If the h-value of this successor node is misinformed, they immediately have a chance to learn a better one since this successor node becomes the current node at the next iteration. This property does not hold with our action-selection rule because a successor node with the smallest f-value may not have the smallest h-value. We will discuss this problem in Section 2.6.1 and will solve it in Section 2.6.2. We call the resulting algorithm FALCONS.

Appendix A contains the formal proofs that FALCONS is guaranteed to reach a goal during each trial and eventually to converge to a shortest path. Our empirical study reported in Section 2.7 will

Table 1: Speedup of FALCONS over LRTA*

Domain	Heuristic	Number of actions to convergence	Number of trials to convergence
8-Puzzle	M	60%	73%
	T	20%	44%
	Z	10%	47%
Gridworld	N	41%	52%
	Z	14%	38%
Permute-7	A	5%	18%
	Z	3%	36%
Arrow	F	15%	23%
	Z	6%	38%
Towers of Hanoi	D	18%	49%
	Z	17%	53%
Words	L	30%	44%
	Z	4%	30%

demonstrate that FALCONS converges faster than LRTA*, a state-of-the-art real-time search algorithm [98]. In thirteen different experimental conditions (each characterized by a standard benchmark domain and a heuristic function), FALCONS needs fewer actions than LRTA* to converge to a shortest path. The corresponding speedups are listed in the second column of Table 1. In addition, while our goal was to reduce the number of actions to convergence, FALCONS also reduces the number of trials to convergence, as shown in the third column of the table. This is a nice property because in domains from real-time control, the setup for each trial may be expensive and thus it is important to keep the number of trials small. Finally, [153] has shown that FALCONS also reduces the memory consumption of LRTA*. Because it focuses the search around what it believes to be a shortest path, FALCONS ends up visiting (and thus storing) fewer nodes.

In conclusion, FALCONS improves on a state-of-the-art real-time search algorithm in terms of both speed of convergence and memory consumption. Vadim Bulitko at the University of Alberta is in the process of extending FALCONS (for example with a larger lookahead [16]), while Shan et al. [149] are planning to apply FALCONS to constraint-based routing in ad-hoc networks, having already applied LRTA* to this task. More generally, we believe that our new action-selection rule is quite relevant to the reinforcement-learning community, since the vast majority of existing methods in this area use h-based action-selection rules when exploiting heuristic information. Our results

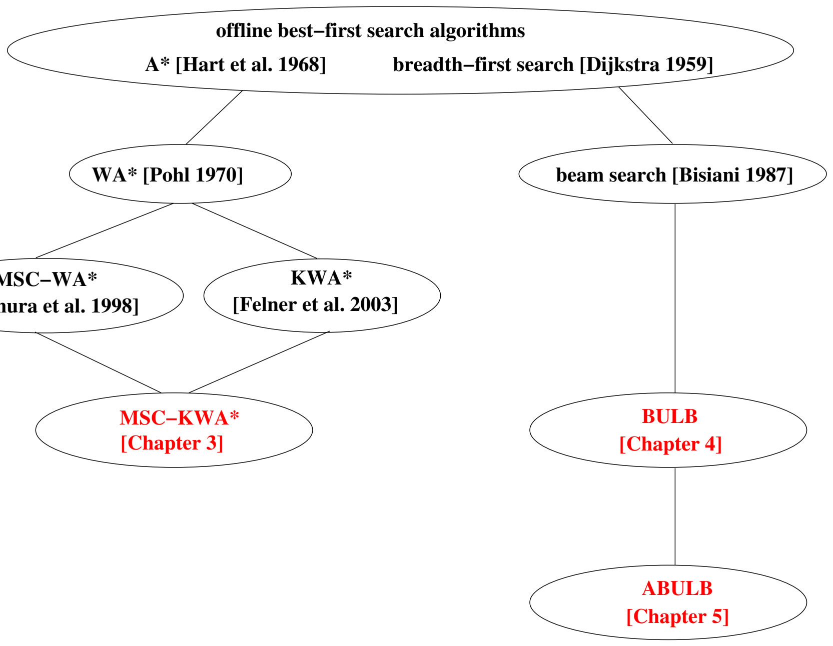

offline best–first search algorithms
A* [Hart et al. 1968]
breadth–first search [Dijkstra 1959]
WA* [Pohl 1970]
beam search [Bisiani 1987]
MSC–WA* [Kitamura et al. 1998]
KWA* [Felner et al. 2003]
MSC–KWA* [Chapter 3]
BULB [Chapter 4]
ABULB [Chapter 5]

Figure 2: Lineage of our new offline heuristic search algorithms

suggest that significantly faster learning could result from an f-based exploitation rule.

1.5 Overview of our contributions to offline search

In the case of offline search, our primary goal is to scale up existing algorithms so that they can solve larger problems (that is, problems with larger underlying graphs) without running out of memory. When comparing algorithms that scale up to problems of similar sizes, our secondary goal is to find low-cost solutions in a reasonable amount of time (on the order of minutes, as opposed to days or weeks). We will build on two existing approaches for scaling up best-first search to larger problems while sacrificing solution optimality, namely greedy variants (such as WA* [132]) and pruning variants (such as beam search [7]) of best-first search. Our main contribution in each case is a new algorithm. Chapter 3 will describe MSC-KWA*, which scales up to larger problems than existing variants of WA*. Chapter 4 will describe BULB, which scales up to larger problems than an existing variant of beam search. Figure 2 shows the lineage of our new algorithms.

This section provides a high-level summary of our contributions to offline search. A more detailed and more technical summary will be given in Chapter 7.

1.5.1 Our contributions to greedy best-first search

WA* is a variant of A* in which the f-value of each node n is calculated as $f(n) = g(n) + Wh(n)$, where W is a real number larger than or equal to 1 [132]. A* is the special case of WA* when $W = 1$. When $W > 1$, WA* puts more weight on the h-value than it does on the g-value. The search is said to be greedy because, by minimizing f-values, WA* favors nodes that are (believed to be) close to the goal (since small h-values lead to small f-values). On the one hand, increasing W makes the search more greedy, which reduces the number of nodes WA* generates. This reduction speeds up the search and also enables WA* to solve larger problems than A*. On the other hand, increasing W increases the cost of the solution found by WA*, which is not admissible anymore. [24] shows that the cost of the solution returned by WA* exceeds the optimal cost by a multiplicative factor equal to W in the worst case. In practice, the solution cost returned by WA* is much lower than this upper bound (see, for example, [99] as well as our experimental results in Chapter 3). In the past few years, the scaling behavior of WA* has been improved in two ways, namely with diversity or commitment.

First, diversifying the search means expanding $K \geq 1$ nodes in parallel at each iteration, resulting in the KWA* algorithm [37]. By expanding only one node at a time, WA* may visit large goal-free regions of the graph as a result of putting a large weight on misleading heuristic values. By expanding in parallel the most promising K nodes, KWA* is more likely to expand a node with a well-informed h-value. In effect, KWA* introduces a breadth-first search component into WA*. The right level of diversity (controlled by K) can significantly reduce the number of node generations needed to find solutions of a given cost [37]. With too much diversity, KWA* degenerates into breadth-first search (when $K = \infty$).

Second, committing the search means focusing it on a sub-set of the candidate nodes for expansion, resulting in the MSC-WA* algorithm [88]. MSC-WA* controls the level of commitment with a parameter C, namely the maximum number of nodes that are considered for expansion at each iteration. When $C = \infty$, MSC-WA* reduces to WA* since then, all generated but unexpanded nodes are considered for expansion at each iteration. When C has a finite value (larger than or equal to one), only the C nodes with the lowest f-values are considered for expansion. Any additional nodes are moved to a *reserve list*. These nodes are not pruned since the full reserve list is stored in memory. Instead, this list is used to replenish the set of nodes under consideration every time its size becomes smaller than C. Keeping C small serves to focus the search on a limited number of nodes. If the heuristic values are well informed, this can cut down the exponential explosion of the search. In effect, MSC-WA* introduces a depth-first search component into WA*. The right level of commitment (controlled by C) can reduce the number of node generations significantly [88].

In Chapter 3, we will show empirically that increased levels of commitment and diversity are orthogonal and complementary ways of improving on WA*. We will also show empirically that they can, in combination, scale up WA* to even larger problems. We call MSC-KWA* our new algorithm resulting from the combination of MSC-WA* and KWA*. Furthermore, we will discuss the similarities between MSC-KWA* and beam search. Note that Appendix B contains all of the graphs detailing the performance of WA*, KWA*, MSC-WA*, and KWA* in the N-Puzzle domain. The data in these graphs will only be summarized in Chapter 3 due to space considerations.

1.5.2 Our contributions to beam search

Beam search is a variant of best-first search that prunes some generated nodes (pruned nodes are not stored in memory, in contrast to nodes in the reserve list maintained by MSC-WA* and MSC-KWA*) [7, 170, 144]. Pruning nodes from the set under consideration for expansion focuses the search on a restricted number of possible paths, thereby cutting down on the exponential explosion of the search. However, pruning nodes is more radical than keeping them in reserve because the only way to bring these nodes back under consideration is to find another path to them during the search. Beam search is not complete because all paths to the goal may become cut off due to pruning. The same reasoning applied to optimal paths explains why beam search is not admissible.

In Chapter 4, we will focus on a standard variant of beam search based on breadth-first search [7, 43, 170, 151, 180]. In this case, beam search expands in parallel all nodes under consideration (starting with the set containing only the start node), orders the set of all their successor nodes by increasing h-values (all nodes under consideration at each iteration have the same g-value), and only keeps the best B nodes to make up the set of nodes under consideration at the next iteration. B is called the beam *width*. Since all discarded nodes are purged from memory, the memory consumption of beam search is proportional to B times the depth of the search (that is, the number of iterations or levels of the search). By keeping a maximum of B nodes at each level, beam search makes the memory consumption linear in the solution length. Since beam search stops as soon as the goal is generated, the length of (or the number of edges in) the solution path is equal to the depth of the search.

There are three situations in which beam search may terminate without a goal. First, if B is too small, the beam may become empty before finding a goal. This can happen because beam search never re-visits a node and all successor nodes may have been visited earlier. Solutions to this problem include increasing the value of B or finding a better heuristic function. Second, the shallowest goal may be so far away from the start that beam search with a given B value runs out of memory before reaching it (i.e., the total memory needed for all nodes in the beam down to the goal is larger than the available memory). The solution to this problem requires decreasing the value of B. Third, in the intermediate case, beam search may run out of memory at a given depth (say, d) because the heuristic function leads it astray. If there is a goal at level d (or closer to the start), solutions to this problem include finding a better heuristic function or a memory-purging strategy that continues searching "against" the heuristic values to find out where they are misleading.

In Chapter 4, we will follow this latter strategy. Our goal will be to scale up beam search to larger problems by dealing with the cases in which the goal is reachable with the current value of B but the heuristic function used to order the nodes at each level is misleading. Our main contribution in Chapter 4 will be to apply existing backtracking strategies to beam search. By backtracking on its pruning decisions, beam search can solve larger problems. In order to keep the search reasonably fast, we will need a smart backtracking strategy. We will show that backtracking based on limited discrepancy search [61] combines nicely with beam search to yield a new algorithm called BULB.

1.5.3 Summary of empirical results

We will test all of our offline search algorithms on (a sub-set of) the same standard benchmark domains, namely the N-Puzzle with values of N ranging from 8 through 80, the 4-peg Towers of Hanoi domain, and the Rubik's Cube domain. Our domains (and corresponding heuristic functions) will be described in Sections 3.6.1 through 3.6.3, respectively.

Table 2: Scaling behavior in our three benchmark domains

Domain	Heuristic	Memory (10^6 nodes)	WA*	MSC-WA*	KWA*	MSC-KWA*	beam search	BULB
8-Puzzle	MD	6	✓	✓	✓	✓	✓	✓
15-Puzzle	MD	6	✓	✓	✓	✓	✓	✓
24-Puzzle	MD	6	✓	✓	✓	✓	✓	✓
35-Puzzle	MD	6		✓	✓	✓	✓	✓
48-Puzzle	MD	6				✓	✓	✓
63-Puzzle	MD	4						✓
80-Puzzle	MD	3						✓
Rubik's Cube	Korf's	1					✓	✓
Towers of Hanoi	13-disk PDB	1				✓		✓

Table 2 contains a preview of our results that demonstrates to which extent we have achieved our primary goal of scaling up offline search to larger problems in these domains. The first three columns define an empirical condition as the combination of a domain, a heuristic function and the available memory (measured as the number of storable nodes in millions). The remaining columns list the tested algorithms. A check mark in a cell means that the algorithm in the corresponding column solves the full set of random instances in the empirical condition defined by the row.

First, the table shows that MSC-KWA* scales up to larger problems than either KWA* or MSC-WA* can handle since it can solve all of our random instances of the 48-Puzzle and of the Towers of Hanoi domain. Even though MSC-KWA* does not solve all of our random instances of the Rubik's Cube domain, neither do the other variants of WA* (this can be inferred from Table 15 where the available memory is twice the one listed here), but MSC-KWA* solves the highest percentage of instances (see Table 15).

Second, the table shows that BULB is the only tested algorithm that solves all random instances in our three benchmark domains. In addition, the table shows that beam search, which BULB extends, is also a strong contender. Nevertheless, beam search does not solve all of our random instances of the Towers of Hanoi domain, whereas BULB does. Furthermore, what the table does not show is that, when both beam search and BULB scale up to problems of the same size, BULB always finds solutions with lower costs than beam search and it does so in a reasonable amount of time. In the 48-Puzzle, beam search reaches its best average solution cost at about 11,700 in a fraction of a second (see Table 17 when $B = 5$), while BULB can reduce the average solution cost by an order of magnitude down to below 1,000 and it does so with an average runtime of 10 seconds (see Figure 49 when $B = 2,000$). In the Rubik's Cube domain, beam search reaches its best average solution cost at about 55 in about 10 seconds (see Table 20 when $B = 7,000$), while BULB can cut the average solution cost nearly in half down to about 30 and it does so with an average runtime of 40 seconds (see Figure 54 when $B = 20,000$). This is a significant decrease in solution cost given the already low solution cost exhibited by beam search. Indeed, the median and worst solution costs in this domain are estimated to be 18 and 20, respectively [101]. In fact, the solution obtained by BULB in a matter of minutes (namely, about 23 when $B = 50,000$) is significantly lower than that obtained by a recent, powerful Rubik's Cube solver based on macro-operators, even though this solver uses both a larger number of pattern databases to build the macro-operators and a post-processing step on solution paths [63]. Therefore, we believe that BULB is a state-of-the-art solver in this domain (in terms of the trade-off between solution cost and runtime) even though it is a pure-search, domain-independent algorithm that uses neither pre- nor post-processing.

1.5.4 Algorithm selection

With respect to our goal of scaling up offline search to larger problems, BULB presents several advantages over MSC-KWA*. First, Table 2 shows that BULB scales better than MSC-KWA* across domains. (In contrast, neither beam search nor MSC-KWA* clearly scales better than the other algorithm across domains. However, when both algorithms solve all of our random instances of the 48-Puzzle, MSC-KWA* yields a better average solution cost of about 4,000 (see Table 16) against about 12,000 for beam search (see Table 17).)

Second, BULB is easier than MSC-KWA* to apply in practice since it only takes one parameter (namely B) against three for MSC-KWA* (namely, W, C, and K). Indeed, Chapter 3 will show that obtaining the best scaling behavior of MSC-KWA* requires the fine tuning of its C and K parameters (W is typically kept close to one for the best scaling). Nevertheless, choosing an appropriate value of B to give as input to BULB (and ABULB) remains a challenge and this difficulty constitutes the main limitation of BULB.

Third, a crucial difference between BULB and MSC-KWA* is that BULB is a memory-bounded algorithm while MSC-KWA* is not. Through B, the user can control how deep BULB searches without ever running out of memory. Like for all variants of WA*, such control is not possible in the case of MSC-KWA*.

Fourth, because it is memory-bounded, BULB lends itself nicely to anytime extensions, as described in the next sub-section.

For all these reasons, and despite the fact that MSC-KWA* is easier to implement than BULB, we believe that BULB is the algorithm of choice among the ones we have tested when it comes to scaling offline search to larger problems. It remains future work to find a way to determine or learn the best B value a priori based, for example, on the domain description and the heuristic function. In this work, the value of B is determined by trial and error.

1.5.5 ABULB: Anytime variants of BULB

In Chapter 5, we will present a new family of anytime heuristic search algorithms generically called ABULB (for Anytime BULB). ABULB is a local (or neighborhood) search algorithm in the space of solution paths. ABULB uses BULB to find both an initial solution and restarting solutions. ABULB can also take advantage of ITSA* for local path optimization.

ITSA* is a new local path optimization algorithm. ITSA* imposes a neighborhood structure on the space of solution paths based on our definition of distance between paths. ITSA* interleaves the construction and the searching of the neighborhood using breadth-first and A* search, respectively. Successive iterations return paths with non-increasing costs. ITSA* is thus an anytime algorithm in its own right. ITSA* performs gradient descent on the surface whose connectivity and elevation result from the neighborhood structure and the solution cost, respectively. Each time ITSA* reaches a (possibly local) minimum on the surface, ABULB generates a new restarting solution of higher quality.

Our empirical study will show that, while ITSA* reduces the solution cost over time when used as an anytime algorithm in the 48-Puzzle and the Rubik's Cube domain, an even larger reduction in solution cost is achieved by continuing BULB's execution with the same beam width when it finds a solution (ABULB 1.0) or by restarting it with a larger, automatically computed beam width (ABULB 2.0). Furthermore, combining ITSA* with either variant of ABULB yields an even larger reduction in solution cost in the 48-Puzzle.

1.5.6 Application of ABULB to the multiple sequence alignment problem

In Chapter 6, we will use the Multiple Sequence Alignment (MSA) problem in molecular biology as an additional benchmark domain for ABULB. We will explain how the MSA problem of maximizing the similarity score of an alignment of n biological sequences reduces to the shortest-path problem of minimizing the cost of a path between two opposite corners of an n-dimensional hypercube. We will also discuss the minor modifications needed for the application of ABULB to this domain.

Our empirical results will show that, on our MSA test problems, both ABULB 1.0 and ABULB 2.0 scale up to larger problems than Anytime A*, another anytime heuristic search algorithm based on WA*. Our results will also show that ABULB 2.0 reduces the solution cost more quickly than ABULB 1.0.

CHAPTER II

SPEEDING UP THE CONVERGENCE OF REAL-TIME SEARCH[*]

2.1 Introduction

Real-time (heuristic) search methods interleave planning (via local searches) and plan execution, and allow for fine-grained control over how much planning to perform between plan executions. They have successfully been applied to a variety of planning problems, including traditional search problems [98], moving-target search problems [81], STRIPS-type planning problems [119, 14], project scheduling with resource constraints or PSRC problems [154], robot navigation and localization problems with initial pose uncertainty [94], robot exploration problems [90], ad-hoc network routing problems [149], totally observable Markov decision process problems [6], and partially observable Markov decision process problems [53]. Learning-Real Time A* (LRTA*) is probably the most popular real-time search method [98]. It converges to a shortest path when it solves the same planning task repeatedly. Unlike traditional search methods, such as A* [128], it can not only act in real time (which is important, for example, for real-time control) but also amortize learning over several planning episodes. This allows it to find a sub-optimal path fast and then improve the path until it follows a shortest path. Thus, the sum of planning and plan-execution time is always small, yet LRTA* follows a shortest path in the long run.

Recently, researchers have attempted to speed up the convergence of LRTA* while maintaining its advantages over traditional search methods, that is, without increasing its lookahead. Ishida, for example, achieved a significant speedup by sacrificing the optimality of the resulting path [83, 79]. We, on the other hand, show how to achieve a significant speedup without sacrificing the optimality of the resulting path. FALCONS (FAst Learning and CONverging Search), our novel real-time search method, looks similar to LRTA* but selects successors very differently. LRTA* always greedily minimizes the estimated cost to go (in A* terminology: the sum of the cost of

[*]This chapter first appeared as [49].

moving to a successor and its h-value). FALCONS, on the other hand, always greedily minimizes the estimated cost of a shortest path from the start to a goal via the successor it moves to (in A* terminology: the f-value of the successor). This allows FALCONS to focus the search more sharply on the neighborhood of an optimal path. We use our experiments with FALCONS to support our hypothesis that real-time heuristic search converges faster to a shortest path when it selects actions leading to states with a minimum estimated cost of a shortest path going from the start through the state and to the goal. Our results on standard search domains from the artificial intelligence literature show that FALCONS indeed converges typically about twenty percent faster and in some cases even sixty percent faster than LRTA* in terms of travel cost. It also converges typically about forty percent faster and in some cases even seventy percent faster than LRTA* in terms of trials, even though it looks at the same states as LRTA* when it selects successors and even though it is not more knowledge-intensive to implement.

In addition to its relevance to the real-time search community, this research also sends an important message to reinforcement-learning researchers. Indeed, they are typically interested in fast convergence to an optimal behavior and use methods that, just like LRTA*, interleave planning (via local searches) and plan execution and converge to optimal behaviors when they solve the same planning task repeatedly [6, 85, 161]. Furthermore, during exploitation, all commonly-used reinforcement-learning methods, again just like LRTA*, always greedily move to minimize the expected estimated cost to go [165]. Our results therefore suggest that it might be possible to design reinforcement-learning methods that converge substantially faster to optimal behaviors than state-of-the-art reinforcement-learning methods, by using information to guide exploration and exploitation that is more directly related to the learning objective.

This chapter is structured as follows. Section 2.2 defines terminology and spells out our assumptions. Section 2.3 introduces LRTA*. Section 2.4 provides motivation for our new action-selection rule. Section 2.5 shows how we can significantly reduce the number of actions until convergence by breaking ties among successor states with equal cost-to-goal estimates in favor of one with minimal f-value. Section 2.6 demonstrates that FALCONS, our proposed algorithm, achieves an even larger reduction in the number of actions until convergence, by selecting as the next state one with minimal f-value and by making the cost-to-goal estimates a secondary criterion used only for breaking ties.

Section 2.7 provides empirical evidence for this reduction in several domains. Sections 2.8 & 2.9 discuss related and future work, respectively. Finally, Section 2.10 summarizes our contributions.

2.2 Definitions and assumptions

Definitions. Throughout this chapter, we use the following notation and definitions. S denotes the finite state space; $s_{start} \in S$ denotes the start state; and $s_{goal} \in S$ denotes the goal state.[1] $succ(s) \subseteq S$ denotes the set of successors of state s, and $pred(s) \subseteq S$ denotes the set of its predecessors. $c(s, s') > 0$ denotes the cost of moving from state s to successor $s' \in succ(s)$. The goal distance $gd(s)$ of state s is the cost of a shortest path from state s to the goal, and the start distance $sd(s)$ of state s is the cost of a shortest path from the start to state s. Each state s has a g-value and an h-value associated with it, two concepts known from A* search [128]. We use the notation $g(s)/h(s)$ to denote these values. The h-value of state s denotes an estimate of its true goal distance $h^*(s) := gd(s)$. Similarly, the g-value of state s denotes an estimate of its true start distance $g^*(s) := sd(s)$. Finally, the f-value of state s denotes an estimate of the cost $f^*(s) := g^*(s) + h^*(s)$ of a shortest path from the start to the goal through state s. H-values are called admissible iff $0 \leq h(s) \leq gd(s)$ for all states s, that is, if they do not overestimate the goal distances. They are called consistent iff $h(s_{goal}) = 0$ and $0 \leq h(s) \leq c(s, s') + h(s')$ for all states s with $s \neq s_{goal}$ and $s' \in succ(s)$, that is, if they satisfy the triangle inequality. It is known that zero-initialized h-values are consistent, and that consistent h-values are admissible [131]. The definition of admissibility can be extended in a straightforward way to the g- and f-values, and the definition of consistency can be extended to the g-values [50].

Assumptions. In this chapter, we assume that the given heuristic values are admissible. Almost all commonly-used heuristic values have this property, including straight-line distances for maps or Manhattan distances for sliding-tile puzzles. If $h(s, s')$ denotes $h(s)$ with respect to goal s', then we initialize the g- and h-values as follows: $h(s) := h(s, s_{goal})$ and $g(s) := h(s_{start}, s)$ for all states s. We also assume that the domain is safely explorable, that is, the goal distances of all states are finite, which guarantees that the task remains solvable by real-time search methods since they

[1]We assume that there is only one goal throughout this chapter (with the exception of Figure 8) to keep the notation simple. All of our results continue to hold in domains with multiple goals.

```
1. $s := s_{start}$
2. $s' := \arg\min_{s'' \in succ(s)}(c(s, s'') + h(s''))$
   Break ties arbitrarily
3. $h(s)$ := **if** $(s = s_{goal})$ **then** $h(s)^{\dagger}$
             **else** $\max(h(s), \min_{s'' \in succ(s)}(c(s, s'') + h(s'')))$
4. **If** $(s = s_{goal})$ **then** stop successfully
5. $s := s'$
6. **Go to** Line 2
```

Figure 3: The LRTA* algorithm

cannot accidentally reach a state with infinite goal distance.

2.3 Learning Real-Time A* (LRTA*)

In this section, we describe Learning Real-Time A* (LRTA*) [98], probably the most popular real-time search method. LRTA* (with lookahead one) is shown in Figure 3. Each state s has an h-value associated with it. LRTA* first decides which successor to move to (action-selection rule, Step 2). It looks at the successors of the current state and always greedily minimizes the estimated cost-to-goal, that is, the sum of the cost of moving to a successor and the estimated goal distance of that successor (that is, its h-value). Then, LRTA* updates the h-value of its current state to better approximate its goal distance (value-update rule, Step 3). Finally, it moves to the selected successor (Step 5) and iterates the procedure (Step 6). LRTA* terminates successfully when it reaches the goal (Step 4). A more comprehensive introduction to LRTA* and other real-time search methods can be found in [79].

The following properties of LRTA* are known: First, its h-values never decrease and remain admissible. Second, LRTA* terminates [98]. We call a *trial* any execution of LRTA* that begins at the start and ends in the goal. Third, if LRTA* is reset to the start whenever it reaches the goal and maintains its h-values from one trial to the next, then it eventually follows a shortest path from the start to the goal [98]. We call a *run* any sequence of trials from the first one until convergence is detected. We say that LRTA* “breaks ties systematically” if it breaks ties for each state according to an arbitrary ordering on its successors that is selected at the beginning of each run. If LRTA* breaks ties systematically, then it must have converged when it did not change any h-value during a trial. We use this property to detect convergence. Another advantage of systematic tie-breaking is discussed in Section 2.7.3. Our approach differs slightly from that of Korf [98] whose version of LRTA* breaks ties non-systematically and thus finds *all* shortest paths from the start to the goal. We are satisfied with finding only one shortest path. To represent the state of the art, we use LRTA* that "breaks ties randomly," meaning that ties are broken systematically according to orderings on the successors that are randomized before each run.

2.4 Motivation for our new action-selection rule

The premise of this work is that convergence to an optimal solution path can be sped up by consistently maintaining the focus of the search upon its long-term objective, namely an optimal path from the start to a goal, as opposed to the short-term objective of reaching a goal as fast as possible from the current state. We thus advocate a radically different way of focusing the search. In this section, we make this intuitive search strategy operational and motivate 1) the need for a new action-selection rule and 2) our choice of the action-selection rule that minimizes f-values.

Because it is agent-centered, real-time search is limited to local search around the current state of the agent [91]. In particular, this means that the agent can only expand states in its neighborhood. In fact, in the standard approach to real-time search with lookahead one, which we adopt in this chapter, the agent can only expand the current state. The obvious implication is that the agent must first move to a state in order to expand it. In other words, changing the search strategy requires changing the action-selection strategy of the agent. This is different from standard best-first search in which any state in the OPEN list could be expanded next, since its merit only depends on its evaluation function, not on its proximity in the search space to the previously expanded state.[2]

Having motivated our need for a new action-selection rule, we now motivate our specific proposal for an f-based rule. First note that, even though it remembers heuristic values for all visited states (in a hash table, say), real-time search does *not* save the search tree in memory. Doing so may speed up state re-expansions, but at the expense of memory usage. This would only be beneficial if expansions are time-consuming and space is not a problem. For the same reason,

[†]This test could be eliminated by moving Step 4 before Step 2 so that the h-value of s_{goal} is never modified. However, we prefer the current (equivalent) formulation since it makes the value-update rule for the h-values completely symmetrical with the value-update rule for the g-values to be introduced in FALCONS.

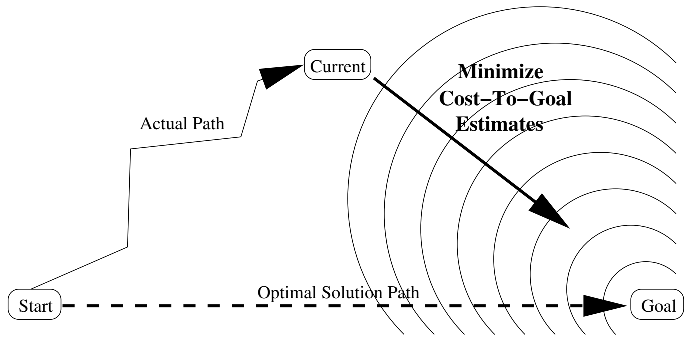
[2]A recent version of A* called PHA* also takes into account the cost of physically moving from one state to another in the OPEN list [39].

real-time search only maintains point-to-point heuristic information, namely estimates of the shortest distance from each visited state to the goal, but it does *not* explicitly maintain previously found solution paths from the start to the goal. Therefore, we would like to focus the search, if not on an explicit solution path, at least on an area of the search space believed to contain an optimal solution. The research question thus becomes

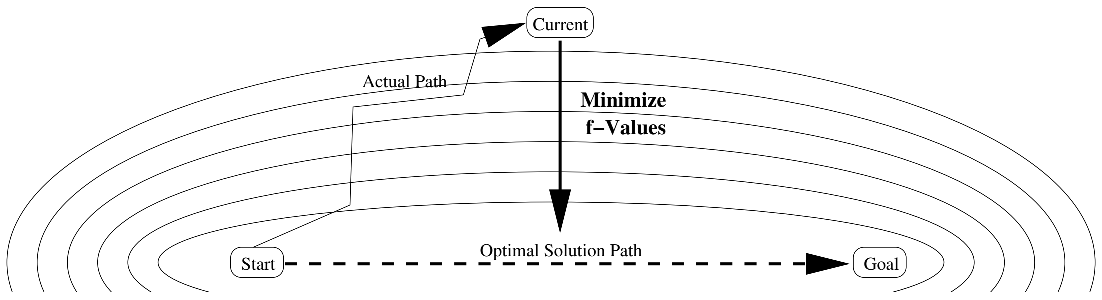
How to estimate, for each state, how far it is from an optimal path?

The answer is not trivial because 1) such heuristic information needs to estimate the distance from each state to a path (not another state), and 2) no solution path is explicitly identified. We solve this problem in the following way. Recall that the f-value $f(s)$ of a state s in A* estimates the cost $f^*(s)$ of a minimum-cost path from the start to the goal constrained to go through s. The main insight of our approach is to use the property that the f*-values of all states on any minimum-cost path from the start to the goal are all equal to the cost C^* of any minimum-cost path, while the f*-values of all other states are all strictly larger than C^* (otherwise, these states would be on some shortest path, by definition of f^*) [59]. Since f-values estimate f*-values, and if we assume that each state has associated with it an f-value, we propose to select actions so as to always minimize f-values. This way, if the f-values are perfectly informed, the agent will follow directly a minimum-cost path from the start to the goal (provided that ties among states with equal f-values are broken in favor of states with smaller h-values, that is, in the direction of the goal state). Otherwise, since heuristic values are continuously updated, the agent will gather more informed heuristic information and will thus be able to switch its focus to another area of the search space that looks more likely to contain an optimal solution. In short, we will use lowest f-values to focus the search toward previously identified *regions likely to contain an optimal solution*.

Figure 4 graphically represents our new action-selection rule in comparison to that used by LRTA*. In this figure, we have assumed that the only optimal solution is the straight line between the start and goal states. Note that the agent has strayed off of the optimal path, as typically happens when heuristic information is not perfect. In this case, minimizing cost-to-goal estimates, as LRTA* does, may waste search effort in areas that do not seem likely to contain an optimal path (because,



a) Action−Selection Rule of LRTA*



b) Proposed Action−Selection Rule

Figure 4: Two action-selection rules for real-time search. Curves represent iso-contours for a) cost-to-goal estimates and b) f-values.

despite having low cost-to-goal estimates, they also have high f-values). This is because the optimal path from the *current* state to the goal may have little overlap with an optimal path from the *start* state to the goal. In such cases, greedily aiming for the goal may not serve the long-term objective of finding an optimal path. By embedding this learning objective directly into the action-selection strategy, we expect to focus the search onto a narrower region of the search space. This reduced number of visited states will likely be accompanied by a reduction in the total number of actions until convergence (including repeated visits to some states).

In the next section, we show that keeping the action-selection rule of LRTA* but breaking ties in favor of states with smaller f-values already reduces the number of actions needed to converge. In the following section, we demonstrate that directly selecting actions that minimize f-values reduces this number even more.

2.5 Breaking ties in favor of smaller f-values

LRTA* terminates and eventually follows a shortest path no matter how its action-selection rule breaks ties among successors. In this section, we demonstrate, for the first time, that the tie-breaking criterion crucially influences the convergence speed of LRTA*. We present an experimental study that shows that LRTA* converges significantly faster to a shortest path when it breaks ties towards successors with smallest f-values rather than, say, randomly or towards successors with largest f-values. Recall that, in the A* search method, $f(s)$ is equal to the sum of $g^*(s)$ and $h(s)$, for all states s. To implement our new tie-breaking criterion, LRTA* does not have the g*-values available but can approximate them with g-values. It can update the g-values in a way similar to how it updates the h-values, except that it uses the predecessors instead of the successors. Note that the g-values in our real-time search algorithms do not have the same semantics as the g-values in offline search. Here, a g-value is an underestimate of the cost of a minimum-cost path from the start to the state, not the cost of the best path found so far. Figure 5 shows TB-LRTA* (Tie-Breaking LRTA*), our real-time search method that maintains g- and h-values and breaks ties towards successors with smallest f-values, where $f(s) := g(s) + h(s)$ for all states s. Remaining ties can be broken arbitrarily (but systematically). We compared TB-LRTA* against versions of LRTA* that break ties randomly or towards successors with largest f-values. We performed experiments in thirteen combinations

```
1. $s$ := $s_{start}$
2. $s'$ := $\arg\min_{s'' \in succ(s)}(c(s,s'') + h(s''))$
   Break ties in favor of a successor $s''$ with a smallest f-value, where $f(s'') := g(s'') + h(s'')$
   Break remaining ties arbitrarily (but systematically)
3. $g(s)$ := **if** $(s = s_{start})$ **then** $g(s)$
             **else** $\max(g(s), \min_{s'' \in pred(s)}(g(s'') + c(s'', s)))$
   $h(s)$ := **if** $(s = s_{goal})$ **then** $h(s)$
             **else** $\max(h(s), \min_{s'' \in succ(s)}(c(s,s'') + h(s'')))$
4. **If** $(s = s_{goal})$ **then** stop successfully
5. $s$ := $s'$
6. **Go to** Line 2
```

Figure 5: The TB-LRTA* algorithm

of standard search domains from the artificial intelligence literature and heuristic values, averaged over at least one thousand runs each. Section 2.7 contains information on the domains, heuristic values, and experimental setup, including how we tested for statistical significance. Table 3 shows that in all cases but one (Permute-7 with the zero (Z) heuristic)[3] breaking ties towards successors with smallest f-values (statistically) significantly sped up the convergence of LRTA* in terms of travel cost (action executions).

2.6 FALCONS: Selecting actions that minimize f-values

In this section, we show that turning f-value minimization into the primary action-selection criterion is not trivial. The obvious, naive approach leads to non-termination or convergence to a non-optimal path. We then show how to solve these problems in our final version of FALCONS.

2.6.1 FALCONS: A naive approach

We just showed that TB-LRTA* converges significantly faster than LRTA* because it breaks ties towards successors with smallest f-values. We thus expect real-time search methods that implement this principle more consequently and always move to successors with smallest f-values to converge even faster. Figure 6 shows Naive FALCONS (FAst Learning and CONverging Search), our real-time search method that maintains g- and h-values, always moves to successors with smallest f-values, and breaks ties to minimize the estimated cost-to-goal. Remaining ties can be broken

[3]This exception will disappear in our results with FALCONS.

Table 3: Travel cost to convergence with different tie-breaking rules

domain and heuristic values		LRTA* that breaks ties ... towards a largest f-value	randomly	towards a smallest f-value (TB-LRTA*)
8-Puzzle	M	64,746.47	45,979.19	**18,332.39**
	T	911,934.40	881,315.71	**848,814.91**
	Z	2,200,071.25	2,167,621.63	**2,141,219.97**
Gridworld	N	116.50	97.32	**82.08**
	Z	1,817.57	1,675.87	**1,562.46**
Permute-7	A	302.58	298.42	**288.62**
	Z	**16,346.56**	16,853.69	16,996.51
Arrow	F	1,755.42	1,621.26	**1,518.27**
	Z	7,136.93	7,161.71	**7,024.11**
Tower of Hanoi	D	145,246.55	130,113.43	**116,257.30**
	Z	156,349.86	140,361.39	**125,332.52**
Words	L	988.15	813.66	**652.95**
	Z	16,207.19	16,137.67	**15,929.81**

arbitrarily (but systematically). To understand why ties are broken to minimize the estimated cost-to-goal, consider g- and h-values that are perfectly informed. In this case, all states on a shortest path have the same (smallest) f-values and breaking ties to minimize the estimated cost-to-goal ensures that Naive FALCONS moves towards the goal. (All real-time search methods discussed in this chapter have the property that they follow a shortest path right away if the g- and h-values are perfectly informed.) To summarize, Naive FALCONS is identical to TB-LRTA* but switches the primary and secondary action-selection criteria. Unfortunately, we show in the remainder of this section that Naive FALCONS does not necessarily terminate nor converge to a shortest path. In both cases, this is due to Naive FALCONS being unable to increase misleading f-values of states that it visits, because they depend on misleading g- or h-values of states that it does not visit and thus cannot increase.

Naive FALCONS can cycle forever. Figure 7 shows an example of a domain where Naive FALCONS does not terminate for g- and h-values that are admissible but inconsistent. Naive FALCONS follows the cyclic path $s_0, s_1, s_2, s_3, s_2, s_3, \ldots$ without modifying the g- or h-values of any state. For example during the first trial, Naive FALCONS updates $g(s_2)$ to one (based on $g(s_7)$) and $h(s_2)$ to one (based on $h(s_6)$), and thus does not modify them. $g(s_7)$ and $h(s_6)$ are both zero and

```
1. $s := s_{start}$
2. $s' :=$ arg $\min_{s'' \in succ(s)} f(s'')$, where $f(s'') := g(s'') + h(s'')$
   Break ties in favor of a successor $s''$ with the smallest value of $c(s, s'') + h(s'')$
   Break remaining ties arbitrarily (but systematically)
3. $g(s) :=$ if $(s = s_{start})$ then $g(s)$
              else $\max(g(s), \min_{s'' \in pred(s)}(g(s'') + c(s'', s)))$
   $h(s) :=$ if $(s = s_{goal})$ then $h(s)$
              else $\max(h(s), \min_{s'' \in succ(s)}(c(s, s'') + h(s'')))$
4. If $(s = s_{goal})$ then stop successfully
5. $s := s'$
6. Go to Line 2
```

Figure 6: Naive FALCONS (initial, non-functional version)

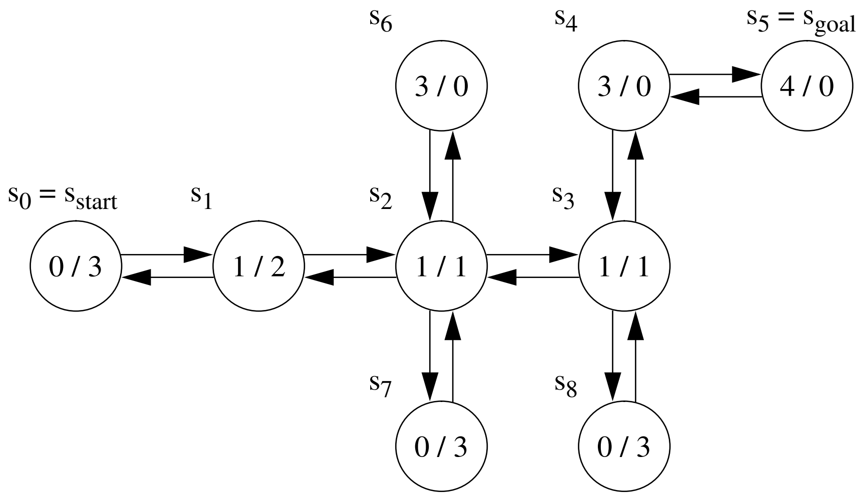


Figure 7: Naive FALCONS cycles forever (Each circle represents a state with its g-value/h-value)

thus strictly underestimate the true start and goal distances of their respective states. Unfortunately, the successor of state s_2 with the smallest f-value is state s_3. Thus, Naive FALCONS moves to state s_3 and never increases the misleading $g(s_7)$ and $h(s_6)$ values. Similarly, when Naive FALCONS is in state s_3 it moves back to state s_2, and thus cycles forever.

Naive FALCONS can converge to sub-optimal paths. Figure 8 shows an example of a domain where Naive FALCONS terminates but converges to a sub-optimal path even though the g- and h-values are consistent. Naive FALCONS converges to the sub-optimal path s_0, s_1, s_2, s_3, and s_4. The successor of state s_2 with the smallest f-value is state s_3. $f(s_3)$ is two and thus clearly underestimates $f^*(s_3)$. Even though Naive FALCONS moves to state s_3, it never increases its f-value

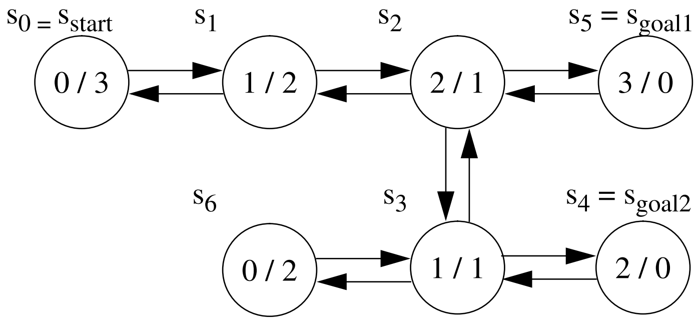


Figure 8: Naive FALCONS converges to a sub-optimal path (Each circle represents a state with its g-value/h-value)

```
1. $s := s_{start}$
2. $s' := \arg\min_{s'' \in succ(s)} f(s'')$, where $f(s'') := \max(g(s'') + h(s''), h(s_{start}))$
   Break ties in favor of a successor $s''$ with the smallest value of $c(s, s'') + h(s'')$
   Break remaining ties arbitrarily (but systematically)
3. $g(s)$ := **if** $(s = s_{start})$ **then** $g(s)$
             **else** $\max(g(s),$
                        $\min_{s'' \in pred(s)}(g(s'') + c(s'', s)),$
                        $\max_{s'' \in succ(s)}(g(s'') - c(s, s'')))$
   $h(s)$ := **if** $(s = s_{goal})$ **then** $h(s)$
             **else** $\max(h(s),$
                        $\min_{s'' \in succ(s)}(c(s, s'') + h(s'')),$
                        $\max_{s'' \in pred(s)}(h(s'') - c(s'', s)))$
4. **If** $(s = s_{goal})$ **then** stop successfully
5. $s := s'$
6. **Go to** Line 2
```

Figure 9: The FALCONS algorithm (final version)

because it updates its g-value to one (based on $g(s_6)$) and $h(s_3)$ to one (based on $h(s_4)$), and thus does not modify them. Naive FALCONS then moves to state s_4. Thus, the trial ends and Naive FALCONS has followed a sub-optimal path. Since no g- or h-values changed during the trial, Naive FALCONS has converged to a sub-optimal path.

2.6.2 FALCONS: The final version

In the previous section, we showed that Naive FALCONS does not necessarily terminate nor converge to a shortest path. Figure 9 shows the final (improved) version of FALCONS that solves both problems. Appendix A contains proofs that the following theorems hold under our assumptions.

Theorem 1 *Each trial of FALCONS terminates.*

Theorem 2 *FALCONS eventually converges to a path from the start to the goal if it is reset to the start whenever it reaches the goal and maintains its g- and h-values from one trial to the next one.*

Theorem 3 *The path from the start to the goal that FALCONS eventually converges to is a shortest path.*

We now give some intuitions behind the new value-update and action-selection rules and show that they solve the problems of Naive FALCONS for the examples introduced in the previous section.

FALCONS terminates. The new value-update rules of FALCONS cause it to terminate. We first derive the new value-update rule for the h-values. It provides more informed but still admissible estimates of the h-values than the old value-update rule, by making better use of information in the neighborhood of the current state. The new value-update rule makes the h-values locally consistent and is similar to the pathmax equation used in conjunction with A* [121]. If the h-values are consistent, then there is no difference between the old and new value-update rules. To motivate the new value-update rule, assume that the h-values are admissible and FALCONS is currently in some state s with $s \neq s_{goal}$. The old value-update rule used two lower bounds on the goal distance of state s, namely $h(s)$ and $\min_{s'' \in succ(s)}(c(s, s'') + h(s''))$. The new value-update rule adds a third lower bound, namely $\max_{s'' \in pred(s)}(h(s'') - c(s'', s))$. To understand the third lower bound, note that the goal distance of any predecessor s'' of state s is at least $h(s'')$ since the h-values are admissible. This implies that the goal distance of state s is at least $h(s'') - c(s'', s)$. Since this is true for all predecessors of state s, the goal distance of state s is at least $\max_{s'' \in pred(s)}(h(s'') - c(s'', s))$. The maximum of the three lower bounds then is an admissible estimate of the goal distance of state s and thus becomes its new h-value. This explains the new value-update rule for the h-values. The new value-update rule for the g-values can be derived in a similar way.

As an example, we show that Naive FALCONS with the new value-update rules now terminates in the domain from Figure 7. When Naive FALCONS is in state s_2 during the first trial, it increases both $g(s_2)$ and $h(s_2)$ to two and then moves to state s_3. The successor of state s_3 with the smallest f-value is state s_4, and no longer state s_2, because $f(s_2)$ was increased to four. Thus, Naive FALCONS now moves to state s_4 and breaks the cycle. Unfortunately, the new value-update rules are not sufficient to guarantee that Naive FALCONS converges to a shortest path. The domain from Figure 8 still provides a counterexample.

FALCONS converges to a shortest path. The new action-selection rule of FALCONS causes it to converge to a shortest path by using more informed but still admissible estimates of the f*-values. In the following, we assume that the g- and h-values are admissible and we present two lower bounds on $f^*(s)$. First, $f^*(s)$ is at least $g(s) + h(s)$, since the g- and h-values are admissible. Second, $f^*(s)$ is at least as large as the cost of a shortest path from the start to the goal, a lower bound of which is $h(s_{start})$, since the h-values are admissible. The maximum of the two lower bounds is an admissible estimate of $f^*(s)$ and thus becomes the new f-value of s. This explains the new calculation of the f-values performed by the action-selection rule. The other parts of the action-selection rule remain unchanged. The new f-value of state s, unfortunately, cannot be used to update its g- or h-values, because it is unknown by how much to update the g-value and by how much to update the h-value.

As an example, we show that FALCONS now converges to a shortest path in the domain from Figure 8. When FALCONS reaches state s_2 in the first trial, $f(s_3)$ is now three. All three successors of state s_2 have the same f-value and FALCONS breaks ties in favor of the one with the smallest h-value, namely state s_5. Thus, the trial ends and FALCONS has followed a shortest path. Since no g- or h-values changed, FALCONS has converged to a shortest path.

2.7 Experimental results

In this section, we present our empirical evaluation of FALCONS, which we compared to LRTA* that breaks ties randomly and TB-LRTA*. We describe, in turn, our domains and heuristic functions, our performance measures, our empirical setup, and finally our results.

2.7.1 Domains and heuristics

For our empirical study, we used the following domains from the artificial intelligence literature.

The **8-Puzzle** domain [98] consists of eight tiles (numbered one through eight) in a 3x3 grid, leaving one position blank. A move is performed by sliding one of the tiles adjacent to the blank into the blank position. Since tiles are not allowed to move diagonally, the number of possible moves in

Table 4: Travel cost to convergence with different action-selection rules

domain and heuristic values		LRTA* that breaks tie randomly		TB-LRTA*	FALCONS	
8-Puzzle	M	45,979.19	(100%)	**18,332.39**	**18,332.39**	(39.87%)
	T	881,315.71	(100%)	848,814.91	**709,416.75**	(80.50%)
	Z	2,167,621.63	(100%)	2,141,219.97	**1,955,762.18**	(90.23%)
Gridworld	N	97.32	(100%)	82.08	**57.40**	(58.98%)
	Z	1,675.87	(100%)	1,562.46	**1,440.02**	(85.93%)
Permute-7	A	298.42	(100%)	288.62	**284.95**	(95.49%)
	Z	16,853.69	(100%)	16,996.51	**16,334.67**	(96.92%)
Arrow	F	1,621.26	(100%)	1,518.27	**1,372.62**	(84.66%)
	Z	7,161.71	(100%)	7,024.11	**6,763.49**	(94.44%)
Tower of Hanoi	D	130,113.43	(100%)	116,257.30	**107,058.94**	(82.28%)
	Z	140,361.39	(100%)	125,332.52	**116,389.79**	(82.92%)
Words	L	813.66	(100%)	652.95	**569.71**	(70.02%)
	Z	16,137.67	(100%)	15,929.81	**15,530.42**	(96.24%)

each configuration is at most four: up, right, down or left. The goal state is the configuration with the blank in the center and the tiles positioned in increasing order, starting at the upper left corner and proceeding in a clockwise fashion. We used 1000 randomly selected start states among those from which the goal is reachable. In this domain, we experimented with the Manhattan distance (the sum, for all tiles, of their horizontal and vertical distances from their respective goal positions), abbreviated *M*, and the "Tiles Out Of Order" heuristic (the number of misplaced tiles), abbreviated *T*.

For the **Gridworld** domain [79], we used a set of 20x20 grids in which 35 percent of the 20^2 grid cells were randomly selected as untraversable obstacles. For each grid, the start and goal positions were chosen randomly, while making sure that the goal was reachable from the start. Since we allowed moves to any of the traversable neighboring locations (including diagonal moves), we modified the Manhattan distance heuristic to be the sum, over all tiles, of the maximum of the tile's horizontal and vertical distances to its goal position. This heuristic was abbreviated *N*.

In the **Permute-7** domain [68], a state is a permutation of the integers 1 through 7. Therefore, the state space has $7! = 5040$ states. There are 6 operators. Each operator Op_k $(k = 2, \ldots, 7)$ is applicable in all states and reverses the order of the first k integers in the state it is executed in. For example, the execution of Op_4 in state 7654321 leads to state 4567321. The goal state is 1234567. The *adjacency heuristic* (abbreviated A) computes for each state s the number of pairs of adjacent digits in the goal state that are not adjacent in s. For instance, $A(7321645) = 3$ since exactly three pairs are adjacent in the goal but not in s, namely $(3, 4)$, $(5, 6)$ and $(6, 7)$. We experimented with all 5040 states as start state.

We also used a version of the **Tower of Hanoi** domain [68] with 7 disks and 3 pegs. In the goal state, all disks are on the same peg, say peg number three. We experimented with 1000 randomly chosen start states. The *D* heuristic simply counts the number of disks that are not on the goal peg.

The **Words** domain [70] is a connected graph whose 4493 nodes are 5-letter English words that are pairwise connected if they differ in exactly one letter. The goal state is the word "goals". We experimented with 1000 randomly chosen start states. The *L* heuristic computes the number of positions (between 1 and 5) for which the letter is different from the letter at the same position in the goal state.

In the **Arrow** domain [95], a state is an ordered list of 12 arrows. Each arrow can either point up or down. There are 11 operators that can each invert a pair of adjacent arrows. The goal state has all arrows pointing up. We experimented with 1000 randomly chosen start states among those from which the goal is reachable. The *F* heuristic returns the largest integer that is not larger than the number of arrows that need to be flipped divided by two.

In addition to the above domain-dependent heuristic values, we also experimented in all domains with the constant function Zero (Z). Note that all of our domains share the following two properties: (1) they are undirected, which means that for every action leading from state s to state s' with cost *c*, there is a reverse action from s' to s with cost *c*, and (2) they have uniform costs, which means that all action costs are one. Finally, all of these domains and heuristic functions satisfy our assumptions.

2.7.2 Performance measures

So far, we have motivated our new action-selection rule in terms of an expected reduction in the runtime to convergence. In this section, we discuss this and other relevant performance measures.

Number of actions until convergence. The number of expansions is a common way of measuring the performance of heuristic search algorithms in general [131, 96, 105]. Since real-time heuristic search with lookahead one only expands the states it visits, the number of expansions is equal to the number of actions it executes. The number of actions until convergence (also referred to as "travel cost to convergence") is therefore our primary performance measure in this chapter. The reason this performance measure is used in lieu of the runtime itself is because the latter is typically sensitive to both the implementation and the architecture of the machine on which it is run. In contrast, the number of actions depends only on the algorithm itself and thus makes it easier for different research teams to compare and reproduce empirical results. Nevertheless, since the time needed for each action selection is bounded by a constant (in domains with a finite maximum branching factor), the total runtime of real-time search algorithms is equal to the product of this constant and of our primary performance measure. A decrease in the latter must be weighed against any increase of the constant itself.

Number of trials until convergence. Since a real-time search agent is reset into the start state whenever it reaches the goal, its behavior is episodic. We have called each episode a trial. So far, we have implicitly assumed that the total time until convergence is equal to the sum of the times spent in all trials. This assumes that the inter-trial time is negligible. However, there are domains (for example, when a robot is learning to juggle) in which resetting the agent into its initial state is time-consuming. In such domains, reducing the number of trials may significantly reduce the total learning time in practice. In other cases, such as robot simulations, inter-trial time *is* negligible. To take this factor into account in a domain-independent way, we propose to use the number of trials until convergence as another performance measure.

Number of actions in the first trial. In the learning behavior of real-time search agents, there is a possible trade-off between how many times they reach the goal (that is, the number of trials) and how much effort they spend reaching the goal (that is, the effort per trial). It is possible that additional exploration of the state space within a trial will reduce the total number of trials needed to converge. This is a trade-off between short-term (getting to the goal as fast as possible) and long-term (converging to an optimal solution) objectives. We therefore propose to measure the effort spent in the first trial as an indication of how much exploration is performed at the beginning of learning. Our last performance measure will thus be the number of actions in the first trial.

2.7.3 Empirical setup

In order for FALCONS to converge to a unique path, the secondary tie-breaking criterion must be systematic (*systematic tie-breaking* was defined in Section 2.3). We enforced systematicity by (1) choosing an arbitrary ordering for the successors of each state and (2) breaking remaining ties according to that ordering. The ordering was selected randomly at the beginning of a run and did not change during the run.

An *experiment* refers to a sequence of n runs of an algorithm in one domain with a given set of heuristic values. To attain statistical significance, we averaged our results over $n = 1000$ runs, except in the Permute-7 domain for which each experiment consisted of $7! = 5040$ runs, one for each possible start state. In general, the n runs of an experiment only differed from the other runs in the same experiment in two respects: (1) the start state, and (2) the random ordering selected at the beginning of each run to be used for systematic tie-breaking. In addition, in the Gridworld domain, each run used a different grid and goal state.

There are two advantages to using systematic tie-breaking. First, it ensures that FALCONS will converge to a unique path. If tie-breaking is not systematic, then FALCONS may not converge to a unique path. Instead, it may converge to a set of shortest paths and randomly switch between them after the heuristic values have converged, just like LRTA* [98]. Systematic tie-breaking thus facilitated the detection of convergence, which happens when no heuristic value changes in the course of a run.

Second, systematic tie-breaking allowed us to carefully control our experimental conditions. In particular, we compared pairs of experiments that only differed in the algorithm tested (for example, FALCONS versus LRTA*). We only compared pairs of experiments in the same domain and with the same heuristic values. In addition, we used the same (random) ordering of successor states for systematic tie-breaking in all pairs of runs to be compared. In other words, when comparing algorithm 1 with algorithm 2, run 1 of both experiments used the same ordering, run 2 of both experiments used the same ordering (but different from that of run 1), etc. Furthermore, each pair of corresponding runs used the same start state (and the same grid and goal state in the Gridworld domain). Now, assume that we wanted to compare the travel cost to convergence of FALCONS in a

Table 5: Trials to convergence with different action-selection rules

domain and heuristic values		LRTA* that breaks tie randomly		TB-LRTA*	FALCONS	
8-Puzzle	M	214.37	(100%)	**58.30**	**58.30**	(27.20%)
	T	1,428.57	(100%)	1,214.63	**797.26**	(55.81%)
	Z	1,428.59	(100%)	1,227.74	**756.47**	(52.95%)
Gridworld	N	6.06	(100%)	5.01	**2.90**	(47.85%)
	Z	32.02	(100%)	26.30	**19.77**	(61.74%)
Permute-7	A	26.91	(100%)	25.55	**22.10**	(82.13%)
	Z	117.82	(100%)	92.63	**75.22**	(63.84%)
Arrow	F	114.94	(100%)	110.60	**89.01**	(77.44%)
	Z	171.50	(100%)	135.13	**105.92**	(61.76%)
Tower of Hanoi	D	214.47	(100%)	177.96	**109.13**	(50.88%)
	Z	216.77	(100%)	166.55	**101.44**	(46.80%)
Words	L	32.82	(100%)	22.72	**18.40**	(56.06%)
	Z	71.86	(100%)	55.77	**50.10**	(69.72%)

particular domain and with a particular set of heuristic values (experiment 1) with that of LRTA* in the same domain and with the same set of heuristic values (experiment 2). Our experimental setup guaranteed that the only difference between run i ($i = 1, \ldots, n$) of experiment 1 and run i of experiment 2 was the algorithm tested, whereas each run was made under different conditions (namely, start state and ordering of successor states) from all of the other runs in the same experiment. This setup enabled us to test our results for statistical significance using the paired-samples Z test.

2.7.4 Results

Tables 4, 5, and 6 report the travel cost (action executions) until convergence, the number of trials until convergence, and the travel cost of the first trial, respectively.

Table 4 shows that, in all cases, FALCONS converged to a shortest path with a smaller travel cost (action executions) than LRTA* that breaks ties randomly and, in all cases but one, faster than TB-LRTA*. The percentages in the last column compare the travel cost of FALCONS with that of LRTA*. FALCONS converged 18.57 percent faster over all thirteen cases and in one case even 60.13 percent faster. All the comparisons stated above are significant at the five-percent confidence level. The heuristic values for each domain are listed in order of their decreasing informedness (sum of

Table 6: Travel cost of the first trial with different action-selection rules

domain and heuristic values		LRTA* that breaks tie randomly		TB-LRTA*	FALCONS	
8-Puzzle	M	**311.18**	(100%)	452.84	452.84	(145.52%)
	T	1,342.75	(100%)	**970.87**	1,057.86	(78.78%)
	Z	81,570.22	(100%)	81,585.44	**81,526.34**	(99.95%)
Gridworld	N	**12.15**	(100%)	12.70	20.92	(172.18%)
	Z	**182.37**	(100%)	182.55	183.13	(100.42%)
Permute-7	A	8.14	(100%)	**7.75**	8.13	(99.88%)
	Z	**2,637.86**	(100%)	2,639.13	2,639.13	(100.05%)
Arrow	F	**15.85**	(100%)	16.62	33.61	(212.05%)
	Z	**1,016.33**	(100%)	1,016.83	1,016.83	(100.05%)
Tower of Hanoi	D	4,457.86	(100%)	**3,654.80**	3,910.46	(87.72%)
	Z	4,839.49	(100%)	4,803.81	**4,801.84**	(99.22%)
Words	L	**24.27**	(100%)	27.79	37.80	(155.75%)
	Z	**2,899.73**	(100%)	2,900.36	2,900.68	(100.03%)

the heuristic values over all states). For example, the (completely uninformed) zero (Z) heuristic is listed last. Table 4 then, shows that the speedup of FALCONS over LRTA* was positively correlated with the informedness of the heuristic values. This suggests that FALCONS makes better use of the given heuristic values. Notice that it cannot be the case that FALCONS converges more quickly than LRTA* because it looks at different (or more) states than LRTA* when selecting successor states. FALCONS looks at both the predecessors and successors of the current state while LRTA* looks only at the successors, but all of our domains are undirected and thus every predecessor is also a successor. This implies that FALCONS and LRTA* look at exactly the same states.

Table 5 shows that, in all cases, FALCONS converged to a shortest path with a smaller number of trials than LRTA* that breaks ties randomly and, in all cases but one, faster than TB-LRTA*. FALCONS converged 41.94 percent faster over all thirteen cases and in some cases even 72.80 percent faster.

To summarize, Table 4 and Table 5 show that FALCONS converges faster than LRTA* and even TB-LRTA*, both in terms of travel cost and trials.

We originally expected that FALCONS would increase the travel cost during the first trial, since the action-selection rule of LRTA* (minimize the cost-to-goal) has experimentally been shown to

Table 7: Travel cost to convergence with different action-selection rules, and with or without g updates for FALCONS

domain and heuristic values		LRTA*		FALCONS		FALCONS without g updates	
8-Puzzle	M	45,979.19	(100%)	**18,332.39**	(39.87%)	19,222.08	(41.81%)
	T	881,315.71	(100%)	**709,416.75**	(80.50%)	817,078.12	(92.71%)
Gridworld	N	97.32	(100%)	**57.40**	(58.98%)	58.82	(60.44%)
Permute-7	A	298.42	(100%)	284.95	(95.49%)	**263.00**	(88.13%)
Arrow	F	1,621.26	(100%)	**1,372.62**	(84.66%)	1,533.11	(94.56%)
T. of Hanoi	D	130,113.43	(100%)	**107,058.94**	(82.28%)	128,987.97	(99.14%)
Words	L	813.66	(100%)	569.71	(70.02%)	**547.35**	(67.27%)

result in a small travel cost during the first trial under various conditions. Table 6 shows that, in four of the thirteen cases, the travel cost of FALCONS during the first trial was larger than that of LRTA*; in seven cases it was approximately the same (99 percent to 101 percent); and in two cases it was lower. The travel cost of FALCONS during the first trial was 19.35 percent larger than that of LRTA* over the thirteen cases. Overall, there is no systematic relationship between the travel cost of FALCONS and LRTA* during the first trial, and the sum of planning and plan-execution times is always small for FALCONS, just like for LRTA*.

So far, our main performance measure has been the travel cost to convergence. One may complain that the speedup exhibited by FALCONS over LRTA* comes at an extra computational cost, namely an extra value update per action execution. To decrease the total computational cost (value updates), FALCONS would have to cut the travel cost to convergence at least in half. However, it reduces the travel cost by only 18.57 percent. We also compared FALCONS with a variant of LRTA* that performs two value updates per action execution. This can be done in various ways. Among the ones we tried, our best results were obtained with a variant of LRTA* that first updates $h(s')$ (where s' is the successor of the current state s with the smallest $c(s, s') + h(s')$), then updates $h(s)$, and finally selects the successor s'' of s with the smallest $c(s, s'') + h(s'')$, which may be different from s'. Empirically, this algorithm had a smaller travel cost to convergence than FALCONS.

However, we can modify FALCONS so that it never updates the g-values, resulting in one value-update per action execution, just like LRTA*. Table 7 reports experimental results that clearly show that FALCONS without g updates had a smaller travel cost to convergence than LRTA* (with lookahead one). The speedup was 22.28 percent on average, and up to 58.19 percent. Additional results show that the number of trials to convergence for FALCONS without g updates was 25.97 percent less than for LRTA* on average (and up to 68.71 percent less), and that FALCONS executed an average of 57.51 percent more actions than LRTA* in the first trial.[3] These results are important for two reasons. First, they support the claim that the action-selection rule of FALCONS speeds up convergence by making better use of the available heuristic knowledge and is able to decrease both the travel cost and computational cost to convergence. Second, they suggest that FALCONS may benefit from an enhanced action-selection rule that focuses the search even more sharply around an optimal path by speeding up the learning of more accurate g-values, while still making efficient use of the initial heuristic knowledge.

2.8 Related work

[79] presents an overview of real-time heuristic search algorithms and their application to moving-target search [81, 76, 82, 152] and bidirectional search [77, 78]. Multi-agent extensions of real-time search have also been discussed [89, 87, 171, 80]. We now focus on single-agent real-time search approaches that are more closely related to LRTA* and FALCONS.

HLRTA* [164] is representative of a class of methods (such as ELRTA* [164] and SLRTA* [34]) that speed up the convergence of LRTA* by using a different value-update rule.[4] This is in contrast to FALCONS that uses a different action-selection rule. Both improvements to LRTA* are orthogonal and guarantee the optimality of the final (that is, converged) solution. In [51, 46], we present the first thorough empirical evaluation of HLRTA* and show that it and FALCONS have complementary strengths that can be combined. We call the resulting real-time search method eFALCONS (for Even FAster Learning and CONverging Search) and show that it converges with fewer actions to a minimum-cost plan than LRTA*, HLRTA*, and FALCONS, even though it looks

[3]In domains with uniform costs, with consistent h-values, and with zero-initialized g-values, FALCONS without g updates reduces to LRTA*. Thus, Table 7 does not show results for completely uninformed heuristic values and our averages do not include them.

[4]We thank Stefan Edelkamp for introducing us to HLRTA* and Richard Korf for making Thorpe's thesis about HLRTA* available to us.

at the same states when it selects successors on undirected graphs and is not more knowledge-intensive to implement. However, the main drawback of eFALCONS is that its runtime overhead per state expansion is larger than that of its component algorithms since it maintains more heuristic values per state. This overhead is typically larger than the runtime savings due to the reduction in the number of actions to convergence.

δ-search is another variant of LRTA* that uses a pruning mechanism to control its exploration [153]. This mechanism is orthogonal to the aforementioned improvements on LRTA*. Shimbo et al. show that FALCONS converges faster than δ-search in the 8-Puzzle and gridworld domains.[5] Furthermore, their experiments demonstrate that FALCONS significantly reduces space requirements over LRTA*. This is important since it provides empirical evidence that the speedup exhibited by FALCONS is accompanied by a sharper focus of the search around the optimal solution it converges to. [153] also introduces another variant of LRTA* called ϵ-search that speeds up its convergence while sacrificing the optimality of the converged solution by putting more weight on the h-values when computing the estimated cost to the goal. Again, this variation is orthogonal to the aforementioned ones.

Finally, there exists a class of real-time search algorithms that add a backtracking mechanism to LRTA* in order to speed up convergence by propagating value updates backward. This class includes SLA* [155], SLA*T [156], and γ-Trap [17]. SLA* and γ-Trap use essentially the same backtracking strategy but were developed independently. SLA*T, which extends SLA* with an additional parameter to control the amount of backtracking and thus the amount of exploration (and the rate of learning), was applied to project scheduling problems with resource constraints (PSRC). The idea of backtracking is orthogonal to the action-selection rule used by FALCONS. However, all the backtracking algorithms make the extra assumption that all actions in the domain are reversible (that is, the graph is undirected). In contrast, FALCONS is applicable to both undirected and directed graphs.

[5]Nevertheless, δ-search does converge faster than LRTA*, even though the motivation for it was different (δ-search was designed to distribute and control the learning across trials).

2.9 Future work

We envision at least two directions for future work in real-time search.

First, the family of real-time search algorithms has grown quite large. There is a need for an exhaustive empirical comparison of these algorithms. Due to the different assumptions they make, this empirical study will require a wide variety of benchmark domains[6] in order to discover a mapping between classes of domains (based, for example, on structural features of their associated state spaces) and the classes of algorithms that are most efficient on them. Not only is the efficiency of real-time search algorithms expected to be domain-dependent, there are numerous ways to measure performance (such as the number of actions to convergence or per trial, the number of trials to convergence, the solution quality as a function of learning or after convergence, the rate of learning, the trade-off between exploration and exploitation, etc.). Furthermore, most of the foregoing enhancements to LRTA* are orthogonal, including changes to the value-update rule, changes to the action-selection rule, the increased weight on the h-values, the addition of pruning rules, the use of a larger and even variable lookahead [17], etc.). The number of algorithms resulting from possible combinations is extremely large. Yet, it would be useful to know which combinations work best together and for what tasks. [16] has started such an investigation.

Second, a particularly interesting extension of FALCONS is its application to domains from real-time control. These domains require real-time action selection and convergence to optimal behaviors but, at the same time, the setup for each trial is expensive and thus it is important to keep the number of trials small. For learning how to balance poles or juggle devil-sticks [146], for example, the pole needs to be picked up and brought into the initial position before every trial. Domains from real-time control are typically directed and sometimes probabilistic, and we have not yet applied FALCONS to domains with these properties. Of course, FALCONS can be applied without modification to directed domains since all of our theoretical results continue to hold. The main difficulty of applying FALCONS to probabilistic domains is to adapt the notion of f-values to such domains. When the effects of actions are probabilistic, the agent learns a policy that only has a probability (typically smaller than one) of visiting some states. In this context, it is not obvious

[6]Note that all the domains used in our evaluation of FALCONS happen to be undirected.

how to even define the concept of a minimum path cost through a state (such as an f-value) when the policy is not guaranteed to visit the state.

2.10 Contributions

Our research on real-time heuristic search has yielded the following contributions:

- We have extended the scope of applicability of A*'s principle for ordering state expansions to the real-time search setting. We have shown that, by making real-time search less greedy (namely, by moving towards states with minimum f-values instead of h-values), we can significantly reduce its number of actions to convergence in several domains. Furthermore, this reduction in runtime can be accompanied by a reduction in the number of visited states, and therefore in the space requirements of real-time search, as results in [153] indicate.

- We designed a new action-selection rule for online, dynamic programming methods. We have shown how to implement it in the case of deterministic task domains. This rule has great potential relevance to the reinforcement-learning community, since the vast majority of existing methods in this area use the greedy action-selection rule when exploiting heuristic information. Our results suggest that significantly faster learning could result from a less greedy exploitation rule.

- We successfully implemented FALCONS, a new algorithm for real-time heuristic search. FALCONS exhibits significantly higher performance than state-of-the-art real-time heuristic search methods. More precisely, FALCONS reduces both the number of actions and the number of trials it takes to converge. We resolved non-trivial problems in order to guarantee that FALCONS terminates and converges to a shortest path.

CHAPTER III

SCALING UP WA* WITH COMMITMENT AND DIVERSITY[*]

3.1 Introduction

Adding greediness is a standard way of scaling up A* search to larger problems while sacrificing solution quality. Weighted A* (or WA*) embodies this trade-off by varying the weight it puts on the heuristic values. Recently, two improved versions of WA* have been proposed to speed it up and scale it up to even larger domains. These recent variants of WA* were developed independently and have never been compared. In this chapter, we first compare them empirically in three benchmark domains. Then, we demonstrate the additional benefit of combining them, since the resulting algorithm scales up to larger problems. Finally, we observe the strong similarity between our new algorithm and an existing algorithm called *beam search*. This fuller understanding of the behavior of beam search enables us to propose possible variations on the standard beam search algorithm.

The starting point of this work is the existence of two separate lines of research that have produced two distinct variants of WA*. First, K-Best-First Search (KBFS) introduces diversity in WA* (resulting in the KWA* algorithm [37]) in order to avoid focusing too much search effort in areas of the search space where the heuristic function is misleading. Second, Multi-State Commitment search introduces commitment in WA* (resulting in the MSC-WA* algorithm [88]) in order to give it a stronger depth-first component. While WA* only scales up to the 24-Puzzle in our empirical setup, each of these algorithms scales up to the 35-Puzzle.

Following an empirical comparison of these three algorithms, we proceed to show that the ideas of commitment and diversity can be combined and applied to WA* (see Figure 10). The resulting algorithm, which we call MSC-KWA*, scales up to the 48-Puzzle, while neither MSC-WA* nor KWA* does in our empirical setup. These and similar empirical results with our two other benchmark domains support our hypothesis that WA* solves larger problems when it is enhanced with

[*]This chapter first appeared as [48].

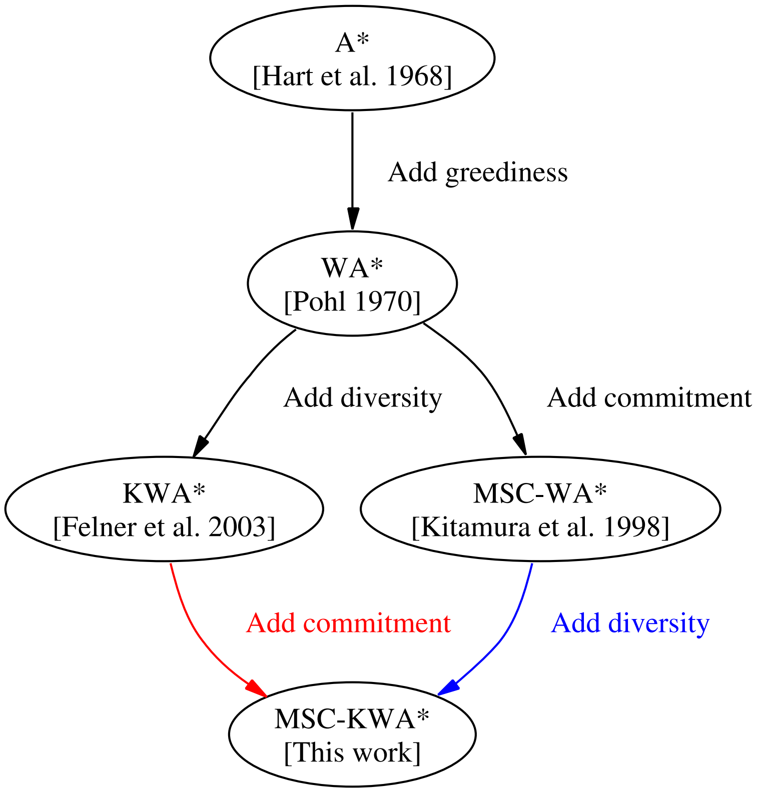


Figure 10: Roadmap for this research

both diversity and commitment.

This chapter is structured as follows. Sections 3.2, 3.3, and 3.4 present the WA*, KWA* and MSC-WA* algorithms, respectively. Section 3.5 motivates and describes the new MSC-KWA* algorithm that results from adding both diversity and commitment to WA*. Section 3.6 reports on our empirical evaluation of all four algorithms in three benchmark domains. Sections 3.7 and 3.8 discuss related and future work, respectively. Finally, Section 3.9 concludes by summarizing our contributions.

3.2 The WA algorithm*

A* [59, 60] is a best-first search algorithm since it always expands next a most promising state in the current list of candidates. This list, called *OPEN*, contains all the generated states that have not yet been expanded. The promise of an open state is represented by an estimate of the cost of a shortest path from the start via the open state to a goal state. This estimate, called the f-value of the state, is the sum of the cost of the shortest path found so far from the start to the state (its g-value)

```
1. procedure WA*($s_{start}$, $heuristic(.)$, $w_g$, $w_h$): solution cost
2.  $g(s_{start}) := 0$; $h(s_{start}) := heuristic(s_{start})$; $OPEN := \{s_{start}\}$; $CLOSED := \emptyset$
3.  while ( $OPEN \neq \emptyset$ ) do
4.   $state :=$ arg $\min_{s \in OPEN} \{ w_g \times g(s) + w_h \times h(s) \}$
5.   $OPEN := OPEN \setminus \{state\}$
6.   $CLOSED := CLOSED \cup \{state\}$
7.   $g := g(state) + 1$
8.   for each successor $s$ of $state$ do
9.    if ( $s = s_{goal}$ ) then return $g$
10.   if ( $s \notin OPEN \cup CLOSED$ ) then $g(s) := g$;$h(s) := heuristic(s)$;$OPEN := OPEN \cup \{s\}$
11.   else if ( $g < g(s)$ ) then
12.     if ( $s \in OPEN$ ) then $OPEN := OPEN \setminus \{s\}$ else $CLOSED := CLOSED \setminus \{s\}$
13.     $g(s) := g$; $OPEN := OPEN \cup \{s\}$
14.  end for
15. end while
16. return $\infty$
```

Figure 11: The WA* algorithm

and the cost of the remaining path from the state to a goal (its h-value). Since a path from an open state to a goal is not yet known, its h-value is computed using a heuristic function. If this function is admissible (that is, no h-value overestimates the true cost of a shortest path to a goal), then A* is complete, it returns an optimal solution, and it is optimally efficient among all admissible best-first search algorithms [131].

Its exponential space- and time-complexity prevents A* from solving large problems. One way to scale up A* is to make the search more greedy by putting a larger weight on h than on g when adding them up to compute the f-value of each state [132, 52]. WA* uses this new definition of the f-value of a state s: $f(s) = w_g \times g(s) + w_h \times h(s)$, $w_h \geq w_g \geq 0$. Equivalently, W is defined as the relative weight on the h-value, that is $W = \frac{w_h}{w_g + h_h}$. $f(s)$ can then be re-written as $(1 - W)g(s) + W \times h(s)$. When $W = 0.5$ (equivalently $w_g = w_h > 0$), WA* reduces to A*. When $W > 0.5$ (equivalently $w_h > w_g$), WA* is more greedy than A*. Increasing W increases the cost of the solution found by WA* (which is not admissible anymore) but it also speeds up WA* by reducing the number of states it generates. [24] shows that, in the worst case, the cost of the solution returned by WA* exceeds the optimal cost by a factor equal to w_h/w_g. In practice, the solution cost returned by WA* is much lower than this upper bound (see, for example, [99], as well as our experimental results reported below).

Figure 11 contains the pseudo-code for our implementation of WA*, which embodies the following assumptions: 1) there is a single goal state denoted s_{goal}, and 2) each operator (or action) has a uniform cost of, say, one. These assumptions, while not essential to the behavior of WA*, make the pseudo-code more concise.[1] WA* takes as input the start state (and implicitly the domain operators), a heuristic function (which associates an h-value with each state), and the weights applied to the g- and h-values in the f-value computations (or, equivalently, a single parameter W). The OPEN list is initialized with the start state (Line 2). Each generated state is stored in memory with its g- and h-value. The former is computed as states are generated during the search (Lines 2&7), while the latter is computed using the given heuristic function (Lines 2&10). The CLOSED list, also initialized on Line 2, contains the set of expanded states. The main loop (Lines 3-15) is executed until the OPEN list is empty, in which case there exists no solution path (Line 16), or until the goal is generated for the first time (Line 9). During each iteration, the best state is selected in OPEN (Line 4). It is moved from OPEN to CLOSED (Lines 5&6), since it is about to be expanded (Lines 8-14). If a successor is newly generated, it is inserted into OPEN (Line 10). Otherwise, it must be in either OPEN or CLOSED. In both cases, if the newly found path is shorter than the best one found so far (Line 11), the successor is removed from its current location (Line 12) and inserted into OPEN with its new, reduced g-value (Line 13). Otherwise, the successor is discarded.

The two extensions of WA* described in Sections 3.3 & 3.4 are alternatives to the following two characteristics of WA*. First, WA* expands only one state per iteration. In contrast, KWA* expands K ($K \geq 1$) states per iteration. Second, WA* keeps (in OPEN) all generated states as potential candidates for expansion. In contrast, MSC-WA* reduces the size of the set of candidates for expansion to a small constant. The next two sections describe these two variants of WA*, respectively.

3.3 The KWA algorithm: Introducing diversity in WA**

WA* is a greedy version of A*. This has two implications. First, like A*, WA* is a best-first search algorithm that always expands next the most promising node, namely an open node with the

[1]The pseudo-code also omits the management of back-pointers that enable the recovery of the solution path when the search terminates at the goal state.

smallest f-value. Second, since it is greedy, WA* puts more weight on the h-value than A* does, and the more so, the closer to 1 the value of W. As a result, WA* is likely to get trapped in regions of the search space in which the h-values are misleading (namely, too low). This problem is common among greedy search algorithms, which are attracted to local minima.

K-best-first search (KBFS) is an answer to this problem: it reduces the greediness of the search by re-introducing a breadth-first component into best-first search [37]. The idea is to expand K nodes ($K \geq 1$) at each iteration of the main loop, instead of just one as in standard best-first search. Applied to WA*, this idea results in the KWA* algorithm.

Figure 12 contains the pseudo-code for KWA*. The only difference with WA* is that Lines 4-6 in Figure 11, in which a single most promising node is selected for expansion, are replaced with Lines 4-10 in Figure 12, in which K most promising nodes are selected for expansion. Later (see Lines 11-21 in Figure 12), these nodes are expanded in parallel so that the set of all their successors is added to OPEN before the next iteration begins. This is the crucial point in order to avoid focusing the search greedily (and often wastefully) around a local minimum.

To illustrate this point, let us consider the following example. Assume the start state has two successors with f-values equal to 10 and 20, respectively. Note that, since we are interested in cases with $W > 1$, the f-values are not monotonically increasing when going down the tree. Further assume that the goal is only reachable via the second successor. Therefore, choosing the first successor to expand next is a mistake for WA*. If the first successor is the root of a large sub-tree whose states also have misleading h- and thus f-values, WA* keeps exploring this sub-tree until all f-values in the OPEN list become larger than or equal to 20. This happens because WA* focuses the search on a single successor of the start state. In contrast, when expanding the K best nodes in OPEN (for $K = 2$, say), KWA* expands the start state in the first iteration. At this point, the OPEN list contains the two successors of the start state. KWA* expands them both in the second iteration. All their successors are added to the OPEN list. Two of them are then selected for expansion. The difference now is that there is a chance that one of them (or both) is in the ‘good’ sub-tree, namely the one rooted at the second successor of the start state. This is a way of adding diversity to the search, as opposed to focusing on a single sub-tree. Of course, this diversity may be advantageous at various levels in the search tree, not just at the start state. Generally speaking, the larger K, the

```
1. procedure KWA*($s_{start}$, $heuristic(.)$, $w_g$, $w_h$, $K$): solution cost
2.   $g(s_{start}) := 0$; $h(s_{start}) := heuristic(s_{start})$; $OPEN := \{s_{start}\}$; $CLOSED := \emptyset$
3.   while ( $OPEN \neq \emptyset$ ) do
4.       $SET := \emptyset$
5.       while ( $(OPEN \neq \emptyset)$ and $(|SET| < K)$ ) do
6.           $state :=$ arg $\min_{s \in OPEN} \{w_g \times g(s) + w_h \times h(s)\}$
7.           $SET := SET \cup \{state\}$
8.           $OPEN := OPEN \setminus \{state\}$
9.           $CLOSED := CLOSED \cup \{state\}$
10.      end while
11.      for each $state$ in $SET$ do
12.          $g := g(state) + 1$
13.          for each successor $s$ of $state$ do
14.              if ( $s = s_{goal}$ ) then return $g$
15.              if ( $s \notin OPEN \cup CLOSED$ ) then
16.                  $g(s) := g$; $h(s) := heuristic(s)$; $OPEN := OPEN \cup \{s\}$
17.              else if ( $g < g(s)$ ) then
18.                if ( $s \in OPEN$ ) then $OPEN := OPEN \setminus \{s\}$ else $CLOSED := CLOSED \setminus \{s\}$
19.                $g(s) := g$; $OPEN := OPEN \cup \{s\}$
20.          end for
21.      end for
22.  end while
23. return $\infty$
```

Figure 12: The KWA* algorithm

larger the probability that a node on the optimal path will be selected during the next iteration. In the extreme case, when $K = \infty$, KWA* reduces to breadth-first search since then all nodes at a given level in the search tree are expanded in parallel. The probability of one of them being on an optimal path is equal to 1. The price to pay for this increased diversity is the risk of generating much more nodes at each iteration than is necessary, as is clear in the extreme case when KWA* reduces to uninformed search.

3.4 The MSC-WA* algorithm: Introducing commitment in WA*

As a greedy version of A*, WA* puts more weight on the h-value than A* does. Therefore, WA* with $W > 0.5$ expands early on some nodes with low h-values even when their g-value is high enough for A* to delay their expansion. As a result, WA* search exhibits a stronger depth-first characteristic than A* search. This enables WA* to solve larger problems than is possible with WA* (at the expense of solution quality). For example, because its memory consumption is reduced, WA* can solve any random instances of the 15-Puzzle, while A* often runs out of memory. To summarize, WA* increases greediness in order to scale up A*.

Another way to scale up A* with a depth-first component, is to focus the search on a sub-set of nodes in the OPEN list. This is the idea behind Multi-State Commitment (MSC) search [88]. Applied to WA*, this idea results in the MSC-WA* algorithm.

WA* keeps in OPEN the end points (that is, nodes) of all paths currently under construction. So WA* does not commit to any region of the search space: based on the lowest f-value, WA* can expand next any node in OPEN, even if it is very distant (in the search space) from the most recently expanded node. WA* can jump around the state space indiscriminately among open nodes. This is beneficial since it allows WA* 1) to stop exploring a promising region of the search space when later expansions reveal that its h-values are misleading and 2) to restart exploration from a different node in OPEN. Unfortunately, this "insurance" against mistakes comes at the cost of growing a wide search front. In this sense, WA* does not make any commitment: any node in OPEN can be expanded next if its f-value warrants it. In short, lack of commitment leads to a wide search front, which in turn results in large memory requirements and thus poor scaling (as demonstrated by the performance of WA* in the 35-Puzzle, see Table 9).

To address this problem, [88] introduces the notion of *commitment*, according to which only a sub-set of the OPEN nodes are currently active and stored in the COMMIT list. Only one of them can be expanded next. The other nodes in OPEN are on the RESERVE list. They are used to refill the COMMIT list when it becomes smaller than its predefined size C. Decreasing C means sharpening the focus of the search to a smaller sub-set of the open nodes. In turn, this introduces a stronger depth-first search component in order to scale up the search to larger problems. Applied to WA* this idea results in MSC-WA*.

The first difference with WA* is that the OPEN list is split into the COMMIT and RESERVE lists. While the former plays the role of a smaller OPEN list, the latter is used as storage for generated nodes that are not currently committed to but that the search may return to later on. This is useful because of the second difference with WA*, namely the fact that MSC-WA* never re-expands a node, even if a shorter path to it is later found (the designers of MSC-WA* made that design choice because they were more interested in finding solutions quickly than in the solution cost [88]). As a result, the COMMIT list may not be filled to capacity or may even become empty.

Figure 13 contains the pseudo-code for MSC-WA*. The algorithm takes one more parameter

```
1. procedure MSC-WA*($s_{start}$, $heuristic(.)$, $w_g$, $w_h$, $C$): solution cost
2.   $g(s_{start}) := 0$; $h(s_{start}) := heuristic(s_{start})$; $COMMIT := \{s_{start}\}$;
3.   $RESERVE := \emptyset$; $CLOSED := \emptyset$
4.   while ( $COMMIT \neq \emptyset$ ) do
5.       $state :=$ arg $\min_{s \in COMMIT}$ $\{ w_g \times g(s) + w_h \times h(s) \}$
6.       $COMMIT := COMMIT \backslash \{state\}$
7.       $CLOSED := CLOSED \cup \{state\}$
8.       for each successor $s$ of $state$ do
9.           if ( $s = s_{goal}$ ) then return $g$
10.          if ( $s \notin COMMIT \cup RESERVE \cup CLOSED$) then
11.              $g(s) := g(state) + 1$; $h(s) := heuristic(s)$; $COMMIT := COMMIT \cup \{s\}$
12.      end for
13.      while ( $|COMMIT| > C$ ) do
14.          $state :=$ arg $\max_{s \in COMMIT}$ $\{ w_g \times g(s) + w_h \times h(s) \}$
15.          $COMMIT := COMMIT \backslash \{state\}$
16.          $RESERVE := RESERVE \cup \{state\}$
17.      end while
18.      while ( ($|COMMIT| < C$) and ($|RESERVE| > 0$) ) do
19.          $state :=$ arg $\min_{s \in RESERVE}$ $\{ w_g \times g(s) + w_h \times h(s) \}$
20.          $RESERVE := RESERVE \backslash \{state\}$
21.          $COMMIT := COMMIT \cup \{state\}$
22.      end while
23.  end while
24. return $\infty$
```

Figure 13: The MSC-WA* algorithm

than WA*, namely the size C of the COMMIT list. The COMMIT list is initialized to the start state, while the RESERVE and CLOSED lists are initially empty (Lines 2&3). Then the main loop (Lines 4-23) is executed until COMMIT is empty (in which case there is no solution, Line 24) or until the goal is generated for the first time (Line 9). At each iteration, the best node in COMMIT is selected for expansion and moved to the CLOSED list (Lines 5-7). Every newly generated node is added to the COMMIT list (Lines 10-11). If the COMMIT list is too large, the least promising nodes are moved to the RESERVE list (Lines 13-17). but if the COMMIT list not full, the most promising nodes in the RESERVE (if any) are used to fill it up (Lines 18-22).

To illustrate the behavior of MSC-WA*, we now consider two extreme cases. When $C = 1$, MSC-WA* performs a greedy search. Starting with the start node, the current node is repeatedly removed from COMMIT (which is now empty) and expanded. All its successors are added into COMMIT. All but the most promising ones are immediately moved to the RESERVE. At the end of each iteration, the COMMIT list only contains the best successor of the current node. This depth-first search stops when the goal is reached or when a dead-end is reached (either because there exists a dead-end in the search space or because no *new* state is reachable from the current one). In the latter case, the search is restarted from the best state in the RESERVE.

When $C = \infty$, the COMMIT list never fills up and the RESERVE list remains empty. As a result, COMMIT behaves like the original OPEN list. In this case, MSC-WA* reduces to a version of WA* that never re-expands a node. When C ranges from 2 to infinity, MSC-WA* becomes less and less focused and resembles more and more best-first search with no commitment. MSC-WA* grows a search tree in which at most C leaves are active at any time. Some children of the just-expanded node may be preferred over nodes in the RESERVE with smaller f-values. This sharper focus explains how MSC-WA* is able to scale up greedy best-first search to larger domains than WA* can handle.

3.5 The MSC-KWA* algorithm: Combining diversity and commitment

In this section, we first contrast the effects of diversity and commitment on the performance of best-first search. Since these ideas are orthogonal, we then show how to combine them into a new algorithm (MSC-KWA*) that scales up to larger problems than is possible with diversity (KWA*) or commitment (MSC-WA*) alone.

3.5.1 Comparing the behaviors of KWA* and MSC-WA*

When assessing the behavior of our heuristic search algorithms, we use four standard performance measures, namely memory consumption, solution cost, search effort, and runtime. Since our primary objective is to scale up best-first search to larger domains, and since the memory usage is the main obstacle in this respect, the memory consumption is our primary focus. When the available memory is enough for an algorithm to terminate successfully on all instances, the solution cost is our secondary performance measure. This preference (over search effort and runtime) is justified by the facts that:

- The search effort (that is, the number of generated nodes) is strongly correlated with the memory consumption, as our empirical results have demonstrated for all the algorithms under consideration.
- The runtime is on the order of seconds since the memory typically available on current PC's is quickly filled up by best-first search algorithms such as variants of WA*.[2]

In summary, our comparison focuses primarily on the functional relationship between the memory consumption and the solution cost.

In this sub-section, we use a preview of our empirical results in the N-puzzle to compare the behaviors of KWA* and MSC-WA*. Since each of the KWA* and MSC-WA* algorithms has two parameters (namely, W and either K or C), our full empirical results are structured as two sets of memory-cost functions parameterized on each parameter (see Section 3.6). Here in contrast, we abstract away the influence of these parameters and use graphs that associate with each solution cost the minimum memory consumption over all settings of the parameters. We thus obtain a single curve per algorithm. In essence, we consider an idealized situation in which an oracle tells us the parameter settings that minimize the memory consumption for a given solution cost. The resulting curves are shown in Figure 14 for the N-Puzzle with $N = 8, 15, 24, 35$.

We observe the following trends:

[2]In this chapter, we do not consider memory-bounded algorithms (e.g., [96, 19, 143, 86, 177]) because, even though they do not run out of memory, they exhibit unacceptably large runtimes due to their node-regeneration overhead.

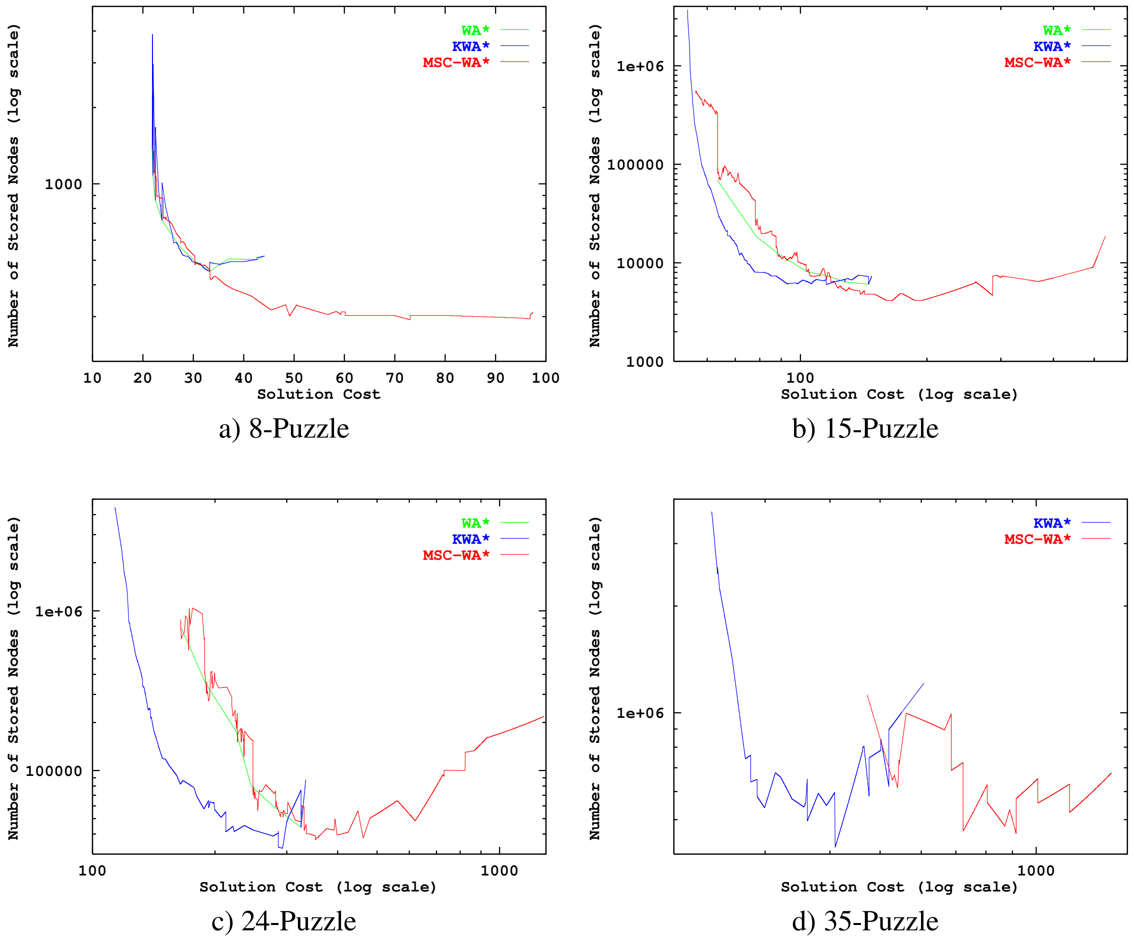


Figure 14: Performance comparison: WA*, KWA*, and MSC-WA* in the N-Puzzle

- The larger the domain, the more KWA* reduces the memory consumption (for a given solution cost) over WA*. While there is no significant improvement in the 8-Puzzle, the reduction is significant in the 15-Puzzle and even larger (about an order of magnitude) in the 24-Puzzle. Because of this effect, KWA* is able to find shorter solutions than WA* in both the 15- and 24-Puzzle within the available memory. In other words, KWA* enlarges the range of reachable solution costs toward the small end (that is, toward the left in the figures) while reducing the memory consumption (that is, a shift toward the bottom in the figures). Because of its reduced memory consumption, KWA*, unlike WA*, is able to solve the 35-Puzzle.

- While MSC-WA* does not improve over the memory consumption of WA* (over the range of solution costs obtainable by WA*), its main advantage, from our perspective, is that it enlarges this range toward the large end (that is, toward the right in the figures) while reducing its memory consumption (that is, toward the bottom in the figures). Unfortunately, the magnitude of this effect seems to decrease as N increases. Nevertheless, MSC-WA* can solve the 35-Puzzle, which WA* cannot.

- As a result, KWA* and MSC-WA* improve over WA* in two different ways. Because of its stronger breadth-first search component, KWA* improves solution quality so that it moves (and extends) the WA* curve toward the left. Because of its stronger depth-first search component, MSC-WA* improves its memory consumption so that it extends the WA* curve toward the bottom (and the right).

Table 8: Comparison of WA*, KWA*, and MSC-WA* in the N-Puzzle

N	Perf.	WA*			KWA*				MSC-WA*			
	Measure	Value	W	Best	Value	W	K	Best	Value	W	C	Best
8	Min Cost	21.85	0.50	✓	21.85	0.50	2	✓	22.01	0.50	800	
	Min Sto.	452	0.86		464	0.86	2		292	0.80	2	✓
	Min Gen.	514	0.86		519	0.99	4		296	0.80	2	✓
	Min Time											
15	Min Cost	63.51	0.67		53.85	0.67	50K	✓	56.29	0.60	80K	
	Min Sto.	6,050	0.99		6,028	0.99	8		4,113	0.95	20	✓
	Min Gen.	6,972	0.99		6,704	0.99	8		4,191	0.95	20	✓
	Min Time	0.003	0.99		0.003	0.99	5		0.002	0.99	6	✓
24	Min Cost	165.16	0.75		113.56	0.99	20K	✓	164.56	0.75	90K	
	Min Sto.	44,097	0.99		32,567	0.99	5	✓	36,907	0.99	300	
	Min Gen.	56,070	0.99		43,578	0.99	4		37,832	0.99	300	✓
	Min Time	0.027	0.99		0.021	0.99	4	✓	0.021	0.99	50K	✓
35	Min Cost				236.50	0.99	7K	✓	472.10	0.90	3K	
	Min Sto.				417,675	0.95	20	✓	456,777	0.99	90	
	Min Gen.				652,100	0.95	500		467,586	0.99	90	✓
	Min Time				0.377	0.95	500		0.297	0.99	90	✓
48	Min Cost											
	Min Sto.											
	Min Gen.											
	Min Time											

Table 8 provides a different view of the empirical comparison of WA*, KWA*, and MSC-WA* — one that moves from curves to single points, and one that encompasses all four of our performance measures. Each row (one for each size N of the puzzle and each performance measure) reports the minimum value obtained by each algorithm for this performance measure. Each row also reports, for each algorithm, one parameter setting with which the algorithm reaches the minimum value of this performance measure. Finally, a check mark indicates that the algorithm is within one percent of the minimum value reported in the row. A check mark thus means that this algorithm is the best according to the performance measure and for this puzzle size. Empty cells indicate that the algorithm does not solve all instances of this N-Puzzle (except for the 8-Puzzle, in which case runtimes are not reported because they are insignificant).

First, both KWA* and MSC-WA* improve on WA* since only they can solve all instances of the 35-Puzzle. Second, because of its breadth-first search component, KWA* is always the best algorithm according to solution quality. Third, because of its depth-first (greedy) component, MSC-WA* is always the best algorithm according to the search effort (number of generated nodes) and the runtime. Fourth, both KWA* and MSC-WA* beat WA* according to the memory consumption (number of stored nodes). However, no algorithm consistently beats the other in this dimension. Finally, none of the algorithms is able to solve all instances of the 48-Puzzle in our empirical setup.

3.5.2 The MSC-KWA* algorithm

Our preview of results in the N-Puzzle suggests that KWA* and MSC-WA* have distinct advantages over WA*. It is natural to wonder whether these advantages can be cumulated by combining the concepts of diversity and commitment in order to scale up to even larger domains, such as the 48-Puzzle. We now turn to this question.

By forcing the search to commit to a sub-set of the OPEN list (namely the COMMIT list), MSC-WA* uses COMMIT in the role of OPEN (while RESERVE is only used to refill COMMIT when necessary). As a result, MSC-WA* is more focused than WA* and is thus as likely (or even more) to be led by misleading heuristic values into goal-free regions of the search space. One way to alleviate this problem is to add diversity to the MSC-WA* search. We propose to introduce into MSC-WA* the mechanism used by KWA*: instead of expanding the best node in COMMIT at each iteration, we propose to expand a sub-set of the best $K \leq C$ nodes in COMMIT in parallel at each iteration. We call the resulting algorithm *MSC-KWA**.

Figure 15 contains the pseudo-code for MSC-KWA*. It takes three parameters, namely W, C, and K. After initializing the COMMIT and RESERVE lists identically to MSC-WA* (Lines 2 & 3),

```
 1. **procedure** MSC-KWA*($s_{start}$, $heuristic(.)$, $w_g$, $w_h$, $C$,$1 \leq K \leq C$): *solution cost*
 2.   $g(s_{start})$ := 0; $h(s_{start})$ := $heuristic(s_{start})$; $COMMIT$ := $\{s_{start}\}$;
 3.   $RESERVE$ := $\emptyset$; $CLOSED$ := $\emptyset$
 4.   **while** ( $COMMIT \neq \emptyset$ ) **do**
 5.     $SET$ := $\emptyset$
 6.     **while** ( ($COMMIT \neq \emptyset$) **and** ($|SET| < K$) ) **do**
 7.       $state$ := arg $\min_{s \in COMMIT}$ $\{w_g \times g(s) + w_h \times h(s)\}$
 8.       $SET$ := $SET \cup \{state\}$
 9.       $COMMIT$ := $COMMIT \backslash \{state\}$
10.       $CLOSED$ := $CLOSED \cup \{state\}$
11.     **end while**
12.     **for each** $state$ **in** $SET$ **do**
13.       $g$ := $g(state) + 1$
14.       **for each** successor $s$ of $state$ **do**
15.         **if** ( $s = s_{goal}$ ) **then return** $g$
16.         **if** ( $s \notin COMMIT \cup RESERVE \cup CLOSED$) **then**
17.           $g(s)$ := $g(state)+1$; $h(s)$ := $heuristic(s)$; $COMMIT$ := $COMMIT \cup \{s\}$
18.       **end for**
19.     **end for**
20.     **while** ( $|COMMIT| > C$ ) **do**
21.       $state$ := arg $\max_{s \in COMMIT}$ $\{ w_g \times g(s) + w_h \times h(s) \}$
22.       $COMMIT$ := $COMMIT \backslash \{state\}$
23.       $RESERVE$ := $RESERVE \cup \{state\}$
24.     **end while**
25.     **while** ( ($|COMMIT| < C$) **and** ($|RESERVE| > 0$) ) **do**
26.       $state$ := arg $\min_{s \in RESERVE}$ $\{ w_g \times g(s) + w_h \times h(s) \}$
27.       $RESERVE$ := $RESERVE \backslash \{state\}$
28.       $COMMIT$ := $COMMIT \cup \{state\}$
29.     **end while**
30.   **end while**
31. **return** $\infty$
```

Figure 15: The MSC-KWA* algorithm

Figure 16: The 15-Puzzle

the main loop (Lines 4-30) is executed until the COMMIT list is empty, in which case there exists no solution path (Line 31), or until the goal is generated for the first time (Line 15). At each iteration, the best K nodes in COMMIT are selected for expansion (Lines 5-11) and all newly generated nodes are added into COMMIT (Lines 12-19). Finally, either excess nodes in COMMIT are moved into RESERVE (Lines 20-24), or COMMIT is refilled using RESERVE nodes (Lines 25-29), if necessary and possible.

3.6 Empirical evaluation

We have tested WA*, KWA*, MSC-WA* and MSC-KWA* in three standard benchmark domains. First, we introduce these domains. Then, we describe our empirical setup. Finally, we present and discuss our results.

3.6.1 The N-Puzzle domain

The N-Puzzle is a famous sliding-tile puzzle [84] that has often been used as a single-agent search benchmark domain by the heuristic search community (e.g., [30, 131, 96, 98, 99, 108, 42, 105]). When $N = k^2 - 1$ (for $k = 3, 4, 5, \ldots$), the N-Puzzle is a $k \times k$ square board that contains N numbered square tiles and an empty location called the 'blank' (see Figure 16 for a picture of the 15-Puzzle sold by Thinkfun, formerly Binary Arts). The goal of the puzzle is to repeatedly slide a tile adjacent to the blank into its location until a random initial configuration of the puzzle is transformed into a given goal configuration. While it is relatively easy to find a solution by hand, the heuristic search community has typically focused on finding (near-)optimal solutions to the puzzle. An *optimal* solution minimizes the cost of the solution, where a *solution* is a sequence of tile movements that transform the start configuration into the goal and its *cost* is the number of movements in the sequence. The N-Puzzle has been analyzed both empirically [147, 138] and theoretically [137]. The latter reference contains a proof that optimally solving the N-Puzzle is NP-hard.

In general, the size of the state space for the N-Puzzle, that is the number of states reachable from the start state, is equal to $(N + 1)!/2$. For the 8-Puzzle, this size is relatively small so that the whole state space can fit into memory and non-heuristic search techniques (e.g., breadth-first search) can solve it optimally in no time. The 15-Puzzle was first solved optimally using the IDA* algorithm [96] (a linear-space heuristic search algorithm) with the Manhattan distance heuristic function. The Manhattan distance is the sum, over all tiles, of the tile's distance from its current location to its goal location in both the horizontal and vertical directions. Since each tile must move at least that many times to reach its goal position and each sliding action only moves one tile, the Manhattan distance (henceforth referred to as 'MD') is an admissible heuristic for the N-Puzzle. This heuristic was also used to solve larger versions of the puzzle (that is, $k > 4$) with real-time search algorithms [98].

An enhancement to MD called the *linear conflict* heuristic (LC) increases the pruning power of MD and reduces the number of generated states by a factor of 8 in the 15-Puzzle [58]. Due to the additional overhead involved in computing the heuristic value for each node, LC yields a 5-fold runtime speedup over MD [108]. However, only when LC was extended to a more general way of generating admissible heuristics (called *pattern databases* [22, 23]) was the 24-Puzzle solved optimally for the first time [108]. In a later paper [105], it is estimated (based on analytical results developed in [107]) that optimally solving a random instance of the 24-Puzzle with MD would take on average about 50,000 years. In contrast, the average solving time with pattern databases is 2 days (with a rather large variance).

Despite impressive recent improvements in the pattern database technology (e.g., [69, 36, 38]), the performance of admissible search in the 35-Puzzle can only be predicted (e.g., [36]) and it is expected that this puzzle will not be solved optimally in the near future. For this reason, and because it is often preferable to obtain suboptimal solutions fast than to wait too long for an optimal one, suboptimal search is an active area of research. In this context also, the N-Puzzle is often used both for testing domain-dependent search and learning algorithms [42] and as a benchmark for domain-independent suboptimal search [136, 98, 88, 37].

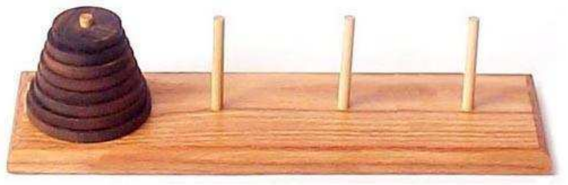


Figure 17: The 4-peg Towers of Hanoi

In short, the N-Puzzle has been and remains a standard tool for measuring progress in both optimal and suboptimal heuristic search algorithms (and heuristic functions). In this work, we use it to measure the performance of WA* and some of its recent variants. In our experiments using MD, WA* can only solve all random instances of the N-puzzle for N up to and including 24. Both KWA* and MSC-WA* scale up to the 35-Puzzle but run out of memory on some random instances of the 48-Puzzle. Finally, the combination of KWA* and MSC-WA* (MSC-KWA*) does scale up to the 48-Puzzle.

3.6.2 The 4-peg Towers of Hanoi domain

The Towers of Hanoi problem is an old and famous problem [118, 32, 45, 160] that has been used and studied by mathematicians, and cognitive and computer scientists alike (e.g., [157, 62, 168, 127, 9, 8, 141, 117, 26, 64, 5, 145, 36]). In addition to being a worthwhile object of mathematical study in its own right, the Towers of Hanoi problem is a useful tool for demonstrating the behavior of recursive algorithms, as well as a benchmark task for the study (both physiological and computational) of cognitive processes.

A problem instance is characterized by the number $p \geq 3$ of pegs (or towers), the number $D \geq 1$ of disks, and the initial location (that is, peg) of each disk. No two disks have the same diameter. One peg is identified as the destination peg. The objective is to find a sequence of moves that gets all of the disks stacked on the destination peg. A move consists in moving a single disk from the top of a stack on any peg onto any other peg. The only constraint is that a larger disk can never lie on top of a smaller disk.

Computational studies of the Towers of Hanoi problem typically focus on the length of the solution sequence. The most commonly studied class of instances has $p = 3$. In this case, the optimal solution cost (that is, the length of a shortest sequence) is exponential in D in the worst case. This class of problems is considered solved since there is a known algorithm that is guaranteed to find optimal solutions. In contrast, the case $p = 4$ (called *Reve's puzzle*) is still open. Existing algorithms are only conjectured to be optimal [9, 8, 145]. Since Reve's puzzle is becoming popular in the heuristic search community [103, 104, 38, 36], we use it as one of our benchmarks (see Figure 17 for a picture of a 4-peg Towers of Hanoi set).

Since each of the D disks can be on any of the four pegs and there is only one way to stack a given set of disks on a peg, the D-disk Reve's puzzle contains 4^D distinct states. A simple memoryless heuristic for this puzzle is the number of misplaced disks, where a disk is *misplaced* when it is not on the destination peg. This heuristic is clearly admissible since each misplaced disk must move at least once. A slightly more informed and still admissible heuristic is the so-called *infinite-peg* heuristic [36]. It is obtained by solving optimally a relaxed version of Reve's puzzle, namely one in which there are infinitely many pegs (or equivalently $D + 4$ pegs). Unfortunately, both of these heuristics are rather poorly informed. Therefore, we use a pattern database as our heuristic function. In [103, 104, 38, 36], a pattern database for a Towers of Hanoi problem with D disks is a lookup table of the minimum costs of solving all possible configurations of $d \leq D$ disks. This heuristic is clearly admissible and takes into account some of the disk interactions that memoryless heuristics ignore. The larger d, the more interactions are accounted for and the more informed the heuristic. In the extreme case, where $d = D$, the heuristic is perfectly informed. The largest instances solved optimally by heuristic search with pattern databases have $D = 18$ [38][3]. We set $D = 22$ and $d = 13$. The first value was chosen in order to demonstrate the advantage of suboptimal searches for scaling up. The second was chosen based on our available memory: Our 13-disk pattern database uses 64Mbytes of memory. The rest of the available memory is used to store search nodes.

[3]In [38], such large instances of the Towers of Hanoi are solved with enhancements to the simple idea of a pattern databases, namely additive pattern databases and compressed pattern databases. Our heuristic does not take advantage of these enhancements. Note that [104] solves optimally one instance with $D = 20$ using breadth-first search and external memory.

Figure 18: The Rubik's Cube

3.6.3 The Rubik's Cube domain

Invented by Erno Rubik in 1974, the Rubik's Cube is a very popular puzzle since more than 100 millions units have been sold worldwide. In addition to its appeal to puzzle-lovers and mathematicians, the Rubik's Cube has been used by many in the artificial intelligence community in the context of path-planning [31], machine learning [97, 41, 133], and especially heuristic search [162, 15, 35, 106, 142, 63, 69].

The six-face cube is made up of twenty smaller, movable cubies, namely eight corner cubies and twelve edge cubies (the six face-center cubies and the one cube-center cubie are not movable). The cubies are grouped into three layers of nine cubies each, both horizontally and vertically. Each move rotates one full layer by 90, 180 or 270 degrees. Starting from a random position and orientation of the twenty movable cubies, the objective is to find a sequence of moves that puts the cubies into a pre-defined goal configuration (e.g., one in which each face of the cube is uniformly colored). See Figure 18 for a picture of the Rubik's Cube®, a registered trademark of Seven Towns Limited.

The search space for the Rubik's Cube is an instance of a permutation group that contains approximately 4.3×10^{19} distinct states. It is believed that any cube configuration can be solved in no more than twenty moves [101]. One crucial difference between the Rubik's Cube and our other two domains is that each move displaces several cubies. In contrast, each move in the N-Puzzle or Reve's puzzle displaces only one tile or disk at a time, respectively. This explains why it is comparatively harder to find good, admissible heuristic functions for the Rubik's Cube. Because simple memoryless heuristics are not informed enough to make search efficient on random instances, [101] applies the idea of pattern databases to this domain and, for the first time, finds optimal solutions to random instances using as heuristic function the maximum of three pattern databases. At the time of publication, the average runtime was on the order of two days (with a large variance). As the size of RAM increases, larger pattern databases can be used, yielding more informed heuristic functions and reduced runtimes. In our experiments, we use Korf's heuristic function, and any additional available RAM is used to store the generated nodes needed by our variants of WA*.

3.6.4 Empirical setup

Problem instances. In the N-Puzzle, we vary k from 3 to 7 (recall that $N = k^2 - 1$). In each case, the goal state is the puzzle configuration in which the blank is in the upper left corner of the board and the tiles are positioned in increasing order of their number (1 through N) from left to right within a row, from the top row down. Start states are randomly selected among the set the states reachable from the goal. Due to timing considerations, we choose a random number of problem instances (each characterized only by the start state) equal to 1000 and 100 for the 8- and 15-Puzzle, respectively.[4] For larger puzzles, we use 50 random instances in each case ($N = 5, 6, 7$).[5]

In the 4-peg Towers of Hanoi problem, we set D equal to 22 and d equal to 13. The goal state has all disks stacked up on an arbitrary peg. Fifty initial states are selected by randomly selecting a peg as the starting position for each disk.

In the Rubik's Cube problem, the goal state is the original configuration of the cube. Fifty initial states are obtained by performing 500-long sequences of random moves starting in the goal state.

Implementation details. Our implementations of the WA* variants described in this chapter share the following features.[6]

- They all use the same data structures for packed states and the same hashing scheme in each domain.
- They all use the same successor-generator function which determines the systematic (that is, non-random) order in which applicable moves are considered during expansion.
- They all prune (that is, do not generate) the successor of a node that is identical to the parent of the node in the search tree. This simply eliminates cycles of length equal to two whose

[4]The 100 random instances used in the 15-Puzzle are the standard set first used in [96].
[5]The 50 random instances used in the 24-Puzzle are the first 50 instances in the 100-long set first used in [37].
[6]Our code for all variants of WA* results from modifications to the WA* code graciously sent to us by Richard Korf.

existence results from the fact that each move in the puzzle is reversible (that is, the state-space is really an undirected graph). In addition, the same disk (layer) cannot be moved twice in a row in the Towers of Hanoi problem (Rubik's Cube) since the state resulting from the sequence of two such moves can be reached in a single move. Finally in the Rubik's Cube, opposite faces are arbitrarily ordered (pairwise) and successive moves of opposite faces are forced to follow this order so that redundant, commutative sequences of two moves are avoided. These constraints applied during node expansion are standard, domain-dependent ways of reducing the branching factors of each domain to make search as efficient as possible.

- They share the same termination condition, since they all stop as soon as the goal state is generated for the first time.

Our experiments in the N-Puzzle (in the Rubik's Cube and Towers of Hanoi, respectively) were performed on a Pentium-IV PC clocked at a 2.2 GHz (on a Pentium-III PC clocked at 1.4 GHz, respectively) and with enough memory to store a few million nodes. This number of stored nodes is the size of our hash table. The timings reported below always exclude the timings of initializations (of the hash table, OPEN lists, etc.) since these are identical across instances and variants of WA*. Runtimes are always for the search phase itself.

Performance measures. All algorithms are evaluated according to the following performance measures:

- The cost (that is, the path length) of the solution found,
- the number of nodes generated, which is a machine-independent measure of the search effort,
- the actual runtime (in seconds) of the search, and
- the number of nodes stored, which is a measure of memory consumption.

There are only two reasons why an algorithm terminates: either it has generated the goal or it has run out of memory. Except where noted, all of our averages are computed for experimental conditions in which all instances are solved within the memory constraints.

3.6.5 Empirical results in the N-Puzzle domain

3.6.5.1 Empirical evaluation of WA in the N-Puzzle*

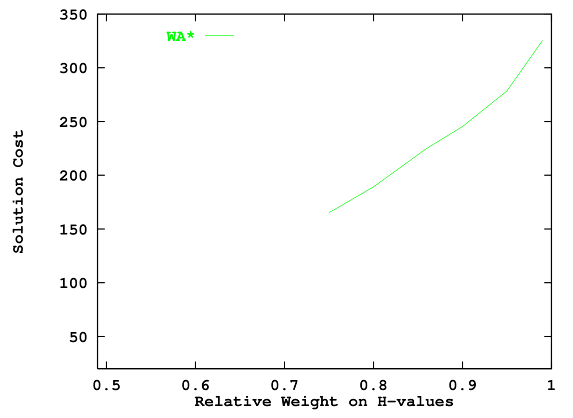
Figure 19 summarizes the performance of WA* on the 8-, 15-, and 24-Puzzle. It contains six sub-figures. Each of the first four sub-figures (a through d) plots one of our selected performance measures against values of W. We use the following values of W; 0.50, 0.56, 0.60, 0.67, 0.75, 0.80, 0,86,0.90, 0.95, 0.99.[7] Only for the 8-Puzzle do we omit the runtime plot since WA* (and all of its variants) are too fast for the runtimes to be significant. The last two sub-figures (e & f) plot the memory usage and runtime as a function of the solution cost, respectively. Note that the plot of the search effort as a function of the solution cost is omitted since, in all cases, it would be very similar to sub-figure f (because, as sub-figures b & d demonstrate, the actual runtime is very strongly correlated with the search effort as measured by the number of generated nodes).

Our results with WA* do not contain any surprises and reproduce the trends exhibited by earlier studies (e.g., [99] and [37] for the 15-Puzzle and 24-Puzzle, respectively). Since the performance of WA* constitutes our baseline for this research, we have decided to include these results, which we now briefly describe. We observe the following general trends:

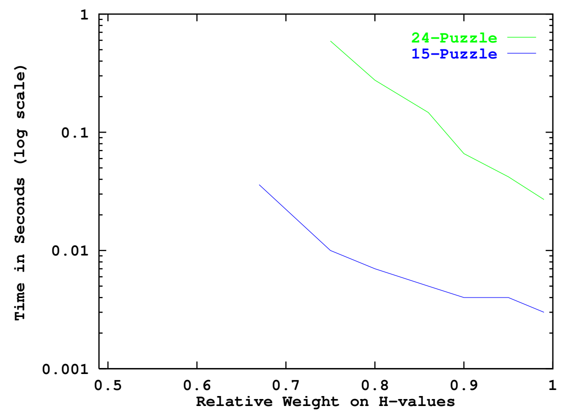
- The solution cost increases with W (sub-figure a). As expected, making WA* more greedy leads to a degradation in solution quality. Furthermore, as observed in earlier studies, the degraded solution quality, measured as a multiplicative factor of the optimal solution cost, is much smaller than the theoretical upper bound of W [24]. For example in the 15-Puzzle, where the known average optimal cost is 53.05 [96], WA* degrades solution quality by a factor smaller than 3 when $\frac{w_h}{w_g} = 99$ (the average solution cost corresponding to the top-rightmost point of the 15-Puzzle curve in Figure 19a is 145.27).

- The search effort decreases as W increases (sub-figures b & d). As expected, making WA* more greedy leads to a significant speedup. For example in the 15-Puzzle, we observe an order of magnitude speedup between the top-leftmost and bottom-rightmost ends of the curve

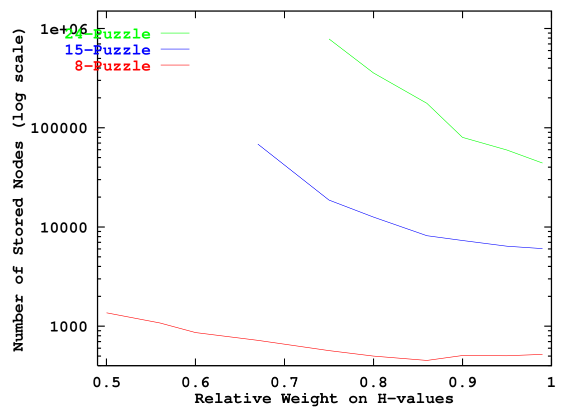
[7]These values of W were chosen 1) to be approximately evenly distributed over the [0.5, 1] interval and 2) to correspond to small, relatively prime integer values of w_g and w_h. These values, listed in Table 9, are a strict sub-set of those used in [99, 37]. The only reason for not using all of their values for W was the considerable cumulated runtimes involved (on the order of days) in running all variants of WA* on 50 to 1000 instances with various values for N, W as well as values for K (see KBFS) and the size of the commitment list (see MSC-WA*).



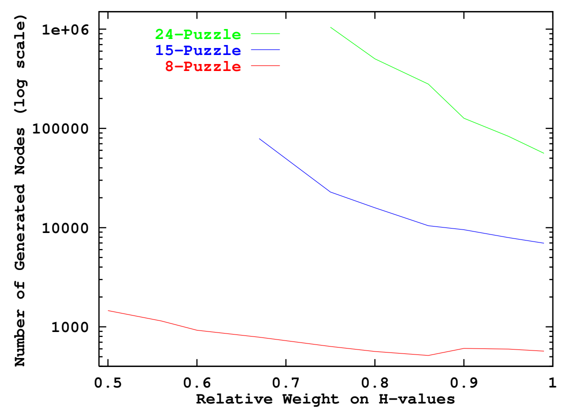
a) Solution cost versus W



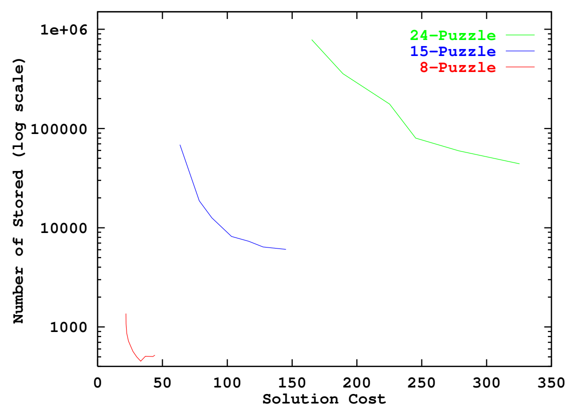
b) Runtime versus W



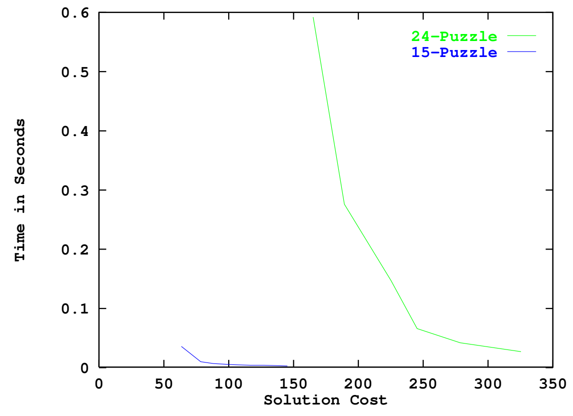
c) Memory usage versus W



d) Search effort versus W



e) Memory usage versus solution cost



f) Runtime versus solution cost

Figure 19: Performance of WA* in the N-Puzzle with varying W

in Figure 19b. As a multiplicative factor, this speedup is thus larger than the degradation in solution quality.[8] This confirms that WA* is an effective way to trade off solution quality for runtime (or, equivalently, search effort). This trade-off is plotted in sub-figure e.

- The memory usage decreases as W increases (sub-figure c). This is crucial from our point of view, since we are interested in scaling up these algorithms to larger problems. Again, this is not surprising: the memory consumption is proportional to the number of stored nodes, which itself is strongly correlated with the number of generated nodes (since they are stored). The resulting trade-off between solution cost and memory usage is plotted in sub-figure e. Furthermore, while WA* does regenerate some nodes (see Lines 11 through 13 in Figure 11), these represent a small percentage of the total number of node generations, as can be seen by comparing sub-figures c and d. Each curve in sub-figure c is similarly shaped, but slightly lower, than that in sub-figure d.

Finally, as N increases, the range of W values for which WA* solves all instances shrinks (and the curves are increasingly truncated on their left side in Figure 19). The 35-Puzzle is the smallest puzzle for which WA* cannot solve all random instances, even with values of W close to 1. Since WA* solves a different sub-set of the test instances for different values of W, as suggested by the last column in Table 9, it is not meaningful to use the other numbers in the table in order to study the trade-offs observed in the smaller puzzles.

3.6.5.2 Empirical evaluation of KWA in the N-Puzzle*

Figures 89 through 94 (see Appendix B.1) show the performance of KWA* in the 8-, 15-, and 24-Puzzle domains. Figures 20 and 21 summarize the performance of KWA* in the 35-Puzzle domain (these figures demonstrate the better scaling of KWA* over WA* and are thus included in the main text below).

Since KWA* adds one more parameter (namely K) to W already used by WA*, we vary each

[8]Furthermore, note that the top-leftmost point in the figure does not correspond to WA*. The curve is truncated to the left because, for smaller values of W including $W = 0.50$, WA* runs out of memory. A* (equivalently, WA* with $W = 0.50$) would in fact be much slower. For a suggestive comparison, the curve is truncated at $W = 0.67$ where the number of nodes generated is equal to 78,870 (see Figure 19d). IDA*, on the other hand, generates 363,028,090 nodes in order to guarantee optimal solutions [99]. Even allowing for the duplicated effort incurred by IDA* over WA*, this is at least 4 orders of magnitude worse than the 6,972 nodes generated by WA* when $W = 0.99$ (see Figure 19d).

Table 9: Performance of WA* in the 35-Puzzle with varying W

W	w_g/w_h	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
0.50	1/1	N/A	N/A	N/A	N/A	0
0.56	4/5	N/A	N/A	N/A	N/A	0
0.60	2/3	N/A	N/A	N/A	N/A	0
0.67	1/2	N/A	N/A	N/A	N/A	0
0.75	1/3	304.39	2,696,266	2,105,719	1.840	36
0.80	1/4	341.82	2,416,819	1,688,718	1.635	66
0.86	1/6	393.00	1,474,970	922,542	0.984	88
0.90	1/9	459.41	1,936,818	1,055,140	1.280	98
0.95	1/19	531.81	1,241,877	670,980	0.749	96
0.99	1/99	628.26	2,281,228	944,441	1.451	94

parameter separately. The values used for W are the same as those used for WA*. In addition, we use the following values for K: $1, 2, 3, \ldots, 10, 20, 30, \ldots, 100, \ldots, 1000$ through 100,000. Each curve in the figures corresponds to either varying W while fixing K, or varying K while fixing W, depending on the figure.

Our results are in agreement with those reported in [37] for the 15- and 24-puzzle (see the detailed discussion of our results below). In addition, our results demonstrate that:

- KWA* does not significantly improve on WA* in the 8-Puzzle (see Figure 89e). For small values of K (say, $K \leq 10$), there is no significant gain, as the decrease in the solution cost is approximately compensated by the node-generation overhead of KWA*. For larger values of K, the added breadth-first component causes an overwhelming node-generation overhead.

- KWA* does improve on WA* in the 35-Puzzle: KWA* can solve all instances of the 35-Puzzle for several values for W and K (see Figures 20 & 21), while there is no value of W for which WA* can solve all instances of the 35-Puzzle (see Table 9).[9] However, the set of parameter values for which KWA* solves all instances is relatively small and no trend can be seen in the figures. In fact, in smaller N-Puzzle problems, there is no decrease in memory

[9] In [37], the authors report that their implementation of KWA* still cannot solve all instances of the 35-Puzzle even though their memory could store up to nine million nodes (compared to our six million nodes). The reason for this discrepancy is that their management of the OPEN and CLOSED lists was less space-efficient than ours since it allowed for the same state to be stored in multiple nodes (personal communication with Ariel Felner, November 2003).

consumption as K increases. For example, the curves for the 24-Puzzle in Figure 94c (Appendix B) are essentially horizontal (or have a positive slope for large values of K). Only noisy data points allow KWA* to reduce memory consumption. In conclusion, while increasing K does not reduce the memory consumption of KWA*, KWA* does reduce the memory consumption of WA* *for a given solution cost* but only with appropriate values for *both* K *and* W.

In the 8-, 15-, and 24-Puzzle domains, we observe the following trends:

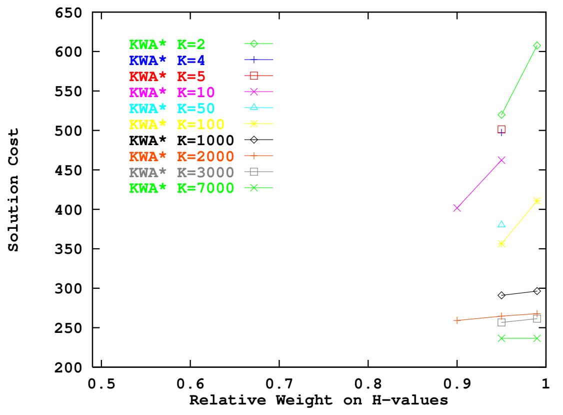
- The solution cost increases with W and decreases with K (sub-figure a). As expected, making KWA* more greedy, either by increasing W or decreasing K, or both, leads to a degradation in solution quality.

- The search effort remains more or less constant for small values of K (see sub-figures 90c, 92d, and 94d). However, when K is large enough, the search effort increases (and the value of W matters less and less) as K increases. Furthermore, the demarcation point between the flat and inclined portions of the curve shifts to the right (that is, the corresponding K value increases) as N increases. It is approximately equal to 10, 100, and 1,000 for the 8-, 15-, and 24-Puzzle, respectively.

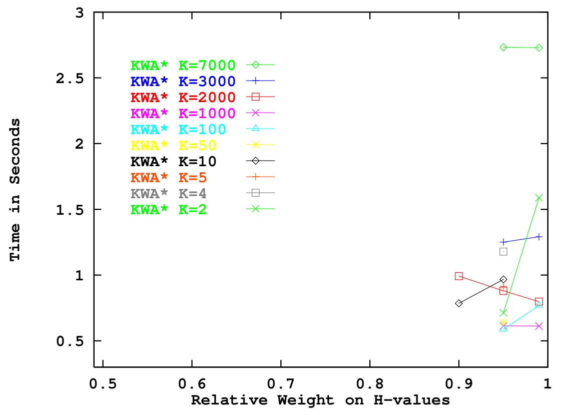
- The search effort decreases as W increases, for all values of K except large ones (see sub-figures 89c, 91d, and 93d).

- The memory usage follows the same trend as the search effort, since like WA*, KWA* stores all the nodes it generates. Furthermore, the number of stored nodes is only slightly smaller than the number of generated nodes, which indicates that relatively few nodes are re-generated (compare sub-figures b and c for the 8-Puzzle, and sub-figures c and d for the 15- and 24-Puzzle).

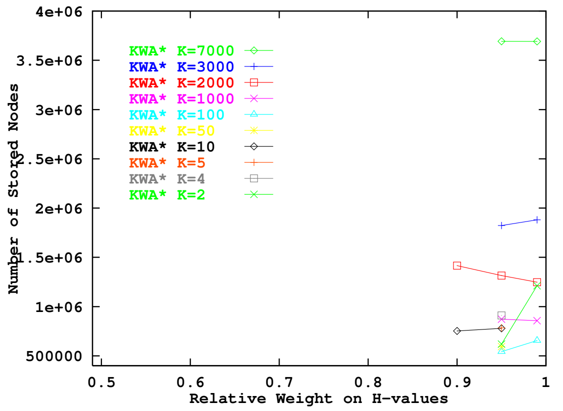
- The runtime follows the same trend as the search effort (see sub-figure b in the 8- and 15-Puzzle). Since there is no additional overhead per node generation for KWA*, the runtime is essentially proportional to the number of node generations, like in WA*.



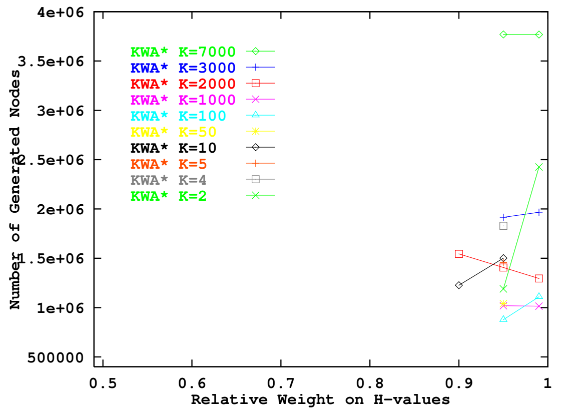
a) Solution cost versus W



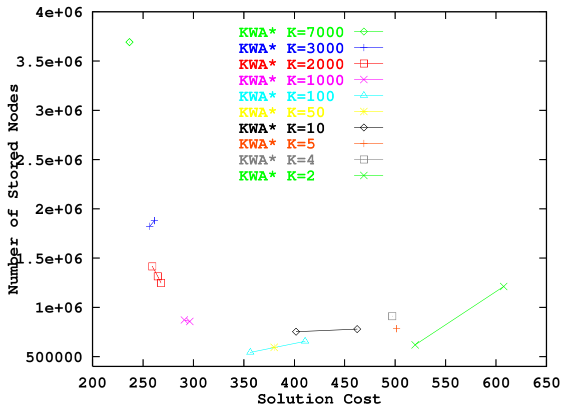
b) Runtime versus W



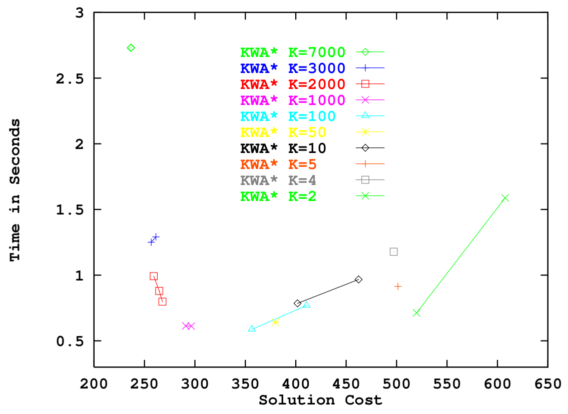
c) Memory usage versus W



d) Search effort versus W

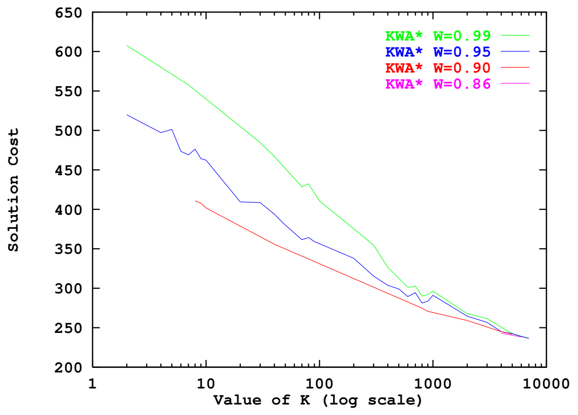


e) Memory usage versus solution cost

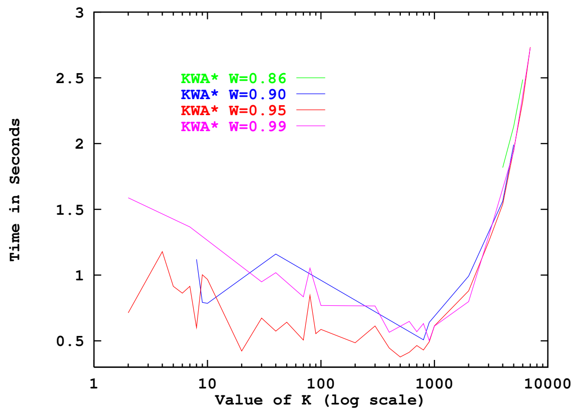


f) Runtime versus solution cost

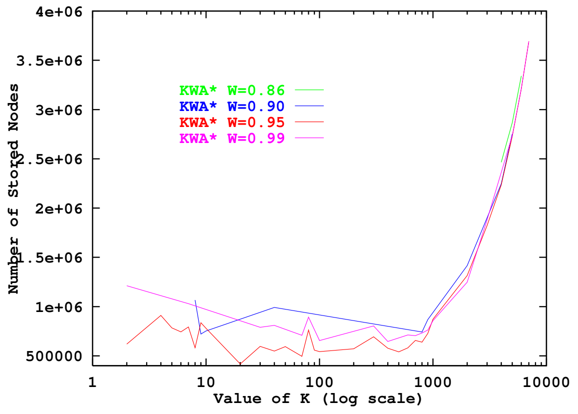
Figure 20: Performance of KWA* in the 35-Puzzle with varying W



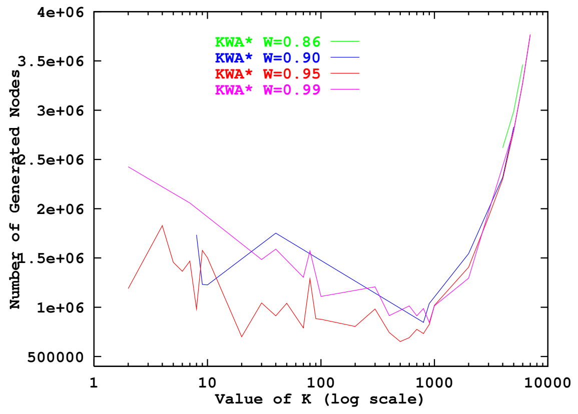
a) Solution cost versus K



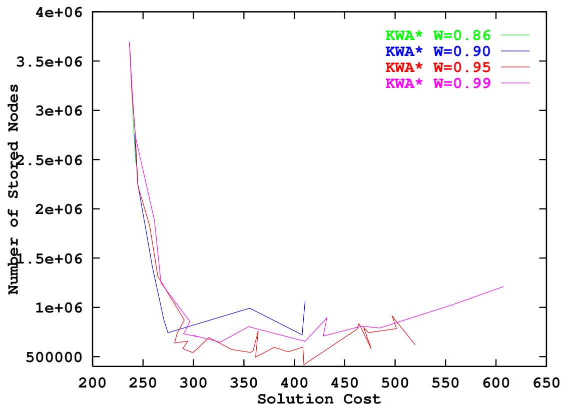
b) Runtime versus K



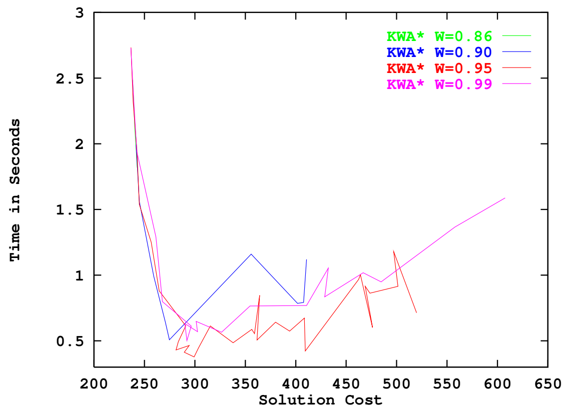
c) Memory usage versus K



d) Search effort versus K



e) Memory usage versus solution cost



f) Runtime versus solution cost

Figure 21: Performance of KWA* in the 35-Puzzle with varying K

The main motivation for the development of KWA* was to make WA* less greedy (by increasing K, and thus the role of the breadth-first component) so that it avoids useless node generations in areas of the search space where the heuristic function is misleading. In all domains larger than the 8-Puzzle, the advantage of KWA* over WA* is obvious in terms of both search effort and memory usage for a given solution cost. For example in Figure 91f, KWA* with K=50 is about twice as fast as WA* to generate a solution with a cost of 80. In the 24-Puzzle (see Figure 93f), KWA* with K=100 is about an order of magnitude faster than WA* to generate a solution with a cost of 165.

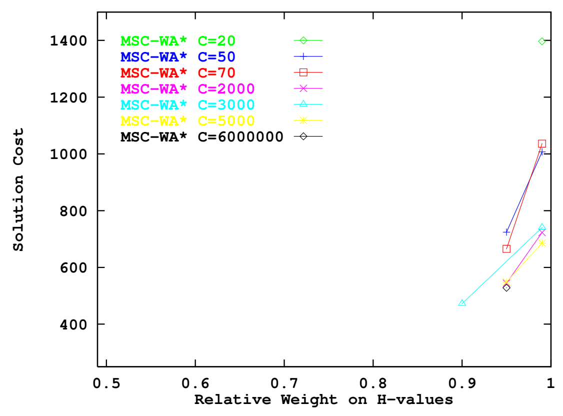
With respect to our goal of scaling up WA* to larger domains, the improvement of KWA* over WA* in terms of memory usage is of the same magnitude. This is because its runtime is proportional to the number of node generations and the latter is strongly correlated with the number of stored nodes. Such a reduction in the memory usage explains why KWA* can solve all instances of the 35-Puzzle while WA* cannot. However, as we have already noted, the improved scaling of KWA* is probably due to noisy data points since simply increasing K while keeping W constant does not lead to a steady reduction in memory requirements.

Finally, Table 10 contains the performance data for KWA* in the 48-Puzzle for experimental conditions in which at least two-third of the instances are solved. KWA* never solves more than 76 percent of them. We report these data as a baseline for future comparison with other variants of WA*.

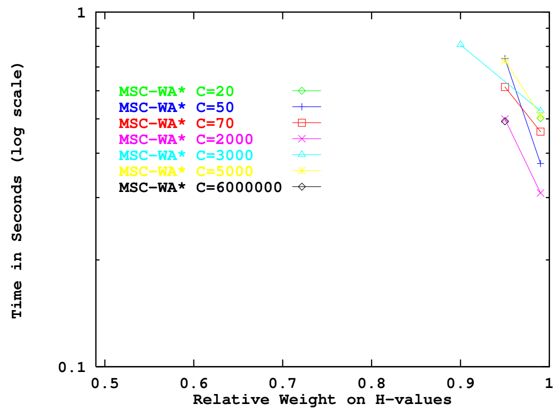
3.6.5.3 Empirical evaluation of MSC-WA in the N-Puzzle*

The empirical study in [88] evaluates MSC-WA* in the N-Puzzle and gridworld domains. While their results do not show any improvement of MSC-WA* over WA* in gridworlds, they do demonstrate a scale-up in the N-puzzle. With this study, we confirm the latter trend. In addition, we improve on their study by varying both W and C over their range of values. In contrast, the empirical evaluation in [88] keeps W constant (and equal to 1) and varies C from 1 to 6 only. Here, we report results for values of N equal to 8, 15, 24, 35, and 48. Later, we experiment with MSC-WA* in two additional domains.

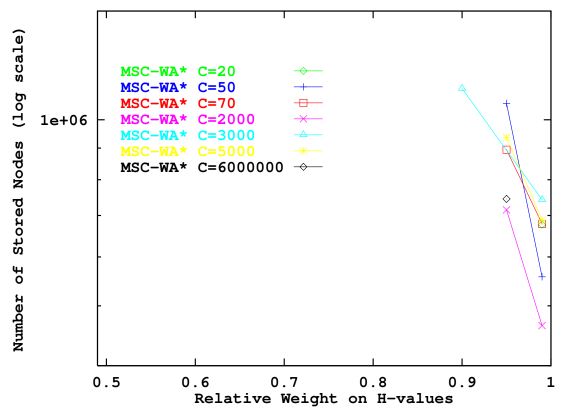
The main result of this study is that MSC-WA* scales up to the 35-Puzzle, while WA* cannot



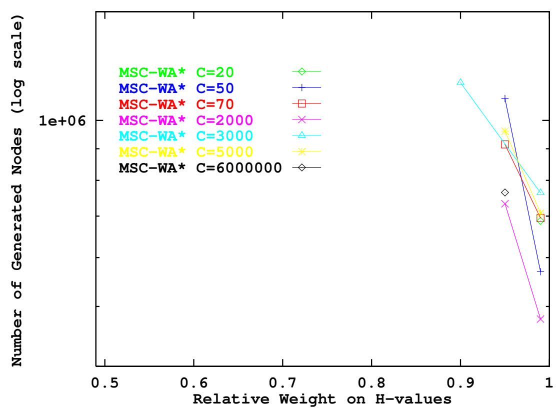
a) Solution cost versus W



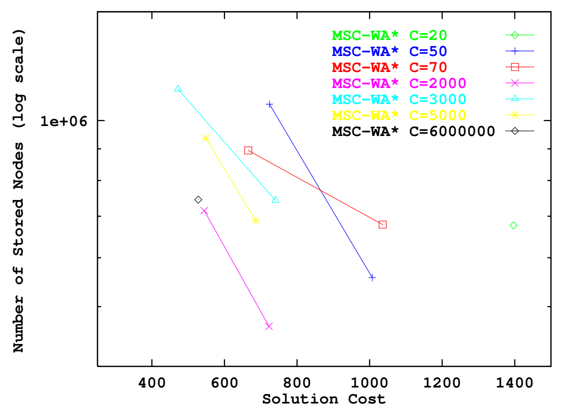
b) Runtime versus W



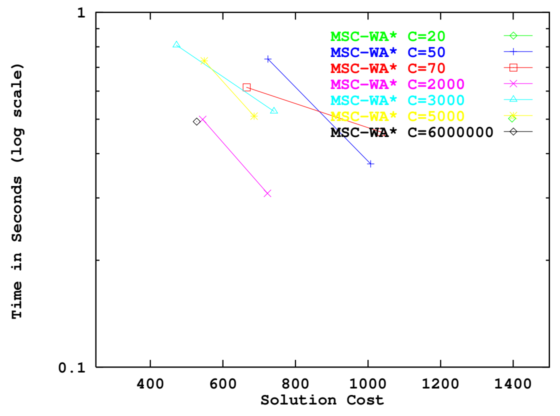
c) Memory usage versus W



d) Search effort versus W

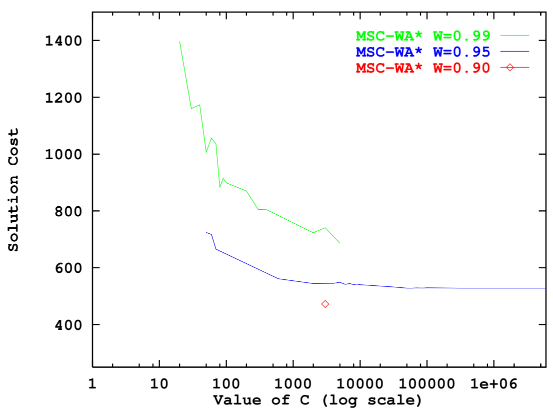


e) Memory usage versus solution cost

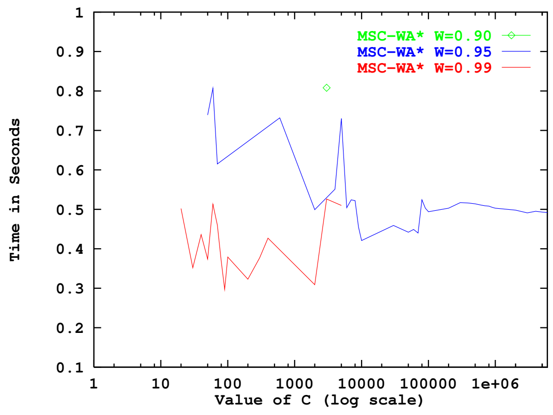


f) Runtime versus solution cost

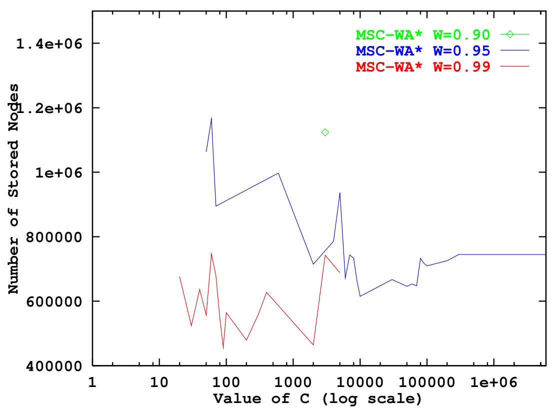
Figure 22: Performance of MSC-WA* in the 35-Puzzle with varying W



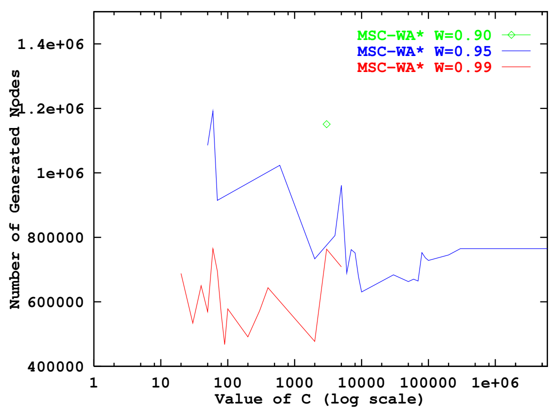
a) Solution cost versus C



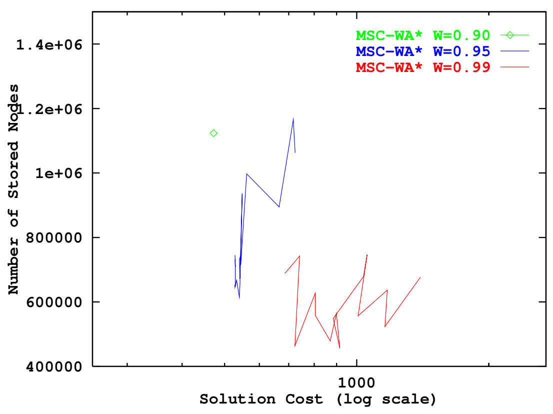
b) Runtime versus C



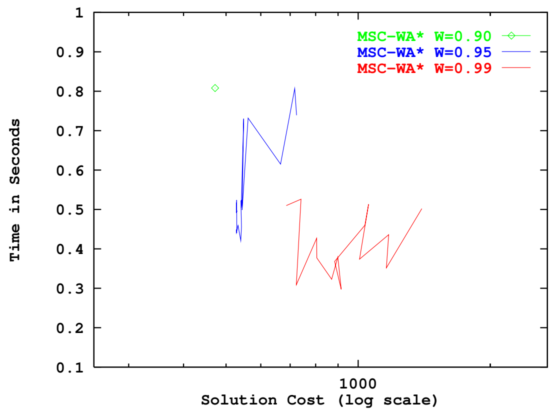
c) Memory usage versus C



d) Search effort versus C



e) Memory usage versus solution cost



f) Runtime versus solution cost

Figure 23: Performance of MSC-WA* in the 35-Puzzle with varying C

Table 10: Performance of KWA* in the 48-Puzzle when solving at least two thirds of the instances

K	W	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
2	0.95	860.97	2,931,245	1,712,080	2.120	68
4	0.90	731.35	3,165,464	2,101,378	2.314	68
4	0.95	834.23	3,083,273	1,778,545	2.209	70
6	0.90	691.63	3,461,305	2,208,772	2.582	70
9	0.95	773.18	2,578,358	1,462,580	1.861	76
50	0.95	655.47	3,354,891	2,007,508	2.449	76
200	0.95	563.03	2,857,603	1,775,200	2.074	70
300	0.90	512.83	2,781,818	1,935,235	2.033	70
500	0.90	493.59	2,643,401	2,054,547	1.936	68
600	0.95	518.65	2,552,990	1,829,689	1.857	74
700	0.90	471.12	2,223,390	1,835,408	1.621	68
700	0.95	507.91	3,076,285	2,195,437	2.401	68
800	0.95	515.89	3,266,025	2,304,743	2.574	72
3,000	0.95	440.54	3,430,356	3,262,424	2.797	70
3,000	0.99	444.09	3,273,160	3,182,089	2.647	70
4,000	0.90	425.24	4,274,189	4,149,173	3.799	68
4,000	0.99	434.06	4,203,531	4,112,240	3.762	70

solve all random instances given the same amount of memory (see Table 9). This is because MSC-WA* focuses the search more sharply, leading to a degradation in solution quality while reducing the search effort and the memory consumption (see Figures 22 and 23 below for the 35-Puzzle, and Figures 95 through 100 in Appendix B.2 for the smaller N-Puzzles).

We observe the following trends:

- The solution cost increases with W. It also increases as C decreases, that is as the depth-first component becomes stronger. This second trend is less clear for small values of C in large puzzles.

- The search effort (both in terms of node generations and runtime) and the memory consumption both decrease as W increases. In the 8-Puzzle, both increase with C. In the 15-Puzzle, both decrease and then increase as C increases, with a minimum around $C = 10$. However in the 24- and 35-Puzzle, both decrease and then stabilize as C increases.

- For all values of N, there is a trade-off between solution quality on one hand and search effort and memory consumption on the other, when increasing W. In the 8- and 15-puzzle, a similar

Table 11: Performance of MSC-WA* in the 48-Puzzle when solving at least two thirds of the instances

C	W	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
60	0.99	1,861.61	2,239,906	2,190,618	1.735	72
90	0.99	1,763.03	2,412,524	2,353,818	1.983	72
200	0.99	1,540.26	1,998,260	1,947,644	1.526	76
400	0.99	1,292.91	2,304,667	2,243,993	1.781	68
900	0.99	1,273.15	2,581,034	2,507,904	2.049	78
30,000	0.95	878.29	2,256,880	2,201,531	1.747	68

trade-off exists when decreasing C (the trend is clearer for small values of W and large values of C). In the 24- and 35-puzzle, the trend is reversed since increasing C reduces the solution cost, the search effort, and the memory consumption (except for small values of W and large values of C).

The results in large domains indicate that a large W is crucial to reducing the memory consumption. After fixing W close to 1, C should probably be increased as much as possible until memory runs out, since increasing C reduces both the solution cost and the memory consumption, up to a certain point. Indeed, this is confirmed in the 48-Puzzle (see Table 11). The highest success rates are achieved for high values of W. Additionally, larger values of C seem to decrease the solution cost while maintaining the memory consumption approximately constant, up to a point which unfortunately is not sufficient for MSC-WA* to solve all random instances of the 48-Puzzle.

3.6.5.4 Empirical evaluation of MSC-KWA in the N-Puzzle*

MSC-KWA* has three parameters. These, when crossed with the different values for N, generate a large number of experimental conditions. So for now, we eliminate one parameter by making C equal to K. This section reports results for this special case. The general case (where K varies between 1 and C) is discussed in Section 3.8 .

Appendix B.3 contains all of the results of this study. We summarize them here by including a representative sub-set of the figures pertaining to the 35-Puzzle (see Figures 24 through 28). We observe the following trends:

- MSC-KWA* shares with WA*, KWA* and MSC-WA* the property that its search effort (that is, the number of generated nodes), runtime (in seconds), and memory consumption (that is, the number of stored nodes) are so strongly correlated that their performance curves are hardly distinguishable. Again, we focus on the memory consumption.

- In contrast to KWA* and MSC-WA*, the value of W has a much smaller effect on performance than the other parameters. In fact, in contrast to WA*, KWA* and MSC-WA*, the value of W has hardly any effect on performance (see Figures 24 & 25), except for small values of $K = C$. This is advantageous since it reduces the effort needed to determine an optimal parameter setting by effectively eliminating one of the parameters (we recommend to use a value of W close to 1). W's lack of effect can be explained as follows. Assume that C is large enough that COMMIT never needs refilling (that is, Lines 25-29 in Figure 15 are never executed). At every iteration, the $K = C$ nodes in COMMIT are removed from COMMIT (Line 9), which is now empty. Since only newly generated nodes are added into COMMIT (Line 17) and their g-value is one more than that of their parent (Line 17), it follows that, at each iteration, all nodes in COMMIT have the same g-value (simple induction based on the fact that the initial COMMIT list has only one node, namely the start node). Therefore, changing the constant weight on the h-values does not alter their ordering, nor the behavior of MSC-KWA*. When W does have an effect, it is because C is small enough that nodes in RESERVE (which do have different g-values since they were added during different iterations) are used to refill COMMIT. The latter now contains nodes with different g-values and W has an effect on their ordering.

- The cost of solutions found by MSC-KWA* decreases when K increases, except for small values of K (see Figure 26). This is a consequence of the strong breadth-first component of the algorithm.

- For the same reason, the memory consumption increases with K, when K is large enough (see Figure 27). But for small enough values, increasing K does reduce the memory consumption (as well as the search effort and runtime). This is diversity in action: because it does not focus exclusively on the best node in COMMIT, MSC-KWA* avoids getting trapped into exploring (and storing) irrelevant parts of the search space. When K gets too large, the benefit of diversity is traded off with the additional overhead of parallel expansions. This overhead

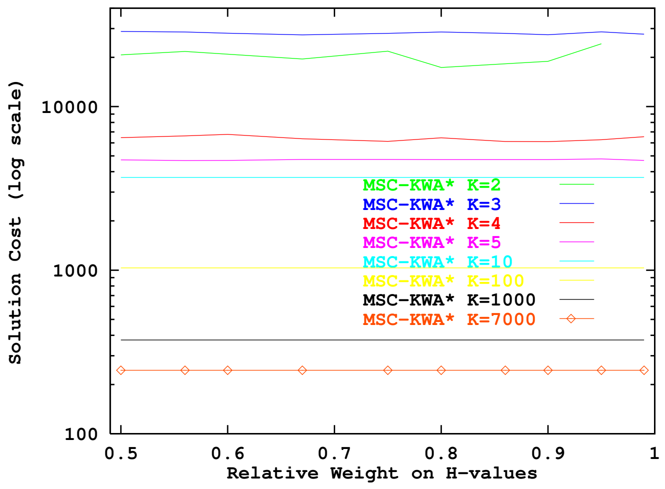


Figure 24: Solution cost versus W for MSC-KWA* ($K = C$) in the 35-Puzzle

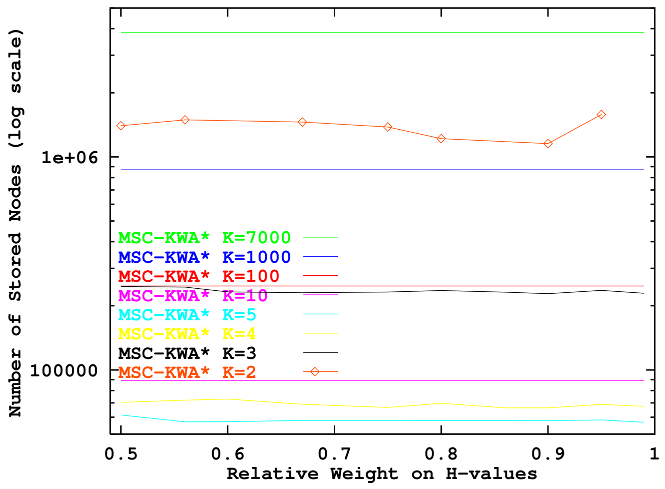


Figure 25: Memory usage versus W for MSC-KWA* ($K = C$) in the 35-Puzzle

increases with K and it cancels out the benefit of diversity for very large values of K.

- Figure 28 illustrates the trade-off exhibited by MSC-KWA* between solution cost and memory requirements: decreasing K (up to a point) sacrifices solution quality for a reduced memory consumption. The reason why each curve in the figure (except for small values of K) reduces to a point is that the curve represents performance variations caused by varying W; and we explained earlier why varying W has little effect.

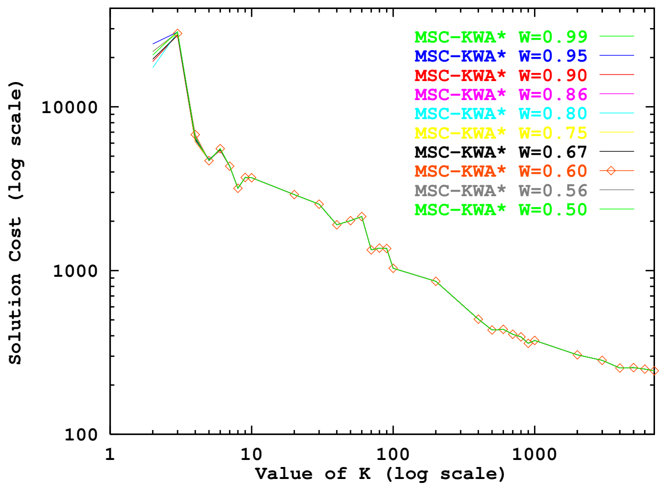


Figure 26: Solution cost versus K for MSC-KWA* ($K = C$) in the 35-Puzzle

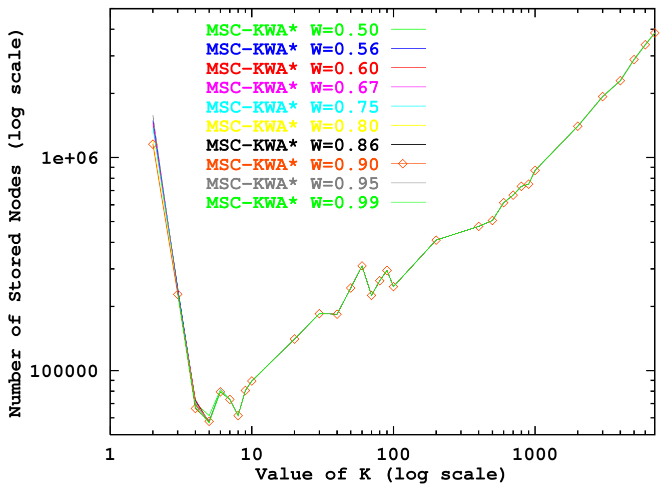


Figure 27: Memory usage versus K for MSC-KWA* ($K = C$) in the 35-Puzzle

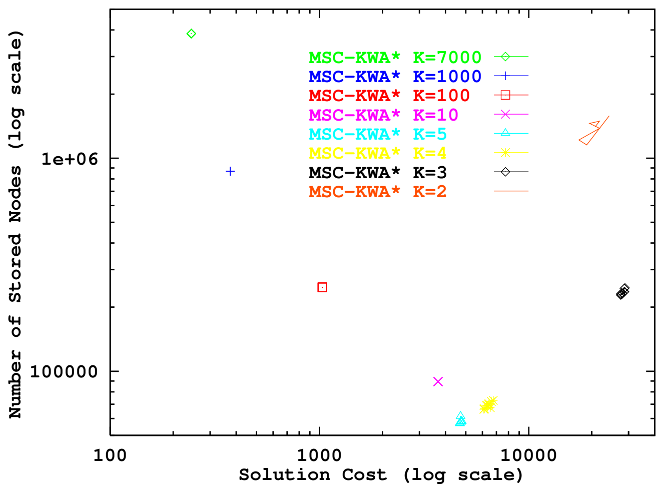
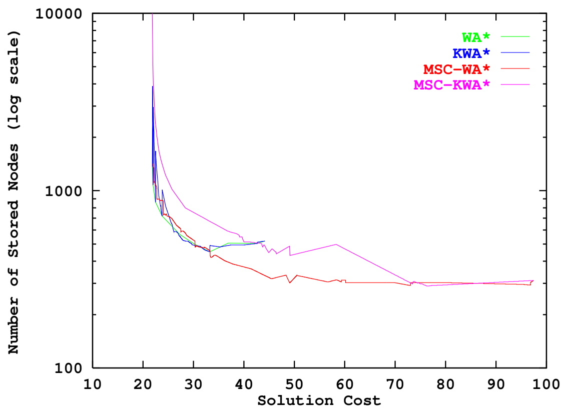
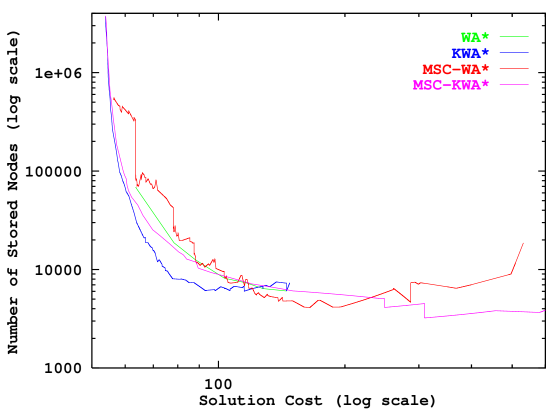


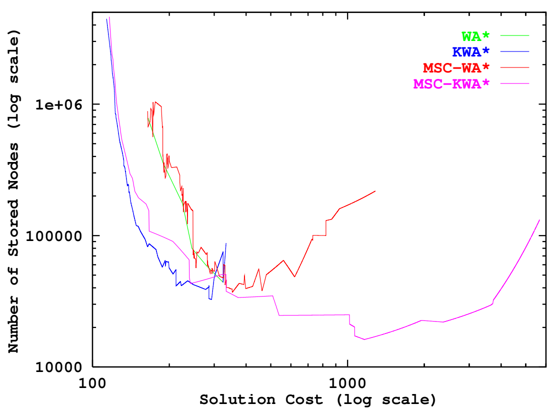
Figure 28: Memory usage versus solution cost for MSC-KWA* ($K = C$) in the 35-Puzzle with varying W



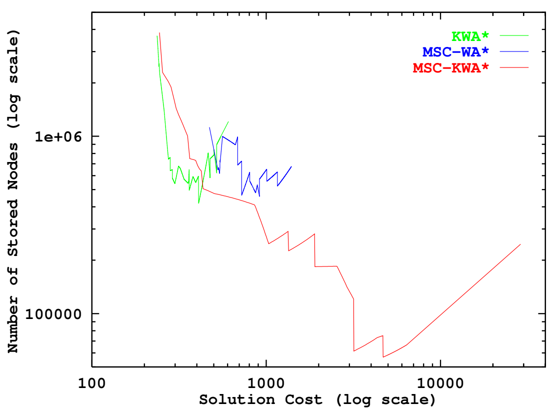
a) 8-Puzzle



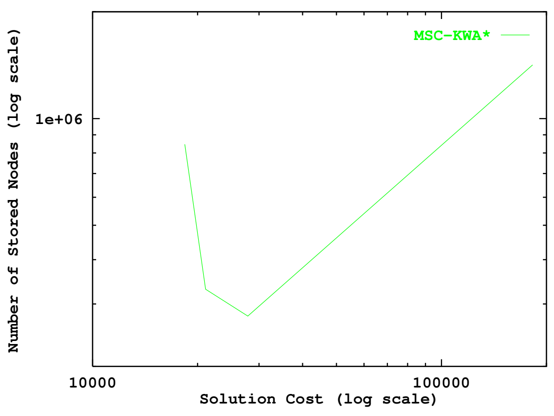
b) 15-Puzzle



c) 24-Puzzle



d) 35-Puzzle



c) 48-Puzzle

Figure 29: Performance comparison: WA*, KWA*, MSC-WA*, and MSC-KWA* in the N-Puzzle

3.6.5.5 *Empirical comparison of WA*, KWA*, MSC-WA*, and MSC-KWA* in the N-Puzzle*

In order to obtain a more synthetic view of the behavior of MSC-KWA*, we perform the same curve-minimization process described in Section 3.5.1 to represent the performance of MSC-KWA* with a single curve. We plot this curve against the corresponding ones for WA*, KWA*, and MSC-WA* for different values of N. Figure 29 shows that:

- MSC-KWA* approximates the behavior of KWA* for large values of K, yielding solutions of high quality.
- MSC-KWA* approximates, and even significantly improves on, the behavior of MSC-WA* for smaller values of K, yielding a low memory consumption.
- The improvement of MSC-KWA* over both KWA* and MSC-WA* in terms of memory consumption grows larger when the problem gets larger. For $N = 8$, there is no significant improvement over MSC-WA*. For $N = 15$ through 35, the reduction in the memory consumption both increases in amplitude and starts with larger and larger values of K (that is, more and more toward the left of the figure). Finally, MSC-KWA* is the only algorithm among the four tested that can solve all instances of the 48-Puzzle.

To conclude, Table 12 builds on Table 8 by adding the performance of MSC-KWA*. We observe that MSC-KWA* dominates all other algorithms in terms of memory consumption, search effort, and runtime (when using each algorithm's ideal settings for each performance measure) for all tested sizes of the N-Puzzle. Furthermore, MSC-KWA* dominates WA* and MSC-WA* in terms of solution quality for all sizes of the N-Puzzle. Finally, MSC-KWA* is always within about three percent of the best algorithm (namely KWA*) in terms of solution quality.

3.6.6 Empirical results in the 4-peg Towers of Hanoi domain

We have tested the variants of WA* in the 4-peg Towers of Hanoi domain. Table 13 shows the parameter settings (and resulting performance data) for which each algorithm solves the largest percentage of instances (and with the smallest average solution cost) with up to one million nodes in memory. None of the WA*, KWA*, and MSC-WA* algorithms can solve all of our fifty random instances. Nevertheless, both MSC-WA* and KWA* scale up slightly better than WA*. Finally,

Table 12: Comparison of WA*, KWA*, MSC-WA*, and MSC-KWA* in the N-Puzzle

N	Perf.	WA*			KWA*				MSC-WA*				MSC-KWA*			
	Measure	Value	W	Best	Value	W	K	Best	Value	W	C	Best	Value	W	K	Best
8	Min Cost	21.85	0.50	✓	21.85	0.50	2	✓	22.01	0.50	800	✓	21.85	0.99	600	✓
	Min Sto.	452	0.86		464	0.86	2		292	0.80	2	✓	290	0.95	2	✓
	Min Gen.	514	0.86		519	0.99	4		296	0.80	2	✓	294	0.95	2	✓
	Min Time															
15	Min Cost	63.51	0.67		53.85	0.67	50K	✓	56.29	0.60	80K		53.89	0.99	50K	✓
	Min Sto.	6,050	0.99		6,028	0.99	8		4,113	0.95	20		3,223	0.99	5	✓
	Min Gen.	6,972	0.99		6,704	0.99	8		4,191	0.95	20		3,259	0.99	5	✓
	Min Time	0.003	0.99		0.003	0.99	5		0.002	0.99	6		0.001	0.95	3	✓
24	Min Cost	165.16	0.75		113.56	0.99	20K	✓	164.56	0.75	90K		116.32	0.99	20K	
	Min Sto.	44,097	0.99		32,567	0.99	5		36,907	0.99	300		16,178	0.99	6	✓
	Min Gen.	56,070	0.99		43,578	0.99	4		37,832	0.99	300		16,331	0.99	6	✓
	Min Time	0.027	0.99		0.021	0.99	4		0.021	0.99	50K		0.007	0.99	6	✓
35	Min Cost				236.50	0.99	7K	✓	472.10	0.90	3K		244.14	0.99	7K	
	Min Sto.				417,675	0.95	20		456,777	0.99	90		56,807	0.99	5	✓
	Min Gen.				652,100	0.95	500		467,586	0.99	90		57,291	0.99	5	✓
	Min Time				0.377	0.95	500		0.297	0.99	90		0.033	0.99	5	✓
48	Min Cost												18,379.32	0.60	5	✓
	Min Sto.												275,293	0.80	4	✓
	Min Gen.												277,282	0.80	4	✓
	Min Time												0.181	0.80	4	✓

Table 13: Best performance of all algorithms in the Towers of Hanoi domain (memory = 1 million nodes)

Algorithm	C	K	W	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
WA*	∞	1	0.99	629.88	2,323,424	762,638	1.582	84
KWA*	N/A	3	0.99	629.68	2,278,109	749,782	1.551	88
MSC-WA*	600	N/A	0.99	672.66	2,059,218	750,125	1.587	94
MSC-KWA*	100	100	1.00	3,762.33	1,431,303	675,450	0.976	72
MSC-KWA*	40,000	177	1.00	2,261.76	1,664,383	750,326	1.139	100

Table 14: Performance of MSC-KWA* in the Towers of Hanoi domain when solving all instances (memory = 1 million nodes)

C	K	W	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
20,000	93	1.00	3,877.66	1,516,603	727,983	1.058	100
20,000	117	1.00	2,949.94	1,540,506	709,424	1.095	100
20,000	120	1.00	2,922.56	1,664,459	755,601	1.193	100
30,000	75	1.00	4,261.12	1,529,075	723,398	1.120	100
30,000	96	1.00	3,492.08	1,450,032	688,717	0.991	100
30,000	111	1.00	3,387.04	1,671,093	766,913	1.167	100
30,000	129	1.00	2,576.00	1,506,882	693,956	1.056	100
40,000	132	1.00	2,866.26	1,621,793	748,193	1.117	100
40,000	177	1.00	2,261.76	1,664,383	750,326	1.139	100
50,000	111	1.00	3,406.30	1,639,262	763,766	1.254	100
50,000	135	1.00	2,742.02	1,619,731	744,240	1.212	100

MSC-KWA* is the only one of our contenders that can solve all of our random instances within the same memory bound. The next-to-last line in the table reports the best performance of MSC-KWA* when $K = C$.

Table 14 shows parameter settings (and the resulting performance data) for which MSC-KWA* is complete over our full set of random instances. We observe the following trends. First, the relative weight on the h-values is always maximum (namely equal to 1). Second, the level of commitment is relatively low (i.e., C is relatively large, on the order of a few tens of thousands) but still higher than that of WA* (for which $C = \infty$). Third, the level of diversity is relatively small (in particular, $K < C$) but still two orders of magnitude larger than for WA* (namely, K is on the order of 100 versus 1 for WA*). In conclusion, in our empirical setup for this domain, MSC-KWA* scales up to larger (namely, 22-disk) problems than any of the other tested algorithms. Therefore, the right mix of commitment and diversity boosts the performance of WA* significantly in this domain.

3.6.7 Empirical results in the Rubik's Cube domain

Table 15 shows the parameter settings (and resulting performance data) for which each algorithm solves the largest percentage of instances in the Rubik's Cube domain with up to two million nodes in memory. No algorithm solves all instances. However, each variant of WA* improves upon it. MSC-KWA* solves more instances than MSC-WA*, which in turn solves more instances than KWA*. Note that the best scaling behavior of MSC-KWA* is obtained for different values of K

Table 15: Best performance of all algorithms in the Rubik's Cube domain (memory = 2 million nodes)

Algorithm	C	K	W	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
WA*	∞	1	1.00	4,649.89	1,227,880	898,748	2.561	38
KWA*	N/A	70	1.00	570.67	951,451	893,778	2.002	66
MSC-WA*	50	N/A	1.00	2,388.95	1,291,810	989,394	2.618	80
MSC-KWA*	90	90	1.00	591.03	705,449	703,566	1.529	64
MSC-KWA*	30	3	0.99	2,593.11	740,982	738,952	1.591	90

and C, namely $C = 30$ and $K = 3$ (see the last line in the table). The next-to-last line in the table reports the best performance of MSC-KWA* when $K = C$.

This domain differs from the previous two in that its search tree is quite wide and shallow since its branching factor is approximately equal to 13 and its maximum depth is estimated at 20 [101].[10] As a result the range of possible values of the heuristic function is quite restricted.[11] The number of ties is thus large. This means that WA* grows a wide search front with rather poor discrimination among its frontier nodes. MSC-WA* significantly improves on WA* by restricting the width of this front. On the other hand, one benefit of increased diversity (that is, of a stronger breadth-first search component) is that KWA* and MSC-KWA* (with $K = C$) exhibit lower solution costs than the other two algorithms. But when comparing the average solution costs of KWA* and MSC-KWA*, the increase in solution cost is probably the result of introducing commitment (that is, a stronger depth-first search component) into the search.

3.7 Related work

In this section, we describe two related approaches. First, MSC-RTA* applies the idea of varying levels of commitment to the RTA* algorithm (as opposed to WA*). Second, beam search is a close relative of MSC-KWA*.

[10]In contrast, both the N-Puzzle and 4-peg Towers of Hanoi domain search trees have branching factors smaller than 5, and their average solution length (for large enough values of N and D) are several times larger than 20

[11]Our heuristic function has values between 0 and 11 and averages around 9 [101].

```
 1. procedure RTA*($s_{start}$, $heuristic(.)$): solution cost
 2.   $g(s_{start})$ := 0; $h(s_{start})$ := $heuristic(s_{start})$
 3.   $state$ := $s_{start}$
 4.   $bestH$ := $\infty$; $secondBestH$ := $\infty$; $bestS$ := $\perp$
 5.   for each successor $s$ of $state$ do
 6.       if ( $s = s_{goal}$ ) then return $g(state) + 1$
 7.       if ( $s$ is newly generated ) then
 8.           $g(s)$ := $g(state) + 1$; $h(s)$ := $heuristic(s)$
 9.       if ( $h(s) \leq bestH$ ) then $secondBestH$ := $bestH$; $bestH$ := $h(s)$; $bestS$ := $s$
10.       else if ( $h(s) < secondBestH$ ) then $secondBestH$ := $h(s)$
11.   end for
12.   $h(state)$ := max( $h(state)$ , $1 + secondBestH$ )
13.   if ( $bestS = \perp$ ) then return $\infty$
14.   else $state$ := $bestS$
15.   go to Line 4
```

Figure 30: The RTA* algorithm

3.7.1 Multi-state commitment applied to RTA* search

In [88], the idea of multi-state commitment is applied, not only to WA*, but also to RTA* [98]. In this section, we describe the RTA* and resulting MSC-RTA* algorithms.

3.7.1.1 The RTA algorithm*

The RTA* algorithm is a variant of the LRTA* algorithm [98]. The only difference between LRTA* and RTA* is the way they update the h-value of the current state. While LRTA* sets it to the estimated cost of a shortest path from the state to a goal, RTA* sets it to the second best estimate. The motivation for RTA* is that, since the algorithm always moves to a successor that minimizes the estimated cost to a goal, re-visiting the state is only worth it if one of its other successor becomes the most promising (since the best one was just visited). There are two important consequences of this change in the value-update rule. First, RTA* typically finds shorter solutions that LRTA* [98]. Second, even if the initial h-values are admissible, the updated ones may not be. Therefore, RTA* (unlike LRTA*) is not guaranteed to converge to an optimal path after repeated trials. In this work (like in [88]), we use RTA* as an approximation algorithm to the shortest-path problem, not as a real-time search algorithm (that is, one that interleaves planning and plan execution). We are only interested in the first trial of RTA*, namely until it reaches a goal state for the first time.

Figure 30 contains the pseudo-code for RTA*. As always in this chapter, 1) we assume that there is a single goal, 2) we assume that all actions have unit cost, and 3) we omit the description of the mechanism (that is, the management of back-pointers) used to reconstruct the solution found.

3.7.1.2 The MSC-RTA algorithm*

[88] points out that WA* and RTA* are the two extremes of a spectrum characterized by the notion of commitment. WA* does not make any commitment since any open node can be expanded next. In contrast, RTA* is maximally committed since the next node to be expanded is always selected among the successors of a single node, namely the latest node expanded. Multi-state commitment search is a middle ground between RTA* and WA*, since the set of candidate nodes for expansions is kept constant between one and the size of the OPEN list. [88] describes two approaches for multi-state commitment search. First, adding commitment to WA* results in MSC-WA*. Second, decreasing commitment in RTA* results in MSC-RTA*, which we focus on in this section.

Figure 31 contains the pseudo-code for MSC-RTA*. Like MSC-WA*, MSC-RTA* maintains a COMMIT list (of fixed maximum size C) among which it chooses the next node to expand. Nodes in COMMIT are ordered according to increasing h-values. The list is initialized with the start state (Line 2). During each iteration (Lines 3-32), the best state is selected as the current state (Line 4) and removed from COMMIT (Line 5), and its successors are added into COMMIT (Lines 6-26). Then, the h-value of the current state is updated, like in RTA* (Line 27). Finally, the worst nodes in COMMIT are pruned if necessary to keep its size no greater than C (Lines 28-31). The algorithm terminates when the goal state is generated for the first time (Line 8) or when COMMIT becomes empty (Line 3).

In our version of MSC-RTA* (these details are not provided in [88]), a successor is only added into COMMIT if either the list is not full (Lines 11&19) or the h-value of the successor is less than that of the worst node in COMMIT (Lines 13&21), in which case this worst node is replaced with the successor (Lines 14-15 & 22-23). Only nodes that have been in COMMIT are stored in memory. The other generated successors are not stored. Whereas RTA* only stores the successor of the current state it moves to, MSC-RTA* stores a percentage (between 0 and 100 percent included) of the successors of the current state, depending on the distribution of their h-values and those of

```
1. procedure MSC-RTA*($s_{start}$, $heuristic(.)$, $C$): solution cost
2.   $g(s_{start}) := 0$; $h(s_{start}) := heuristic(s_{start})$; $COMMIT := \{s_{start}\}$
3.   while ( $COMMIT \neq \emptyset$ ) do
4.       $state := \arg\min_{s \in COMMIT} \{ h(s) \}$
5.       $COMMIT := COMMIT \backslash \{state\}$
6.       $bestH := \infty$; $secondBestH := \infty$
7.       for each successor $s$ of $state$ do
8.           if ( $s = s_{goal}$ ) then return $g(state) + 1$
9.           if ( $s$ is newly generated ) then
10.              $g(s) := g(state) + 1$; $h(s) := heuristic(s)$
11.              if ( $|COMMIT| < C$ ) then $COMMIT := COMMIT \cup \{s\}$
12.              else $worst := \arg\max_{s \in COMMIT} \{ h(s) \}$
13.                  if ( $h(s) < h(worst)$ ) then
14.                      $COMMIT := COMMIT \backslash \{worst\}$
15.                      $COMMIT := COMMIT \cup \{s\}$
16.          else
17.              $g(s) := \min( g(s), g(state) + 1 )$
18.              if ( $s \notin COMMIT$ ) then
19.                  if ( $|COMMIT| < C$ ) then $COMMIT := COMMIT \cup \{s\}$
20.                  else $worst := \arg\max_{s \in COMMIT} \{ h(s) \}$
21.                      if ( $h(s) < h(worst)$ ) then
22.                          $COMMIT := COMMIT \backslash \{worst\}$
23.                          $COMMIT := COMMIT \cup \{s\}$
24.          if ( $h(s) \leq bestH$ ) then $secondBestH := bestH$; $bestH := h(s)$
25.          else if ( $h(s) < secondBestH$ ) then $secondBestH := h(s)$
26.      end for
27.      $h(state) := \max( h(state) , 1 + secondBestH )$
28.      while ( $|COMMIT| > C$ ) do
29.          $state := \arg\max_{s \in COMMIT} \{ h(s) \}$
30.          $COMMIT := COMMIT \backslash \{state\}$
31.      end while
32.  end while
33.  return $\infty$
```

Figure 31: The MSC-RTA* algorithm

nodes currently in COMMIT. In contrast, WA* stores all generated nodes (in OPEN or CLOSED). This is another dimension along which MSC-RTA* lies between RTA* and WA*.

When $C = 1$, MSC-RTA* starts and ends each iteration (except the last one) with only one node in COMMIT. During the iteration, the current state is removed from COMMIT and successively replaced with (a sub-set of) its successors in decreasing order of their h-values until the best successor remains in COMMIT at the end of the iteration. Since this node is selected as current node at the beginning of the next iteration, MSC-RTA* with $C = 1$ is functionally equivalent to RTA* (but with the additional overhead of maintaining the COMMIT list).

When $C = \infty$, the COMMIT list never fills up. All generated nodes are thus added into COMMIT (and stored in memory). In this case, MSC-RTA* grows a full search frontier, like WA*. The main differences are that MSC-RTA* can re-expands some nodes and that it updates the h-value of the expanded node.

MSC-RTA* is to RTA* as MSC-WA* is to WA*: both multi-state commitment search algorithms vary the level of commitment of the basic algorithm they modify. Furthermore, MSC-RTA*, like MSC-WA*, only expands one node at each iteration. Studying the effect of introducing diversity into MSC-RTA is an interesting direction for future research (see Section 3.8.3).

3.7.2 Beam search

Beam search is another algorithm that takes advantage of a strong commitment to scale up heuristic search to larger problems [7, 170]. Beam search is typically a variant of best-first search in which only a fixed number (called the *width* of the beam) of nodes are considered for expansion at each iteration. The beam width is thus equivalent to the size of the commitment list in MSC-WA* and MSC-KWA*.

Furthermore, beam search typically expands all the nodes under consideration in parallel [7, 170]. In other words, beam search uses a full measure of diversity similarly to MSC-KWA* with $K = C$.

Therefore, one contribution of this work is the observation that beam search can be construed as the combination of two distinct principles (namely commitment and diversity) applied to best-first search. In contrast, the standard view of beam search considers its strong commitment as the only factor in its improved scaling behavior. Our research suggests otherwise, with diversity (that is, parallel expansion) as another contributing factor, as demonstrated by our experiments in the N-Puzzle, the Rubik's Cube, and to a smaller degree in the 4-peg Towers of Hanoi domain. The isolation of these two separate factors raises the following questions:

- What are the relative contributions of these two factors, and how do they vary across domains and with different values of K?
- Are there any benefits to using different levels of commitment and diversity (namely different values for C and K)? (see Section 3.8.2 for preliminary results in the N-puzzle)

Finally, one important difference between beam search and MSC-KWA* is that, while MSC-KWA* keeps all discarded nodes in memory (in the RESERVE list), beam search never stores them at all. So when the beam becomes empty, beam search has no reserved nodes available to refill it and it must stop. On the other hand, when the beam is wide enough that it never empties out, beam search wastes no memory on the RESERVE list. There is thus a continuum of algorithms between beam search and MSC-KWA*. They are characterized by the size R of the RESERVE list. As R increases from 0 (beam search) to infinity (MSC-KWA*), the risk of terminating without a goal because of an empty beam decreases, whereas the risk of running out of memory before a goal is found increases. The trade-off between these risks depends on both the structure of the search graph (e.g., its average branching factor) and the value of K (or beam width). The empirical evaluation of these variants of beam search, as well as their comparison against the variants of WA* discussed in this chapter, constitute another interesting avenue of research.

3.8 Future work

In this section, we discuss possible directions for future work and present preliminary results when available.

3.8.1 Domain-dependent behaviors of MSC-KWA*

The relative performance of heuristic search algorithms typically varies from domain to domain. This and previous research have shown that domain-dependent effects on the relative performance of our variants of WA* do exist. [88] suggests that MSC-WA* may not significantly improve on WA* in the gridworld domain. Our research shows that the clear improvements exhibited by MSC-KWA* over WA* and its other variants in the N-Puzzle and Towers of Hanoi domains do not carry over to the Rubik's Cube domain. Further work remains to be done to understand which characteristics of the domain (e.g., its average branching factor, the depth of the shallowest goal, the distribution of goal states in the search tree, etc.) or the heuristic function (e.g., how well it discriminates among states, how many ties it creates when ordering nodes for expansion, etc.) can predict the performance of these variants of WA*. Our empirical study only uses one heuristic function per domain. It would be interesting to study, within each domain, the effects of varying the level of informedness of the heuristic function on the relative performance of these algorithms.

3.8.2 MSC-KWA* versus beam search

As noted earlier, MSC-KWA* is a close relative of beam search [7, 170], since they only differ in whether to use some of the available memory for a reserve list of nodes. [47] presents the results of our empirical comparison of beam search and MSC-KWA* in the same three domains used in this study. Further work is also needed to study the continuum of methods identified earlier between these two search algorithms when varying the size of the reserve list (see Section 3.7.2).

Furthermore, our description of both MSC-KWA* and beam search as committed and diversified variants of best-first search suggests the possibility of teasing out the effects of these two distinct principles on the overall behavior. In particular, we have so far only considered the situation where the levels of commitment and diversity are equal, which is the case both in MSC-KWA* with K=C and in beam search. But could different levels of diversity and commitment lead to better performance? We briefly address this issue for MSC-KWA* in the following sub-section.

3.8.2.1 Preliminary study of MSC-KWA with $1 \leq K \leq C$ in the N-Puzzle*

Since MSC-KWA* makes it possible to fine-tune the relative contributions of commitment (through C) and diversity (through K), we vary both C and K (in addition to W) and run MSC-KWA* in the 48-Puzzle. Our goal is to determine empirically whether some combinations of distinct values for C and K improve the behavior of MSC-KWA*.

Table 16: Performance of MSC-KWA* with varying C and K in the 48-Puzzle when solving all instances with an average solution cost of less than 10,000

W	C	K	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)
0.56	30	10	9,331.20	619,561	610,862	0.413
0.56	30	16	9,903.56	683,113	672,019	0.462
0.60	30	6	9,357.16	641,944	633,731	0.428
0.67	8	3	8,959.04	992,811	983,515	0.721
0.67	9	3	7,519.12	964,381	955,302	0.677
0.67	20	3	3,603.20	1,647,091	1,629,928	1.235
0.67	20	4	9,178.16	898,218	887,005	0.623
0.67	30	4	6,619.72	1,274,879	1,257,713	0.912
0.75	9	3	9,177.56	811,015	803,034	0.569
0.75	20	3	4,415.60	710,776	702,688	0.492
0.75	30	5	9,449.52	640,595	631,162	0.428
0.80	20	3	4,867.72	769,685	759,838	0.533
0.80	30	4	8,755.00	911,618	897,308	0.631
0.86	20	3	6,136.64	858,298	847,044	0.612

Table 16 contains the performance data for MSC-KWA* in the 48-Puzzle for experimental conditions in which MSC-KWA* solves all test instances with an average solution cost inferior to 10,000. This additional constraint was imposed because 1) it keeps the table small (there are much more combinations of values for W, C and K for which MSC-KWA* is complete in the 48-Puzzle), and 2) this value is significantly lower than the minimum solution quality achieved by MSC-KWA* with $K = C$ (namely 18,379.32).

While the large number of parameters and the (relatively) small amount of testing (since we do not vary N) makes it hard to detect general trends, we can still make the following observations in the 48-Puzzle.

- For all parameter settings in the table, C is significantly larger than 1 (the most committed or depth-first setting) and K is less than or equal to about half the value of C. This seems to indicate that a small amount of diversity within a somewhat larger width of commitment (but still much smaller than the length of the full OPEN list) is enough for good scaling performance. A larger value of C or K makes the search too unfocused, whereas a smaller value of C or K makes the search rely too much on the h-values of the very few best nodes. In both cases, the excessive memory consumption prevents successful termination because the

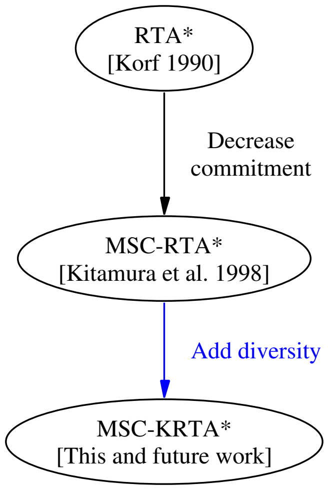


Figure 32: Proposed evolution of RTA*

search front grows too wide or too deep (in the wrong direction), respectively. It is not clear at this point how to find (near-)optimal settings for these parameters, other than empirically on a domain-by-domain basis. The table shows that finding the right combination of (possibly distinct) values for C and K can result in a significant improvement in solution quality (five-fold in this case, from about 18,000 down to 3,600).

- Even though our results do not exhibit a clear trend in the effect of W, the fact that there exists such an effect is worth commenting upon. Recall that the best scaling behavior of MSC-KWA* with $K = C$ is obtained for values of W that are close to one. When $K \neq C$, only a sub-set of the COMMIT list is expanded in parallel. Therefore, in contrast to the situation in which $K = C$ discussed above, nodes with different g-values cohabit in COMMIT and the relative weight on their h-values does influence their ordering. This is why the behavior of MSC-KWA* is affected by the value of W in this case. The effect can be significant. For example, the performance reported as the first row in Table 16 is degraded significantly when increasing the value of W. With $W = 0.80$, the average solution cost is approximately doubled to about 18,000. With $W = 0.99$, MSC-KWA* only solves 92 percent of the test instances.

3.8.3 Introducing diversity in MSC-RTA*

Section 3.7.1 describes how [88] varies the level of commitment in RTA*, thus transforming the real-time search algorithm into a best-first search algorithm called Multi-State Commitment RTA* (MSC-RTA*). Our research naturally suggests a direction for future work, namely to study the effects of introducing diversity into MSC-RTA*. MSC-KRTA*, the resulting algorithm, is to MSC-RTA* as MSC-KWA* is to MSC-WA*. The proposed evolution of RTA* into MSC-KRTA* is depicted in Figure 32. In the next sub-section, we describe MSC-KRTA* and report on a preliminary empirical evaluation.

3.8.3.1 The MSC-KRTA algorithm*

MSC-RTA* is less committed than RTA*: as C increases, its depth-first component weakens and its best-first component strengthens. Like WA* and RTA*, MSC-RTA* can temporarily and wastefully get trapped in goal-free regions of the search space due to misleading h-values. To address this problem, we propose to add diversity in the MSC-RTA* search by expanding several nodes in parallel at each iteration.

Figure 33 contains the pseudo-code for MSC-KRTA*. The only change from MSC-RTA* are Lines 4-9 where K nodes in COMMIT are selected for parallel expansion (in Lines 1-33). When $K = 1$, MSC-RTA* reduces to MSC-RTA*. When $K = C$, all nodes in COMMIT are expanded in parallel.

MSC-KRTA* adds one parameter (namely K) to the only parameter (namely C) of MSC-RTA*. As with MSC-KWA*, we eliminate one parameter by making K equal to C and we run MSC-RTA* and MSC-KRTA* in the N-Puzzle. Figure 34 compares the performance of all algorithms.

Overall, both MSC-RTA* and MSC-KRTA* seem to perform at least as well as MSC-KWA*. In fact, as the domain gets larger, they both seem to improve on MSC-KWA* in terms of memory consumption for large values of K. So much so that both algorithms exhibit a significantly better memory-solution quality trade-off in the 48-Puzzle. Since one important difference between MSC-KWA* and MSC-KRTA* is that the latter revisits nodes and can learn better informed h-values for them, these preliminary results suggest as an interesting direction for future work the study of how

```
1. **procedure** MSC-KRTA*($s_{start}, heuristic(.), C, 1 \leq K \leq C$): *solution cost*
2.   $g(s_{start}) := 0; h(s_{start}) := heuristic(s_{start}); COMMIT := \{s_{start}\}$
3.   **while** ( $COMMIT \neq \emptyset$ ) **do**
4.      $SET := \emptyset$
5.      **while** ( ($COMMIT \neq \emptyset$) **and** ($|SET| < K$) ) **do**
6.         $state :=$ arg min$_{s \in COMMIT}$ $\{ h(s) \}$
7.         $SET := SET \cup \{state\}$
8.         $COMMIT := COMMIT \backslash \{state\}$
9.      **end while**
10.     **for each** $state$ **in** $SET$ **do**
11.        $bestH := \infty; secondBestH := \infty$
12.        **for each** successor $s$ of $state$ **do**
13.           **if** ( $s = s_{goal}$ ) **then return** $g(state) + 1$
14.           **if** ( $s$ is newly generated ) **then**
15.              $g(s) := g(state) + 1; h(s) := heuristic(s)$
16.              **if** ( $|COMMIT| < C$ ) **then** $COMMIT := COMMIT \cup \{s\}$
17.              **else** $worst :=$ arg max$_{s \in COMMIT}$ $\{ h(s) \}$
18.                 **if** ( $h(s) < h(worst)$ ) **then**
19.                    $COMMIT := COMMIT \backslash \{worst\}$
20.                    $COMMIT := COMMIT \cup \{s\}$
21.           **else**
22.              $g(s) :=$ min( $g(s), g(state) + 1$ )
23.              **if** ( $s \notin COMMIT$ ) **then**
24.                 **if** ( $|COMMIT| < C$ ) **then** $COMMIT := COMMIT \cup \{s\}$
25.                 **else** $worst :=$ arg max$_{s \in COMMIT}$ $\{ h(s) \}$
26.                    **if** ( $h(s) < h(worst)$ ) **then**
27.                       $COMMIT := COMMIT \backslash \{worst\}$
28.                       $COMMIT := COMMIT \cup \{s\}$
29.           **if** ( $h(s) \leq bestH$ ) **then** $secondBestH := bestH; bestH := h(s)$
30.           **else if** ( $h(s) < secondBestH$ ) **then** $secondBestH := h(s)$
31.        **end for**
32.        $h(state) :=$ max( $h(state)$ , $1 + secondBestH$ )
33.     **end for**
34.     **while** ( $|COMMIT| > C$ ) **do**
35.        $state :=$ arg max$_{s \in COMMIT}$ $\{ h(s) \}$
36.        $COMMIT := COMMIT \backslash \{state\}$
37.     **end while**
38.  **end while**
39. **return** $\infty$
```

Figure 33: The MSC-KRTA* algorithm

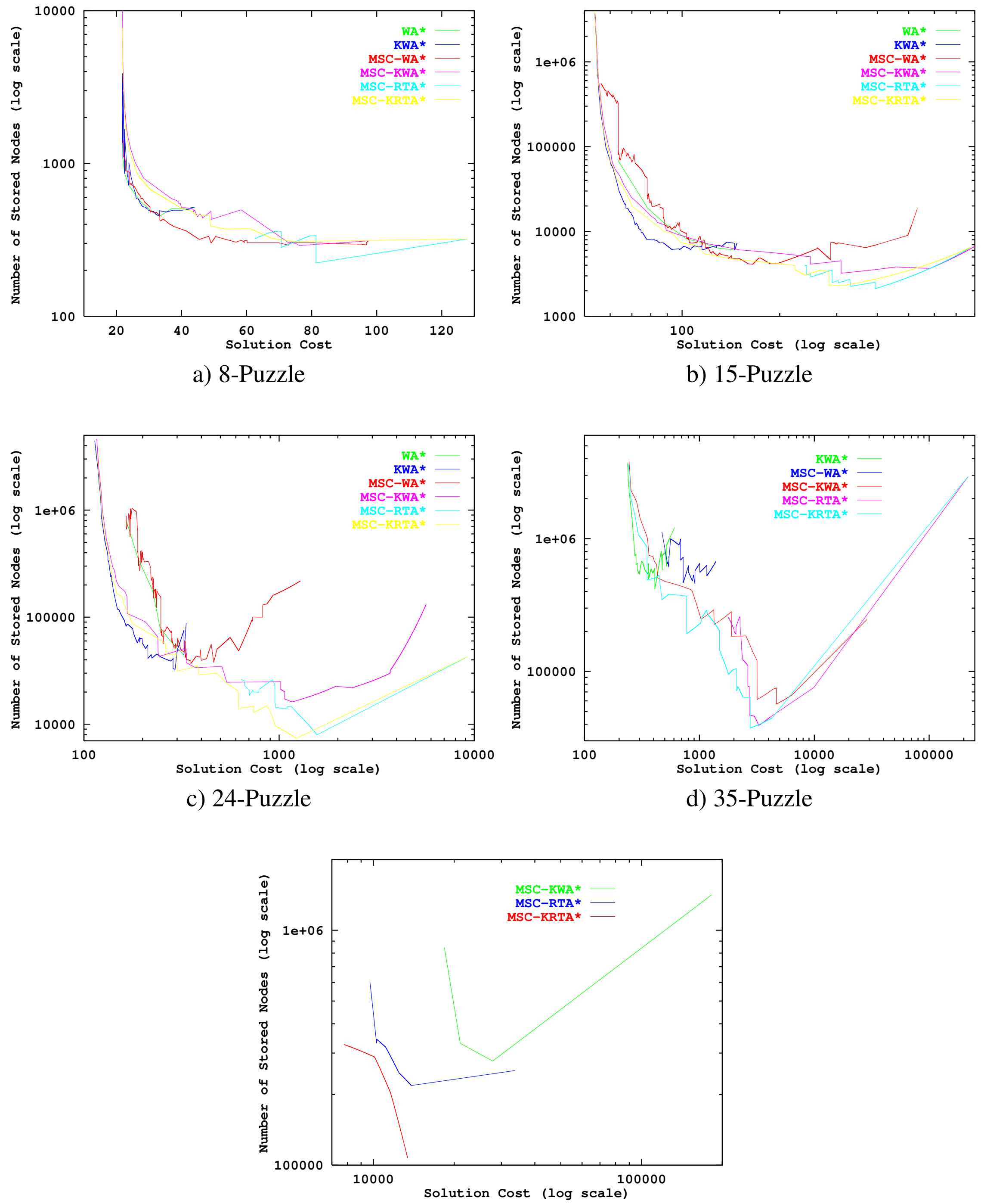


Figure 34: Performance comparison: WA*, KWA*, MSC-WA*, MSC-KWA*, MSC-RTA*, and MSC-KRTA* in the N-Puzzle

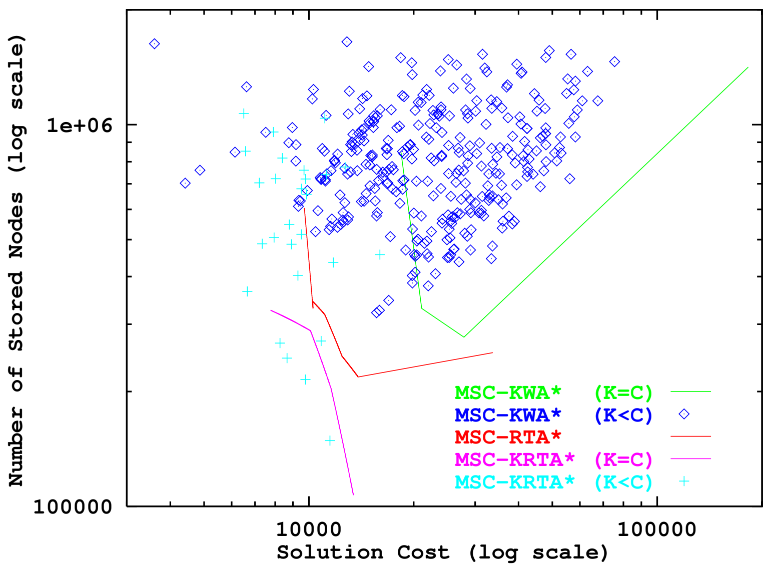


Figure 35: Performance comparison: MSC-KWA*, MSC-RTA*, and MSC-KRTA* in the 48-Puzzle

and when this learning mechanism improves performance when combined with diversity and multi-state commitment in best-first search. Furthermore, the fact that MSC-KRTA* performs better than MSC-RTA* in large instances of this domain provides additional support for our claim that diversity and commitment can be combined with beneficial effects.

Finally, as is the case with MSC-KWA* (Section 3.8.2.1), a potential advantage of MSC-KRTA* is the possibility of using different levels of commitment and diversity by setting K to a strictly smaller value than C. Preliminary results in the 48-Puzzle reported in Figure 35 suggest that doing so may improve the scaling behavior of MSC-KRTA*. Though the effect seems smaller than for MSC-KWA*, further empirical evaluation remains to be done.

3.9 Conclusions

The research described in this chapter makes the following contributions.

- It provides stronger empirical support for the improved scaling behavior of MSC-WA* over WA* by 1) measuring its search effort, solution quality and memory consumption while varying both the size of the commitment list and the relative weight on the h-values (the original study kept this weight fixed), and by 2) testing it in two additional domains.

- It provides stronger empirical support for both the speedup and reduced memory consumption (for a given solution cost) of KWA* over WA* by 1) using a slightly improved implementation of KWA* that does scale it up to the 35-Puzzle (the original implementation did not) and 2) by testing it in two additional domains.

- It provides the first comparison between two existing algorithms that were developed independently. Our empirical comparison highlights the different ways in which they improve on WA*. In the N-Puzzle for example, we show that 1) KWA* takes advantage of its stronger breadth-first search component to improve the solution quality, but that 2) MSC-WA* takes advantage of its stronger depth-first search component to reduce memory consumption.

- It shows how to combine the orthogonal ideas of commitment and diversity into a new algorithm called MSC-KWA* that, in two of our benchmark domains, scales up to even larger problems than either KWA* or MSC-WA* taken individually.

- It underscores the strong similarity between MSC-KWA* and beam search. This observation in turn provides a new view of beam search as the result of applying both commitment and diversity to best-first search.

CHAPTER IV

LIMITED DISCREPANCY BEAM SEARCH[*]

4.1 Introduction

In Chapter 3, we applied the ideas of commitment and diversity to the WA* algorithm, resulting in the MSC-KWA* algorithm and we showed that 1) MSC-KWA* scales up better than existing variants of WA* and 2) MSC-KWA* is functionally similar to beam search. In fact, these two features of MSC-KWA* and beam search are essential to scaling: commitment involves explicit control over the width of the search tree in order to eliminate its exponential growth; and diversity (i.e., the expansion of each full level in parallel) reduces the risk of being led astray by the heuristic function. These observations motivate our interest in beam search.

Beam search is a well-known solution to the problem of the exponential memory requirements of best-first search. Beam search sacrifices solution quality for reduced memory consumption by pruning nodes that are ranked worse than the best B nodes currently under consideration. B, called the *bandwidth* or simply the *width* of the beam,[1] is a constant parameter that is set at the beginning of the search. The smaller B, that is the more pruning occurs at each step of the search, the less memory beam search consumes at each level of the search tree and the larger the problem instances beam search is able to solve. Unfortunately, more pruning typically increases the probability of discarding good nodes and thus often reduces the solution quality. Excessive pruning can even render the search incomplete. Therefore, the choice of a good B value (or equivalently an appropriate pruning power) depends on the amount of available memory, the optimal path cost (or alternatively the minimum depth of the goal in the search tree) and the quality of the heuristic function (which ranks the nodes before pruning occurs).

In this chapter, we review several existing beam search algorithms but we focus on its standard

[*]This chapter first appeared as [47].

[1]Depending on the context, *beam* denotes either the complete set of stored nodes at all levels of the search tree (in most of this chapter), or only the subset of stored nodes at a given level of the search tree (in the pseudo-code of beam search variants and its discussion in the main text).

application to breadth-first search. Breadth-first-based beam search only keeps B nodes at each level of the search tree which is built in a breadth-first manner. Therefore, the memory consumption of beam search is proportional to B times the maximum search depth. Now, the available memory and the goal's depth in the search tree are usually fixed by the machine configuration and the search problem, respectively. Tuning the behavior of beam search thus requires changing the value of B. Assuming that the search time is only a secondary concern,[2] the best value of B is often one that returns the highest solution quality without exhausting the available memory.[3]

The research presented in this chapter is based on the observation that, even in some large problems, beam search can solve most instances using a large B and therefore with a good solution quality. For example, in our experimental setup, beam search with a large beam ($B = 10,000$) solves about eighty percent of random instances in the 48-Puzzle. In this domain, the solution quality it outputs is on average about an order of magnitude higher than that output by the variants of WA$*$ studied in Chapter 3. The research problem addressed in this chapter is how to solve the remaining twenty percent of instances for which beam search runs out of memory. Assuming the value of B and the available memory make it possible for the beam to reach deep enough (namely to the goal's depth) into the search space, its failure to find the goal must be attributed to misguided pruning. In other words, the heuristic function is to blame for wrong rankings of nodes at one or more levels. In order to correct these heuristic failures, we propose to use backtracking. The main contribution of this chapter is to combine the ideas of beam search and backtracking. The result is a memory-bounded algorithm that 1) behaves like beam search until memory runs out and 2) keeps searching after memory runs out by retracting previous node-ranking decisions and searching in new directions. Having thus solved the memory-consumption problem of beam search with large beams, our research issue becomes that of finding an efficient backtracking strategy in order to solve hard instances in a reasonable amount of time.

We first introduce the breadth-first-based beam search algorithm. We extend it into a depth-first beam (DB) search algorithm using chronological backtracking. Then, we describe another existing

[2]This is usually the case because beam search (like breadth-first search and more generally best-first search) runs out of memory in a matter of seconds or minutes on standard benchmarks.

[3]We assume that finding any solution regardless of its quality is the primary objective of the search. In other words, terminating without a solution is equivalent to reaching a solution quality of minus infinity.

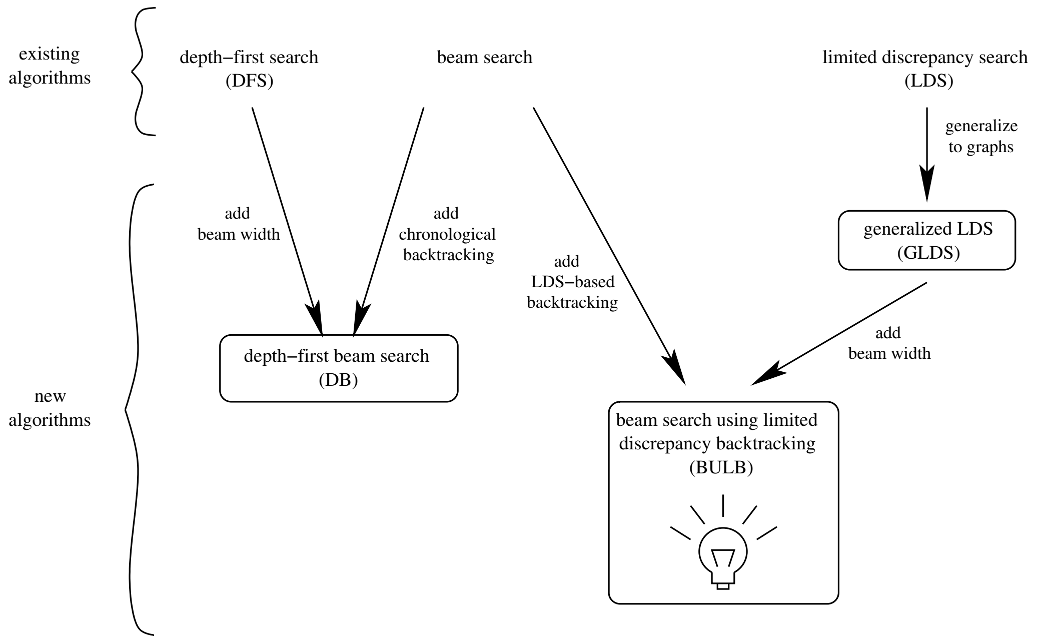

existing algorithms
depth−first search (DFS)
beam search
limited discrepancy search (LDS)
generalize to graphs
add beam width
add chronological backtracking
add LDS−based backtracking
generalized LDS (GLDS)
add beam width
depth−first beam search (DB)
new algorithms
beam search using limited discrepancy backtracking (BULB)

Figure 36: Roadmap for this research

backtracking strategy based on limited discrepancy search and combine it with beam search. The resulting algorithm, called BULB (for Beam search Using Limited discrepancy Backtracking), not only scales up to large domains, just like beam search, it also finds solutions of much higher quality than the best algorithms in Chapter 3 without running out of memory and within reasonable amounts of time. Therefore, our empirical study of BULB supports our hypothesis that beam search quickly solves larger problems when it is enhanced with backtracking based on limited discrepancy search. The roadmap for this research is depicted in Figure 36.

4.2 Beam search

Popular early applications of beam search are numerous and include speech recognition [4, 126], job-shop scheduling [44], and learning [28]. In fact, there are many variants of beam search in the literature. [7] provides this general definition: beam search is any technique "in which a number of [...] alternatives (the beam) are examined in parallel. [It] is a heuristic technique because heuristic rules are used to discard non-promising alternatives in order to keep the size of the beam as small as possible." The two defining characteristics in this definition are that 1) a set of nodes are expanded in parallel and 2) pruning rules are used to discard some nodes. In the extreme case (that is, when

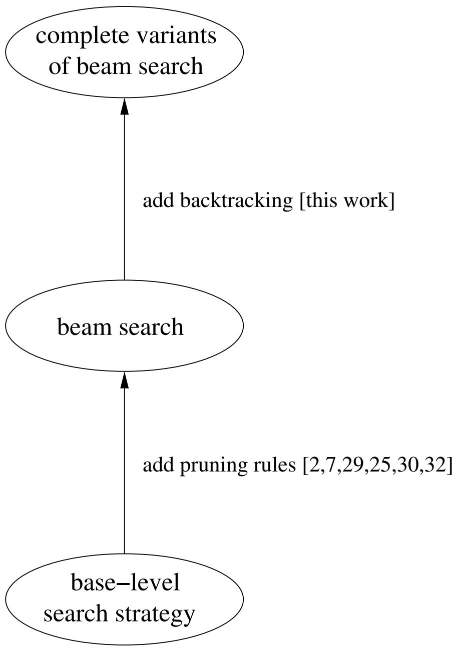

complete variants of beam search
add backtracking [this work]
beam search
add pruning rules [2,7,29,25,30,32]
base–level search strategy

Figure 37: Levels of search strategies

all candidates are expanded in parallel and no pruning rules apply), beam search reduces to breadth-first search. An even more general definition of beam search has been proposed that drops the first characteristic [173]. In this case, any node-ordering strategy (be it breadth-, depth-, or best-first) is acceptable, provided that pruning rules are applied. In this chapter, it is helpful to distinguish three levels of search strategies as depicted in Figure 37. The node-ordering strategy just mentioned is the base-level search strategy. Standard beam search simply adds pruning rules and occupies the next, higher level. Since breadth-first-based beam search is quite common [7, 43, 170, 151, 180], we focus on it in this chapter. Note that the node-ordering strategy used in beam search is the same as the base-level one, namely breadth-first. Finally, the main contribution of this chapter is to introduce a new level defined by a backtracking strategy on top of beam search. At this higher level, we discuss both depth-first and limited discrepancy search strategies.

4.2.1 The beam search algorithm

The main motivation for beam search is to reduce the space complexity of breadth-first search from exponential to linear in the search depth.[4] This is achieved by keeping (at most) a constant number

[4]The same motivation applies for best-first-based beam search in general [20, 140, 129]. However, when the base-level search strategy is depth-first, beam search is aimed at speeding up the search (because pruning avoids visiting goal-free sub-trees) [173].

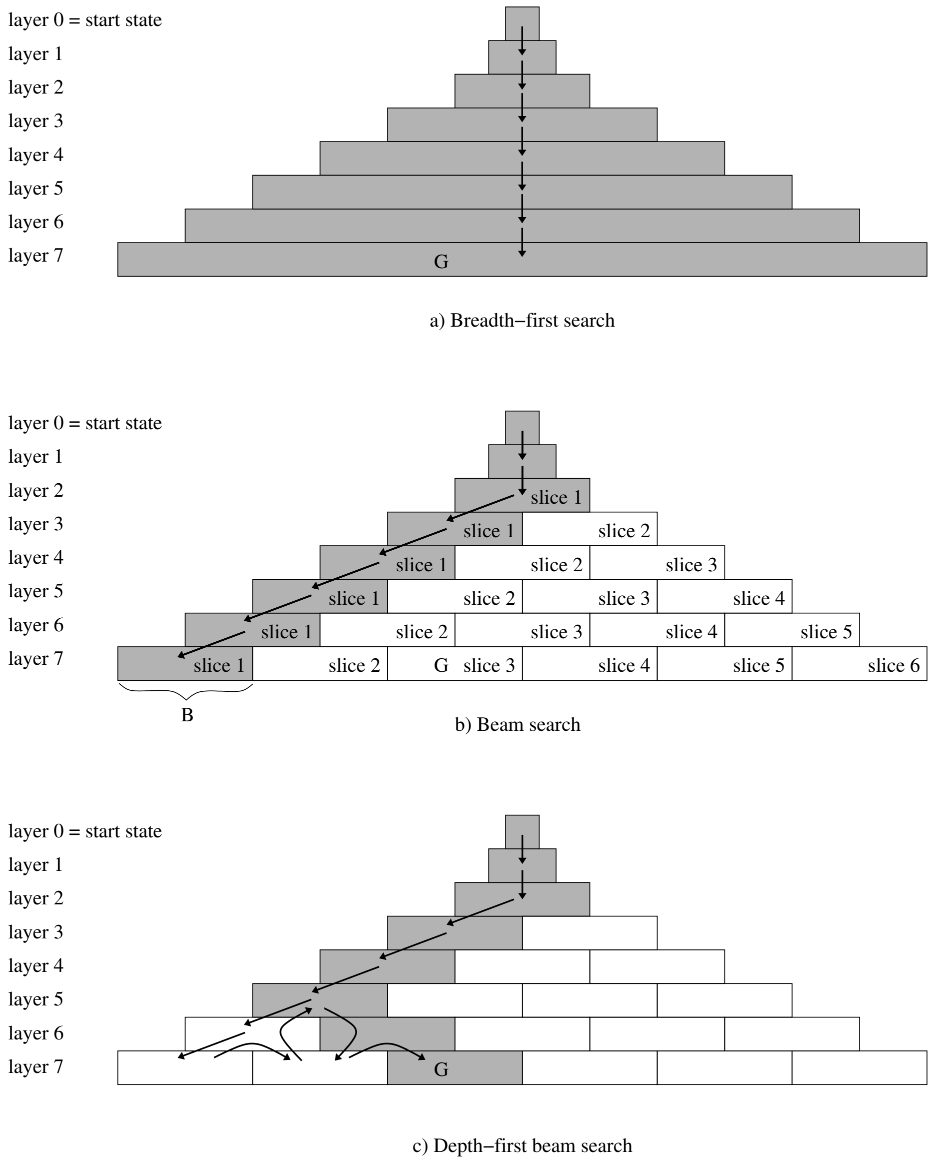


Figure 38: From breadth-first search to beam search to depth-first beam search

B of nodes at each level in the search tree and pruning all additional nodes. B is the size (or bandwidth) of the beam. Pruned nodes are those in excess of B with the worst heuristic values. Thanks to this pruning mechanism, the space complexity of beam search is $O(Bd)$, where d is the maximum search depth. Figure 38 illustrates the memory requirements (shaded areas) of breadth-first search (a) and beam search (b). In the figure, the search tree is built from the top down, starting in the start state. In breadth-first search, each layer is fully generated and stored in memory before the one below it. Thus, in the worst case, the full search tree is stored down to the depth of the goal state. The space complexity of breadth-first search is $O(b^d)$, where b is the branching factor. In beam search, each layer is split into slices containing at most B nodes. During the search, the set of nodes at the current level is limited to a single slice. The next layer is generated, that is the set of successor nodes of the current slice (not the whole layer). Then the nodes in the next layer are ordered according to increasing heuristic values (from left to right in the figure) and the layer is split into slices. Only the best (i.e., leftmost slice) is stored in memory and becomes the current layer at the next level.

Figure 39 contains the pseudo-code for beam search. Beam search takes B as an input parameter. First, it initializes the hash table and the beam with the start state (Lines 2-3). Then it iterates on the main loop (Lines 4-22) until 1) the beam is empty, in which case no solution was found (Line 23), 2) a goal is generated for the first time (Line 8), or 3) the memory is full (Line 20). At each iteration, the set of all successors of states in the beam is built (Lines 5-11), the beam is reinitialized (Line 12) and the search depth is incremented by one (Line 13). Finally, the beam is filled up at the next (deeper) search level (Line 14-21): Until the beam is filled to capacity or there are no more successors to add (Line 14), the best successor is selected (Line 15), removed from the set of successors (Line 16), and added to the hash table (Line 18) and the beam (Line 19), provided it is not already in the hash table (Line 17). In effect, beam search expands all of the nodes in the beam, orders them according to increasing heuristic values, and adds the new ones in order into the next-level beam.

Finally, we examine the behavior of beam search under two extreme conditions defined by the bounding values for B.

When $B = 1$, beam search reduces to greedy search since it always expands next the newly

```
1.  procedure Beam($s_{start}$, $heuristic(.)$, $B$): solution cost
2.    $g$ := 0; $hashTable$ := $\{s_{start}\}$
3.    $BEAM$ := $\{s_{start}\}$
4.    while ( $BEAM \neq \emptyset$ ) do
5.        $SET$ := $\emptyset$
6.        for each $state$ in $BEAM$ do
7.            for each successor $s$ of $state$ do
8.                if ( $s = s_{goal}$ ) then return $g + 1$
9.                $SET$ := $SET \cup \{s\}$
10.           end for
11.       end for
12.       $BEAM$ := $\emptyset$
13.       $g$ := $g + 1$
14.       while ( ($SET \neq \emptyset$) and ($|BEAM| < B$) ) do
15.           $state$ := arg $\min_{s \in SET}$ { $heuristic(s)$ }
16.           $SET$ := $SET \setminus \{state\}$
17.           if ( $state \notin hashTable$ ) then
18.               $hashTable$ := $hashTable \cup \{state\}$
19.               $BEAM$ := $BEAM \cup \{state\}$
20.               if ( $hashTable$ is full ) then return $\infty$
21.       end while
22.   end while
23. return $\infty$
```

Figure 39: The beam search algorithm

generated successor of the current node with the smallest heuristic value. Greedy search may exhibit poor solution quality when the heuristic function is misleading. It may even become stuck in a dead-end where the current node has no new successor.

When $B = \infty$, beam search behaves like breadth-first search since the beam, as large as the largest layer, contains all newly generated nodes at the next level. Breadth-first search is guaranteed to find a minimum-cost path (assuming that all edge costs in the search space are equal, which we assume throughout this chapter). Unfortunately, due to its exponential space requirements, it may run out of memory before doing so.

4.2.2 Motivation for backtracking beam search

The foregoing description of beam search naturally creates the following expectations concerning its behavior.

- For small values of B, beam search may terminate without a solution because the beam becomes empty when the current node has no new successors.
- As B increases, beam search finds solutions of higher quality but also uses more and more time and memory.
- As B increases even more, beam search may terminate without a solution because memory fills up before a goal is found.

Table 17 contains the performance data for beam search averaged over the 50 random instances of the 48-Puzzle used in Chapter 3. These data essentially exhibit the expected trends. For small values of B (namely 1 and 2), beam search solves none of the instances (see last column). For larger values of B (up to 10,000), beam search solves at least some instances, and the solution cost generally decreases as B increases, while the space usage and the runtime follow an opposite trend. Finally, when B is larger than 10, beam search becomes incomplete (that is, it does not solve 100 percent of our instances) again due to memory shortages.

In addition to confirming our expectations, the data underscore two somewhat surprising and complementary features. First, the solution quality exhibited by beam search for large values of B is quite high since the average solution cost over solved instances is on the order of 400. This is about

Table 17: Performance of beam search in the 48-Puzzle

B	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
1	N/A	N/A	N/A	N/A	0
2	N/A	N/A	N/A	N/A	0
3	121,766.68	938,369	365,298	0.615	100
4	24,518.68	247,744	98,071	0.161	100
5	11,737.12	147,239	58,680	0.090	100
6	22,019.68	323,566	132,110	0.208	100
7	22,085.92	387,919	154,591	0.250	100
8	19,463.92	391,254	155,699	0.264	100
9	21,804.96	483,132	196,229	0.309	100
10	36,281.64	904,632	362,799	0.601	100
20	32,879.92	1,655,928	657,549	1.136	96
30	33,732.74	2,561,703	1,011,897	1.879	94
40	24,936.02	2,524,773	997,315	1.944	94
50	25,341.44	3,211,244	1,266,902	2.495	86
60	24,635.84	3,743,041	1,477,936	2.995	90
70	19,537.37	3,469,959	1,367,356	2.727	92
80	11,908.16	2,411,950	952,341	1.843	86
90	12,978.54	2,966,322	1,167,713	2.220	92
100	12,129.88	3,079,594	1,212,579	2.296	86
200	6,266.98	3,176,821	1,252,423	2.349	82
300	3,469.15	2,625,067	1,039,169	1.890	82
400	3,906.33	3,950,665	1,560,311	3.156	78
500	2,302.86	2,899,765	1,148,559	2.205	74
600	2,161.64	3,263,774	1,293,419	2.524	78
700	1,747.58	3,070,222	1,219,046	2.537	90
800	1,527.50	3,059,038	1,216,990	2.484	72
900	1,448.03	3,261,173	1,297,485	2.747	76
1,000	1,337.95	3,346,004	1,331,451	2.822	84
2,000	747.02	3,676,719	1,479,593	3.448	94
3,000	612.42	4,491,479	1,814,334	4.314	86
4,000	527.88	5,131,180	2,079,790	4.746	80
5,000	481.30	5,814,061	2,365,603	5.500	86
6,000	466.49	6,750,775	2,748,609	6.591	82
7,000	449.67	7,588,757	3,087,861	7.662	86
8,000	438.52	8,437,526	3,438,717	8.660	80
9,000	443.27	9,588,420	3,910,897	10.132	88
10,000	440.07	10,569,816	4,312,007	11.307	80
20,000	N/A	N/A	N/A	N/A	0

an order of magnitude smaller than the solution cost output by the best algorithms in Chapter 3. Second, and even more surprisingly, the success rate (see last column) remains high (around 80 percent) even for these large values of B. This means that beam search with a relatively large B value solves a majority of instances with a high solution quality and only runs out of memory in a small number of cases. This observation is the motivation for our research question: how to make beam search complete for large values of B? In this chapter, we consider one possible answer, namely backtracking search.

4.3 Backtracking beam search

The foregoing discussion shows that, at least in one large domain, beam search can use a relatively wide beam to find high quality solutions in most instances. Assuming that the beam is not so large that it cannot reach the goal depth, the reason beam search fails to find a goal in a few instances is that the heuristic values are misleading. For example in Figure 38(b), the goal is mistakenly ordered in the third slice of the seventh layer. Since beam search only visits the first slice of each layer, it misses the goal.[5] Figure 38(c) illustrates one possible solution to this problem: if the goal is not found when the search bottoms out, why not backtrack to a previous choice point where the slices may have been mistakenly ordered?

In this section, we describe two new algorithms. First, we introduce a depth-first extension of beam search called DB. Then we describe an existing, alternative backtracking strategy called *limited discrepancy search* which we use in our new BULB algorithm.

4.3.1 The depth-first beam search (DB) algorithm

The DB algorithm is an extension of beam search that continues the search when it bottoms out because the memory is full. Before moving back to an earlier choice point in the search tree in order to explore, for example, the second slice in the next layer, some empty space must be recovered in memory (since it is now full). The easiest way to do this is to purge one slice and to replace it with the new one. When it comes to choosing which stored slice to purge, there are as many options as there are levels in the current path through the search tree. This section discusses the simplest

[5]Since beam search stops when a goal is *generated*, this example assumes that the third slice of layer 7 is not a successor layer of the first slice of layer 6.

choice, namely purging the lastly inserted slice. This strategy is called chronological backtracking and results in a depth-first extension of beam search which we call DB (for Depth-first Beam search). Note that heuristic search algorithms that repeatedly fill up and then purge memory are often quite complicated [19, 143, 86, 177].[6] In contrast, because 1) DB uses a depth-first search strategy, 2) DB always purges contiguous regions of memory and 3) DB does not need to ensure optimality of the solution, its implementation is (relatively) easy. We now describe DB in detail.

Figure 40 contains the pseudo-code for the depth-first beam search (DB) algorithm. The code contains four functions: the top-level *DB()* function and three component functions. We first discuss the component functions in a bottom-up order, and then the top-level function.

The *generateNewSuccessors()* function (Lines 30-38) takes as input a set of states and a heuristic function. It returns an array of all the successors of the input states in order of increasing heuristic values. The output array only contains newly generated states (i.e., states that are not already in the hash table). Note that the output array may contain duplicate states since two (or more) states in the input set may share a newly generated successor. This function does not have any side effects.

The *nextSlice()* function (Lines 14-29) takes as input the current search depth and the index of the needed slice at the next level (as well as a heuristic function and the value of B). It is assumed that the (non-empty) slice at the current level is already in the hash table. This function calls the preceding one to generate all of the successors of the current slice (Line 16) and it locates the needed slice within the complete set of successors (Lines 19-28) and returns it (Line 29). As a side effect, the slice is inserted one successor node at a time into the hash table (Line 23) before the function returns. The main loop of this function (Lines 19-28) fills up the slice: Starting at *startIndex* with an empty slice (Line 19), it iterates over the index into the array *SUCCS* of successors (Line 27) until the end of the layer is reached or the slice is filled to capacity, that is B (Line 20). Each newly generated successor (Line 21) is inserted into the slice (Line 22) and the hash table (Line 23). If the hash table happens to fill up during the processing of the slice (Line 24), the function aborts (Line 26), but only after having cleared the hash table of the incomplete slice (Line 25).[7]

[6]The difficulty lies mainly in the need to maintain 1) the network of data structures (including for example an array of states for the hash table and one or more linked lists of states for buckets and other ordered lists) in a coherent state, and 2) a set of counters for insuring that the search is admissible.

[7]This must be done since the calling function (namely *DBprobe()*) assumes that either the search must continue and the full slice at the next level is in memory or that the search must backtrack and the memory was not changed in *nextSlice()*.

```
procedure DB($s_{start}$, $heuristic(.)$, $B$): solution cost
    $g(s_{start}) := 0$; $hashTable := \{s_{start}\}$
return DBprobe( 0, $heuristic(.)$, $B$)
procedure DBprobe($depth$, $heuristic(.)$, $B$): solution cost
    $startIndex := 0$
    while ( $true$ ) do
        $\langle SLICE, value, startIndex \rangle :=$ nextSlice($depth$, $startIndex$, $heuristic(.)$, $B$)
        if ( $value \geq 0$ ) then return $value$
        if ( $SLICE = \emptyset$ ) then continue
        $cost :=$ DBprobe($depth + 1$, $heuristic(.)$, $B$)
        for each $s$ in $SLICE$ do $hashTable := hashTable \backslash \{s\}$ end for
        if ( $cost < \infty$ ) then return $cost$
    end while
procedure nextSlice($depth$,$startIndex$,$heuristic(.)$,$B$): ⟨ array of states, integer, integer ⟩
    $currentLayer := \{s \in hashTable \mid g(s) = depth\}$
    $SUCCS :=$ generateNewSuccessors($currentLayer$, $heuristic(.)$)
    if ( $(SUCCS = \emptyset)$ or $(startIndex = |SUCCS|)$ ) then return $\langle \emptyset, \infty, -1 \rangle$
    if ( $s_{goal} \in SUCCS$ ) return $\langle \emptyset, depth + 1, -1 \rangle$
    $SLICE := \emptyset$; $i := startIndex$
    while ( $(i < |SUCCS|)$ and $(|SLICE| < B)$ ) do
        if ( $SUCCS[i] \notin hashTable$ ) then
            $g(SUCCS[i]) := depth$; $SLICE := SLICE \cup \{SUCCS[i]\}$
            $hashTable := hashTable \cup \{SUCCS[i]\}$
            if ( $hashTable$ is full ) then
                for each $s$ in $SLICE$ do $hashTable := hashTable \backslash \{s\}$ end for
                return $\langle \emptyset, \infty, -1 \rangle$
        $i := i + 1$
    end while
    return $\langle SLICE, -1, i \rangle$
procedure generateNewSuccessors($stateSet$, $heuristic(.)$): array of states
    $index := 0$
    for each $state$ in $stateSet$ do
        for each successor $s$ of $state$ do
            if ( $s \notin hashTable$ ) then $SUCCS[index] := s$; $index := index + 1$
        end for
    end for
    Sort states in $SUCCS$ in order of increasing $heuristic(.)$-values
return $SUCCS$
```

Figure 40: The depth-first beam search (DB) algorithm

This function returns a 3-tuple containing the output slice, a flag value, and the index of the following slice to be processed in the next layer. In the normal case (i.e., when the search must continue down the tree from the slice), the flag value equals negative one. However, there are four situations under which this function terminates with a positively-valued flag. All indicate that the search need not continue down the tree below the slice, namely:

- When the set of successors is empty[8] (Line 17);
- When the slice index has gone beyond the last slice[9] (Line 17);
- When the goal is found in the set of successors (Line 18); or
- When the hash table is full (Lines 24-26).

In all four cases, the returned flag is set to a positive value, that is either the solution cost (in the third case) or infinity (in the three other cases). In addition, the returned slice index is irrelevant in all four cases so we arbitrarily set it to negative one.

The *DBprobe()* function (Lines 4-13) takes as input the current search depth (as well as a heuristic function and the value of B). It repeatedly calls the preceding function to generate a new slice at the next level and then recursively calls itself to search the sub-tree below the slice. This is where the chronological backtracking occurs. This function returns a positive value, namely either the cost of the solution found in the sub-tree rooted at the current slice, or infinity.

This function assumes that the slice at the current search depth is already in the hash table and it iterates over the slices at the next layer (Lines 5-13). It calls the previous function to load the next slice into memory (Line 7). If the returned flag value is positive, the search has bottomed out. It returns either the solution cost or infinity (Line 8). Otherwise (i.e., the value of the flag is negative one), the search proceeds down the tree, unless the returned slice turns out to be empty (Line 9). If not, the slice was loaded into memory by *nextSlice()* and *DBprobe()* recursively calls itself on this slice at the next level (Line 10). Upon return, the slice is purged from memory (Line 11) and

[8]This case arises when all of the successors of the current slice are already in the hash table. This is a dead-end for the search,

[9]This is the normal termination condition when the sub-tree under the current slice has been fully searched.

the search proceeds with the sibling slice at the current level, unless the goal was found under the current one (Line 12).

Finally, *DB()* is the top-level function. It takes as input the start state (as well as a heuristic function and the value of B) and returns the cost of a solution (or infinity if none was found). After initializing the hash table with the start state (Line 2), it simply calls *DBprobe()* at depth zero (Line 3).

4.3.2 Limited discrepancy search

While using backtracking to transform beam search into a memory-bounded algorithm solves the memory consumption problem on large instances, is chronological backtracking the most efficient way to do it in terms of runtimes? [61] observes that the chronological backtracking used in depth-first search always revisits its most recent decisions (at the bottom of the tree) before it revisits earlier decisions (at the top of the tree). Since the number of alternatives is exponentially larger at the bottom than at the top of the tree, it takes a long time before it comes back on an early decision. In effect, chronological backtracking puts more trust (or weight) in its early decisions than in its later ones. This is unfortunate because heuristic evaluations are usually more accurate in the proximity of the goal, that is, they are more likely to be inaccurate at the top of the search tree. So, if the algorithm makes a wrong decision at the top of the tree and chooses to explore a large, goal-free sub-tree, depth-first search will waste a lot of time in this sub-tree before it switches to another sub-tree. Our experiments with DB confirm this analysis (see Section 4.4): DB only improves on the solution quality of beam search at the cost of unacceptable runtimes. In this section, we propose to use an existing alternative backtracking strategy called limited discrepancy search which we first describe in its original version and then generalize to graphs.

4.3.2.1 Original limited discrepancy search

[61] proposes a new backtracking algorithm that weighs equally decisions at all depths of the tree. Limited discrepancy search (LDS) assumes that the heuristic ordering of successors is right most of the time. First, LDS searches the tree greedily (that is, with no discrepancy). If no goal is found, it is because (a presumably small number of) wrong turns were made. Thus LDS iteratively searches the tree with an increasing number of allowed wrong turns (or discrepancies). Figure 41 depicts

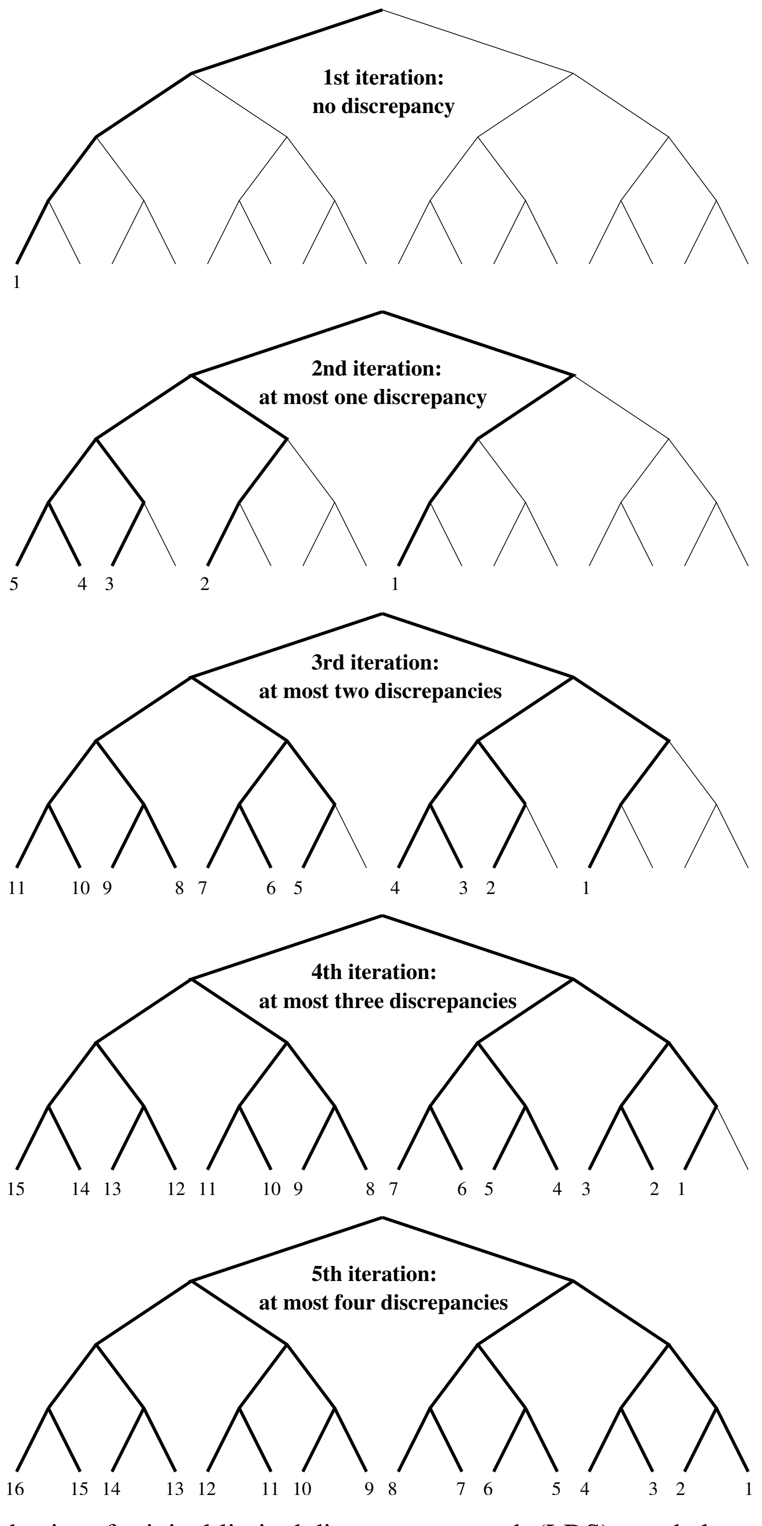


Figure 41: Behavior of original limited discrepancy search (LDS) on a balanced, binary tree

```
1.  procedure LDS($s_{start}$, $heuristic(.)$): solution cost
2.    $Ndiscrepancies$ := 0
3.    while ( $true$ ) do
4.      $cost$ := LDSprobe($s_{start}$, 0, $Ndiscrepancies$, $heuristic(.)$)
5.      if ( $cost < \infty$ ) then return $cost$
6.      $Ndiscrepancies$ := $Ndiscrepancies + 1$
7.    end while

8.  procedure LDSprobe($state$, $depth$, $Ndiscrepancies$, $heuristic(.)$): solution cost
9.    if ( $state$ is a leaf ) then return $\infty$
10.   else $\langle best, second \rangle$ := generateSuccessors($state$)
11.   if ( $(best = s_{goal})$ or $(second = s_{goal})$ ) then return $depth + 1$
12.   if ( $Ndiscrepancies = 0$ ) then return LDSprobe($best$, $depth + 1$, 0, $heuristic(.)$)
13.   else
14.     $cost$ := LDSprobe($second$, $depth + 1$, $Ndiscrepancies - 1$, $heuristic(.)$)
15.     if ( $cost < \infty$ ) then return $cost$
16.     return LDSprobe($best$, $depth + 1$, $Ndiscrepancies$, $heuristic(.)$)
```

Figure 42: The original limited discrepancy search (LDS) algorithm (for balanced binary trees)

this behavior. LDS is designed to work on finite binary trees. In the figure, the children of each node are ordered from left to right by increasing heuristic values. So the heuristic function always recommends going down the left branch while going down the right branch is a wrong turn (or discrepancy) according to the heuristic. The first iteration is a greedy search with no discrepancy. In general, the n^{th} iteration allows for at most $n-1$ discrepancies. Within each iteration, discrepancies occur first at the top of the tree, then further down. Numbers at the bottom of each sub-figure indicate the order in which leaf nodes are visited during the iteration.

Figure 42 contains the pseudo-code for LDS. The top-level function *LDS()* repeatedly performs a limited discrepancy search from the start state (Line 4) by calling *LDSprobe()* with an increasing number of allowed discrepancies (Line 6), starting with no discrepancy (Line 2). Unless the current node is a leaf of the tree (Line 9), *LDSprobe()* generates its successors and recursively calls itself on them. If the maximum number of allowed discrepancies is zero, then only the sub-tree below the best successor is visited with no discrepancy allowed (Line 12). Otherwise, the sub-tree under the worst successor is visited with one less discrepancy allowed (since one was just consumed, Line 14), then the sub-tree under the best successor is visited with the same number of allowed discrepancies (since none was consumed at the current level by following the heuristic recommendation, Line 16). LDS stops when the goal is generated (Line 11).

4.3.2.2 Generalized limited discrepancy search

In order to apply LDS to beam search, we need to make it work on general graphs, not just on binary trees. This requires two modifications. First, it must be able to handle varying branching factors (especially branching factors larger than 2). Second, it must perform cycle detection to avoid following an infinite branch. In LDS, a discrepancy means taking a wrong turn, namely going down the rightmost branch. When the branching factor is larger than 2, choosing any but the best successor is a wrong turn or discrepancy.[10] One approach would be to discard all successors but the best two, thus falling back onto the binary case. This approach makes the search incomplete since some branches are never explored. Therefore, we choose a different approach that views a move down any but the best successor as an acceptable discrepancy. Furthermore, the same weight is assigned to all discrepancies at a given level. This means that each discrepancy at a given level (be it the second, third, or n^{th} best successor) is consumed as fast as possible, and in this order. This takes care of the first issue. Second, cycle detection is done with the hash table scheme used in beam search. We call the resulting algorithm GLDS since it is generalized to work on graphs.

Figure 43 contains the pseudo-code for GLDS. The top-level function is identical to that of LDS except that it initializes the hash table with the start state (Line 2). As for LDS, *GLDSprobe()* performs a limited discrepancy search rooted at its input state. First, the set of its new successor states is built (Lines 9-13). The search backtracks if the goal is found (Line 11), the state is a leaf (Line 14), or the hash table is full (Line 15). Otherwise, the best successor is identified (Line 16). Then two cases can arise. If the maximum number of allowed discrepancies is zero, GLDS calls itself on the best successor with no allowed discrepancies (Line 19). Otherwise, GLDS calls itself successively (Lines 23-31) on the second, third, etc... successor with one less allowed discrepancy (Line 28), before calling itself on the best successor with the same number of allowed discrepancies (Line 33). The behavior of GLDS on a random tree is depicted in Figure 44. Like in Figure 41, the children of each node are ordered from left to right by increasing heuristic values. Numbers at the bottom of each sub-figure indicate the order in which leaf nodes are visited during the iteration.

[10]When there is a tie among best nodes, one of them is chosen arbitrarily and all the other nodes are viewed as discrepancies.

```
1. procedure GLDS($s_{start}$, $heuristic(.)$): solution cost
2.   $Ndiscrepancies$ := 0; $hashTable$ := $\{s_{start}\}$
3.   while ( $true$ ) do
4.     $cost$ := GLDSprobe($s_{start}$, 0, $Ndiscrepancies$, $heuristic(.)$)
5.     if ( $cost < \infty$ ) then return $cost$
6.     $Ndiscrepancies$ := $Ndiscrepancies + 1$
7.   end while

8. procedure GLDSprobe($state$, $depth$, $Ndiscrepancies$, $heuristic(.)$): solution cost
9.   $SET$ := $\emptyset$
10.  for each successor $s$ of $state$ do
11.    if ( $s = s_{goal}$ ) then return $depth + 1$
12.    if ( $s \notin hashTable$ ) then $SET$ := $SET \cup \{s\}$
13.  end for
14.  if ( $SET = \emptyset$ ) then return $\infty$
15.  if ( $hashTable$ has only one empty slot ) then return $\infty$
16.  $best$ := arg $\min_{s \in SET}$ { $heuristic(s)$ }
17.  if ( $Ndiscrepancies = 0$ ) then
18.    $hashTable$ := $hashTable \cup \{best\}$
19.    $cost$ := GLDSprobe($best$, $depth + 1$, 0, $heuristic(.)$)
20.    $hashTable$ := $hashTable \backslash \{best\}$
21.    return $cost$
22.  else
23.    $SET$ := $SET \backslash \{best\}$
24.    while ( $SET \neq \emptyset$ ) do
25.      $state$ := arg $\min_{s \in SET}$ { $heuristic(s)$ }
26.      $SET$ := $SET \backslash \{state\}$
27.      $hashTable$ := $hashTable \cup \{state\}$
28.      $cost$ := GLDSprobe($state$, $depth + 1$, $Ndiscrepancies - 1$, $heuristic(.)$)
29.      $hashTable$ := $hashTable \backslash \{state\}$
30.      if ( $cost < \infty$ ) then return $cost$
31.    end while
32.    $hashTable$ := $hashTable \cup \{best\}$
33.    $cost$ := LDSprobe($best$, $depth + 1$, $Ndiscrepancies$, $heuristic(.)$)
34.    $hashTable$ := $hashTable \backslash \{best\}$
35.    return $cost$
```

Figure 43: The limited discrepancy search algorithm for general graphs (GLDS)

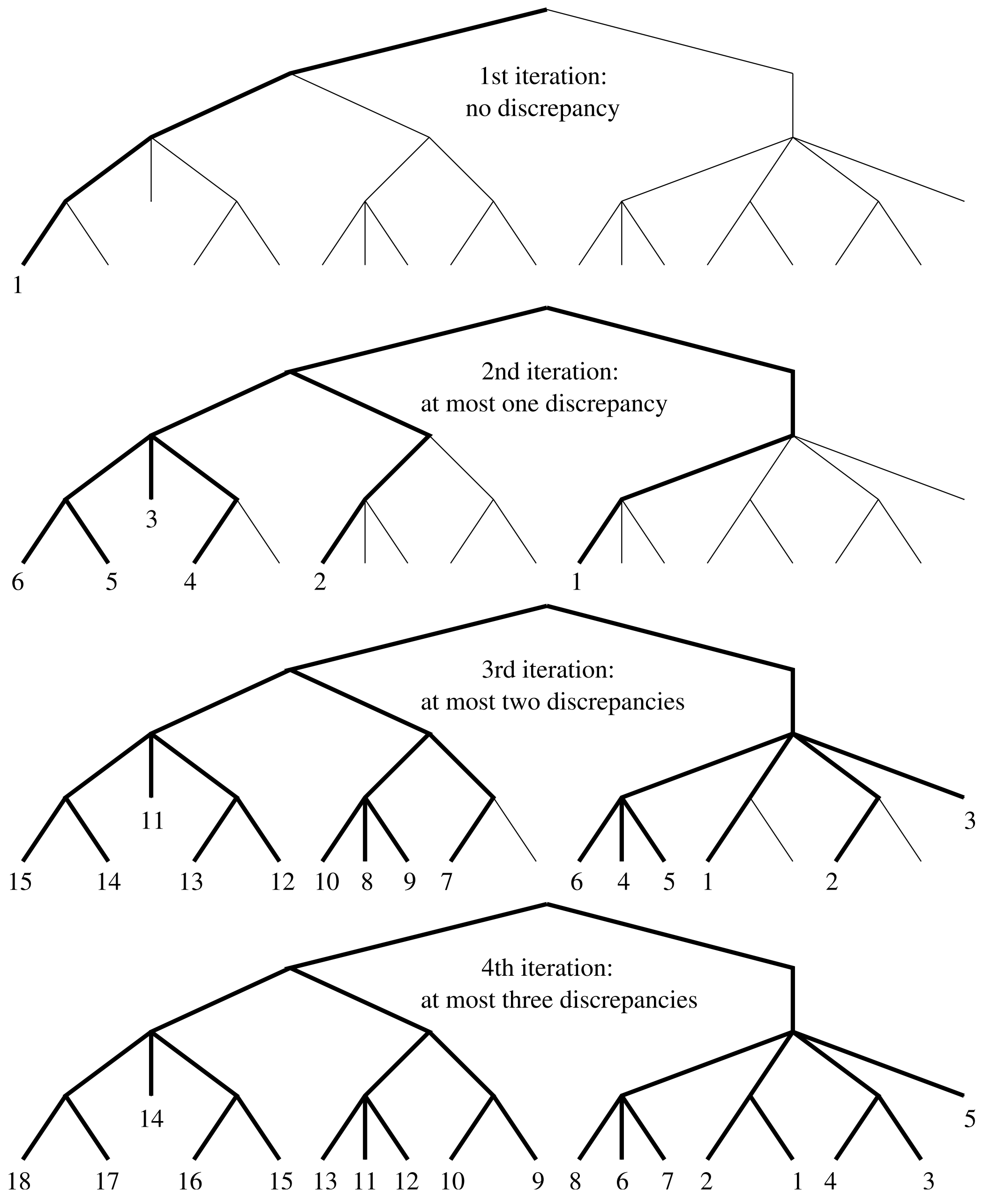


Figure 44: Behavior of GLDS on an irregular tree

4.3.3 Beam search using limited discrepancy backtracking (BULB)

Now that we have extended LDS to work on graphs, we can add backtracking based on limited discrepancy search (instead of chronological backtracking) into beam search. We call the resulting algorithm *BULB* (which stands for Beam search Using Limited discrepancy Backtracking).

Figure 45 contains the pseudo-code for BULB. Its top-level function is the same as the one for GLDS (Lines 1-7 in Figure 43) except that it must initialize the g-value of the start state. The two bottom-level component functions *nextSlice()* and *storeSuccessors()* are the same as the ones for DB (Lines 14-38 in Figure 40). Only the probe function (renamed *BULBprobe()*) differs from the one in both DB and GLDS since it must perform limited discrepancy (not chronological backtracking) on slices (not on individual states).

The function *BULBprobe()* starts by generating the first slice at the next level (Line 9). If any of the four termination conditions shared by DB (see Section 4.3.1) holds, the search has bottomed out and the function returns either the solution cost or infinity (Line 10). Otherwise, two cases may arise (like in LDS and GLDS). The first case is when there is no more discrepancy allowed (Lines 12-15). The search then proceeds to the next level (Line 13) unless the current slice happens to be empty (Line 12). When the sub-tree rooted in the slice is fully explored, the slice is purged from memory (Line 14). The second case is when the number of allowed discrepancies is positive (Lines 17-33). First, the current slice is purged from memory[11] (Line 17). Then, *BULBprobe()* calls itself recursively on the second, third, etc... best slice (Lines 18-27) with one less discrepancy[12] (Line 24). Finally, *BULBprobe()* calls itself recursively on the best slice (Lines 28-33) with the same number of allowed discrepancies as in the current level (Line 31).

[11]This is done because, as discussed in Section 4.3.4.1, we enforce an $O(Bd)$ worst-case space complexity. Therefore, we need to regenerate (and order) the slices at each iteration.

[12]Note that when the search bottoms out for any reason other than having found the goal, the loop should be exited to move on to the best slice. This is the semantics of the *break* statement (Line 22). If there is not enough memory to store this slice, there is not enough memory to store the following slices either. In contrast, when the slice is empty (because all of its states are re-generated), this iteration terminates but the remaining slices must still be processed. This is the semantics of the *continue* statement (Line 23). In this case, the fact that the current slice is empty does not imply that any of the remaining slices will be empty. So they must be checked.

```
 1. **procedure** BULB ($s_{start}$, $heuristic(.)$, $B$): *solution cost*
 2.   $Ndiscrepancies$ := 0; $g(s_{start})$ := 0; $hashTable$ := $\{s_{start}\}$
 3.   **while** ( $true$ ) **do**
 4.       $cost$ := BULBprobe( 0, $Ndiscrepancies$, $heuristic(.)$, $B$)
 5.       **if** ( $cost < \infty$ ) **then return** $cost$
 6.       $Ndiscrepancies$ := $Ndiscrepancies + 1$
 7.   **end while**
 8. **procedure** BULBprobe($depth$, $Ndiscrepancies$, $heuristic(.)$, $B$): *solution cost*
 9.   $\langle SLICE, value, startIndex\rangle$ := nextSlice($depth$, 0, $heuristic(.)$, $B$)
10.   **if** ( $value \geq 0$ ) **then return** $value$
11.   **if** ( $Ndiscrepancies = 0$ ) **then**
12.       **if** ( $SLICE = \emptyset$ ) **then return** $\infty$
13.       $cost$ := BULBprobe($depth + 1$, 0, $heuristic(.)$, $B$)
14.       **for each** $s$ **in** $SLICE$ **do** $hashTable$ := $hashTable \backslash \{s\}$ **end for**
15.       **return** $cost$
16.   **else**
17.       **if** ( $SLICE \neq \emptyset$ ) **then for each** $s$ **in** $SLICE$ **do** $hashTable$ := $hashTable \backslash \{s\}$ **end for**
18.       **while** ( $true$ ) **do**
19.           $\langle SLICE, value, startIndex\rangle$ := nextSlice($depth$, $startIndex$, $heuristic(.)$, $B$)
20.           **if** ( $value \geq 0$ ) **then**
21.               **if** ( $value < \infty$ ) **then return** $value$
22.               **else break**
23.           **if** ( $SLICE = \emptyset$ ) **then continue**
24.           $cost$ := BULBprobe($depth + 1$, $Ndiscrepancies - 1$, $heuristic(.)$, $B$)
25.           **for each** $s$ **in** $SLICE$ **do** $hashTable$ := $hashTable \backslash \{s\}$ **end for**
26.           **if** ( $cost < \infty$ ) **then return** $cost$
27.       **end while**
28.       $\langle SLICE, value, startIndex\rangle$ := nextSlice($depth$, 0, $heuristic(.)$, $B$)
29.       **if** ( $value \geq 0$ ) **then return** $value$
30.       **if** ( $SLICE = \emptyset$ ) **then return** $\infty$
31.       $cost$ := BULBprobe($depth + 1$, $Ndiscrepancies$, $heuristic(.)$, $B$)
32.       **for each** $s$ **in** $SLICE$ **do** $hashTable$ := $hashTable \backslash \{s\}$ **end for**
33.       **return** $cost$
34. **procedure** nextSlice($depth$,$startIndex$,$heuristic(.)$,$B$): $\langle$ *array of states*, *integer*, *integer* $\rangle$
35.   Same as Lines 15-38 in Figure 40
```

Figure 45: The BULB algorithm: Beam search using limited discrepancy backtracking

Table 18: A taxonomy of beam search methods

B	type of backtracking		
	none	chronological	limited discrepancy
1	greedy search (gradient descent)	guided depth-first search	limited discrepancy search (LDS/GLDS)
intermediate value	beam search	depth-first beam search (DB)	beam search using LD backtracking (BULB)
∞	breadth-first search	breadth-first search	breadth-first search

4.3.4 Properties of the BULB algorithm

4.3.4.1 BULB is a memory-bounded algorithm

BULB, like other memory-bounded algorithms [19, 143, 86, 177], continues searching when memory runs out by purging existing nodes from memory. But what is the space complexity of BULB? Typically, depth-first search algorithms only need to keep in memory the path from the start state to the current state, leading to an $O(d)$ complexity, where d is the maximum search depth. Alternatively, depth-first search with node ordering (sometimes called *guided DFS* [167]), like BULB (and DB), also needs to keep in memory the siblings of nodes on the current path, leading to an $O(bd)$ complexity, where b is the maximum branching factor. In the case of BULB, like for variants of beam search, B nodes must be stored at each level, leading to a space complexity of either $O(Bd)$ or $O(Bbd)$. Since BULB uses slice ordering (as in 'node ordering'), the latter case would typically hold. However, in order to be able to perform deeper searches with wider beams, we make sure that the space complexity of BULB does not exceed $O(Bd)$. This is achieved by only storing one slice at each level. Therefore, our implementation of BULB must re-generate (and order) all the successors of a slice every time it backtracks. In Figure 45, the call to *nextSlice()*, which generates the whole set of successor slices, is called at the beginning of each iteration. The code uses the *startIndex* counter for remembering where the next slice begins in the whole set of successors.

4.3.4.2 BULB generalizes both limited discrepancy search and breadth-first search

BULB is to beam search as LDS is to greedy search since only backtracking need be added to transform one into the other. Table 18 shows the resulting taxonomy. When $B = 1$, slices are reduced to single states and BULB reduces to (G)LDS. When $B = \infty$, slices expand into whole

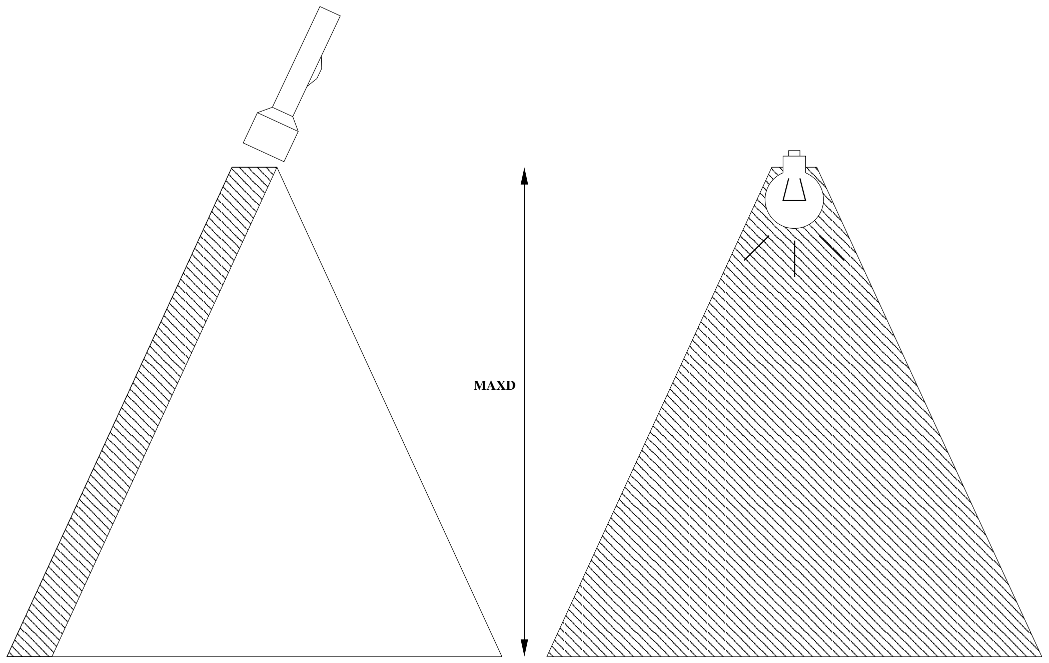


Figure 46: From beam search to BULB search

layers and BULB reduces to breadth-first search. The interesting case is the intermediate one since LDS often terminates without a goal in cyclic graphs (because all successors are being re-generated and the beam becomes empty) and breadth-first search is too memory-intensive in large state spaces. BULB occupies this spot.

4.3.4.3 BULB is a complete algorithm

Let $MAXD$ denote the maximum depth searchable by both beam search and BULB ($MAXD$ is approximately equal to M/B, where M is the number of nodes storable in memory).

The main advantage of BULB over beam search is that the latter, because of misleading heuristic values, may fail to find a goal in some instances where the cost of an optimal path is less than or equal to $MAXD$. In contrast, BULB is complete over the set of all such instances. This is because the backtracking mechanism used in BULB makes sure that all successors of the current slice are potentially generated and expanded if necessary. In other words, BULB is complete because it can potentially visit the whole $MAXD$-deep search tree rooted at the start state (see Figure 46).

The main advantage of BULB over breadth-first search is that it is complete over a much larger set of instances, since breadth-first search can only search a significantly smaller tree before it runs out of memory (the maximum depth searchable by breadth-first search is on the order of $log_b(M)$,

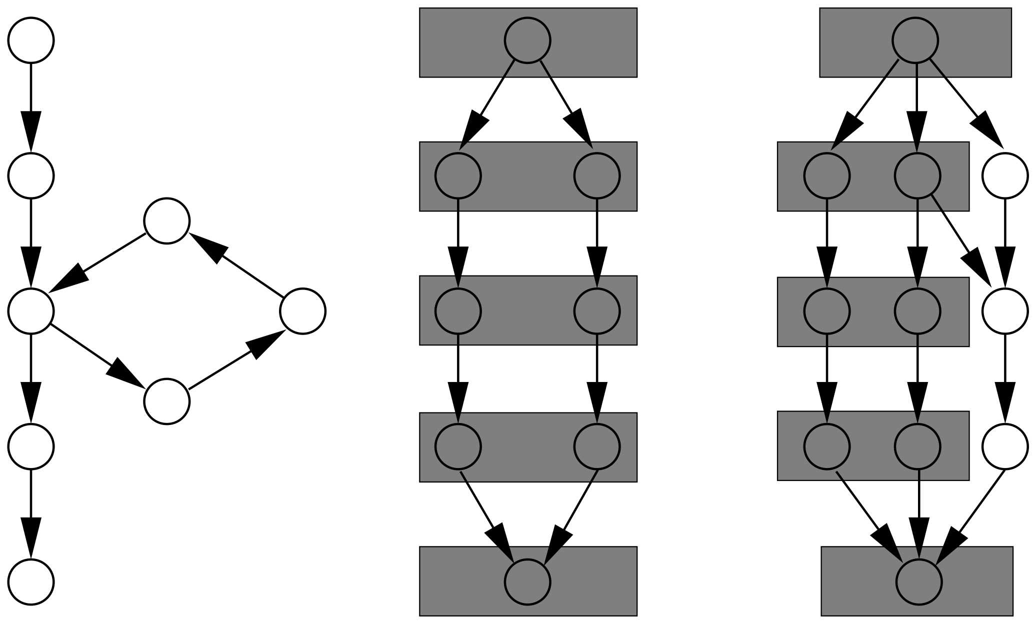


Figure 47: Cycles and transpositions

where b is the average branching factor of the search tree).

4.3.4.4 BULB eliminates all cycles and some transpositions

A cycle is a closed loop in the search space (see Figure 47a), while transpositions are distinct paths between a pair of nodes (see Figure 47b&c).[13] Cycles and transpositions are extremely costly if undetected in depth-first search. Each node (and the sub-tree below it) is searched by depth-first search as many times as there are paths to it. In a 2D grid for example, while the number of expansions grows quadratically with depth for breadth-first search, it grows exponentially for depth-first search [162]. Most implementations of depth-first search do not check for duplicates during node generation. Cycle detection, while easy to perform, consumes valuable runtime. Transpositions are typically extremely numerous and costly to detect both in terms of runtime and memory usage. This is why depth-first search is essentially used in tree-like domains [112] or within iterative deepening search strategies [96]. This discussion also applies to LDS.

Because it is an extension of beam search (which is itself a breadth-first search), BULB automatically eliminates cycles since it saves every expanded node in the hash table and never inserts

[13]A single cycle is thus part of an infinite number of transpositions.

the same state twice in it. On the other hand, BULB does not make any effort at eliminating transpositions. Nevertheless, BULB (like beam search) eliminates some transpositions as a by-product of its memorized beam. For example in Figure 47b, BULB eliminates one of the transpositions to the bottom node since they both fit completely in the beam of width two (shaded area). However in Figure 47c, regardless of how nodes are ordered at each level, BULB can only eliminate one of the two redundant paths to the bottom node.

4.4 Empirical evaluation

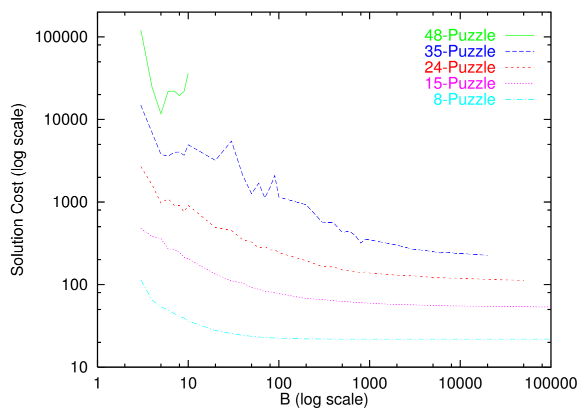
In this section, we present an empirical study of our variants of beam search in three benchmark domains: the N-Puzzle, the 4-peg Towers of Hanoi domain, and the Rubik's Cube domain. An overview of these domains and a description of our empirical setup appear in Chapter 3. All of our experiments in this chapter were performed on a Pentium-IV PC clocked at a 2.2 GHz with enough memory to store one or more million nodes (depending on the domain).

4.4.1 Empirical evaluation in the N-Puzzle domain

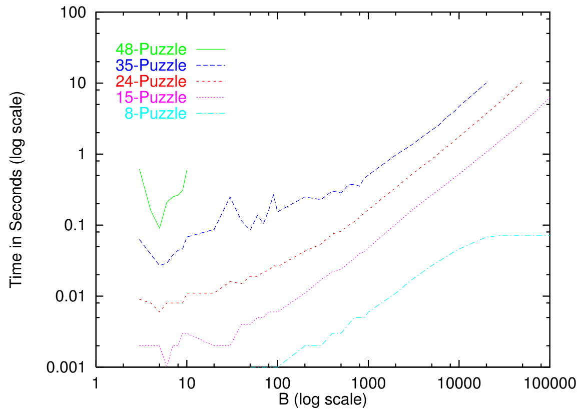
4.4.1.1 Evaluation of beam search in the N-Puzzle

Figure 48 shows the performance of beam search in the N-puzzle for various values of N and B. As B increases, the solution cost tends to decrease (although the trend is sometimes broken in the larger puzzles). For all puzzle sizes, the runtime and the number of nodes stored and generated are strongly correlated. All three measures tend to increase as B increases (again, exceptions to this trend are more numerous for large problem sizes). All these trends confirm our expectations based on the property that beam search approximates breadth-first search more and more closely as B increases. The bottom two graphs illustrate the fact that, in general, obtaining solutions of higher quality (i.e., lower cost) requires generating and storing more nodes. Of course, it also takes more time (not shown, since each runtime to solution cost trade-off curve is not significantly different from the corresponding curve in sub-figure f).

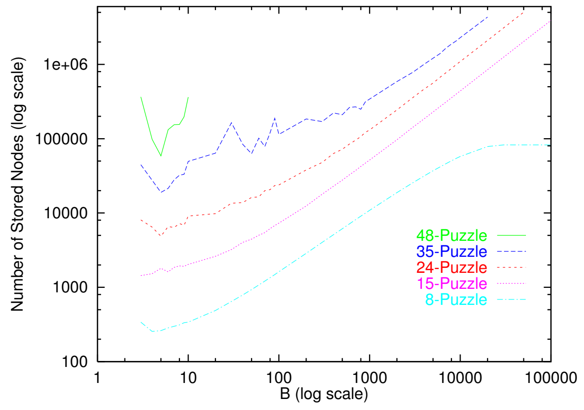
There are three situations in which beam search may terminate without a goal. First, if B is really small (e.g., $B = 1$ or 2 in the N-Puzzle), the beam becomes empty because one of the slices has no new successors. Solutions to this problem include increasing the value of B or finding a better heuristic function. Second, the shallowest goal may be so deep in the search tree that beam search



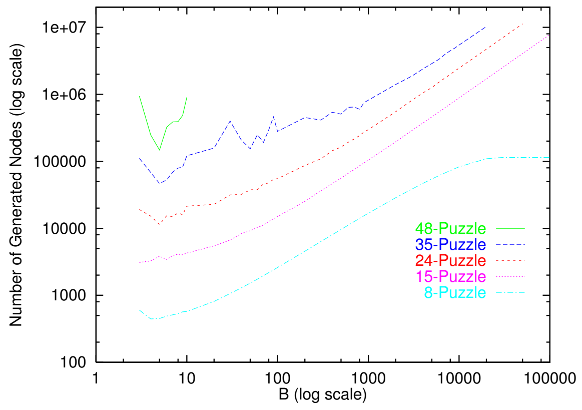
a) Solution cost versus B



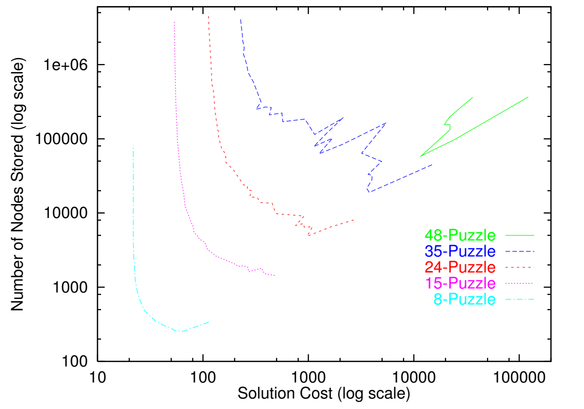
b) Runtime versus B



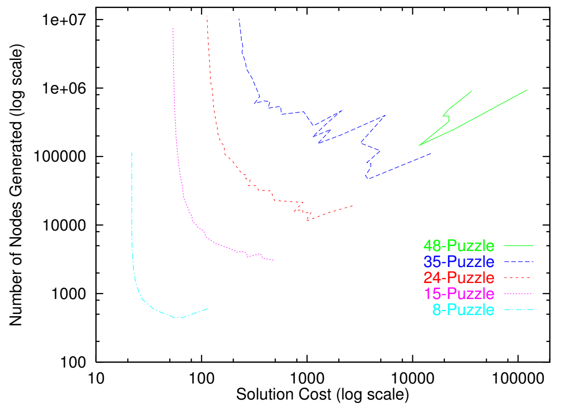
c) Memory usage versus B



d) Search effort versus B



e) Memory usage versus solution cost



f) Search effort versus solution cost

Figure 48: Performance of beam search in the N-Puzzle with varying B

always runs out of memory before reaching it (i.e., the total memory needed for all nodes in the beam down to the goal is larger than the available memory). The solution to this problem requires decreasing the value of B. Third, in the intermediate case, beam search runs out of memory at a given depth (say, d) because the heuristic function leads it astray. Since this situation assumes that there is a goal at level d (or higher in the tree), solutions to this problem include finding 1) a better heuristic function and 2) a memory-purging strategy that continues searching "against" the heuristic recommendation to find out where it is wrong.

The last two cases are failures due to memory shortages. Yet, they are qualitatively different from each other. One requires a new (smaller) value for B while the other does not. In keeping with standard practice, we assume that B is a constant and focus on the last case. Our approach to this problem in this chapter is to use backtracking. Unfortunately, chronological backtracking (implemented in DB) does not solve this problem in a reasonable amount of time. In our experiments, we have not found a problem instance 1) in which beam search fails solely because of the heuristic ordering of nodes and 2) that is solved by DB in a reasonable amount of time (on the order of minutes). This illustrates the weakness of chronological backtracking. Thus, we now turn to discrepancy-based backtracking.

4.4.1.2 Evaluation of BULB in the N-Puzzle

Our experiments show that beam search has an impressive scaling behavior in the N-puzzle (it can solve any random instance of the 48-puzzle in a fraction of a second). For this reason, our negative results with DB might be due to a ceiling effect (that is, there is not much room for improvement over beam search by *any* algorithm). We indeed believe this to be the case, at least for small values of N (up to 35). For example, beam search can solve all of our instances of the 35-Puzzle with B as large as 22,000. In our setup, the memory can contain up to 6 million nodes, thus the maximum searchable depth is about 273 ($= 6,000,000/22,000$). Given that, with these settings, the average (respectively, maximum) solution cost in our sample is 226 (respectively, 274), it is possible (and even likely) that any larger value of B will cause beam search to run out of memory on some instance because the goal is just too deep to reach even with a perfect heuristic function. If this conclusion really holds, then there is no hope of improving upon beam search in the 35-Puzzle by simply adding backtracking. As a further confirmation of this ceiling effect, our experiments with BULB (in addition to those with DB) have also not shown any significant improvement over beam search for $N = 35$ (or smaller).

In contrast, the failure of DB to improve on beam search in the 48-puzzle is not the result of a ceiling effect since BULB *can* significantly improve on beam search (see Figure 49). The graphs show the performance of both beam search and BULB in the 48-Puzzle with the same amount of memory (namely 6 million nodes). Since BULB generates nodes in exactly the same order as beam search, we plot the performance of both algorithms using a single curve in which BULB's curve simply extends the curve of beam search (BULB is complete with much larger B values than beam search). Since BULB is slightly slower than beam search in our recursive implementation of BULB, only the runtime data are plotted as two curves.

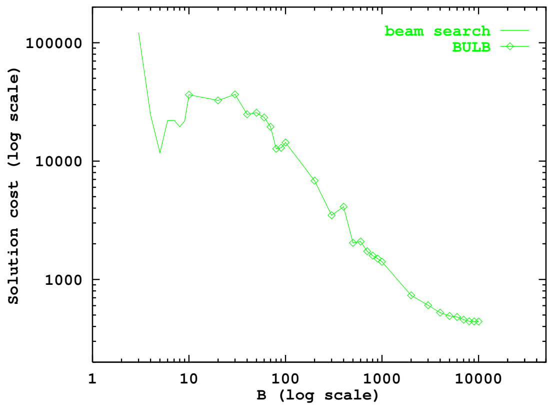
Our results show that, by increasing B to much larger values than beam search can handle in the 48-puzzle, BULB exhibits a 25-fold reduction in the solution cost (from 11,700 for beam search down to 440 for BULB) but still enjoys reasonable maximum average runtimes on the order of 30 seconds.

4.4.1.3 Comparison with variants of multi-state commitment search

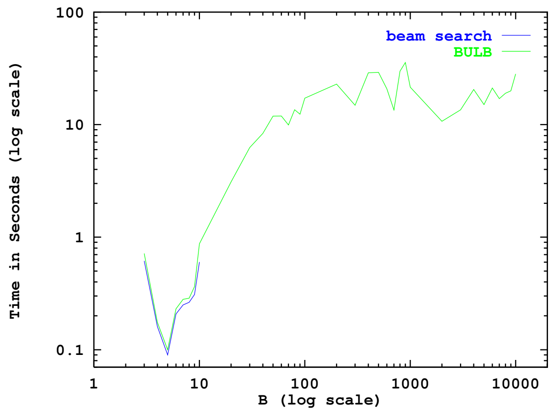
Figure 50 compares the performance of beam search and BULB with that of the best two algorithms in Chapter 3 in the 48-Puzzle. With the same amount of memory, BULB improves the solution quality of MSC-KWA* and MSC-KRTA* by about an order of magnitude while keeping the runtime reasonable (MSC-KWA* and MSC-KRTA* are variants of WA* and RTA*, respectively, whose behavior is similar to that of beam search).

4.4.1.4 BULB scales up to even larger puzzles

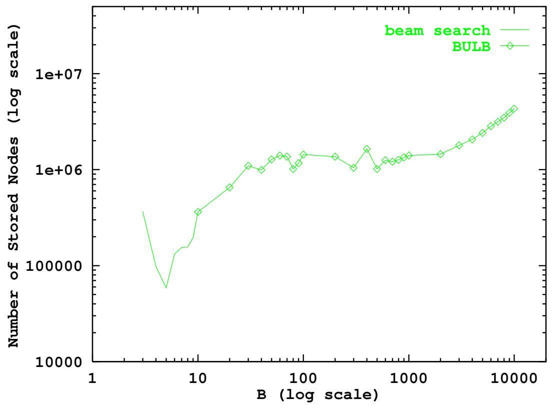
In order to provide additional support for the scaling behavior of BULB, we also report experimental results for larger puzzles (namely for $N = 63$ and $N = 80$). In our empirical setup, while beam search is not able to solve all fifty random instances in either puzzle size, BULB remains complete for a large set of B values. Figures 51 & 52 show the behavior of BULB in these larger N-puzzle domains. On average, BULB solves all instances of the 63- and 80-Puzzle as fast as in about 1 and 10 seconds, respectively. If more time is available to look for better solutions, larger values of



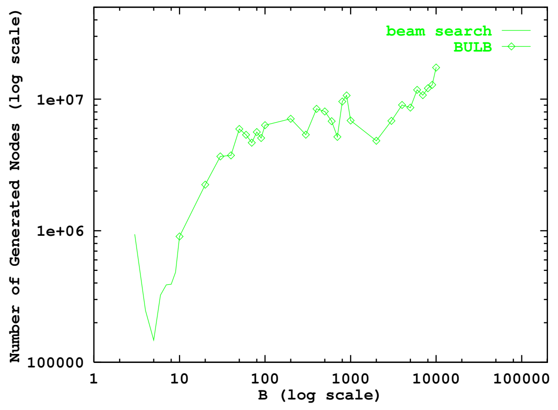
a) Solution cost versus B



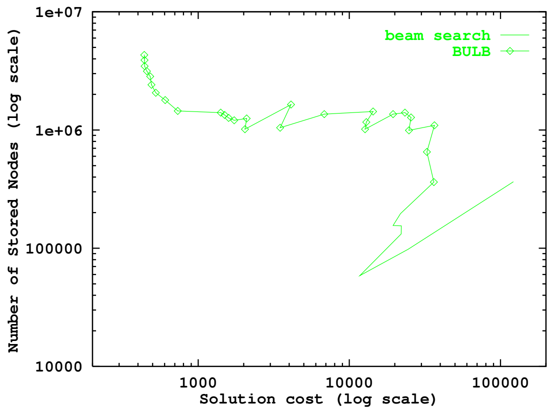
b) Runtime versus B



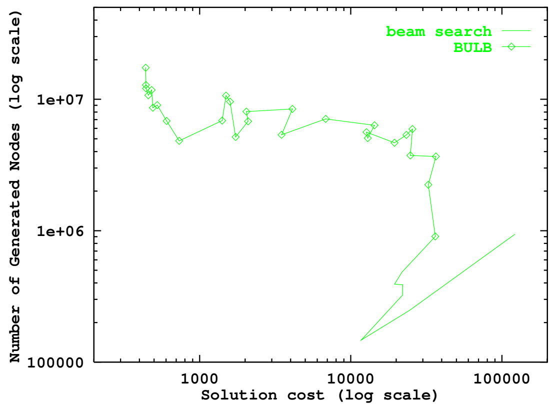
c) Memory usage versus B



d) Search effort versus B



e) Memory usage versus solution cost



f) Search effort versus solution cost

Figure 49: Performance of BULB in the 48-Puzzle with varying B

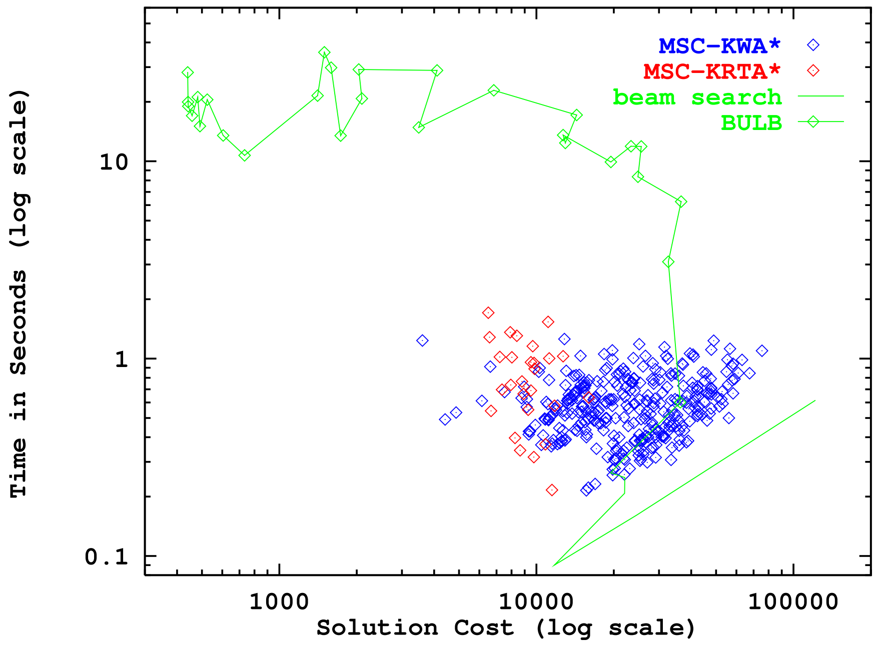


Figure 50: Comparing the performance of beam search and BULB with that of MSC-KWA* and MSC-KRTA* in the 48-Puzzle with varying B

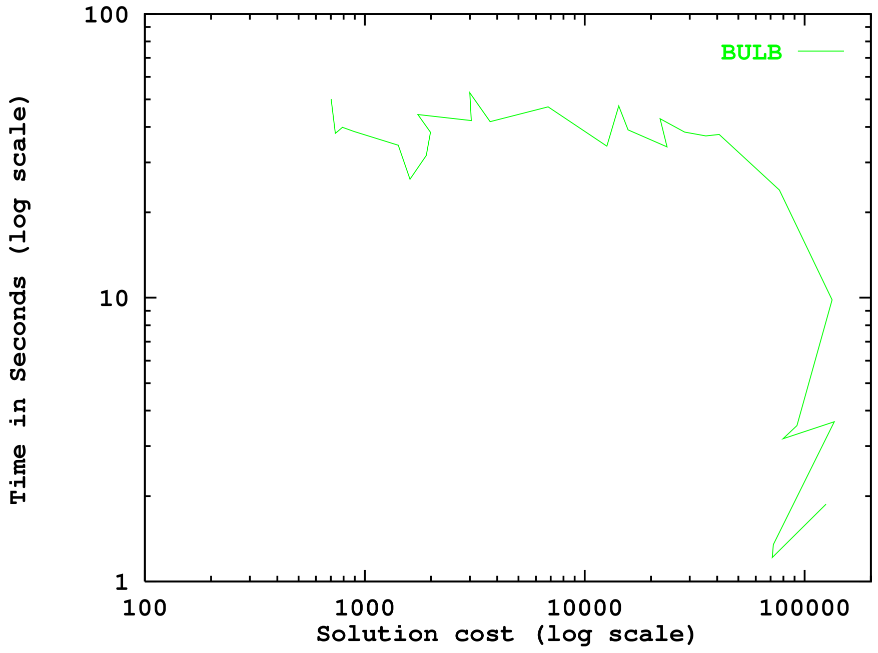


Figure 51: Performance of BULB in the 63-Puzzle with varying B (memory = 4 million nodes)

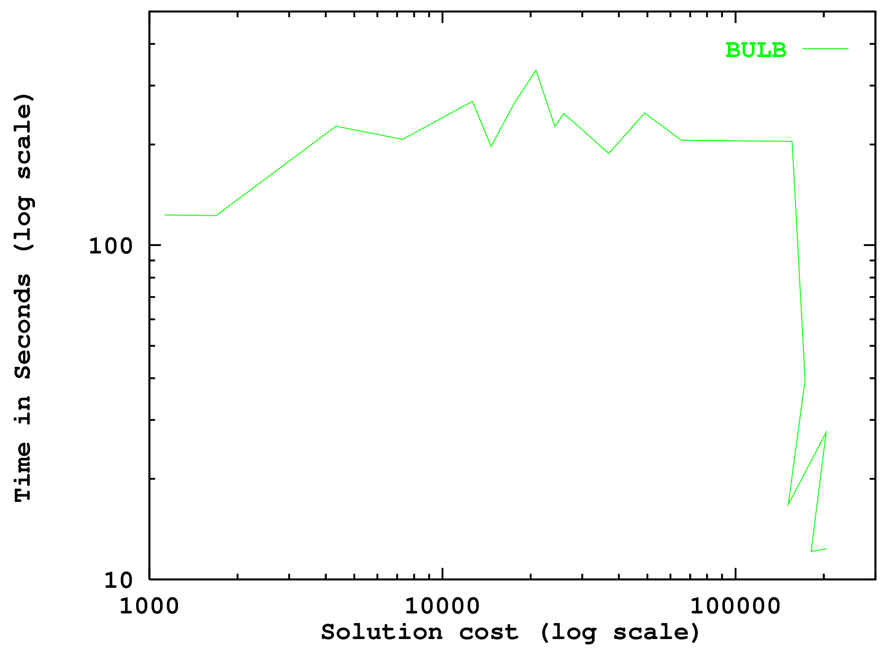


Figure 52: Performance of BULB in the 80-Puzzle with varying B (memory = 3 million nodes)

B (5,000 and 20,000, respectively) yield solutions that are on average (as a conservative estimate) no more than five times larger than optimal (lowest solution costs average about 700 and 1130, respectively). The average runtimes corresponding to such a high solution quality remain reasonable (approximately 50 and 120 seconds, respectively).

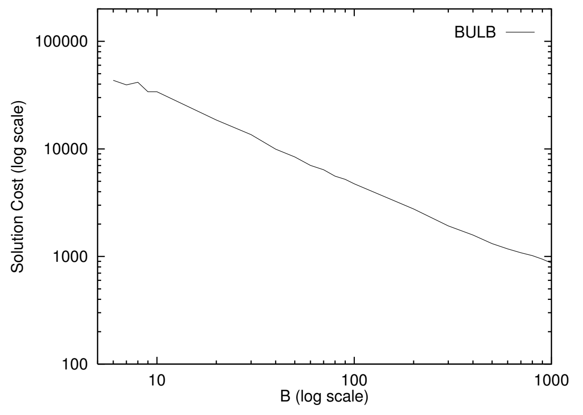
4.4.2 Empirical evaluation in the Towers of Hanoi domain

Our second benchmark domain is the 4-peg Towers of Hanoi domain (also called the Reve's Puzzle, see Section 3.6.2). Our experiments involve fifty random instances with 22 disks in which the goal state has all disks stacked up on the destination peg (or tower). In our empirical setup, the memory can store up to one million nodes. In this setup, none of our test values for B makes beam search complete. Interestingly, Table 19 shows a pattern reminiscent of the behavior of beam search in the 48-Puzzle, namely that beam search with large values of B solves a large proportion of instances (about 70 percent) with a high solution quality.

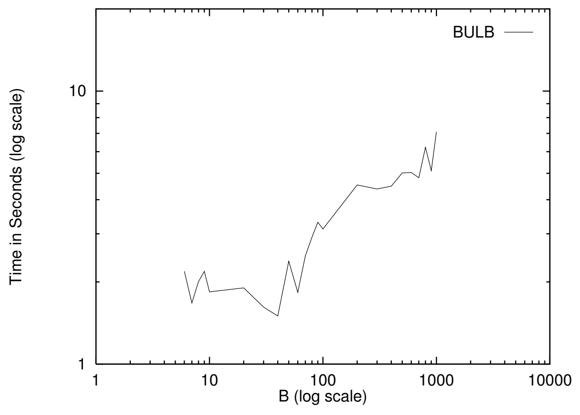
If its superior scaling behavior carries over to this domain, BULB should be able to solve the remaining 30 percent of instances in which beam search runs out of memory. Figure 53 shows that this is indeed the case. The shortest average runtime of BULB is about one and a half second. It is obtained for $B = 40$ and yields an average solution quality of about 10,000. If more time is

Table 19: Performance of beam search in the Towers of Hanoi domain (memory = 1 million nodes)

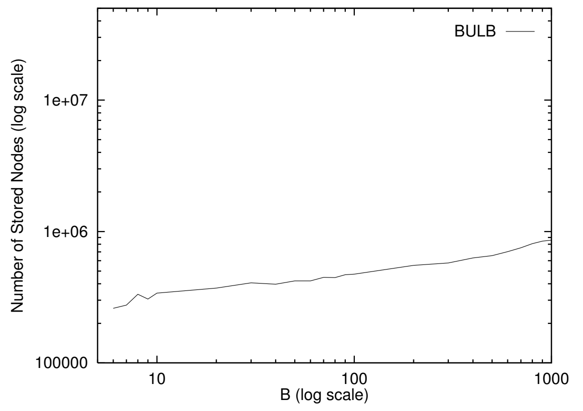
B	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
1	N/A	N/A	N/A	N/A	0
2	115,704.00	890,027	231,401	0.360	2
3	130,153.00	1,519,900	390,451	0.633	24
4	59,836.75	909,528	239,342	0.374	48
5	37,775.12	730,901	188,860	0.306	68
6	41,507.95	942,024	249,012	0.432	40
7	39,229.08	1,030,574	274,588	0.446	50
8	42,626.67	1,279,381	340,982	0.592	42
9	34,337.59	1,163,909	309,019	0.522	44
10	33,489.26	1,261,982	334,850	0.581	46
20	17,588.28	1,330,487	351,700	0.687	50
30	13,414.43	1,531,218	402,270	0.811	70
40	9,843.84	1,503,808	393,571	0.799	76
50	8,468.59	1,619,300	423,103	0.900	68
60	7,073.51	1,625,596	423,951	0.967	86
70	6,533.43	1,755,253	456,849	1.005	70
80	5,562.51	1,710,901	444,503	0.988	70
90	5,189.43	1,798,047	466,470	1.139	74
100	4,629.57	1,784,654	462,443	1.012	70
200	2,745.81	2,122,040	547,955	1.406	74
300	1,948.68	2,261,154	583,020	1.429	76
400	1,579.78	2,442,440	629,629	1.714	72
500	1,363.59	2,632,408	678,792	1.855	74
600	1,172.05	2,712,527	699,411	1.851	76
700	1,081.12	2,916,584	752,281	2.104	66
800	1,020.78	3,143,096	810,973	2.365	54
900	931.00	3,223,097	831,574	2.465	70
1,000	831.90	3,196,242	824,784	2.388	58
2,000	450.33	3,422,296	883,680	2.930	12
3,000	N/A	N/A	N/A	N/A	0



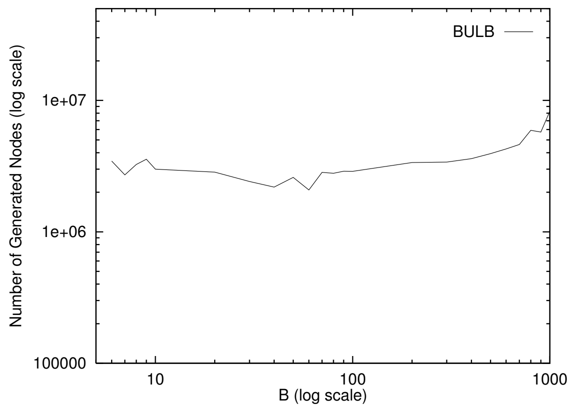
a) Solution cost versus B



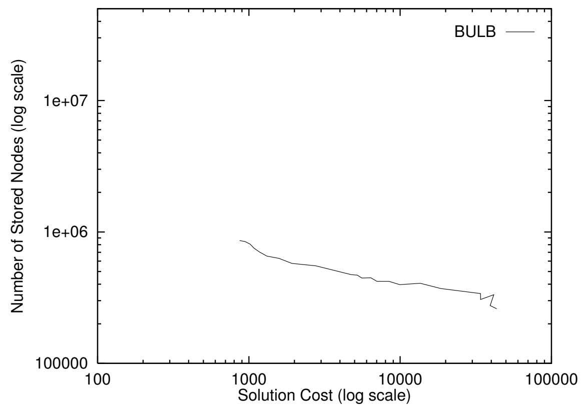
b) Runtime versus B



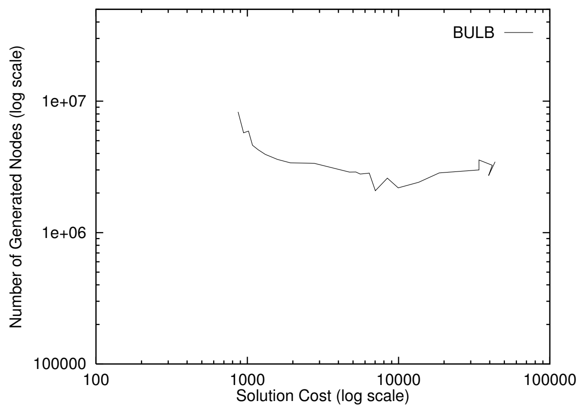
c) Memory usage versus B



d) Search effort versus B



e) Memory usage versus solution cost



f) Search effort versus solution cost

Figure 53: Performance of BULB in the Towers of Hanoi domain with varying B (memory = 1 million nodes)

available to look for better solutions, larger values of B (say, 1,000) yield solutions of significantly higher quality (namely, around 870) in very reasonable runtimes (about seven seconds).

4.4.3 Empirical evaluation in the Rubik's Cube domain

Our third benchmark domain is the Rubik's Cube. Our experiments involve fifty random instances in which the goal state is the original configuration of the cube (where each of the six faces is uniformly colored, see Section 3.6.3). In our empirical setup, where the memory can store up to one million nodes, several of our test values for B make beam search complete (see Table 20). The lowest average solution cost found by beam search is 55.18. This level of solution quality is similar to that found by a recent, powerful Rubik's Cube solver based on macro-operators, even though the latter uses both a larger number of pattern databases to build the macro-operators and a post-processing step on solution paths [63]. Therefore, beam search is a strong contender in this domain.

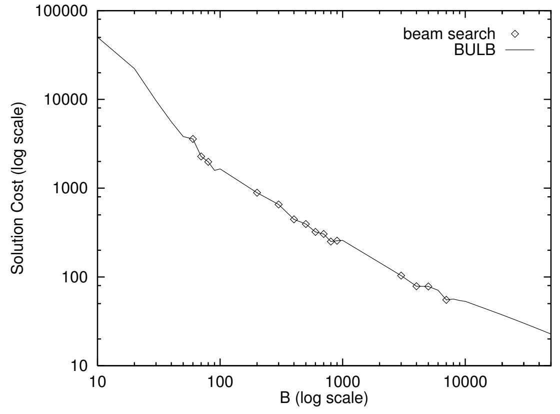
Nevertheless, BULB improves on beam search since it is complete in this domain over the whole range of our test values for B. Figure 54 shows that the lowest average solution cost (namely 22.74) found by BULB is reached in about seven minutes for $B = 50,000$. A slightly higher average solution cost of 25.78 is found by BULB in approximately two minutes for $B = 40,000$. Finally, the best average solution cost found by BULB in under a minute (namely about forty seconds) is 30.14 for $B = 30,000$ (We also ran BULB with $B = 30,000$ and averaged its performance over 10,000 random instances; Table 21 shows that averaging over this much larger set of instances yields similar performance.). This solution quality (obtained in less than a minute) is higher than that of the Rubik's Cube solver presented in [63]. Therefore, we believe that BULB is a state-of-the-art solver in this domain (in terms of the trade-off between solution quality and runtime) even though it is a pure-search (i.e., no pre- nor post-processing), domain-independent algorithm that uses relatively little memory (about 120 Mbytes in our setup, including about 86 Mbytes for the three pattern databases used by the heuristic function and about 32 Mbytes for the hash table).

Table 20: Performance of beam search in the Rubik's Cube domain (memory = 1 million nodes)

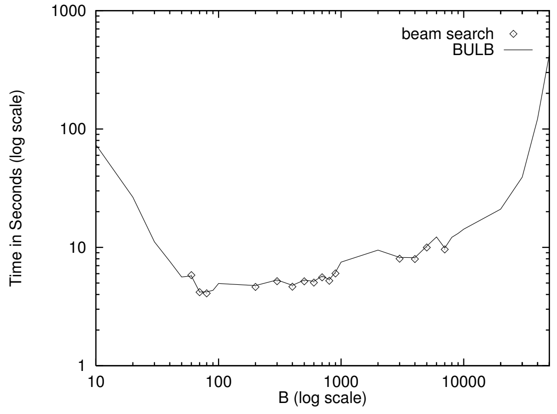
B	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
10	53,909.26	7,146,960	539,084	14.965	38
20	23,343.35	6,184,444	466,846	12.327	62
30	9,805.89	3,896,606	294,136	7.911	88
40	5,748.60	3,046,073	229,883	6.212	94
50	3,882.35	2,570,677	194,036	5.224	98
60	3,586.28	2,850,134	215,076	5.819	100
70	2,274.94	2,108,661	159,125	4.180	100
80	1,978.52	2,095,562	158,141	4.089	100
90	1,587.08	1,890,560	142,676	3.676	98
100	1,679.76	2,223,466	167,795	4.349	98
200	888.76	2,349,712	177,371	4.635	100
300	656.06	2,598,820	196,180	5.171	100
400	446.08	2,349,595	177,494	4.657	100
500	394.84	2,596,065	196,182	5.168	100
600	321.72	2,532,990	191,494	5.040	100
700	305.02	2,799,796	211,676	5.589	100
800	250.92	2,625,262	198,598	5.226	100
900	255.58	3,008,772	227,584	6.020	100
1,000	259.80	3,398,726	257,058	6.798	98
2,000	141.81	3,662,902	277,879	7.389	94
3,000	103.56	3,969,587	301,942	8.015	100
4,000	78.52	3,952,171	301,582	7.975	100
5,000	78.02	4,895,297	373,602	9.977	100
6,000	70.23	5,244,995	400,877	10.726	96
7,000	55.18	4,712,217	361,762	9.610	100
8,000	56.61	5,531,299	424,400	11.344	98
9,000	54.41	5,952,781	457,175	12.269	98
10,000	52.33	6,332,050	486,767	13.087	98
20,000	37.13	8,580,244	666,111	18.033	92
30,000	29.86	9,950,562	779,364	21.124	58
40,000	24.42	10,358,637	820,169	22.098	24
50,000	21.40	10,848,794	866,741	23.256	10
60,000	N/A	N/A	N/A	N/A	0

Table 21: Performance of BULB in the Rubik's Cube domain averaged over 1,000 random instances (memory = 1 million nodes)

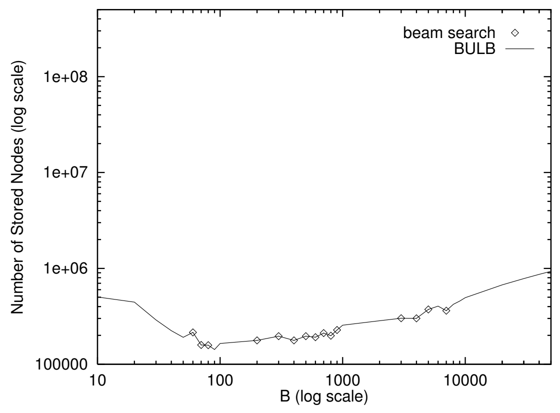
B	Solution Cost	Generated Nodes	Stored Nodes	Time (Seconds)	Percent Solved
30,000	30.58	18,029,740	797,275	39.636	100



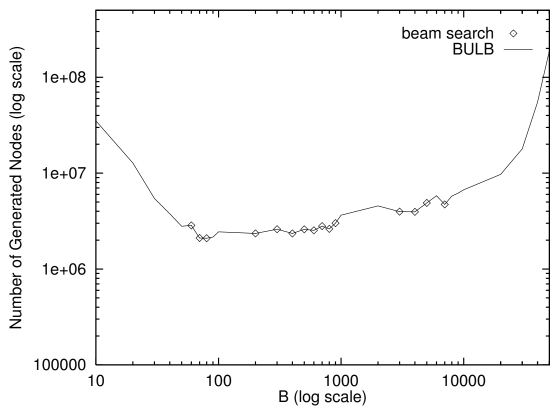
a) Solution cost versus B



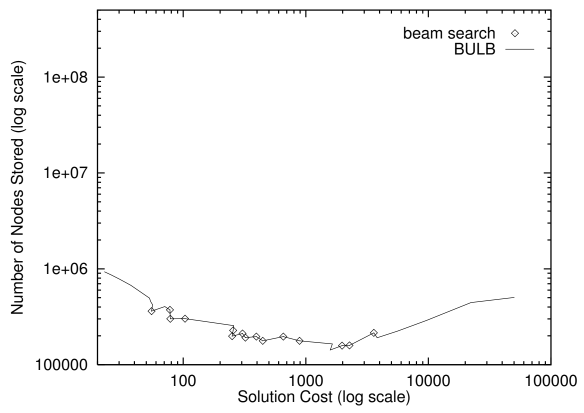
b) Runtime versus B



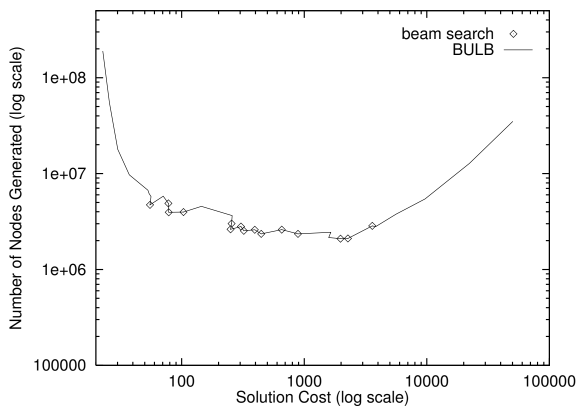
c) Memory usage versus B



d) Search effort versus B



e) Memory usage versus solution cost



f) Search effort versus solution cost

Figure 54: Performance of beam search and BULB in the Rubik's Cube with varying B (memory = 1 million nodes)

4.5 Related work

4.5.1 Band search

Besides beam search itself, the closest existing algorithm to BULB is band search [20]. Band search is a memory-bounded algorithm that maintains a bounded number of nodes (called a *band*) at each level of the search tree, expands these nodes in best-first order, and backtracks when a goal is found or the band is empty. We first describe the behavior of band search in more detail and then compare and contrast it with BULB.

Band search (denoted *BS(W)*) maintains a set (or band) of (at most) W active nodes at each level of the search tree. The W nodes are the ones with the lowest f-values at this level and are ordered according to increasing f-values. Additional nodes are stored in overflow lists, one at each level. These lists are also internally ordered by increasing f-values. Furthermore, the total set of active nodes (i.e., the union of the bands, but not the overflow lists, at all levels) is also ordered by increasing f-values. Finally, a counter is maintained at each level of the search tree to limit the degree of backtracking allowed. Each counter keeps track of how many nodes have been expanded at this level.

BS starts with the start state in the band at level 0 (and its counter initialized with a value of one) and all other lists empty (and all other counters with a value of zero). At each iteration, a node with the smallest f-value among those in all the bands is selected for expansion and removed from its band. Its successors are generated and processed as follows. If a successor's f-value is larger than or equal to the cost of the best solution found so far (initially equal to infinity), it is pruned. If a successor is a goal with a lower cost than the best found so far, the latter is updated (i.e., decreased). Each non-pruned successor is inserted into the band at the next level and its counter is incremented by one. If the band's size exceeds W, excess nodes with the largest f-values are moved into the overflow list at this level and the counter is reset to W. This process continues until the union of all the bands is empty. At this point, if all the overflow lists are also empty, the search terminates. Otherwise, the best W nodes in the deepest overflow list are inserted into the band at their level, all the counters are reset to zero, and the search restarts.

BS(W) is a generalization of both guided depth-first search (GDFS) and best-first search. When $W = 1$, only one node is active at each level, namely the best unexpanded one. In fact, since the expanded node is immediately removed from its band, there is only one non-empty band in the whole search tree at any time. Therefore, BS(1) searches greedily, that is depth-first on the best node. When the bands are all empty, the deepest overflow list is used to provide the next starting point. This enforces the chronological backtracking needed for depth-first search.

When $W = \infty$, all successor nodes are added to the band at the next level. The overflow lists remain empty. Since BS always chooses for expansion a node with the smallest f-value, BS(∞) is functionally equivalent to best-first search.

For values of $W < \infty$, BS can be seen as 1) a best-first version of beam search (which is a breadth-first search) or 2) a beam-search version of best-first search in which the beam width is uniform across levels. The main advantage of band search over best-first search is its bounded memory requirements. By construction, each band cannot contain more than the best W nodes. Furthermore, the overflow list contains the other successors of the nodes in the band at the next higher level. Thus, each level contains at most $W + W.(b - 1) = W.b$ nodes, where b is the maximum branching factor. Finally, if the maximum search depth is d, the worst-case memory requirements for BS(W) are $O(Wbd)$.

A related algorithm is Ibaraki's depth-m search [73] since it simulates best-first search with bounded memory. However, its memory bound is exponential (more precisely, on the order of $O(d^W)$, where d is the maximum search depth).

Similarities between BS and BULB

- Both algorithms are variants of beam search that have a memory bound that is polynomial in the bandwidth and the maximum search depth.

- Both algorithms reduce to variants of depth-first search when $W = 1$, namely to guided depth-first search for BS, and to limited discrepancy search for BULB .

- Both algorithms are anytime algorithms since they may keep searching for better and better solutions after the first one is found.

Differences between BS and BULB

- BS is a variant of best-first search while BULB is a variant of breadth-first search. Furthermore, BS expands one node at a time while BULB expands all nodes at a given level in parallel. In this respect, BULB is more diversified than BS (see Chapter 3 for a discussion of diversification).

- The memory consumption of BS is higher than that of BULB (by a constant factor of b) because BS stores in memory all successors of the bands at all levels, while BULB only stores the successors that are actually in the bands. The flip side of this feature is that BULB must re-generate the discarded successors upon backtracking while BS keeps them in the overflow lists.

- BS uses chronological backtracking when reaching the bottom of the tree while BULB uses limited discrepancy search. Since it is more likely that the heuristic function is misleading at the top of the tree, it seems more fruitful to backtrack there instead of at the bottom of the tree. Our experiments (with DB) have indeed confirmed this.

- BS is essentially designed to work on trees, like depth-first search. While it can be (and has been) applied to general graphs, it does not contain any loop-detection mechanism. Therefore, according to [20], BS "is efficient for solving problems that do not have deep search trees. [...] It should not be applied to problems [...] such as puzzle and maze problems." When applied to graphs such as the N-puzzle, it must be used in the context of iterative-deepening search. However in this case, BS can only be useful in the last iteration of IDA* and the improvement in the 15-puzzle (the largest tested puzzle domain) is small (if any) because the threshold on the last iteration is typically equal to the optimal cost and BS must expand all nodes reachable in the last iteration in best-first order.

- Implementing BS is significantly harder than for BULB because BS maintains two ordered lists at each level (one band and one overflow list). In addition, the set of band lists must also be threaded into a single list (akin to the OPEN list in A*) ordered by increasing f-values. In contrast, BULB maintains a single list (namely, the ordered list of nodes at the current level).

In [21], BS is extended in two ways. First, the bandwidth W is varied dynamically during search. It is decreased by one when a goal is found and increased by one when the search bottoms out without finding a goal. Second, BS is combined with GDFS: The new algorithm performs guided depth-first search to find a first-solution and then switches to BS to refine the first solution. However, these improvements do not overcome the limitations of BS in general graphs. They are only evaluated on known, fixed-depth trees (asymmetric TSP problems) and on maze problems whose search space fits in memory and could be quickly solved optimally by A* or IDA*.

4.5.2 Diversity beam search

Beam search relies on heuristic evaluations to order nodes at each level. When the heuristic function is misleading, beam search can be led astray. Both BULB and BS tackle this issue using backtracking. In contrast, [151] presents a variant of beam search (called *Diversity Beam Search* or DBS) that does not backtrack. Instead, it introduces diversity into the search at all levels of the tree in order to increase the quality of the solution found. Diversity is evaluated by an additional function that takes a set of states and ranks them according to how dissimilar they are. This evaluation is combined with the heuristic function to decide which nodes to keep at the next level. In [151], increased diversity is aimed at improving the solution quality. In our context, it would be used to increase the probability of finding a solution at all before memory runs out. One difficulty with DBS is that it needs an additional function for measuring dissimilarity. The paper applies the idea to the CRESUS expert system for intelligent cash management [150]. In general, a good dissimilarity measure is likely to be domain-dependent and/or hard to find. It should at least statistically beat a trivial approach such as adding to the beam randomly selected nodes with high heuristic evaluations. Another problem with DBS is that it does not provide for a backup mechanism when no solution is found. Since the dissimilarity measure is heuristic in nature, it is likely to lead the search astray in at least some instances. One possible solution would be to combine the idea of diversity with the backtracking mechanisms used in BULB or BS.

4.5.3 Complete anytime beam search

[173] takes a very general view of beam search as any search technique that uses pruning rules to discard non-promising alternatives, regardless of how nodes are ordered for expansion. While the traditional beam search is based on breadth-first search [7, 43, 170, 151] (including this work), other variants are based on best-first search [20, 21]. [173] applies the term to a variant of depth-first search. The idea behind Complete Anytime Beam search (CABS) is to repeatedly call a variant of

depth-first search that uses a pruning rule that is progressively weakened at each iteration. One main contribution of the paper is a new domain-independent pruning rule whose strength is controlled by a single parameter. As the pruning rule is weakened, fewer and fewer nodes are pruned. In the worst case, no pruning occurs and thus CABS is complete. The idea of iterative weakening was introduced in [134]. The particular pruning rule introduced in the paper is similar but different from the idea of iterative broadening [54]. CABS is shown to be both an optimization algorithm and an efficient approximation algorithm for the maximum boolean satisfiability problem as well as the symmetric and asymmetric TSP. Unfortunately, CABS, like most variants of depth-first search, is only well suited to problems with a high density of solutions or a finite search tree [135].

4.5.4 Variants of discrepancy search

One drawback of LDS [61] is that each iteration allows *at most* a given number n of discrepancies. This means that each iteration re-visits all the full paths generated during all previous iterations. [100] proposes ILDS, an improved version of LDS, that only generates paths with *exactly* n discrepancies, with n incremented by one at each iteration.[14] Unfortunately, ILDS needs one additional parameter to behave as described, namely the maximum search depth. This is not always known with precision. To remedy this problem, [169] proposes another variant of LDS, called DDS for *depth-bounded discrepancy search*, that also never re-generates full paths but does not need an upper bound on the search depth. DDS combines the ideas of limited discrepancy search and iterative deepening. At each iteration, DDS does not allow discrepancies below a certain level in the tree, that is: the 0^{th} iteration allows no discrepancy, the first iteration does not allow discrepancies below depth 0, the second iteration does not allow discrepancies below depth 1, etc. With an easy additional mechanism, DDS avoids re-visiting full paths generated during previous iterations.

Conceptually, the evolution from depth-first search, to LDS, to ILDS, and finally to DDS, is best characterized by the way discrepancies are weighted at different search depths.[15]

- Depth-first search weighs discrepancies more at the top of the tree. Thus it goes against the heuristic evaluations (a discrepancy) deep in the tree before it does so toward the top of the

[14]This property only holds when the tree is balanced and its depth is known exactly.

[15]Since a discrepancy is a move *against* the heuristic recommendation, a small-weight discrepancy is preferred over a large-weight one.

tree, resulting in chronological backtracking.

- LDS weighs discrepancies equally at all levels of the tree. Its search strategy is based on the (increasing) number of discrepancies, not their depth in the tree.
- Like LDS, ILDS searches iteratively in order of increasing numbers of discrepancies. But, like depth-first search, it weighs discrepancies more at the top of the tree.
- DDS (in complete opposition to depth-first search and ILDS) weighs discrepancies in proportion to their depth in the tree. Therefore, it goes against the heuristic evaluations at the top of the tree before it does so toward the bottom of the tree. This makes sense when the heuristic function is less accurate at the top of the tree, that is away from the goal.

We have not yet applied DDS in the context of beam search, mainly because in our experiments, LDS is sufficient to show significant improvement over depth-first search. Nevertheless, since the DDS strategy may be even more efficient, its application to beam search is part of our plans for future work. However, we do not plan to implement ILDS since it requires the additional knowledge of an upper bound on the search depth.

Finally, [122] proposes Interleaved Depth-First Search (IDFS). While IDFS is not strictly-speaking a variant of limited discrepancy search, IDFS shares with DDS the property that it weighs deeper discrepancies more heavily. However, instead of limiting the number of discrepancies, IDFS performs parallel depth-first searches on a set of sub-trees (called *active* sub-trees). Note that the parallelism is simulated on a single processor, hence the name *Interleaved* DFS. Since pure IDFS (with full parallelism) has an exponential space complexity, [122] also introduces a linear-space version of IDFS called *Limited IDFS* that only performs limited parallelism. Parallelism only takes place at a limited number of levels (other levels are searched depth-first) and within each parallel level, only a limited number of nodes are searched in parallel (i.e., the number of active sub-trees is bounded). Limited IDFS, like DDS, exhibits significant runtime improvements over depth-first search on hard problems with inaccurate heuristic functions [123]. It is not clear how an application of IDFS to graphs would scale up. Furthermore, IDFS requires the setting of several parameters, namely the number and depths of parallel levels (these need not be consecutive levels), and the amount of parallelism (i.e., number of active sub-trees) within each parallel level, which may be different at every level. We leave an empirical comparison of BULB and IDFS for future work.

4.5.5 Divide-and-conquer beam search

[180] introduces breadth-first heuristic search (BFHS). BFHS is a variant of breadth-first search with two additional features. First, it uses an upper bound on the cost of an optimal solution as well as a heuristic function in order to prune nodes whose f-value is larger than the upper bound. Second, it uses a divide-and-conquer strategy to re-construct the solution path when the goal is found. This allows BFHS to keep in memory only a limited number of layers of closed (i.e., already expanded) nodes while guaranteeing that all duplicate nodes are eliminated (just like in breadth-first search). The motivation for BFHS is to scale up admissible search to larger domains by eliminating the need to store some of the closed nodes. The main contribution of the paper is to demonstrate that, when using a divide-and-conquer strategy, breadth-first is more space-efficient than best-first search.

The main connection between this paper and our work is the common realization that breadth-first search is more efficient than best-first search (e.g., WA*), not only in terms of space complexity (when used in the context of beam search at least) but also in terms of solution quality. [180] uses "distribut[ion of] the search effort" to refer to what we call "diversity."

The other direct connection results from the fact that BFHS is not memory-bounded since it uses as much space as needed to store enough of the layers in order to avoid node duplication. For even better scaling (at the expense of solution quality), [180] proposes a variant of BFHS based on beam search that simply keeps the size of each layer within a given bound (namely B). This variant is called Divide-and-conquer beam search. It only differs from beam search in that it purges some layers from memory (and applies some additional pruning rules that use the upper bound). The divide-and-conquer approach for reducing memory consumption is orthogonal to the backtracking strategy. Both can be combined for better scaling behavior.

Finally, the size of the problems that BFHS can handle depends on how many nodes can be purged from memory, which in turns depends (in part) on how good the upper bound is.[16] BULB

[16]BFHS can also be used without an existing upper bound (by automatically increasing upper bounds in an iterative-deepening fashion similar to IDA*'s. However, running multiple iterations takes time and this variant is slower than basic BFHS.

could therefore be used to provide a good upper bound as input to BFHS.

4.6 Future work

In this section, we list several research directions for possible future work on BULB.

- As discussed in Section 4.5.4, a variant of BULB using DDS as the backtracking strategy (instead of LDS) may be superior to the original version of BULB in terms of runtime. This remains an open empirical question to be settled by future work.

- A possible avenue of research is the use of different weighing schemes for discrepancies. In general graphs (with branching factors larger than two), the definition of a discrepancy needs to be adapted since the original definition assumes that the search space is a binary tree. In this chapter, we have assumed that all but the best successors of a slice are discrepancies with essentially equal weights since we explore them all (in order of increasing h-values) before moving to the next level down the tree. Alternatively, discrepancies with the largest h-values could be weighed more heavily or even pruned altogether. This scheme, which places more confidence in the heuristic values, could be used at each level of the tree, or just in the lower part of the tree. The behavior of such variations of BULB is likely to vary significantly with the level of informedness of the heuristic values.

- One way to improve the runtime behavior of BULB is to limit its backtracking. For example, it is possible to do so at the top of the tree by proceeding in breadth-first manner (down to a level that depends on the amount of available memory and the average branching factor of the search space). One advantage of only performing beam search below this level is that no backtracking is performed higher up in the tree. Interestingly, this strategy is a special case of a family of algorithms in which the beam width varies during search. While, in this chapter, we have followed the common practice of keeping B constant, the discovery of efficient strategies for varying B looks like a promising direction for future work.

- In this chapter, we have focused on making beam search complete using backtracking. In the process, the memory consumption is directly controlled by the value of B (and the maximum depth that one is willing to explore). In contrast, the runtime is a function of both B

and domain-dependent characteristics (including the informedness of the heuristic function). Our experiments have treated the runtime as a dependent variable. While we have observed reasonable runtimes in our benchmark domains, there are situations in which controlling runtimes is crucial. In particular, there may exist an upper bound on the amount of time available for problem solving beyond which the solution becomes useless. When this deadline is soft (or when it is not known in advance), anytime algorithms are very handy since they can be interrupted at any time (hence their name, see [11, 10]) and return the best solution found so far. Transforming BULB into an anytime algorithm is easy since we only need to let it keep searching after it finds the first solution and until time runs out (like in [57] for WA*). Furthermore, such an anytime variant of BULB is admissible: since BULB searches the complete search tree down to $MAXD$, it eventually finds an optimal solution. Interesting future work includes the design of more efficient anytime variants of BULB.

4.7 Conclusion

One main advantage of beam search is that large B values yield solutions of high quality. One main drawback of beam search is that its memory consumption is proportional to the value of B. Therefore, there exists a trade-off between solution quality and completeness.

This research capitalizes on the good scaling behavior of beam search. Even in large domains with simple heuristic functions, beam search can find solutions of good quality in a high percentage of instances. In this chapter, we have addressed the following question: How to make beam search complete on the set of hard instances without sacrificing solution quality (by lowering B)? Our focus has been on backtracking.

The main contribution of this research is to show that introducing backtracking based on limited discrepancy into beam search scales it up to much larger domains (for a given level of solution quality). The resulting algorithm, called BULB (for Beam search Using Limited discrepancy Backtracking) can solve large instances of our benchmark domains in a matter of seconds and with average solution costs that are within a small multiplicative factor of the average optimal cost.

BULB is a complete, memory-bounded variant of beam search. By varying the value of B, BULB generalizes both limited-discrepancy search (for $B = 1$) and breadth-first search (for $B =$

∞). Furthermore, BULB can easily be transformed into an anytime, admissible algorithm (for example, by simply not stopping it when a goal is found). Future work includes the study of different backtracking strategies, of schedules for varying B, and of other efficient anytime extensions.

CHAPTER V

ANYTIME HEURISTIC SEARCH

5.1 Introduction

As artificial intelligence tools keep evolving toward greater usability, the paradigm of flexible computation has gained popularity [71, 27, 184, 72, 65, 176]. Flexible computational tools are able to adapt their runtime to different (and possibly unknown) temporal constraints on task completion. For example, in some variants of competitive chess, the maximum time interval between moves is fixed. Deliberation must end within this interval. Any move chosen after the clock has run out is useless. This is an example of time-dependent planning in which the output of the planning agent is only useful before the deadline. Furthermore, embedded agents have to deal with other agents as well as a dynamic environment. Thus in the real world, the deadlines for deliberation are often situation-dependent and even unknown at the outset of the task. Anytime algorithms are specifically targeted for such tasks. Anytime algorithms are a special class of flexible computational tools that trade off runtime for solution cost[1] [27, 11, 10]. They find a first solution fast and then take advantage of the remaining time (if any) to improve the solution quality. This anytime behavior stands in contrast to that of non-flexible algorithms (such as A*) that return no solution until an optimal one is found.

A well-known class of anytime algorithms is the class of local search algorithms [1]. Local search navigates the space of solutions. After quickly building a starting solution, it repeatedly searches for similar solutions of a higher quality. The search is local because each solution is a neighbor of the previous one. Thus local search requires the definition of a neighborhood structure in the solution space. The behavior of local search can be described as a gradient descent (or hill climbing depending on whether the optimization problem is one of minimization or maximization) on this surface. Once a local optimum is reached, local search is restarted at another point in the

[1]In this work, the solution quality is commensurate with the inverse of the solution or path cost.

solution space and the search proceeds iteratively until time runs out.

Although local search has been extremely popular both in the Operations Research (OR) and Artificial Intelligence (AI) communities, it is not a common approach to the shortest-path problem. One reason is that, unlike in typical optimization problems (such as the Traveling Salesperson Problem or TSP [111]), finding any solution to a shortest-path problem is hard since it requires search. Second, the solution space contains structured elements (namely paths, not states like in standard heuristic search). As a result, it is neither trivial to define a good neighborhood structure on paths (namely one that induces a surface with few local optima) nor is it computationally efficient to search it. Third, since building a starting solution (any solution, regardless of its quality) to the shortest-path problem is computationally expensive, it is an open issue how to efficiently identify additional restarting solutions to get the search agent out of local optima.

In this chapter, we propose a new anytime heuristic search algorithm called ABULB (for Anytime BULB). It is a local search algorithm in the space of solution paths that addresses the three aforementioned issues. In the previous chapter, we showed that BULB efficiently solves large problems. ABULB uses BULB to generate an initial solution. Section 5.2 describes the neighborhood structure imposed on the solution space as well as a new local search algorithm (called ITSA*) that searches the resulting surface. ITSA* has two variants depending on whether each iteration allows for only one or multiple steps on the surface. Section 5.3 describes two ways of transforming BULB into an anytime heuristic search algorithm that efficiently generates restarting solutions. The main difference between these two variants of ABULB is whether the beam width B varies during search. Section 5.3.5 evaluates ABULB in two standard benchmark domains. Section 5.4 discusses related and future work. Section 5.5 concludes by summarizing the contributions of this work.

5.2 ITSA*: Application of local search to the shortest-path problem

In this section, we first motivate our interest in local (or neighborhood) search.[2] We then describe a neighborhood structure on the space of solution paths. Finally, we introduce a new local optimization algorithm (called ITSA*) that takes advantage of this neighborhood structure to greedily explore the optimization surface.

[2]These methods are also called meta-heuristics (e.g., [166, 139]).

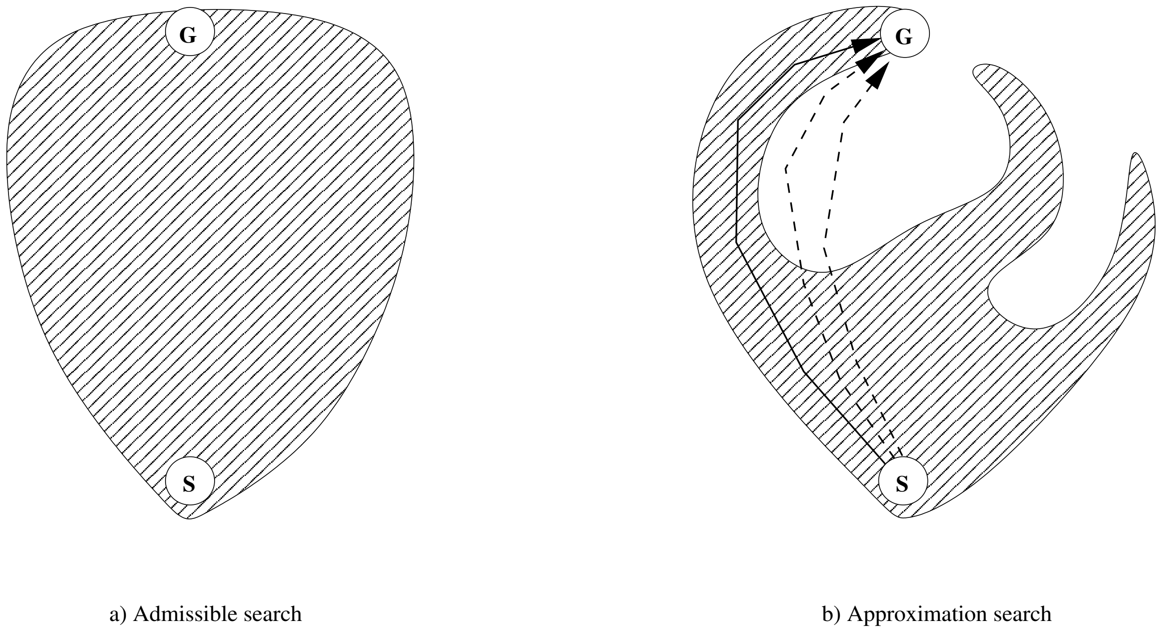


Figure 55: Approximation algorithms explore the search space in a less regular way than admissible algorithms.

5.2.1 Motivation

Approximation algorithms sacrifice solution quality in order to reduce their runtime. Greedy algorithms (such as WA*) rely heavily on the heuristic values by weighing them more than the g-values in the computation of the f-values. Other approximation algorithms (such as beam search) rely even more on the heuristic function by pruning nodes with high h-values. In both cases, the reliance on the h-values gives the search a strong depth-first component. While depth-first search with perfect information leads straight to the goal, it likely leads the search astray when combined with imperfect heuristic values. Nevertheless, both WA* and beam search are often able to use the heuristic information to eventually reach the goal. These two characteristics of approximation algorithms together imply that the solution path they find is often convoluted. They explore the search space less systematically than admissible algorithms: Their search frontier grows in a jagged fashion (see Figure 55b), as opposed to the regular exploration by admissible algorithms such as breadth-first search and A* (see Figure 55a).

Because of the jagged shape of their search frontier, it is possible that approximation algorithms miss shorter solutions that are actually close (in the space of solution paths) to the area of the search space that they have explored (see Figure 55b). This possibility is confirmed empirically. For

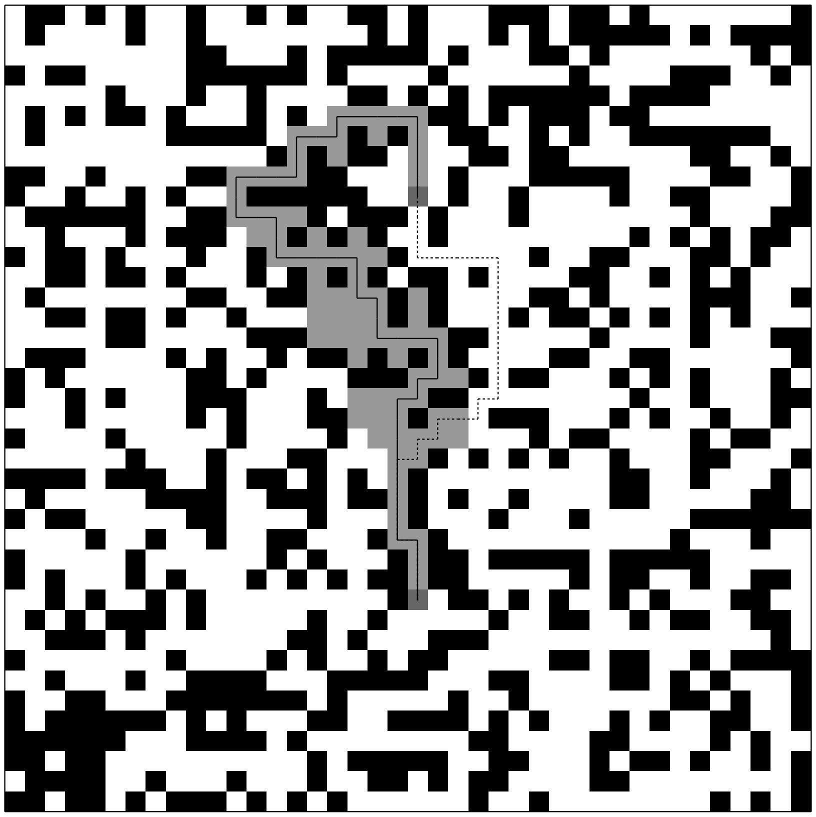


Figure 56: Solutions found (unbroken line) and missed (dashed line) by WA* with $f = g + 5 \times h$ in a gridworld problem.

example, Figure 56 depicts a concrete example of this situation for WA* in the gridworld domain, whose two-dimensional structure enables a direct visualization of its search frontier. Obstacles are black, expanded states are light gray, and the start and goal states are dark gray (with the goal on top). The figure illustrates how the convoluted path found is longer (its cost is equal to fifty steps) than but close to the optimal path (with a cost of thirty steps). Such patterns are also common for beam search (and thus also BULB) as our empirical evaluation in two benchmark domains demonstrates (see Section 5.2.4).

This behavior of approximation algorithms leads us to consider the following approach. In order to avoid leaving some interspersed regions of the state space unexplored, we propose to focus the search around a previously identified solution and to systematically explore its neighborhood. More precisely, we envision a staged search. Prior to invoking the local search, we use an approximation algorithm (such as BULB) to construct a sub-optimal solution. Then memory is purged except for the solution path. A stage consists of the systematic exploration of the state space in expanding concentric regions centered on the solution. When memory runs out, it is completely purged (though the newly found best path is retained) and the stage ends. Successive stages are executed until the quality of the evolving solution does not improve any more (or any other termination condition is satisfied). When two successive stages return the same solution path, the search has reached a (possibly local) minimum on the surface defined by the neighborhood relation and the cost function. The next sub-section formally defines the neighborhood structure.

5.2.2 A neighborhood structure based on path proximity

Our overall approach to anytime heuristic search is to explore systematically the neighborhood of a given solution path P, find a minimum-cost path P' in this neighborhood, then compute and search the neighborhood centered around P', and repeat the process until a local minimum is found. The size of each neighborhood is approximately equal to the maximum number of nodes that simultaneously fit in memory. Each neighborhood is built incrementally by including all nodes that can be reached in one step from any node already in the neighborhood, starting with the initial neighborhood containing the set of nodes in the current solution path. The analogy we use to describe the process of building a neighborhood is that of *iterative tunneling*: If the solution path is seen

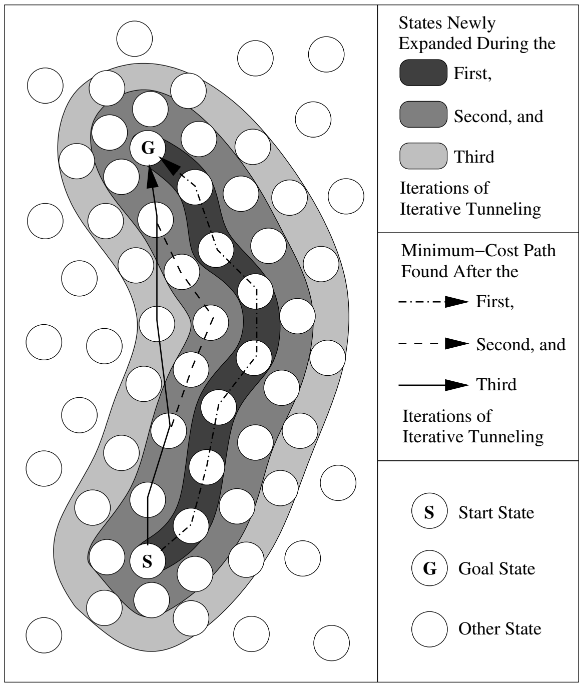

States Newly Expanded During the
First,
Second, and
Third
Iterations of Iterative Tunneling
Minimum−Cost Path Found After the
First,
Second, and
Third
Iterations of Iterative Tunneling
S
Start State
G
Goal State
Other State

Figure 57: Iterative tunneling defines the neighborhood of a path.

as a narrow tunnel (dug into the search space) from start S to goal G, then each iteration enlarges the diameter of the tunnel until there is no available space to heap any additional evacuated debris, assuming memory is viewed as a container for debris/nodes (see Figure 57).

Formally, let (V, E) denote the graph defined by a finite set V of vertices (or nodes) and a finite set E of directed edges $e = (n_1, n_2)$ between pairs of vertices $n_1, n_2 \in V$. S and G are two distinct vertices called the *start* and *goal* vertices, respectively. Let $\mathcal{P}$ denote the set of paths from S to G in the graph. The elements of $\mathcal{P}$ are the states of our neighborhood search. First, we define the distance metric $vd : V \times V \to \mathbb{N} \cup \{\infty\}$ between vertices in V that returns, for each pair of vertices (n_1, n_2), the smallest number $vd(n_1, n_2)$ of edges in any path from n_1 to n_2 in the graph. Second, we define the distance metric $pd : \mathcal{P} \times \mathcal{P} \to \mathbb{N}$ between paths in $\mathcal{P}$ that returns, for each pair of paths (P_1, P_2), the value $\max_{n_2 \in P_2}\{\min_{n_1 \in P_1} vd(n_1, n_2)\}$. pd is commensurate with the "distance" between two paths. Of course it holds that $\forall P \in \mathcal{P}$: $pd(P, P) = 0$. Third, given a path $P \in \mathcal{P}$ and $i \in \mathbb{N}$, let $T(P, i) = \{P' \in \mathcal{P} | pd(P, P') \leq i\}$. Then, the neighborhood $\mathcal{N}(P, M)$ of solution path P given the available memory M is the set $T(P, i)$ such that $T(P, i)$ fits into memory and $\forall i' \in \mathbb{N}$: $i' > i$ implies $T(P, i')$ does not fit into memory.

5.2.3 The ITSA* algorithm

Having defined the neighborhood of a path, we now turn to the issue of how to search it. The task is to find the shortest path from start to goal through states in the neighborhood. We need to answer two questions, namely how to identify such states and how to find the shortest path completely contained in the region of the state space that they occupy. Once the states in the neighborhood are identified, there is a simple answer to the second question, namely to use (a simple variant of) A*.

The identification of the states in the neighborhood of a path P is performed in a way similar to the way a ball is inflated, namely from the center out. The neighborhood is to P as the ball is to its center. As the interior of the ball is the union of concentric spheres of increasing radii around the center (starting with a radius of zero for the center point itself), the states in the neighborhood distribute themselves in layers of increasing distances (i.e., number of edges) from P (starting with a distance of 0, namely the states in P itself). In computational terms, a simple extension of breadth-first search suffices to identify the states layer by layer.

There is therefore a simple two-stage process for finding the shortest-path in the neighborhood of P, namely 1) to generate and store in memory the neighborhood states using breadth-first search applied in parallel to all states in P (this search stops when memory is full), and 2) to use A* to find the shortest path through stored states only. However, this approach has two drawbacks. First, it performs two full explorations of the neighborhood (once with breadth-first search, once with A*) before returning a solution. Second, because A* delays the expansion of states with large f-values, some states in the neighborhood may never be visited by A* and therefore need not be stored in memory. Unfortunately, these states cannot be identified during the first, breadth-first search stage. They occupy memory that might be used by A* for one (or more) extra layer(s).

To address both problems, we propose to interleave the building of the neighborhood with its search by A* in order to take advantage of the pruning power of f-values during construction and thus make the neighborhood as large as possible. We call the resulting algorithm ITSA* (for Iterative Tunneling Search with A*, and pronounced *It's-a-star*). ITSA* iteratively performs modified A* searches from the start to the goal. ITSA* assigns each generated state an iteration (or layer) number. This number corresponds to the distance of the state from (a state in) the path P. Once

Table 22: Performance of one-step ITSA* on paths found by BULB in the 48-Puzzle (with 6 million nodes in memory)

B	BULB		BULB + one-step ITSA*					
	time (seconds)	cost	time (seconds)			cost		
			value	increase over BULB		value	decrease over BULB	
				absolute	relative		absolute	relative
5	0.1	11,737	5.7	5.6	5,600%	3,140	8,597	73%
10	0.9	36,282	6.7	5.8	644%	3,233	33,049	91%
100	6.1	14,354	12.2	6.1	100%	2,052	12,302	86%
1,000	7.3	1,409	12.8	5.5	75%	746	663	47%
10,000	21.7	440	27.7	6.0	28%	428	12	3%

assigned (at the time of the state's generation), this number never changes. Before the call to ITSA*, all states in P are stored in memory with a layer number equal to zero. Then A* is run repeatedly with an increasing iteration number (starting at one) until memory runs out. At the beginning of each iteration, the OPEN list is initialized to the start state. A* proceeds as usual except that 1) it only inserts into the OPEN list states whose layer number is less than the iteration number, and 2) each newly generated state is assigned a layer number equal to one plus that of its parent in the search tree. Each iteration ends when A* is about to expand the goal state (except possibly for the last iteration, which ends when the memory is full).

Instead of performing two full explorations of a pre-existing neighborhood (built with breadth-first search, as discussed above), ITSA* performs several A* searches over a region of the search space that grows around P in a layered fashion. Only the last iteration of ITSA* is a complete A* search over the entire neighborhood. Furthermore, ITSA* outputs a (potentially better) path at the end of each A* iteration. This anytime behavior of ITSA* is discussed in Section 5.3.1. In the next sub-section, we evaluate the performance of ITSA* as a local optimization procedure.

5.2.4 Empirical evaluation of ITSA*

We evaluate the performance of ITSA* as a local optimization algorithm in two standard benchmark domains, namely the 48-Puzzle and the Rubik's Cube. We run ITSA* on solution paths found by BULB in 50 random instances for each domain. We report the solution quality output by ITSA* and its runtime, which we compare to that of BULB (see Tables 22 through 25).

First, we discuss the performance of *one-step ITSA** (see Tables 22 & 23), where ITSA* is

Table 23: Performance of one-step ITSA* on paths found by BULB in the Rubik's Cube (with 3 million nodes in memory)

B	BULB		BULB + one-step ITSA*					
	time (seconds)	cost	time (seconds)			cost		
			value	increase over BULB		value	decrease over BULB	
				absolute	relative		absolute	relative
10	96.9	108,804.8	100.7	3.8	4%	94,346.6	14,458.2	13%
100	5.1	1,893.9	7.9	2.8	56%	679.0	1,214.9	64%
1,000	7.4	275.8	10.2	2.8	38%	178.5	97.3	35%
10,000	13.8	53.6	18.5	4.7	34%	47.3	6.3	12%
50,000	39.2	31.2	46.0	6.8	17%	30.6	0.6	2%
70,000	51.1	30.0	57.3	6.2	12%	28.7	1.3	4%
100,000	74.8	28.1	81.3	6.5	9%	27.6	0.5	2%
120,000	127.2	26.0	134.8	7.6	6%	25.7	0.3	1%

applied only once to the path found by BULB. In each domain, the absolute runtime of ITSA* remains approximately constant (always under ten seconds) when compared to the runtime of BULB. Since ITSA* searches a neighborhood whose size (i.e., its number of states) is fixed by the available memory, its runtime is essentially determined by the time it takes A* to search it. We speculate that the differences in runtimes for various values of B result at least in part from the fact that ITSA* performs varying numbers of iterations of A* depending on the length (or equivalently cost) of the starting solution path. This effect is more prominent in the Rubik's Cube domain, but in either case, the relation between initial solution length and runtime is not monotonic. Other factors influencing runtimes include the overhead of node generation and the branching factor of the search space (both of which are larger in the Rubik's Cube than in the 48-Puzzle). The approximately constant runtime of one-step ITSA*, together with the increasing runtime of BULB, explains why the relative increase in runtime of one-step ITSA* gets smaller as B increases (i.e., as the starting solution cost decreases).

Similarly, the relative improvement in solution cost achieved by ITSA* decreases as B increases. This trend is explained by the fact that, as the initial path length decreases, the path itself becomes less convoluted. Consequently, ITSA* has fewer opportunities to find shortcuts within the neighborhood. In other words, high-quality solutions are more likely to be (closer in solution quality to) local optima in the space of solution paths and, not surprisingly, ITSA* has a harder time

Table 24: Performance of multi-step ITSA* on paths found by BULB in the 48-Puzzle (with 6 million nodes in memory)

B	BULB		BULB + multi-step ITSA*					
	time (seconds)	cost	time (seconds)			cost		
			value	increase over BULB		value	decrease over BULB	
				absolute	relative		absolute	relative
5	0.1	11,737	50.3	50.2	50,200%	2,562	9,175	78%
10	0.9	36,282	53.0	52.1	5,789%	1,808	34,474	95%
100	6.1	14,354	55.2	49.1	805%	1,159	13,195	92%
1,000	7.3	1,409	36.7	29.4	409%	674	735	52%
10,000	21.7	440	42.4	20.7	95%	426	14	3%

Table 25: Performance of multi-step ITSA* on paths found by BULB in the Rubik's Cube (with 3 million nodes in memory)

B	BULB		BULB + multi-step ITSA*					
	time (seconds)	cost	time (seconds)			cost		
			value	increase over BULB		value	decrease over BULB	
				absolute	relative		absolute	relative
10	96.9	108,804.8	111.3	14.4	15%	94,346.6	14,458.2	13%
100	5.1	1,893.9	23.2	18.1	356%	578.5	1,315.4	69%
1,000	7.4	275.8	22.1	14.7	199%	156.8	119.0	43%
10,000	13.8	53.6	28.1	14.3	104%	45.3	8.3	15%
50,000	39.2	31.2	49.8	10.6	27%	30.4	0.8	3%
70,000	51.1	30.0	62.4	11.3	22%	28.7	1.3	4%
100,000	74.8	28.1	86.1	11.3	15%	27.5	0.6	2%
120,000	127.2	26.0	137.4	10.2	8%	25.7	0.3	1%

improving on higher quality solutions (ceiling effect).[3]

Second, we discuss the performance of *multi-step ITSA** (see Tables 24 & 25), where we apply ITSA* iteratively, first to the path found by BULB, and then repeatedly to the best path found during the previous execution of ITSA*. Multi-step ITSA* can stop as soon as ITSA* returns its starting path which is then a local minimum in the space of paths. For efficiency reasons, instead of checking the path itself, we only check its length. So we stop multi-step ITSA* when the solution cost it returns is equal to that of its starting solution.

[3]The case $B = 10$ stands out in Table 23. With such a narrow beam width, BULB reaches deeply into the search space and the cost of the solution is over 100,000. Given that 1) the branching factor in the Rubik's Cube is approximately equal to 13 and 2) the available memory can store up to 3 million nodes, ITSA* can only perform one complete iteration of A* before memory runs out. The neighborhood only extends as far as layer one and the relative improvement in solution quality is thus low. Similarly, because BULB is so slow in this case, ITSA*'s relative increase in runtime is small.

In both domains, the absolute runtime of multi-step ITSA* tends to decrease as B increases. As the initial path cost decreases, it is harder for local search to improve it and ITSA* is called fewer times before it reaches a local minimum. In addition, since the runtime of BULB increases with B, the relative increase in runtime of multi-step ITSA* decreases as B increases. Finally, both the absolute and relative improvements in solution quality decrease as B increases because again, better solutions are harder to improve on.

5.3 ABULB: Anytime BULB

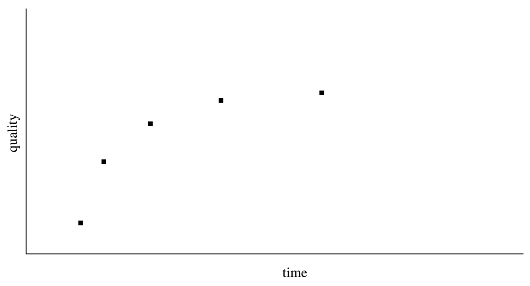
In this section, we discuss three ways of transforming BULB into an anytime algorithm. First, the solution output by BULB is improved by an anytime local search algorithm such as ITSA*. Second, BULB is kept running after it finds a goal. Third, BULB is restarted with a different value of B upon finding a goal. We present these three variants in the first three sub-sections, respectively. Then, we discuss how to combine the use of ITSA* with the last two variants. Finally, we evaluate all resulting variants in our two benchmark domains.

5.3.1 BULB + ITSA*: Local optimization of BULB's solutions

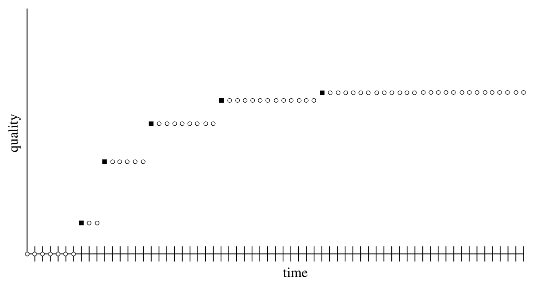
ITSA*, as a local search algorithm, is also an anytime algorithm. First, one-step ITSA* returns solutions of non-increasing cost at the end of each iteration of A*. Second, multi-step ITSA* repeatedly applies ITSA* to solution paths with lower and lower costs until a local minimum is reached.

While ITSA* is a general purpose anytime algorithm for local path optimization, we propose to apply it to the solution output by BULB. More specifically, once BULB terminates with a path P to the goal, all states not in P are purged from memory and multi-step ITSA* is started with all states in P assigned to the zeroth layer and OPEN initialized to the start state (see Section 5.2.3). The only parameter of this anytime algorithm is the value of the beam width B used by BULB. We choose a low value of B so that the first solution is quickly found.

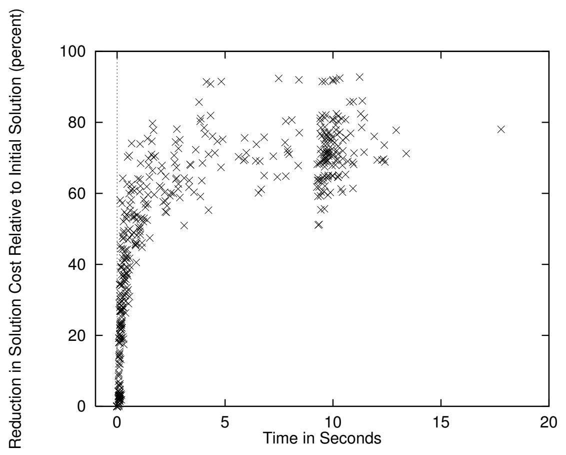
Since this chapter focuses on anytime algorithms (such as ITSA*), we need to be able to represent the performance of an anytime algorithm. The standard representation is a performance profile, that is the expected solution quality output by the algorithm as a function of time. We follow the



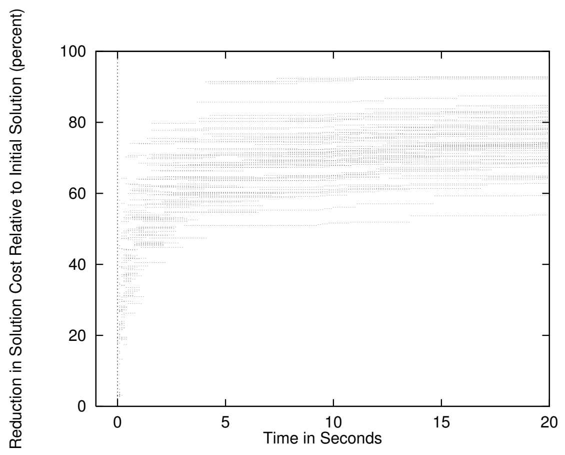
a) Data points obtained for one instance



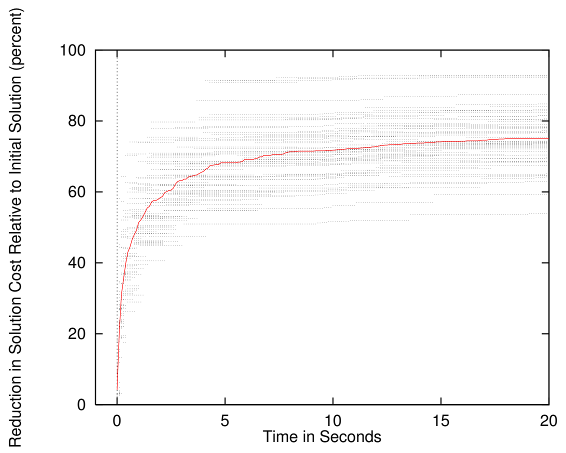
b) Corresponding stair function



c) Data points obtained for fifty instances



d) Corresponding stair functions



e) Corresponding average performance profile

Figure 58: Building a performance profile

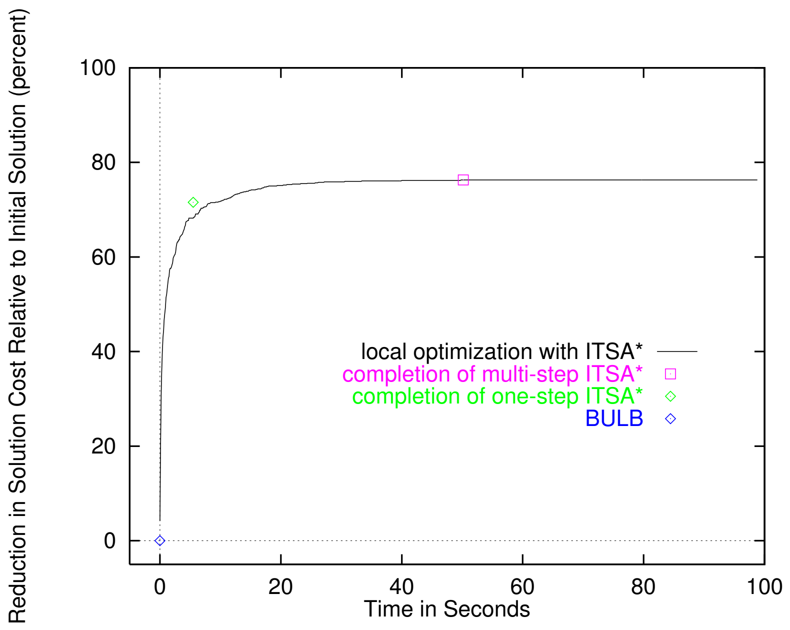


Figure 59: Performance of ITSA* on solutions produced by BULB in the 48-Puzzle (with 6 million nodes and B=5)

standard procedure to obtain the performance profile [183]. First, we compute the quality q as the improvement in solution quality of each new solution path P' relative to the original solution path P (in our case, the one found by BULB). Thus, $q = \frac{cost(P) - cost(P')}{cost(P)}$. The performance profile plots q as a function of t, the time taken to produce P' from P. Thus, $t = runtime(P') - runtime(P)$.[4] Figure 58a shows such data points corresponding to one instance. Note that solutions are found at irregular time intervals. Since the solution quality does not change between two successive data points, we fill in equally spaced points to yield a staircase function (see Figure 58b). Figures 58c & d contain the real and “filled in” data (respectively) for the complete set of fifty test instances. Finally, the performance profile is obtained by fitting a curve through the resulting cloud of points (for each time slice, we compute the average quality improvement of all points in the slice). We call this curve the *performance profile*.

The performance profile in Figure 58e is actually that of ITSA* when applied to solutions found by BULB in the 48-Puzzle. We reproduce it in Figure 59. In addition, the figure shows the average completion time (and the corresponding quality improvement achieved) for one-step and multi-step ITSA*. The reason these points do not sit squarely on the curve is explained in Figure 60 with

[4]Note that defining the runtime as the time elapsed since the first solution is produced abstracts away the variance in the runtimes of BULB over the set of instances.

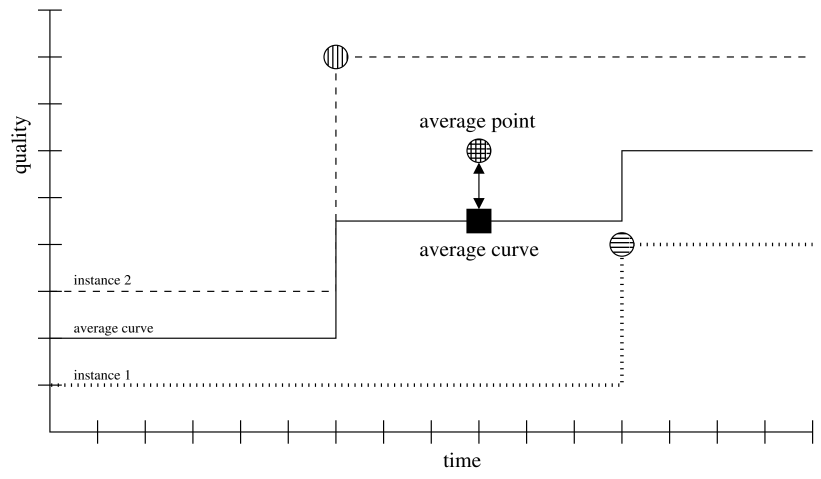


Figure 60: An average point lies above the average curve

only two instances. The discrepancy in quality at the average time slice results from the fact that the average point is based on two actual data points while the average curve is based on two interpolated points.

Finally, Figure 61 shows the performance of ITSA* in the Rubik's Cube domain. In this domain, the final reduction in solution cost is not as large as in the 48-Puzzle, probably because the memory can store fewer nodes and the neighborhood is thus smaller. Nevertheless, the profiles for both domains share similarities. Most of the decrease in solution cost is obtained with one-step ITSA*. Multi-step ITSA* only marginally reduces the solution cost. Both one-step and multi-step ITSA* terminate in a matter of seconds. Their runtime is smaller than in the 48-Puzzle because the neighborhood is smaller and ITSA* performs fewer iterations in the Rubik's Cube domain than in the 48-Puzzle.

5.3.2 ABULB 1.0: Continuous execution of BULB with a constant B value

The previous section describes how to apply local search to a solution found by another algorithm. In this section (and the next one), we focus on a class of anytime algorithms that invoke a single algorithm to generate multiple solutions.

Arguably the simplest way to turn a one-shot search algorithm into an anytime algorithm is to let it run after it finds a goal [56, 57]. The algorithm thus continues to explore the search space with the same strategy and parameter settings. In other words, while the solution is recorded, the search continues as if no solution was found. In the case of BULB, we call the resulting algorithm

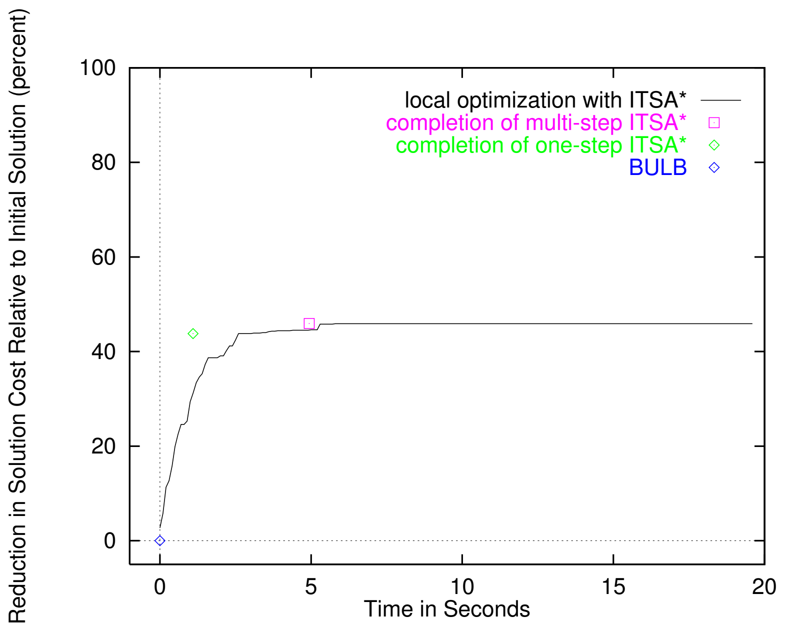


Figure 61: Performance of ITSA* on solutions produced by BULB in the Rubik's Cube domain (with 1 million nodes and B=70)

Anytime BULB or *ABULB* for short. Since the next section describes a different approach also based on BULB, we call this version *ABULB 1.0*.

ABULB 1.0 is a complete and admissible algorithm since it searches the whole search tree rooted at the start state down to a depth whose value depends on the parameter settings. Recall that the maximum depth reached by BULB is approximately equal to M/B, where M is the maximum number of nodes that can fit simultaneously in memory and B is the beam width. This upper bound on the search depth follows from the fact that BULB stores at most B nodes at each level of its search tree. Therefore, ABULB 1.0 is complete and admissible whenever the depth of a shallowest goal state is less than or equal to this bound. In addition, ABULB 1.0 is an anytime algorithm since it finds a sequence of solutions and can at any time return the best solution found so far. Because it maintains a dynamic upper bound (to be described shortly) on the optimal solution cost, ABULB 1.0 is guaranteed to return solutions of monotonically increasing quality.

Figure 62 depicts the behavior of ABULB 1.0. It behaves exactly like BULB until the first goal (G_1 in the figure) is found. Then, two modifications are enforced. First, the search continues as long as time remains to find better and better solutions (G_2, then G_3 in the figure). Second, the depth of successive BULB searches is bounded from above by decreasing values. Indeed, it is easy

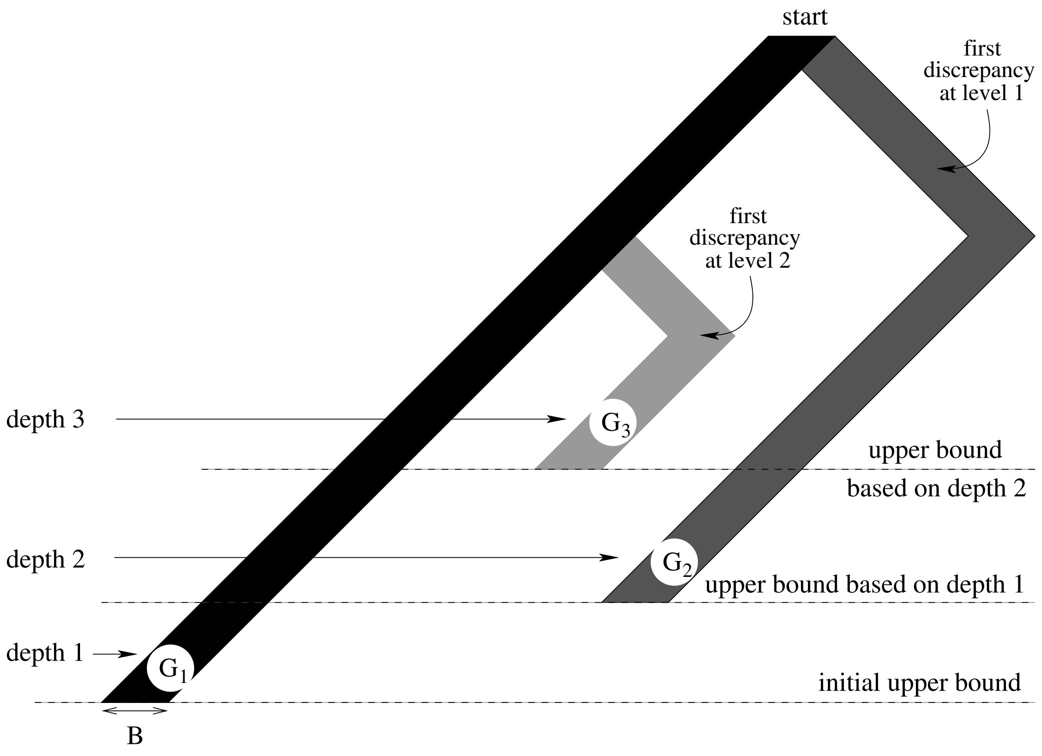

start
first discrepancy at level 1
first discrepancy at level 2
depth 3
G_3
upper bound based on depth 2
depth 2
G_2
upper bound based on depth 1
depth 1
G_1
initial upper bound
B

Figure 62: Behavior of ABULB 1.0

to maintain a dynamic upper bound on the solution cost (or depth).[5] This upper bound is initialized to the maximum searchable depth (namely M/B) until the first goal is found. Then, every time a goal is found, the upper bound is reduced based on the latest goal's cost (or depth). During the search, all candidate nodes whose depth is larger than the upper bound are pruned (i.e., they are not generated). As the search proceeds, the upper bound decreases and the pruning power increases, until an optimal goal is found.

The only input parameter of ABULB 1.0 is B, the beam width. Its value is set at the time the algorithm is called and is never modified. Since the first solution should be found quickly, we set B to a small (domain-dependent and empirically determined) value.

5.3.3 ABULB 2.0: Restart of BULB with varying B values

In this section, we consider another instance in the class of anytime approaches that use a single algorithm to generate multiple solutions. In this instance, a single algorithm is called with different parameter settings. We propose to make successive calls to BULB with different values of B. Recall

[5]Or equivalently, a lower bound on the solution quality.

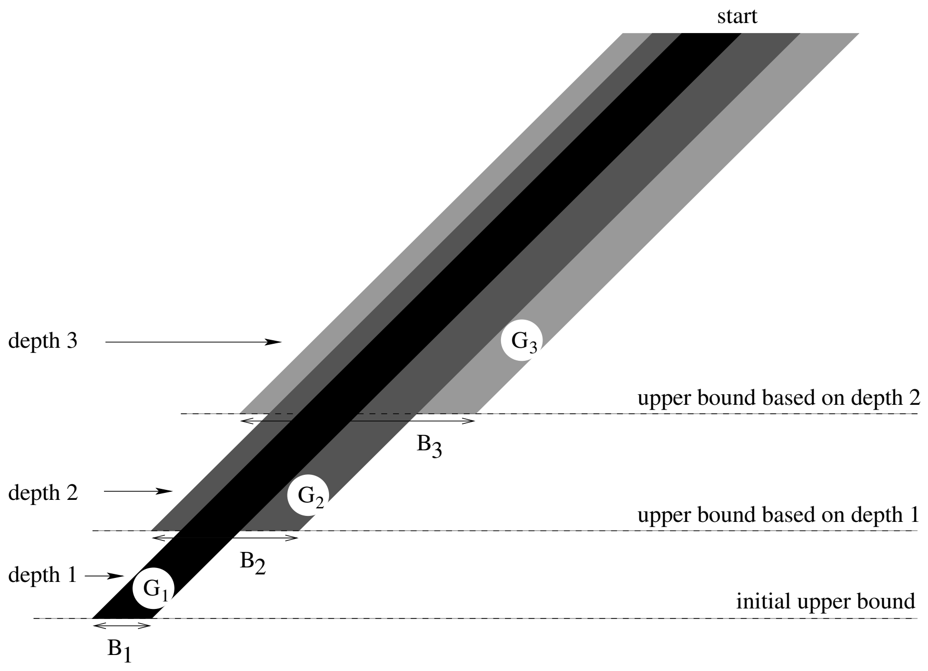


Figure 63: Behavior of ABULB 2.0

that larger values of B tend to generate solutions of higher quality, since BULB behaves more and more like breadth-first search. Therefore, *ABULB 2.0* first calls BULB with a small value of B to quickly find a first solution and then repeatedly calls BULB with increasing values of B to improve the solution quality until time runs out.

Figure 63 depicts the behavior of ABULB 2.0. It behaves exactly like BULB until the first goal (G_1 in the figure) is found. Then, like in ABULB 1.0, a new upper bound is computed based on the cost of the best solution found so far. However, unlike ABULB 1.0, a new value of B is also computed. One important weakness of ABULB 1.0 is that it does not use all the available memory. As time goes by, the upper bound decreases. Thus, the search gets more and more shallow. Since its beam width remains constant (equal to B), ABULB 1.0 uses less and less memory. ABULB 2.0 addresses this issue by increasing the value of B in such a way that the product of B and the upper bound on the search depth remains equal to M (the maximum number of nodes storable in memory). Every time a goal is found (say G_i in the figure), B is updated to a value that is slightly larger than $\frac{M}{\text{depth } i}$.

Like for BULB and ABULB 1.0, the behavior of ABULB 2.0 depends on the initial value of B. A low value typically returns the first solution quickly. In addition, the schedule of B updates also

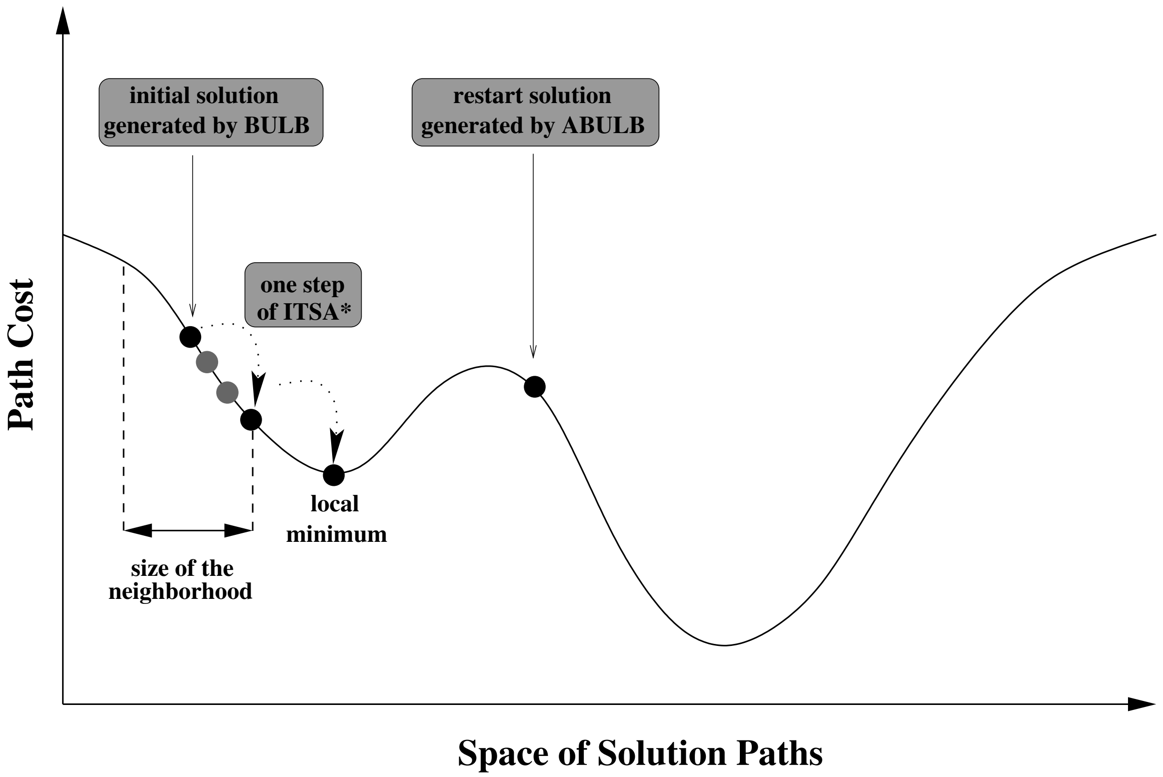


Figure 64: ABULB + ITSA*: A neighborhood search in the space of solution paths

influences the search. A faster increase of B may lead to the solution quality increasing faster with each solution. However, the elapsed time between solutions may also increase substantially. In order to keep ABULB 2.0 admissible, the upper bound and B values should be updated conservatively: the former should be updated to one less than the best cost found so far, while the latter should be set to the largest width that allows the full beam to extend to a depth equal to the upper bound.

5.3.4 ABULB + ITSA*: Local optimization of ABULB's solutions

So far in this chapter, we have discussed three new and distinct anytime algorithms, namely the local search procedure ITSA*, ABULB 1.0, and ABULB 2.0. These methods can be combined in several ways to yield anytime algorithms that use ITSA* for local search and either of the ABULB variants as a generator of restarting solutions (see Figure 64). This section discusses the resulting variants of ABULB.

While ITSA* can in theory be applied to any solution path, its application is limited in practice by the fact that it uses all the available memory to store nodes in the neighborhood. In particular, it starts by purging the memory of all nodes that are off the solution path. Since the ABULB algorithm

Table 26: Versions of Anytime BULB (ABULB)

Version	Changing B	Local optimization
ABULB 1.0	no	none
ABULB 1.1	no	one-step ITSA*
ABULB 1.2	no	multi-step ITSA*
ABULB 2.0	yes	none
ABULB 2.1	yes	one-step ITSA*
ABULB 2.2	yes	multi-step ITSA*

is based on backtracking whose mechanism relies on stored information, combining it with ITSA* requires taking into account possible conflicts for memory needs.

In the case of ABULB 1.0, it is not possible to purge memory every time a goal is found because the anytime algorithm needs to keep the beam in memory in order to perform backtracking (since ABULB 1.0 simply runs BULB as if no goal is found). However, ABULB 1.0 automatically purges memory at the end of each iteration, where an iteration is defined by a given number of allowed discrepancies. Before each iteration starts, the number of allowed discrepancies is incremented by one. Most importantly at that time, memory is not needed by ABULB 1.0 since the next iteration performs a complete search from the start state. We therefore propose to apply ITSA* at the end of each iteration on the best solution found so far.

In the case of ABULB 2.0, BULB is used to find a single solution, then stopped and restarted with a larger value of B. Therefore, each iteration of ABULB 2.0 performs a complete BULB search from the start state until a goal is found. No memory is needed between iterations and ITSA* can be applied to locally optimize each solution found. Finally, since ITSA* comes in two versions (namely, one-step and multi-step), the number of variants of ABULB increases accordingly. We use the version numbering scheme listed in Table 26 to identify the different combinations of ABULB and ITSA*.

5.3.5 Empirical evaluation of ABULB

In this section, we compare the performance of our ABULB variants in two of the benchmark domains from the previous chapters, namely the 48-Puzzle and the Rubik's Cube. For a performance comparison of ABULB 1.0 and ABULB 2.0 in the MSA domain, see Chapter 6. Here, we first

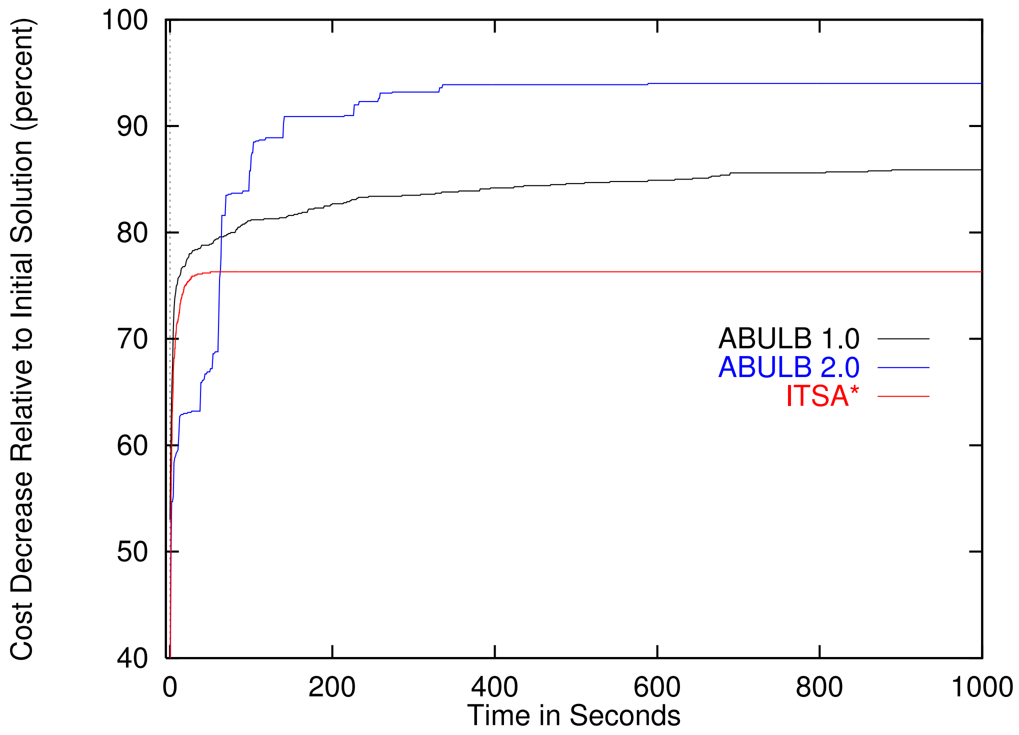


Figure 65: ABULB 1.0 versus ABULB 2.0 in the 48-Puzzle (with 6 million nodes and B=5)

compare ABULB 1.0 and ABULB 2.0 in the two benchmark domains. Then we evaluate the impact of local optimization with ITSA* in the remaining variants of ABULB, namely 1.1, 1.2, 2.1, and 2.2.

Figure 65 and 66 show the performance profiles of ABULB 1.0 and 2.0 in the 48-Puzzle and Rubik's Cube, respectively. The figures also show the performance profile of ITSA*. The profiles are averaged over the same problem instances as those used in Chapter 4. The following observations hold for both domains. First, both ABULB 1.0 and 2.0 eventually reduce the solution cost significantly more than ITSA* does. This shows that, in both domains, limiting the search around the neighborhood of the initial solution (ITSA*) is not as effective as exploring different parts of the solution space with restarts (ABULB).

Second, in both domains, ABULB 2.0 eventually reduces the solution cost more than ABULB 1.0 does. Recall that ABULB 2.0 takes advantage of all the available memory to improve solution quality, while ABULB 1.0 uses less and less memory as the search progresses since its beam width remains constant while its maximum search depth decreases (thanks to the dynamic upper bound). With its small (and constant) beam width, ABULB 1.0 needs a lot of backtracking

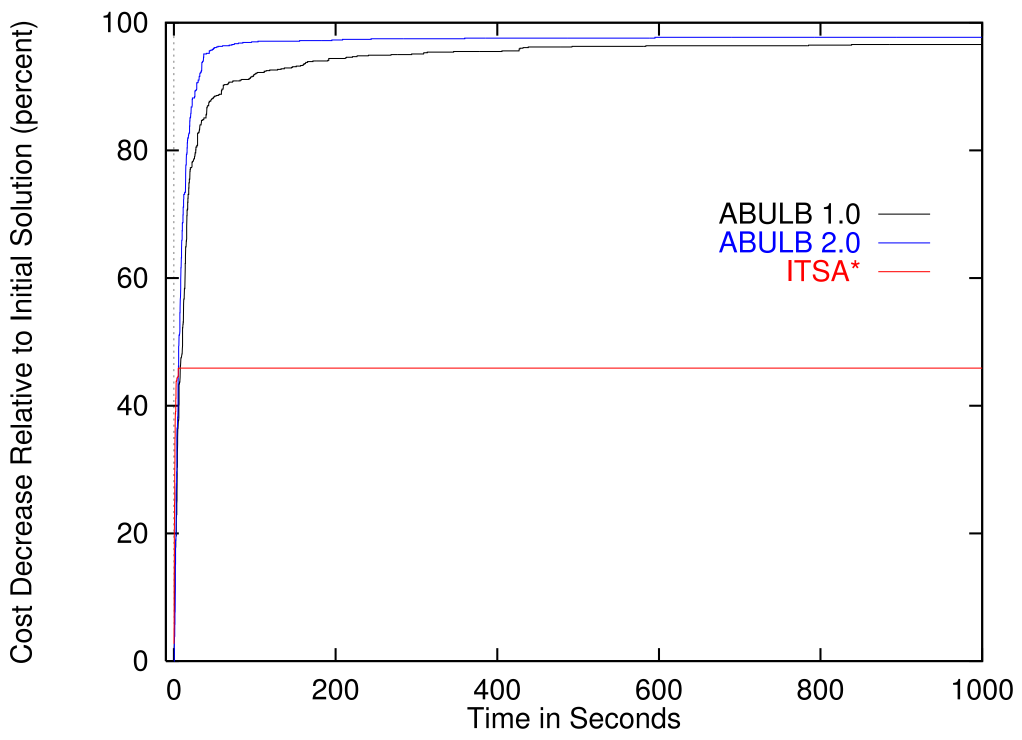


Figure 66: ABULB 1.0 versus ABULB 2.0 in the Rubik's Cube domain (with 1 million nodes and B=70)

and thus is slower to reduce the solution cost. Its performance profile takes longer to level off than that of ABULB 2.0. However, in the 48-Puzzle only, ABULB 1.0 is better than ABULB 2.0 during the first 60 seconds or so.

We now consider the variants of ABULB that use ITSA* for local optimization. Figures 67 and 68 show that, in the 48-Puzzle, local optimization with ITSA* improves the performance of ABULB 1 and ABULB 2.0, respectively. First, in both figures, plain ABULB (i.e., without ITSA*) is the algorithm that eventually exhibits the smallest decrease in solution cost. Therefore, combining ITSA* with ABULB, is always better in the long run in this domain. Second, in both figures, the combination of multi-step ITSA* with ABULB (i.e., ABULB x.2) yields the largest decrease in solution cost in the long run. Overall, ABULB 2.2 is the best algorithm in this domain for long-term performance. In contrast, one-step ITSA* yields the largest decrease in solution cost at the outset (again in both figures). Overall, ABULB 2.1 is the best algorithm in this domain for short-term performance.

Figures 69 and 70 show a different picture in the Rubik's Cube domain. In this domain, neither

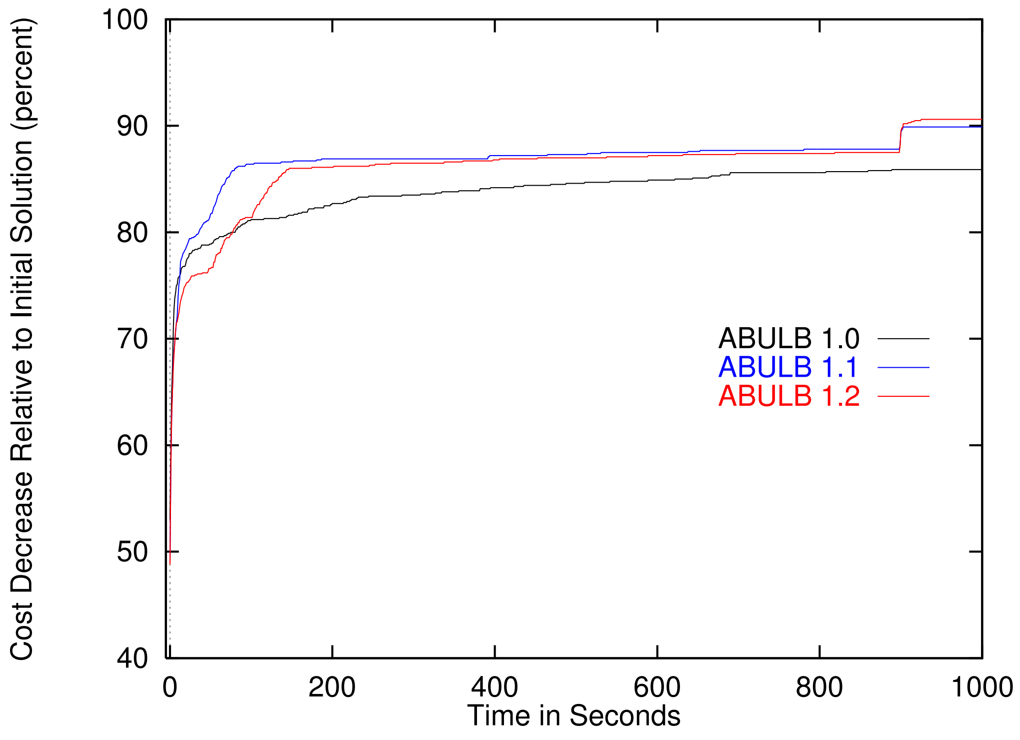


Figure 67: Combining ITSA* with ABULB 1 in the 48-Puzzle (with 6 million nodes and B=5)

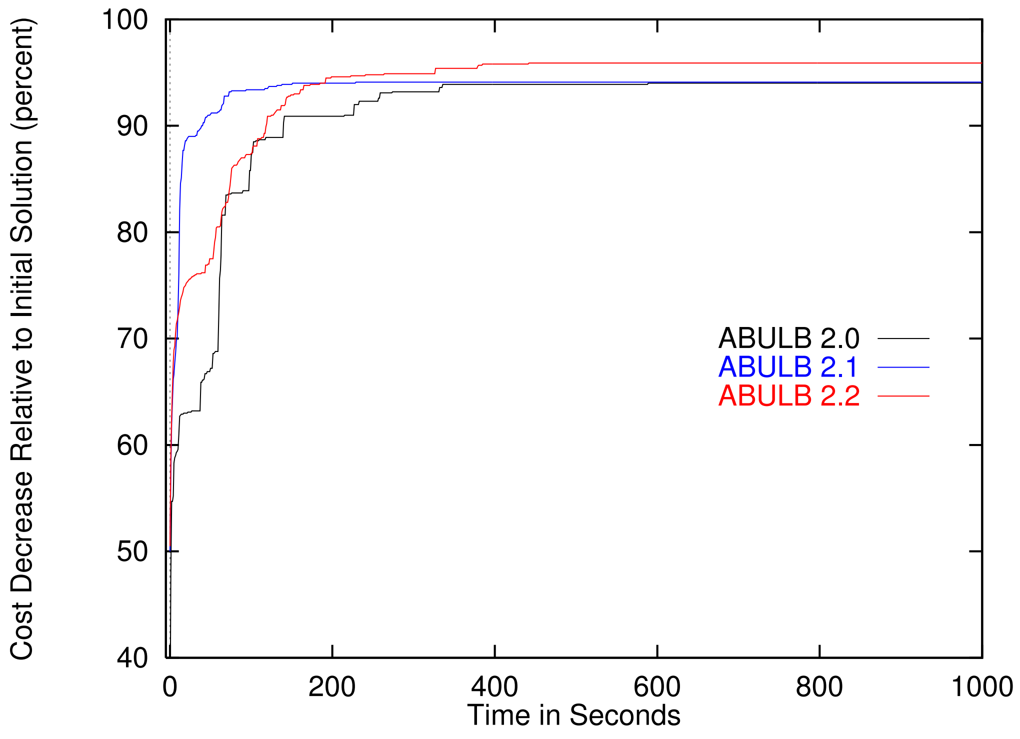


Figure 68: Combining ITSA* with ABULB 2 in the 48-Puzzle (with 6 million nodes and B=5)

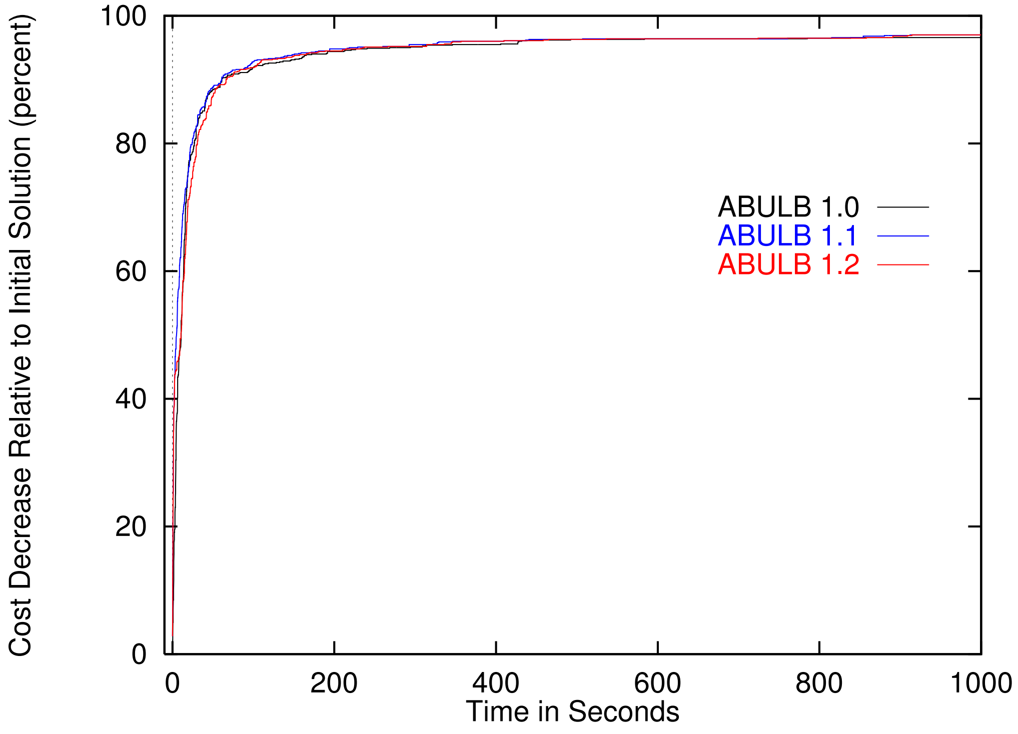


Figure 69: Combining ITSA* with ABULB 1 in the Rubik's Cube domain (with 1 million nodes and B=70)

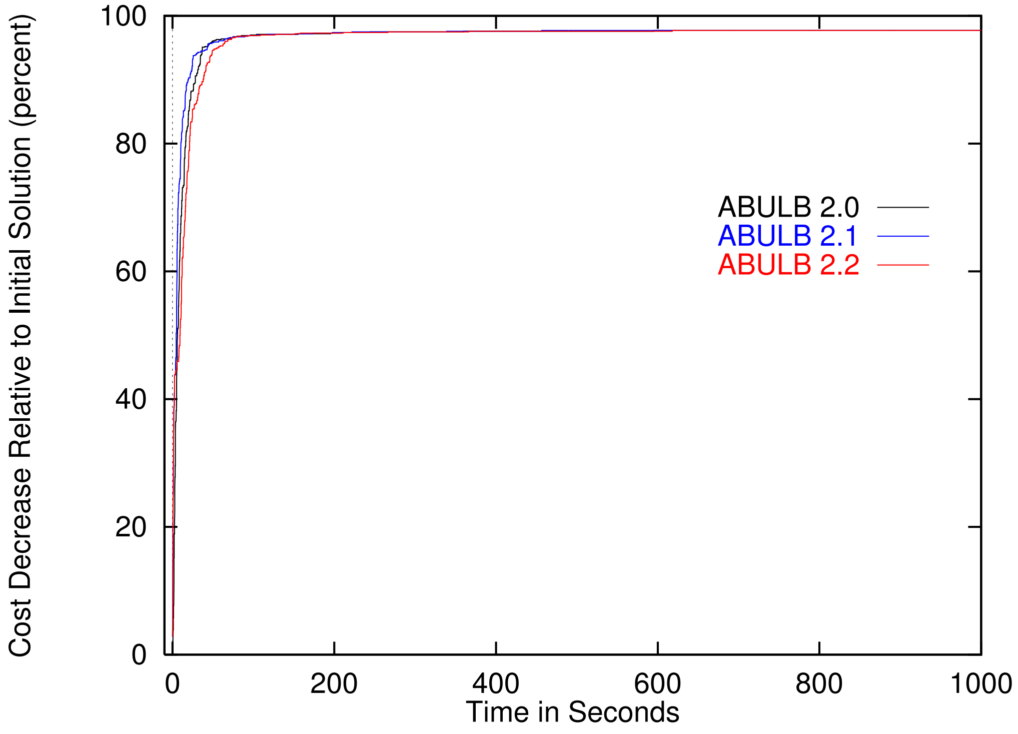


Figure 70: Combining ITSA* with ABULB 2 in the Rubik's Cube domain (with 1 million nodes and B=70)

one-step nor multi-step ITSA* significantly improves the performance profile when combined with ABULB. The decrease in solution cost obtained by letting BULB run (in ABULB 1.0) or by restarting it (in ABULB 2.0) outweighs the decrease in solution cost obtained with ITSA*. The latter is (relatively) small because, as pointed out before, the large branching factor and the tight memory constraints allow for only a small number of A* iterations within ITSA*.

5.4 Related work

In this section, we describe related work in both anytime heuristic search and local search.

5.4.1 Anytime heuristic search

In *time-dependent* tasks, the quality of a solution depends not only on its intrinsic characteristics, but also on how fast it is produced. In AI, the notion of time-dependency was first coined in the context of planning [27]. Since one of the main constraints in time-dependent planning is time, it makes sense to try and make optimal use of this resource. However, when considering the trade-off between planning time and plan quality, approximation and admissible algorithms appear to be two discrete points at each end of a spectrum. On the one hand, approximation algorithms are as fast as possible with or without guarantee on solution cost (that is, by how much it exceeds the shortest path). On the other hand, admissible algorithms guarantee optimality of the solution at the expense of (potentially) exponential time of execution. In other words, admissible algorithms consider time as secondary compared to the solution quality, while approximation algorithms do the reverse. In both cases, the trade-off is fixed.

Anytime algorithms were introduced to provide greater flexibility when trading off time and solution quality [71, 27, 11]. The usefulness of anytime algorithms is obvious when compared to the two following extremes. First, the solution produced by an admissible algorithm can be completely useless if it comes too late. Second, the solution produced by an approximation algorithm can come too early, in the sense that it is produced before it is actually needed as input to a subsequent problem solving stage whose start time has not come yet. This wasted time interval could have been used to improve the approximate solution. Anytime algorithms are designed to provide such flexibility when dealing with uncertain time constraints.

There exist only a few anytime heuristic search algorithms to date. The most relevant ones are variants of WA* (Weighted A*).

Anytime A* (ATA*) [56, 57, 178] is a best-first search algorithm that maintains for each node both an A*-like f-value and a weighted f-value. The only difference between weighted best-first search and Anytime A* is that it does not stop after a solution is found. Instead, the search continues and the cost of the lowest-cost solution found so far is used as an upper bound on the cost of a minimum-cost solution. Such an upper bound serves to prune nodes in the OPEN list whose (unweighted) f-value is larger than or equal to the upper bound. Since this bound decreases over time, the set of nodes that can be pruned increases and Anytime A* eventually finds a minimum-cost solution. ATA* is thus to WA* as ABULB 1.0 is to BULB.

ARA* [114, 115] is another variant of WA* that is more closely related to ABULB 2.0 since it repeatedly calls WA* with decreasing weights on the h-values (whereas ABULB 2.0 repeatedly calls BULB with increasing B values). One difference between ARA* and ABULB 2.0 is that ARA* reuses some of the search effort of the previous WA* search to speed up the current iteration. Combining ideas in incremental search with ABULB is an interesting direction for future research.

However, the main limitation of these variants of WA* is that they do not control their memory consumption (other than by increasing the weight on the h-values). Therefore, both algorithms can only solve problems that WA* can solve. In contrast, we have shown in this dissertation that BULB and ABULB, as memory-bounded algorithms, scale up to much larger domains.

Finally, all other anytime heuristic search algorithms known to us are variants of depth-first search. For example, DFBnB (depth-first branch-and-bound) is a variant of branch-and-bound search [112, 110]. It is a depth-first search that uses h-values to order the successors for expansion and upper bounds to prune some of them. Another example is Complete Anytime Beam search (CABS) [173]. CABS repeatedly calls a variant of depth-first search with a pruning rule that is progressively weakened at each iteration. As the pruning rule weakens, fewer and fewer nodes are pruned. In the worst case, no pruning occurs and thus CABS is complete.

The main weakness of these depth-first search algorithms from our perspective is that they do not scale well to general graph-search problems (such as our benchmark domains and the multiple sequence alignment problem described in Chapter 6) since they cannot detect transpositions and thus tend to suffer from an exponential overhead in node re-generations. Nevertheless, such methods are well suited to problems with a high density of solutions or a finite search tree [135]. DFBnB is very efficient on the Traveling Salesperson Problem [111], for instance [174, 175].

5.4.2 Local search

Local or neighborhood search methods (also called (meta-)heuristics) have been applied by the OR community to a large number of combinatorial problems [166, 139]. However, virtually all these problems share the characteristic that a sub-optimal solution can be found easily (without search). In the Traveling Salesperson Problem (TSP) [111] for example, a feasible solution is a permutation of an explicit set of cities, while in the boolean satisfiability problem (SAT) [148], a feasible solution is an assignment of truth-values to an explicit set of boolean variables. In contrast, in the shortest-path problem, a solution (path) is a sequence of states (or equivalently state-action pairs). Since the states are too numerous to list explicitly, only a small list of action schemata (or operators) are provided and finding any path requires search.

Neighborhood search is sometimes called iterative-improvement search [144, 1], since solution quality is improved by iteratively jumping from one path to another one in the subset of the solution space defined as the neighborhood. This general paradigm leaves open the question of how to select the next solution path within the neighborhood. ITSA* follows a best-improvement [1] or steepest-descent [130] strategy, since it uses a variant of A* to select the solution of highest quality within the neighborhood. Alternatively, a first-improvement strategy would stop searching the neighborhood when any solution is found with higher quality than the current solution.

5.5 Conclusion

In this chapter, we have presented a new family of anytime heuristic search algorithms generically called ABULB (Anytime BULB). ABULB is a local (or neighborhood) search algorithm in the space of solution paths. ABULB uses BULB to find both an initial solution and restarting solutions. ABULB can also take advantage of ITSA* for local path optimization.

ITSA* (Iterative Tunneling Search with A*, pronounced *It's a star*) is a new local path optimization algorithm. ITSA* imposes a neighborhood structure on the space of solution paths based on our definition of distance between paths. ITSA* interleaves the construction and the searching of the neighborhood using breadth-first and A* search, respectively. Each iteration returns a path of smaller cost. ITSA* is thus an anytime algorithm in its own right. ITSA* performs gradient descent on the surface whose connectivity and elevation result from the neighborhood structure and the solution cost, respectively. Each time ITSA* reaches a (possibly local) minimum on the surface, ABULB generates a new restarting solution of higher quality.

Our empirical study has shown that, while ITSA* reduces the solution cost over time when used as an anytime algorithm in the 48-Puzzle and the Rubik's Cube domain, an even larger reduction in solution cost is achieved by letting BULB 1) run after it finds a solution or 2) by restarting it with a larger, automatically computed beam width. Furthermore, combining ITSA* with either anytime modification of BULB yields an even larger reduction in solution cost in the 48-Puzzle. In the Rubik's Cube however, no significant improvement (or degradation) was observed in the performance profile of ABULB when adding ITSA*.

This research has produced the following contributions:

- We have introduced a new local path optimization (namely, ITSA*) based on a search strategy called *iterative tunneling*.
- We have described and empirically evaluated different ways of transforming BULB into an anytime algorithm, both with and without dynamic beam widths.
- To the best of our knowledge, our way of combining ABULB with ITSA* is the first successful application of local (or neighborhood) search to the shortest path problem.

Possible avenues for future work include:

- studying the effect of neighborhood size on the performance of ITSA*; for example, can its runtime be decreased by reducing the neighborhood size to a portion of the available memory with an acceptable loss in solution quality? In this case, how best to take advantage of the freed memory?,
- studying the effect of initial solution quality on the quality of local optima found by ITSA*;

for example, what happens if the quality of restarting solutions does not increase monotonically as is currently the case for ABULB?,

- studying how the trade-off between runtime and solution quality in ABULB (i.e., the slope of the performance profile) is affected by the schedule of B value updates; for example, can smaller increments in B values lead to steeper profiles?, and
- studying domain-dependent properties that influence the performance of our neighborhood search; for example, how does the graph connectivity, say the number of small loops, affect the local optimization performed by ITSA*?

CHAPTER VI

THE MULTIPLE SEQUENCE ALIGNMENT PROBLEM

In this chapter, we use the multiple sequence alignment (MSA) problem as an additional benchmark domain for ABULB. We explain how the MSA problem reduces to the shortest-path problem. We describe how to apply ABULB to the MSA problem and show empirically that ABULB scales up to larger MSA problems than an existing anytime heuristic search algorithm based on WA*.

6.1 Introduction

The primary structure of biological sequences is essentially a linear string of characters over a small alphabet. The primary structure of a nucleic acid such as DNA, for example, is a sequence of characters from the 4-element alphabet {A,C,G,T}, while proteins are sequences of characters from the 20-element set of amino acids (also called residues). Over the past decade or so, the number of available sequences has grown exponentially, as powerful new sequencing technology has emerged. Whenever a new sequence is identified, biologists are interested in determining both its function (for a gene or an enzyme, for example) and its 3-dimensional configuration (for proteins in general). While the most reliable approach remains the empirical one, it is quite time-consuming. This explains the need for, and the great progress of, computational approaches.

One way to infer the function (or 3-dimensional structure) from a linear sequence is to compare the new sequence to available sequences with known functions. For example, the function of a newly discovered protein may be inferred by determining its membership in a known family of proteins. Similarly, the function of a gene (DNA sequence) can be inferred by comparison to similar genes with known functions. However, to assess the similarity between a new sequence and one or more known sequences first requires the alignment of these sequences. This explains why the multiple sequence alignment problem is one of the most important challenges in computational biology.

```
(a)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSDLHAHKL
            G+ +VK+HGKKV  A+++++AH+D++ +++++LS+LH  KL
HBB_HUMAN   GNPKVKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKL

(b)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHV---D--DMPNALSALSDLHAHKL
            ++ ++++H+ KV   + +A  ++           +L+ L+++H+ K
LGB2_LUPLU  NNPELQAHAGKVFKLVYEAAIQLQVTGVVVTDATLKNLGSVHVSKG

(c)
HBA_HUMAN   GSAQVKGHGKKVADALTNAVAHVDDMPNALSALSD----LHAHKL
            GS+ + G +   +D L  ++ H+ D+  A +AL D    ++AH+
F11G11.2    GSGYLVGDSLTFVDLL--VAQHTADLLAANAALLDEFPQFKAHQE
```

Figure 71: Three pairwise alignments (taken from [33])

6.2 Sequence alignment

To align sequences means to write them one under the other, one letter per column. All the letters in a column are said to be aligned. Figure 71(a) depicts the alignment of the same region of two human protein sequences referred to as HBA-HUMAN and HBB-HUMAN in the SWISS-PROT database.[1] The protein sequences occupy the first and third lines in the figure. The middle line is *not* part of the alignment per se. It is added in order to make it easy to visually identify columns that contain a "good" alignment. Since two sequences are being aligned, each column of the alignment contains two letters. When the two letters in a column are identical, the letter is repeated in the middle line to highlight the perfect match. When the two letters are different but represent similar residues, a plus sign is inserted in the middle line to highlight a close but imperfect match (the similarity of residues, or equivalently the closeness of a match, is measured by a scoring function to be described in the next section). Finally, a blank in the middle line highlights a column where two dissimilar residues are aligned. Large numbers of letters and plus signs in the middle line, combined with a small number of blanks, indicate that the human alpha globin (HBA-HUMAN) is very similar to the human beta globin (HBB-HUMAN).

In Figure 71(b), the same region of the human alpha globin is aligned with another sequence (called LGB2-LUPLU). In this case, the proportion of perfect and close matches is smaller than

[1]The SWISS-PROT database is accessible at http://kr.expasy.org/sprot/.

in the alignment of Figure 71(a) but still large enough to indicate a possible relationship between the two globins. Indeed, this alignment is meaningful to biologists since these two globins are evolutionarily related and share structural and functional properties. Notice that this alignment is longer than the one in Figure 71(a), that is, it is made up of a larger number of columns. This is because, in the process of aligning the sequences, five gaps were inserted into the first sequence. Each gap is represented by a dash. In columns containing a dash, the residue in the second sequence is not aligned with any residue in the first sequence. Assuming that the two globins are evolutionarily related, a gap indicates that the residue in the same column was either deleted in the first sequence or inserted into the second sequence during evolution. In contrast, a plus sign in a column suggests that both residues are evolutionarily related through mutations. Either one evolved from the other, or both evolved from a common ancestor. Finally, a perfect match in a column suggests that the corresponding residue remained unchanged through evolution.

6.3 Evaluating alignments

The alignment in Figure 71(c) has approximately the same length and total number of letters and plus signs in the middle line as the alignment in Figure 71(b). Unfortunately, this alignment is spurious because the F11G11.2 protein is neither functionally nor structurally similar to the human alpha globin. Simply counting the number of (close) matches in the alignment is not sufficient to discriminate between meaningful and spurious alignments. A more sophisticated evaluation method is needed.

Biological sequences evolve via the selection of random mutations. We consider three types of mutations. A *substitution* replaces one residue with another, while an *insertion* or a *deletion* adds or removes a residue (or a group of residues). Determining the biological significance of an alignment requires a lot of expert knowledge of phylogenetic constraints on mutations, the properties of various groups of contiguous elements, the influence of higher-order structure, etc. Automating the alignment process requires the design of numerical formulas to capture these constraints. Statistical analyses are used to generate a likelihood ratio for each pair of residues. Each ratio estimates the odds of the pair appearing in the same column due to a phylogenetic relationship as opposed to a random occurrence. The set of ratios for all possible pairs of residues makes up a *score matrix* or

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	2																			
R	-2	6																		
N	0	0	2																	
D	0	-1	2	4																
C	-2	-4	-4	-5	12															
Q	0	1	1	2	-5	4														
E	0	-1	1	3	-5	2	4													
G	1	-3	0	1	-3	-1	0	5												
H	-1	2	2	1	-3	3	1	-2	6											
I	-1	-2	-2	-2	-2	-2	-2	-3	-2	5										
L	-2	-3	-3	-4	-6	-2	-3	-4	-2	2	6									
K	-1	3	1	0	-5	1	0	-2	0	-2	-3	5								
M	-1	0	-2	-3	-5	-1	-2	-3	-2	2	4	0	6							
F	-4	-4	-4	-6	-4	-5	-5	-5	-2	1	2	-5	0	9						
P	1	0	-1	-1	-3	0	-1	-1	0	-2	-3	-1	-2	-5	6					
S	1	0	1	0	0	-1	0	1	-1	-1	-3	0	-2	-3	1	2				
T	1	-1	0	0	-2	-1	0	0	-1	0	-2	0	-1	-3	0	1	3			
W	-6	2	-4	-7	-8	-5	-7	-7	-3	-5	-2	-3	-4	0	-6	-2	-5	17		
Y	-3	-4	-2	-4	0	-4	-4	-5	0	-1	-1	-4	-2	7	-5	-3	-3	0	10	
V	0	-2	-2	-2	-2	-2	-2	-1	-2	4	2	-2	2	-1	-1	-1	0	-6	-2	4

Figure 72: The PAM250 substitution matrix

substitution matrix.

PAM250 is the most widely used matrix for practical protein sequence alignment [25]. This 20×20 matrix is shown in Figure 72. The twenty amino acids $a_i, i = 1 \ldots 20$, appear on both the vertical and horizontal dimensions. Each PAM250$[a_i][a_j]$ is the log likelihood of the pair of residues (a_i, a_j) being an aligned pair, as opposed to appearing separately. The larger this (positive) number, the higher the likelihood that the pair will appear as a pair. The smaller this (negative) number, the higher the likelihood that the pair is unrelated. Since perfect matches appear on the diagonal, each number on the diagonal is larger than any other number in the corresponding row (or column, since the matrix is symmetrical). When working with this matrix, the score of an insertion or deletion, called a *gap cost* (or *gap penalty*), is typically set to -8, which is the smallest number in the matrix.

Finally, biologists often make the assumption that mutations at different places in a sequence take place independently. This independence assumption appears to be reasonable at least for DNA and protein sequences (not so much for RNA sequences). As a result, the total score of the alignment of two protein or DNA sequences is simply the sum of the pairwise scores over all the columns in the alignment. Similarly, a so-called *linear* gap cost (i.e., -8) is added for each column where a gap appears, including adjacent columns.[2] With this scoring scheme, the alignments in Figures 71(b) and 71(c) have scores of 1 and -5, respectively, thereby correctly discriminating between the case of two related sequences (b) and the case of two unrelated sequences (c). Indeed, positive scores indicate that the sequences are more likely in meaningful alignments than in random alignments, while negative scores indicate just the opposite. However, in general, no such scoring scheme is guaranteed to reliably discriminate between meaningful and spurious alignments in all cases. An optimal alignment is only as good as the scoring scheme used to evaluate it.

6.4 Pairwise sequence alignment

From now on, we take for granted the scoring scheme described in the previous section and turn our attention to algorithms used for finding an alignment of two sequences that has a maximum score according to this scheme.

A standard way of building a complete alignment of two sequences is to use optimal alignments of smaller sub-sequences. This is the underlying principle for a *dynamic programming* algorithm called the Needleman-Wunsch algorithm [125, 55]. We explain the behavior of this algorithm using the example in Figure 73(a). Let $C(i, j)$ denote the score of the optimal alignment of the two sequences made up of the first i letters of the first sequence and the first j letters of the second sequence, respectively. Therefore, $C(0, 0) = 0$ since the corresponding alignment is empty, and $C(4, 4)$ is the score of the optimal alignment of the two complete sequences. In general, the value of $C(i, j)$ can be computed using the values of $C(i - 1, j - 1)$, $C(i - 1, j)$ and $C(i, j - 1)$. For example, suppose we want to compute $C(1, 1)$, the score of the optimal alignment of the first letter in each sequence (i.e., 'D'). There are only three ways to build such an alignment using smaller alignments. First, the empty alignment (with zero column) can be extended to a one-column alignment that aligns both first letters with no gap (see Figure 73b). The score of this alignment is $C(0, 0) + \text{PAM250}[D][D] = 0 + 4 = 4$. Second, the one-column alignment with score $C(1, 0)$ can

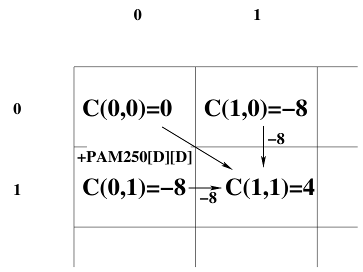
[2]In some cases, an *affine* gap score is used, in which long contiguous insertions or deletions are penalized less than the gap cost times the number of consecutive gaps.

Sequence 1: DQLF   Sequence 2: DNVQ

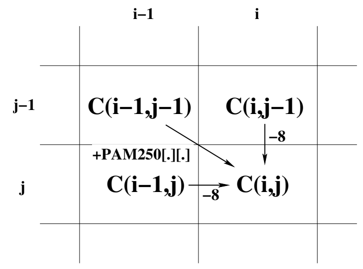
a) Two sequences to be aligned

$\{\text{empty}\} \Rightarrow \begin{matrix}\text{D}\\ \text{D}\end{matrix}$ $\begin{matrix}\text{D}\\ -\end{matrix} \Rightarrow \begin{matrix}\text{D} & -\\ - & \text{D}\end{matrix}$ $\begin{matrix}-\\ \text{D}\end{matrix} \Rightarrow \begin{matrix}- & \text{D}\\ \text{D} & -\end{matrix}$

b) No gap inserted   c) Gap in the first sequence   d) Gap in the second sequence

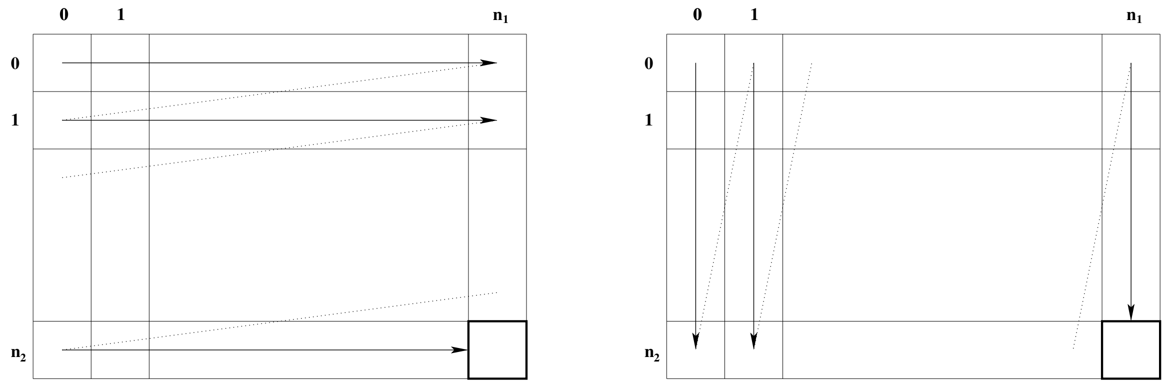


e) $C(1,1) = \max(C(0,0) + \text{PAM250}[D][D], C(0,1) - 8, C(1,0) - 8) = \max(0 + 4, -8 - 8, -8 - 8) = 4$



f) $C(i,j) = \max(C(i-1,j-1) + \text{PAM250}[l_i^1][l_j^2], C(i-1,j) - 8, C(i,j-1) - 8)$

Figure 73: One step in the alignment of two sequences



a) Row by row b) Column by column

Figure 74: The Needleman-Wunsch dynamic programming algorithm

be extended to a two-column alignment that aligns the 'D' in the second sequence with a gap (see Figure 73c). The score of this alignment is $C(1,0) + (-8) = -8 - 8 = -16$, where -8 is the gap cost. Third, the one-column alignment with score $C(0,1)$ can be extended into a two-column alignment that aligns the 'D' in the first sequence with a gap (see Figure 73d). The score of this alignment is $C(0,1) + (-8) = -8 - 8 = -16$. Finally, the optimal alignment of the first letter in each sequence is the one with the largest score, namely the one-column alignment that matches 'D' with 'D', since 4 =max$(4, -16, -16)$.

In general, $C(i,j)$ is calculated as the maximum of three numbers: 1) the sum of $C(i-1, j-1)$ and the PAM250 score of aligning l_i^1 (i.e., the i^{th} letter in the first sequence) and l_j^2 (i.e., the j^{th} letter in the second sequence); 2) the sum of $C(i-1, j)$ and the gap cost; and 3) the sum of $C(i, j-1)$ and the gap cost. The score function $C(i,j)$ is stored as a two-dimensional matrix, with the number i of letters in the first sequence serving as index in the horizontal dimension and the number j of letters in the second sequence serving as index in the vertical dimension. Figure 73e depicts the calculation of $C(1,1)$ using this two-dimensional representation. Figure 73f depicts the general case of $C(i,j)$.

The Needleman-Wunsch dynamic programming algorithm fills up the two-dimensional matrix C either row by row or column by column, as depicted in Figure 74. The score of the optimal alignment of the two original sequences is the last number calculated, that is, the number in the bottom-right cell of the matrix. The result of applying the Needleman-Wunsch algorithm to the example in Figure 73a is shown in Figure 75. Note that each number in the first column (the first row) is a multiple of the gap cost since each cell in this column (row) corresponds to inserting one

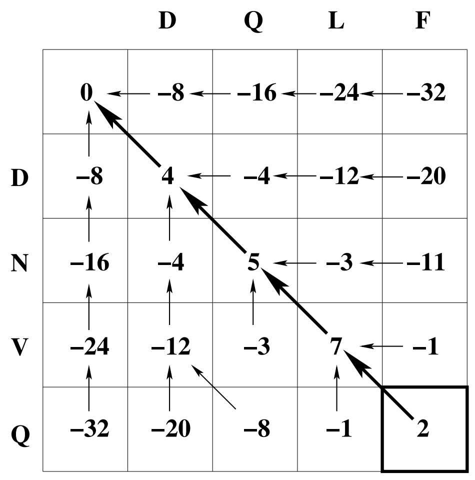


Figure 75: Computing the optimal alignment of the two sequences in Figure 73a

more gap into the first (second) sequence. Also, an arrow leading out of each cell c (except from the top-left one) points to the cell that gave the maximum value when computing the score for c (ties are broken randomly).

Conceptually, building an alignment means starting from the top-left cell (with score 0) and repeatedly moving to the right, downward, or down and to the right, until the bottom-right cell is reached. Each move corresponds to adding a new column to the alignment, starting from the empty alignment. A horizontal move to the right means consuming the next letter in the first sequence (and inserting a gap). A vertical move downward means consuming the next letter in the second sequence (and inserting a gap). A diagonal move corresponds to consuming a letter in each of the two sequences (and inserting no gap). The optimal alignment can be reconstructed from the matrix by following the pointers backward from the bottom-right cell to the top-left cell (these pointers are in bold in the figure). In our example, all the moves are on the diagonal. Therefore, the optimal alignment of the two sequences contains no gap and has four columns. Its score is equal to 2.

6.5 Multiple sequence alignment (MSA)

So far, we have only considered pairwise alignments. We now discuss the multiple sequence alignment (MSA) problem which consists of finding the optimal alignment (i.e., an alignment with the largest score) of n ($n \geq 2$) sequences.

The general MSA problem is a simple extension of the case $n = 2$. To align n sequences means

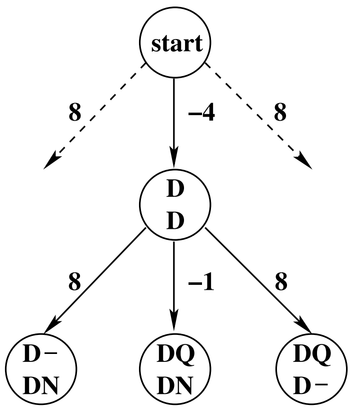


Figure 76: Search tree for the 2-dimensional MSA problem in Figure 73a

to write them down, one under the other, so that each letter belongs to a column. All the letters in a column are said to be aligned. Gaps can be inserted at any position in any of the sequences, as long as no column of the alignment is completely filled with gaps. In other words, each column must consume one letter in at least one sequence. For example, Figure 78b depicts one possible alignment of the three sequences listed in Figure 78a.

The method for computing the score of an alignment is generalized as follows. The total score remains the sum of the scores of all the columns in the alignment. In other words, we continue to make the assumption that random mutations at different positions in a sequence are independent from one another. However, the score of each column is now computed as the sum of the scores of all the pairs of letters in the column. The gap is treated as any other letter, except that any pair containing at least one gap has a score equal to the gap cost, that is, score(–, *any letter*) = score(*any letter*, –) = score(–, –) = -8.

The Needleman-Wunsch dynamic programming algorithm readily extends to the MSA problem. In the general case, the score matrix is an n-dimensional matrix. It is built in row-major or column-major order. The score of the alignment is computed last, in the corner cell opposite the start cell. Pointers are maintained in each cell in order to reconstruct the alignment backward from the corner cell.

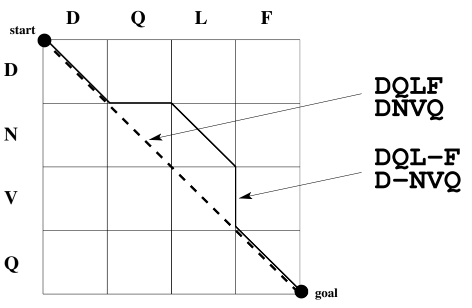


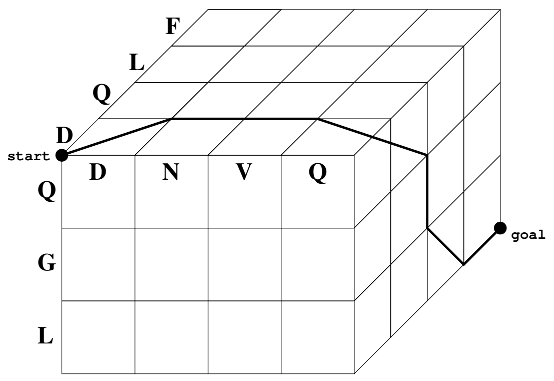
Figure 77: State space for the 2-dimensional MSA problem in Figure 73a

DQLF
DNVQ
QGL

a) Three sequences to be aligned

```
D – – Q – L F
D N V Q – – –
– – – Q G L –
```

b) One possible alignment



c) State space and path corresponding to the alignment

Figure 78: A 3-dimensional MSA problem

6.6 *The MSA problem as a shortest-path problem*

It is well-known that the MSA problem of finding a maximum-score alignment of n sequences is equivalent to the problem of finding a shortest path in an n-dimensional lattice [18]. In this formulation, each node in the graph corresponds to an alignment of some prefix of each sequence. The start node corresponds to the alignment of all empty prefixes. An edge corresponds to adding one column to the right of an existing partial alignment, and the cost of the edge is equal to the score of the added column multiplied by negative one. The goal node corresponds to the alignment of all complete sequences. In this formulation, finding a maximum-score alignment reduces to finding a minimum-cost path from the start node to the goal node. The state space for an n-dimensional MSA problem is an n-dimensional grid. We illustrate this reduction using cases with $n = 2$ and $n = 3$.

Let us reconsider the two sequences in Figure 73a. Assume that in the first step, we decide (out of three choices) to align the first letter 'D' of the first sequence with the first letter 'D' of the second sequence. The resulting alignment (or node) contains only one column. The cost of the edge leading from the start node to this node is -4. At the next step, there are again three choices. First, the letter 'Q' in the first sequence may be aligned with the letter 'N' in the second sequence; second, a gap may be inserted into the first sequence; third, a gap may be inserted into the second sequence. Figure 76 depicts this (partial) search tree. The graph corresponding to the search space for this problem is the 2-dimensional grid shown in Figure 77. Its 25 nodes are the intersection points of the vertical and horizontal lines defining the grid. The start and goal nodes are at opposite corners of the grid. The edges in this graph are the horizontal and vertical line segments connecting pairs of adjacent nodes, as well as the diagonal line segments (not shown in the figure) connecting the top-left corner of each grid cell to its bottom-right corner. The edges are directed from left to right, or from top to bottom, or both (for diagonal moves). Each alignment of the two complete sequences corresponds to a path in this graph from the start node to the goal node. Figure 77 shows two examples of such path-alignment pairs. In general, there is a one-to-one correspondence between the set of all possible alignments of the two sequences and the set of all paths from the start node to the goal node in this grid. Since the cost of each edge is equal to the negative value of the score of the corresponding letter pair (diagonal edge) or the gap cost (horizontal or vertical edge), a minimum-cost path determines an optimal alignment.

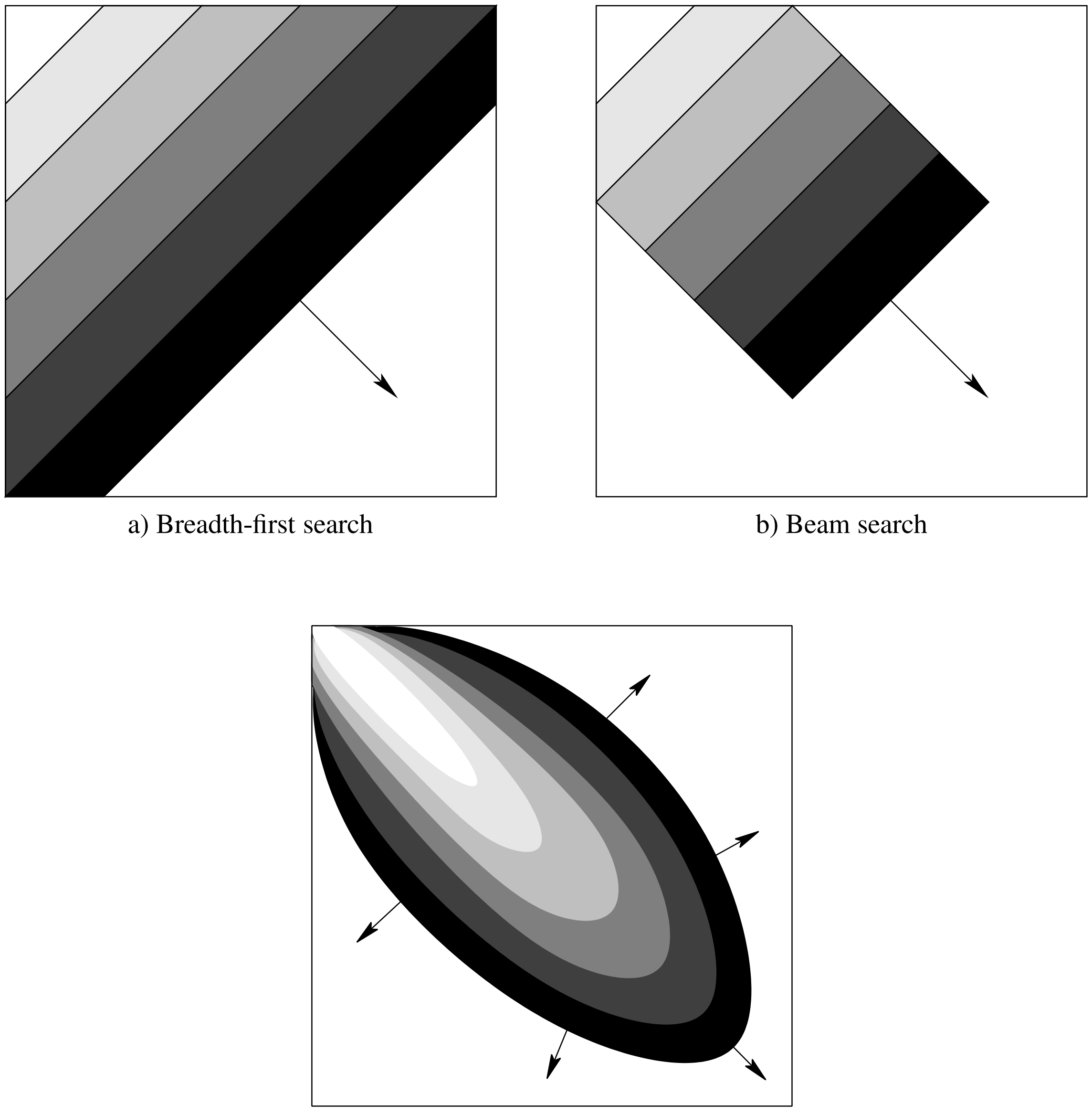
This one-to-one correspondence between paths and alignments extends to MSA problems with three sequences. In this case, the search space is a 3-dimensional grid or cube. Figure 78c shows the search space for the problem of aligning the three sequences in Figure 78a, as well as the path corresponding to the alignment in Figure 78b. In general, the problem of aligning n sequences can be reduced to the problem of finding a shortest path in an n-dimensional hypercube in which each dimension corresponds to one of the sequences to be aligned.

6.7 Solving the MSA problem with search algorithms

Since each MSA problem reduces to a shortest-path problem in a grid, we can use search algorithms to solve it. The defining characteristic of a search algorithm is the order in which it expands nodes. In this context, the Needleman-Wunsch dynamic programming algorithm can be construed as a search algorithm that expands nodes in a fixed order, namely either in row-major or column-major order (see Figure 74). This algorithm takes advantage of the domain-specific structure of the state space when ordering node expansions. But any node ordering strategy may be used.

For example, Figure 79a depicts a breadth-first search strategy for solving the MSA problem in the 2-dimensional case. In contrast to the Needleman-Wunsch algorithm, breadth-first search expands nodes layer by layer, where each layer is perpendicular to the main diagonal of the grid. The main drawback of both the Needleman-Wunsch algorithm and breadth-first search is that they generate the complete search space before finding an optimal alignment. Since both algorithms store every node they generate, and since there are on the order of l^n nodes in an n-dimensional MSA problem of length-l sequences, these algorithms run out of memory before finding a solution to large problems.

For this reason, heuristic search approaches are being used in the AI community to solve the MSA problem. Figures 79b and 79c depict the behavior of beam search and A* search in the 2-dimensional MSA problem. Both heuristic search algorithms are able to find an alignment without generating the whole state space. While A* is guaranteed to find an optimal alignment (when using an admissible heuristic function), beam search is not. However, beam search can explicitly control its memory consumption (via its beam width parameter B) and scales up to larger problems than


a) Breadth-first search

b) Beam search

c) A* search

Figure 79: Solving the MSA problem with search algorithms

A*, as we will show shortly. Both beam search and the A* algorithm need a heuristic function to guide the search. In the next section, we describe a powerful heuristic function for the MSA problem.

6.7.1 An admissible heuristic function for the MSA problem

The cost of an alignment is the sum of the costs of each column in the alignment. The cost of each column is in turn the sum of the costs of each pair of letters (or gaps) in the column. Therefore, the cost of an alignment is also equal to the sum of the costs of the alignments of all the pairs of sequences in the alignment. For example, the cost of an alignment of three (four) sequences is the sum of the costs of three (six) pairwise alignments, since there are three (six) ways of pairing three (four) sequences.

Since the cost of an n-dimensional alignment is the sum of the costs of all pairwise alignments, and since the cost of any pairwise alignment is by definition larger than or equal to the cost of the optimal pairwise alignment, the sum of the costs of the optimal pairwise alignments never overestimates the cost of the optimal n-dimensional alignment. This observation is the basis for a well-known admissible heuristic function for the MSA problem [158], which we now describe using the 3-dimensional MSA problem shown in Figure 78a.

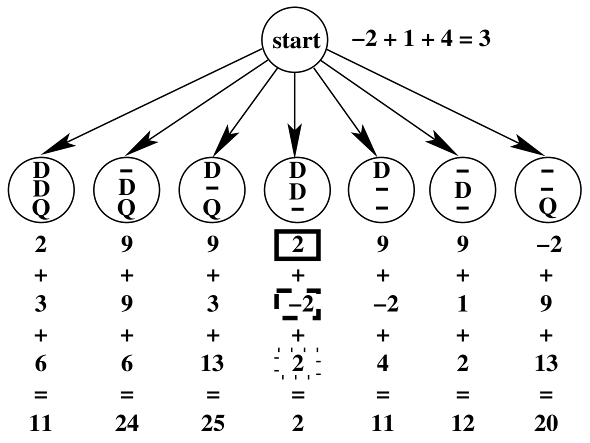
The first step in solving this MSA problem is to compute the optimal cost of each pairwise alignment of the three sequences, as shown in Figure 80a. The h-value of the start node for this problem is equal to 3 (see Figure 80b), which is the sum of the costs in the bottom right cell in each of the three matrices in Figure 80a. The start node has seven successor nodes, since there are seven possible first columns in a three-way alignment, depending on whether the first letter of each sequence is part of the column or not. In order to compute the h-value of these successor nodes, we need to know the costs of the optimal pairwise alignments of the remaining suffixes of the three sequences, that is, the sequences minus the letter consumed in the first column, if any. These costs can be read off the pre-computed matrices shown in Figure 80a. Notice that these matrices compute the costs of aligning the *reversed* sequences, not the original sequences. We illustrate the h-value computation using the middle successor node in Figure 80b, which turns out to have the smallest h-value.

		F	L	Q	D
	0	8	16	24	32
Q	8	5	10	12	20
V	16	9	3	11	14
N	24	17	11	$\boxed{2}$	9
D	32	25	19	9	−2

		F	L	Q	D
	0	8	16	24	32
L	8	−2	2	10	18
G	16	6	2	3	9
Q	24	14	8	$\boxed{-2}$	1

		Q	V	N	D
	0	8	16	24	32
L	8	2	6	14	22
G	16	9	3	6	13
Q	24	17	11	$\boxed{2}$	4

a) Pairwise cost matrices



b) Search tree with h-values

Figure 80: Computing the h-values for the MSA problem

Like for every node in this problem, the h-value is the sum of three costs. First, we consider the pairwise alignment of the first two sequences. Since the first letter of each sequence (i.e., ‘D’) is consumed in the first column of the alignment, we estimate the cost of aligning the suffixes of the sequences, starting with the letters ‘Q’ and ‘N’, respectively. The cost of an optimal alignment of these sub-sequences is stored in the marked cell of the leftmost matrix in Figure 80a. Second, we consider the pairwise alignment of the first and third sequences. Since a gap is inserted in the third sequence, its first letter is not consumed. The cost of an optimal alignment of the first sub-sequence and the complete third sequence is stored in the marked cell of the middle matrix in Figure 80a. Third, we consider the pairwise alignment of the second and third sequences. Since a gap is inserted in the third sequence, its first letter is not consumed. The cost of an optimal alignment of the second sub-sequence and the complete third sequence is stored in the marked cell of the rightmost matrix in Figure 80a. Finally, the h-value of the middle successor of the start node is equal to $2+(-2)+2 = 2$. A similar h-value calculation is used for all generated nodes in the search tree. When a goal node is reached, all letters in each sequence have been consumed. Therefore, the remaining sub-sequences are empty and the values looked up in the three matrices are all equal to zero (the value in the top left cell). The h-value of the goal node is thus zero, as must be the case for an admissible heuristic function.

The heuristic we have just described is a memory-based heuristic [23, 181]. Unlike (memoryless) heuristic functions, such as the Manhattan distance used in the N-Puzzle that compute the h-value from scratch for each state description, memory-based heuristics use lookup tables stored in memory. The pairwise-cost heuristic function for the MSA problem must store in memory all the pairwise cost matrices. However, the time and space requirements for this heuristic are negligible when solving large alignment problems. It is the most popular heuristic function used by the AI search community [74, 124, 178, 67] and we use it in our experiments. More informed heuristics exist that store n-dimensional alignments ($n \geq 3$) instead of pairwise alignments [120]. However, they are much more memory intensive and cannot be stored in full. The mechanisms needed to combine their partial storage with on-demand computation make them much harder to use.

6.7.2 Solving the MSA problem with existing variants of A*

Over the past decade, several variants of the A* algorithm have been used to solve the MSA problem [74, 75, 124, 113, 109, 172, 181, 182]. Since the pairwise-cost heuristic is admissible, its use in the A* algorithm guarantees that the alignment found is optimal. However, the main drawback of A* is that it runs out of memory on large alignments because it stores in memory all the nodes it generates. Much of the research on the application of heuristic search to the MSA problem has focused on reducing the memory consumption of A*, by decreasing the size of either its OPEN list or its CLOSED list. A third approach is based on linear-space search. We describe these approaches in turn.

One characteristic of the MSA problem is its large branching factor. In fact, its branching factor is exponential in the number n of sequences to be aligned: since there are 2^n-1 distinct ways of inserting between 0 and $n - 1$ gaps into each n-dimensional column of the alignment, the number of successors of a node in the MSA problem is $O(2^n)$. For this reason, the nodes in the OPEN list (i.e., the set of nodes that have been generated but not yet expanded) use up a significant percentage of the memory needed by A*. To address this problem, [172] proposed a variant of A* with partial expansion in which only some of the newly-generated successor nodes are added into the OPEN list, namely those with the best estimated cost. The advantage of this approach is that the goal node may be found without having to store many of the nodes in the OPEN list with high f-values.

The second set of approaches aim at reducing the size of the CLOSED list, that is, the set of nodes that have already been expanded. In A*, the CLOSED list has two main functions. First, it serves to construct the solution path (that is, the optimal alignment in the case of MSA) by following pointers backwards from the goal node to the start node. Second, it serves to avoid re-expanding nodes that can be reached from the start node along several paths. Because of the lattice structure of the search space for the MSA problem, there are many paths going through each node. If multiple paths are not detected, each node in the MSA problem may be expanded an exponential number of times. [102, 109] proposed a divide-and-conquer approach that does not store the CLOSED list at all. Instead, once the goal node is found, the solution path is constructed by breaking the current MSA problem into two sub-problems and solving them recursively. Node re-expansions are prevented by maintaining, for each open node, pointers to the incoming edges that have already been visited, as well as by storing additional "dummy" open nodes. In contrast, instead of discarding the whole CLOSED list, [179] proposed to store a fraction of it. This sparse representation of the CLOSED list has several advantages. First, the resulting algorithm behaves like A* when the latter has enough memory to complete the search. Second, the path-constructing phase is more efficient than the recursive approach. Third, the sparse memory-approach can easily be combined with approaches that reduce the size of the OPEN list.

Yet a different approach for reducing the memory consumption of A* is to use linear-space algorithms, such as IDA* [96]. Such algorithms use a depth-first search strategy and therefore only store in memory the path currently being explored. When the goal is found, the solution path is easily reconstructed (again backwards) using the calling stack. Unfortunately, because they do not store generated nodes, they are extremely slow on lattice-like search spaces. [124] applied a new variant of IDA* called *SNC* (for *Stochastic Node Caching*) to the MSA problem. In order to prevent some of the node re-expansions of IDA*, SNC stores a fraction of the generated nodes, namely those that are most likely to be visited often. The main goal of SNC is thus to speed up IDA* by reducing the time spent on re-expansions.

Because of the exponential size and the lattice structure of the MSA search space (and despite the existence of the well-informed pairwise-cost heuristic function), existing variants of A* can only find optimal alignments of n real proteins (of average length equal to approximately 500 amino acids) for values of n that are smaller than 10 on typical current machine configurations. This limitation of optimal heuristic-search-based MSA solvers is one motivation for sacrificing solution optimality in order to scale up to larger problems. Another motivation is the fact that the scoring scheme for the MSA problem is only an approximation of the complex knowledge used by biologists to estimate the phylogenetic significance of an alignment. Since optimality is measured using an imperfect scale, it may be more practical to find good enough or even near-optimal solutions to larger MSA problems than to insist on finding strictly optimal alignments. We follow this practical approach for the experiments reported in this chapter, in which we apply our anytime approximation algorithms to the MSA problem. In the next section, we compare the performance of ABULB to a variant of Weighted A* (or WA* [132, 131]) called Anytime A* (ATA* [178]). ATA*, like WA*, puts more weight on the h-value than on the g-value when computing the f-value of each node. WA* scales up to larger problems than A* at the expense of solution quality. ATA* simply extends WA* in an anytime fashion by continuing its execution even after a goal is found.

6.8 Solving the MSA problem with ABULB

In this section, we motivate and describe the application of ABULB to the MSA problem. We then report our empirical study.

6.8.1 Adapting ABULB to the MSA problem

Since heuristic search algorithms are directly applicable to the MSA problem, we can directly test our variants of both WA* and ABULB on this problem. Being interested in the scaling behavior of our algorithms, we evaluate them under tight memory constraints. In our experiments, each algorithm can only store up to one million nodes in memory. Because variants of WA* are not memory-bounded, they run out of memory for large enough problems. In our empirical setup, they cannot align more than $n = 10$ real proteins whose average length is $l \approx 500$. First, observe that the minimum length of the alignment, measured as the number of columns, is equal to l. Therefore, the number of nodes along any solution path is greater than or equal to l. Second, since the branching factor in the MSA problem is $\leq 2^n - 1$, each node on the solution path of an 11-dimensional problem has up to 2,047 successors. Even with a perfect heuristic function (i.e., when the WA* search goes straight to the goal), such a linear tree of depth d contains more than $2000d$ nodes. In conclusion, even in the best case (i.e., with perfect h-values), WA* runs out of memory before reaching the goal, which lies at depth $d \geq 500$. This limitation is shared by all of our variants of WA*. One solution to this problem is to avoid storing all successor nodes. One such approach, proposed in [172], only partially expands nodes. More precisely, it fully expands each node but only stores successor nodes with small f-values. Another solution is to bound the number of stored nodes at each depth in the tree. This is the basis for beam search. Therefore, it is natural to evaluate our extension of beam search, called ABULB, on the MSA problem, since we expect ABULB to scale up to much larger problems than any variant of WA*.

ABULB is an anytime variant of BULB that continues searching after a goal is found. BULB

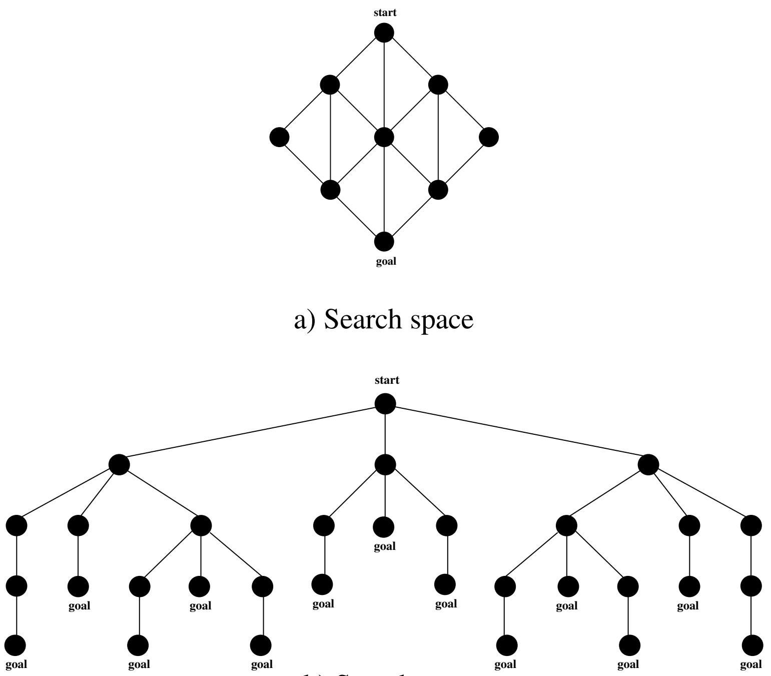


Figure 81: Search space and corresponding search tree for an MSA problem with $n = l = 2$

itself is a backtracking variant of beam search. Finally, beam search is a memory-bounded variant of breadth-first search. In the previous chapter, we applied ABULB to domains with uniform action costs, in which the cost of a path is equal to its length. The MSA problem does not share this property since each edge cost in its search space is the sum of pairwise costs, each of which can take several integer values between -8 and 17 (see the PAM250 matrix in Figure 72). As a result, nodes at a given depth of the search tree typically have different g-values. The only modification we need to apply to ABULB is to order the nodes within each layer by increasing f-values, instead of h-values. Both the g- and h-value of each node are calculated using the pre-computed cost matrices of each possible pairwise alignment of the n sequences in the problem.

Running ABULB requires choosing a value for its only parameter B, the beam width. In general, ABULB with a small value for B returns a solution quickly, if it finds one. In some of our domains in Chapter 4, too small values for B led to dead-ends because the search would run out of new successor nodes to visit. In the MSA problem, this situation cannot occur. The search space for this problem is a direct acyclic graph (or *dag*). Figure 81a depicts the dag for a 2-dimensional problem with sequences of length two (i.e., $n = l = 2$). Note that all edges in the figure are really directed edges: they can only be traversed downward. This search space contains no cycles. The search always progresses downward toward the goal. Figure 81b depicts the complete search tree corresponding to this search space, assuming that duplicate nodes are not detected. Since there exist many paths from the start node to most nodes in the search space (i.e., the dag contains many transpositions), the number of nodes in the search tree (31) is significantly larger than the number of distinct nodes in the search space (9). Despite this combinatorial explosion, the search tree for the MSA problem has two nice properties. First, it is finite and its maximum depth, equal to $n \times l$, is known a priori. This upper bound makes it easy to estimate the largest value of B for which ABULB is still guaranteed to find a goal without running out of memory. Second, every leaf of the tree is a goal. This property is sufficient to guarantee that ABULB with $B = 1$ will always find the first solution path quickly since reaching the first goal involves no backtracking. In the anytime context, we are interested in finding the first solution as fast as possible and we therefore set the initial value of B equal to one.

6.8.2 Empirical evaluation

We evaluate the performance of ABULB using a set of real biological sequences. We first describe our empirical setup and then report and discuss our results.

6.8.2.1 Empirical setup

BAliBASE (Benchmark Alignment dataBASE) is a database of multiple sequence alignments that are categorized by difficulty.[3] The difficulty of an MSA problem depends on the number n of sequences, their average length l, and the degree of similarity (or the percentage of residue identity) among the sequences, among other factors. To evaluate the scaling behavior of ABULB, we select one difficult set of sequences in BAliBASE and align sub-sets of it of increasing size n. The *myrosinase* set belongs to the “Reference 3/long” category.[4] It contains 21 sequences of average length equal to 405 and of maximum length equal to 482. This MSA problem is considered difficult because of its large l value and because its percentage of residue identity is less than 25%. In our experiments, n varies from 8 to 13. For each value of n, five sub-sets of n sequences each are

[3]BAliBASE is accessible at http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE/.

[4]The myrosinase set is available at http://www-igbmc.u-strasbg.fr/BioInfo/BAliBASE/ref3/test/2myr_ref3.html.

selected at random among the twenty-one sequences in the set. Thus each reported measurement represents the performance averaged over five data points.

We run both ABULB 1.0 and ABULB 2.0 with $B = 1$ and with enough memory to store up to one million nodes at a time. Each program runs for twelve minutes (or 720 seconds) on each sub-set of sequences. We use the PAM250 substitution matrix (see Figure 72) to compute the g- and h-value of each node. Since the elements of this matrix constitute a measure of similarity to be maximized, we use their negative values in our shortest-path problem reformulation. Furthermore, since the matrix contains both positive and negative integers, we add the constant value 17 to each element so that all edge costs and h-values remain non-negative integers. Note that this simple algebraic manipulation changes the scoring function in non-trivial ways. For example, two alignments of the same sequences but of different lengths see their score increase by the addition of a multiple of 17 whose magnitude depends on the number of columns in the alignment.

We also compare ABULB to ATA* which, to the best of our knowledge, is the only other anytime heuristic search algorithm to have been applied to the MSA problem [178]. ATA* behaves like WA* until it finds a goal and then continues running with the same value of W until it terminates (when the smallest sum of the g- and h-value of any node in the OPEN list is greater than or equal to the dynamic upper bound, ATA* has found the optimal path and can stop) or runs out of memory. Since the performance profiles in the next section report the decrease in solution cost relative to the cost of the first solution found, the comparison between different algorithms is only meaningful if they find initial solutions with similar costs. This is not an issue when comparing ABULB 1.0 and ABULB 2.0 since both behave identically to BULB with $B = 1$ until the first solution is found. For ATA*, we set the value of W equal to 1.1. With this parameter setting, ATA* and ABULB find initial solutions whose average costs never differ by more than 0.05%. The reason why such a small W is acceptable is that the heuristic values available in the MSA problem are extremely well-informed. In fact, the authors of [178] use a value of W that is even closer to one.

6.8.2.2 *Empirical results*

Figure 82 through 87 are the performance profiles of ABULB 1.0, ABULB 2.0, and ATA* obtained on sets of 8 through 13 proteins. The x- and y-axes in the figures have the same meaning as those

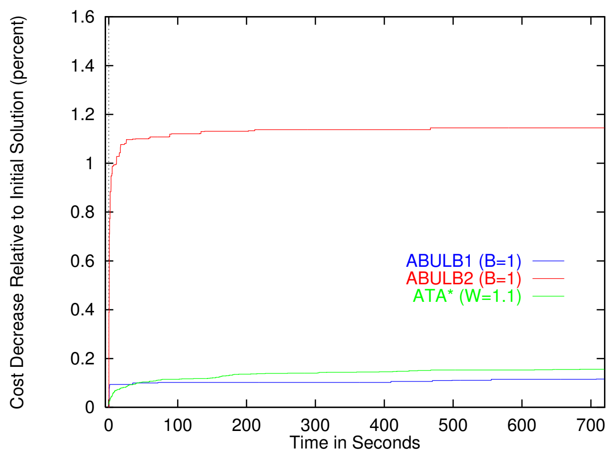


Figure 82: MSA problems with 8 proteins

for the performance profiles in Chapter 5.

First, we observe that the range of values on the y-axis (see, for example, Figure 82) is much smaller than the corresponding ranges in the performance profiles of Chapter 5. This difference is not symptomatic of a poor performance of the algorithms in this domain. Instead, the small range is due to the fact that the cost of the first solution found is close to optimal. In fact, the average solution cost initially found by each algorithm is never more than 5% from optimal.[5] This result is not surprising in the case of ATA*, since a weight of 1.1 guarantees that the solution cost will never be more than 10% from optimal. However, no such guarantee exists for BULB (or, equivalently, the first iteration of ABULB). The fact that BULB with $B = 1$ finds near-optimal solutions means that the heuristic values for this domain are quite accurate, even for this relatively hard set of instances.

Second, we observe that, even though they start with a close-to-optimal solution, both ABULB 1.0 and 2.0 are able to decrease the solution cost over time. However, with all problem sizes, the decrease is much larger for ABULB 2.0 than for ABULB 1.0. This trend mirrors the one observed with the benchmark domains of Chapter 5. Our conclusion in that chapter was that ABULB 2.0, by quickly increasing the value of B, makes better use of the available memory than ABULB 1.0. In the case of the MSA domain, the (relatively) poor performance of ABULB 1.0 can further be explained by the structure of the search space. With a constant B-value of 1, ABULB 1.0 visits the search space without detecting duplicates. Its search tree is thus much larger than the

[5]Since we do not know the optimal cost, we (under)estimate it using the h-value of the start state.

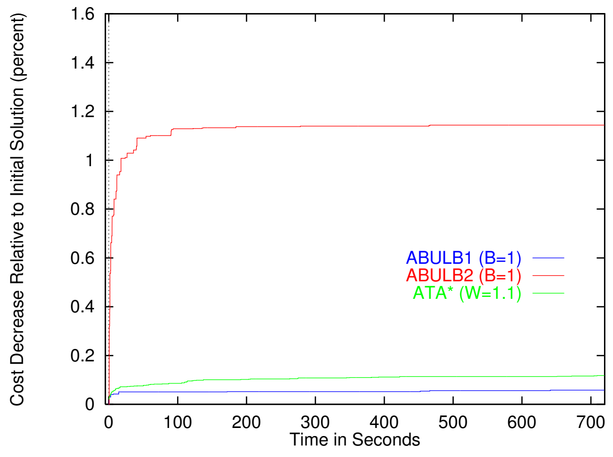


Figure 83: MSA problems with 9 proteins

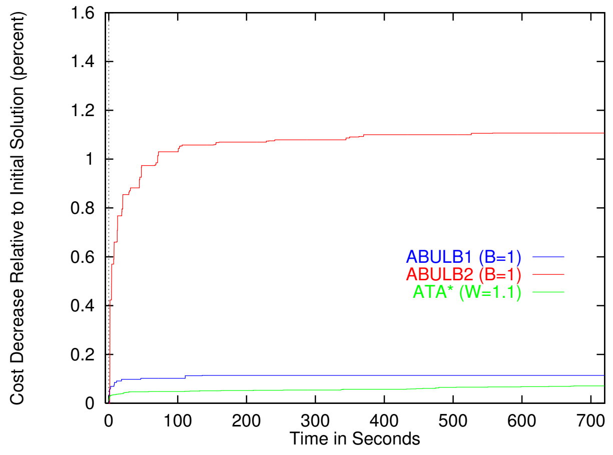


Figure 84: MSA problems with 10 proteins

search space (see Figure 81). In contrast, ABULB 2.0 with larger and larger B-values is able to eliminate more and more transpositions and is thus faster at visiting a larger set of paths to the goal.

ABULB 1.0 with $B = 1$ is similar to another anytime algorithm called Depth-First Branch-and-Bound (or DFBnB [112, 175]). DFBnB is to depth-first search as ABULB 1.0 is to BULB. DFBnB, like ABULB 1.0, simply keeps running after a solution is found and maintains a dynamic upper bound on the solution cost in order to prune more and more nodes as the cost of the solutions found decreases. Therefore, ABULB 1.0 with $B = 1$ only differs from DFBnB in its backtracking mechanism. Given the typical superiority of limited-discrepancy-based backtracking (used by ABULB) over chronological backtracking (used by DFBnB), we expect the performance profile of DFBnB to lie below that of ABULB 1.0. This comparison remains to be performed in future work.

Finally, the performance profiles show that both versions of ABULB scale up better than ATA*

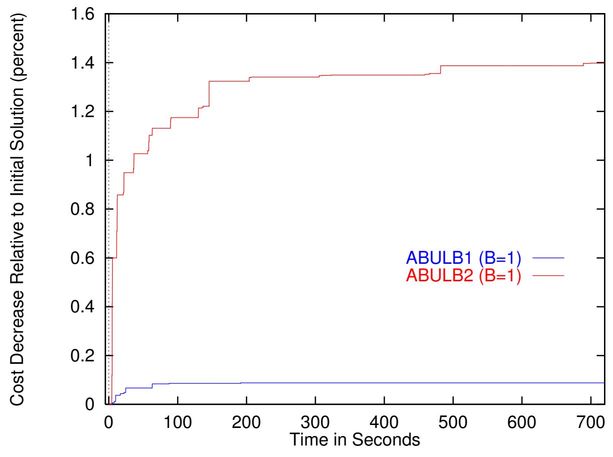


Figure 85: MSA problems with 11 proteins

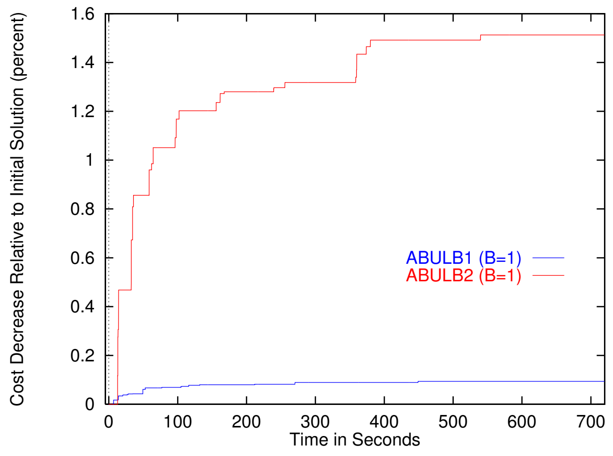


Figure 86: MSA problems with 12 proteins

since the latter algorithm can only solve the MSA problems with n smaller than or equal to 10. The absence of a curve for ATA* in Figures 85 through 87 reflects the fact that ATA* runs out of memory before finding a first solution in at least one of the instances corresponding to $n = 11, 12$, or 13. When it does find a solution before running out of memory, ATA* reduces the solution cost at a rate similar to that of ABULB 1.0.

To summarize our results, the ABULB algorithms scale up to larger MSA problems than the ATA* algorithm, and ABULB 2.0 reduces the solution cost faster than ABULB 1.0 in our test cases.

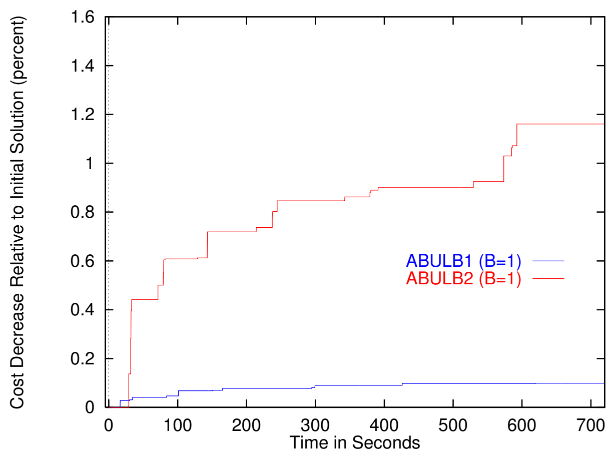


Figure 87: MSA problems with 13 proteins

6.9 Conclusion

In this chapter, we have used the Multiple Sequence Alignment (MSA) problem as an additional benchmark domain for ABULB. We explained how the MSA problem of maximizing the similarity score of an alignment of n biological sequences reduces to the shortest-path problem of minimizing the cost of a path between two opposite corners of an n-dimensional hypercube.

Our empirical results show that, on our MSA test problems, both ABULB 1.0 and ABULB 2.0 scale up to larger problems than Anytime A*, another anytime heuristic search algorithm based on WA*. Our results also show that ABULB 2.0 reduces the solution cost more quickly than ABULB 1.0.

While our goal in this chapter was not to improve the state of the art in this domain, this is a worthwhile goal for the future. The first step in this direction requires the study of the relevant tools already developed within the bioinformatics community. This is no small undertaking given the vast array of existing algorithms for this problem. However, someone familiar with this tool box should be able to take advantage of it and to compound its power with ideas from heuristic search. The use of heuristics to guide the search could likely be applied with benefit in the context of other formalizations of this problem.

Most of the approaches originating in the bioinformatics community are approximations and do not use the shortest-path formulation. For example, a popular approach is called *progressive* *multiple alignment* because it progressively builds the alignment, starting with a pairwise alignment, by adding one complete sequence at a time to the unmodified alignment obtained so far [40]. This approach differs from the heuristic search approach since it adds one complete sequence at a time to an existing alignment of a growing subset of sequences, while heuristic search builds an alignment by adding one column at a time to all sequences from left to right. Progressive multiple alignment methods have been enhanced using statistical profiles that discover and use position-dependent information, in a way that pairwise alignments cannot do. An example of this extension is the CLUSTALW program [163]. Other statistical, profile-based methods exist (e.g., [33]). Finally, because of the high computational complexity of the global MSA problem with gaps, a lot of effort in bioinformatics has been spent on computationally more tractable yet very useful variants of it. These led to such algorithms as BLAST and its variants, that are very fast at finding local matches in large databases [2, 3]. An interesting direction for future work is to investigate the possible cross-fertilization of ideas between these progressive and local approaches on the one hand, and heuristic search approaches on the other.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK IN OFFLINE SEARCH

This chapter concludes the second and last part of the dissertation by summarizing our contributions to offline search and outlining some directions for future work in this area.

7.1 Our contributions to offline search

In the second part of this dissertation (Chapters 3 through 6), we have improved on state-of-the-art offline heuristic search methods. Our primary goal was to design new heuristic search algorithms that scale up to larger problems. Our secondary goal was to find low-cost solutions in a reasonable amount of time. This second part of the dissertation is itself split into two parts. First, Chapters 3 and 4 introduced two new one-shot heuristic search algorithms (namely MSC-KWA* and BULB). Second, Chapters 5 and 6 introduced a new family of anytime heuristic search algorithms (namely the variants and combinations of ABULB and ITSA*) and their application to the multiple sequence alignment problem. Figure 1 shows the position of our new algorithms in the space of heuristic search algorithms, while Table 2 and the accompanying discussion demonstrate that we have achieved both our primary and secondary goals. We now summarize in turn our contributions to one-shot and anytime search.

7.1.1 Our contributions to one-shot search

In Chapter 3:

- We provided stronger empirical support for the improved scaling behavior of MSC-WA* over WA* by 1) measuring its search effort, solution cost and memory consumption while varying both the size of the commitment list and the relative weight on the h-values (the original study kept this weight fixed), and by 2) testing it in two additional domains.

- We provided stronger empirical support for both the speedup and reduced memory consumption (for a given solution cost) KWA* over WA* by 1) using a slightly improved implementation of KWA* that does scale it up to the 35-Puzzle (the original implementation did not) and 2) by testing it in two additional domains.

- We showed empirically that KWA* and MSC-WA* improve WA* in two orthogonal ways. Because of its stronger breadth-first search component (i.e., diversity), KWA* improves its solution quality for a given level of memory consumption. Because of its stronger depth-first search component (i.e., commitment), MSC-WA* improves its memory consumption.

- We combined the orthogonal and complementary ideas of diversity and commitment, resulting in the MSC-KWA* algorithm. We showed empirically that MSC-KWA* scales up to larger domains than either KWA* or MSC-WA*. For example, in our experimental setup, MSC-KWA* scales up to the 48-Puzzle, while KWA* and MSC-WA* only scale up to the 35-Puzzle. WA* only scales up to the 24-Puzzle.

- We discussed the similarities between MSC-KWA* and beam search. This discussion promoted the role of diversity (i.e., the parallel expansion of the set of candidate nodes) in the performance of beam search, while the initial motivation for beam search was the idea of commitment. As an aside, the similarities between MSC-KWA* and beam search also motivated our focus on beam search in the following two chapters.

In Chapter 4:

- We combined the ideas of beam search and backtracking. We showed that, when beam search runs out of memory, backtracking is a relatively simple way (when compared to existing memory-bounded algorithms such as SMAG*) of purging nodes from memory in order to continue the search in another direction. Furthermore, applying backtracking to beam search reduces the runtime over depth-first-based memory-bounded searches (such as IDA*, RBFS or LDS) because, as B increases, beam search eliminates more and more transpositions (i.e., it detects more and more distinct paths to the same node) and thus reduces the associated node-regeneration overhead. This first contribution is refined into the following two contributions comparing the effect on the runtime of two different ways of performing backtracking with beam search.

- We applied chronological backtracking to beam search, resulting in the DB algorithm. We showed empirically that the runtime of DB is typically unacceptably large. This confirms the well-known observation that chronological backtracking is not efficient when the heuristic is misleading in nodes that are far from the goal, since chronological backtracking retracts decisions lower in the tree (where the heuristic is often better informed) before it does so closer to the root (where the heuristic is often less informed).

- We applied limited-discrepancy-based backtracking to beam search, resulting in the BULB algorithm. Like DB, BULB is memory-bounded and can solve larger problems than beam search for each value of B. Furthermore, by varying the value of B, BULB generalizes both limited-discrepancy search (when $B = 1$) and breadth-first search (when $B = \infty$). We showed empirically in three benchmark domains that BULB not only scales up to large problems but also exhibits small runtimes (on the order of seconds or minutes) in these problems.

7.1.2 Our contributions to anytime search

In Chapter 5:

- We introduced a new local path optimization called ITSA*. ITSA* is based on a search strategy called *iterative tunneling*. ITSA* is also an anytime algorithm and we showed that it reduces the solution cost output by BULB in the 48-Puzzle and the Rubik's Cube.

- We described and empirically evaluated different ways of transforming BULB into an anytime algorithm, both with and without dynamic beam widths. We showed empirically that restarting BULB with increasing values of the beam width (a variant called ABULB 2.0) leads to a faster reduction in the solution cost than simply letting BULB run with the same (constant) beam width after a goal is found (a variant called ABULB 1.0). ABULB 2.0 is able to take advantage of all the available memory, while ABULB 1.0 uses less and less memory as the solution cost decreases.

- To the best of our knowledge, our way of combining ABULB with ITSA* is the first successful application of local (or neighborhood) search to the shortest path problem. This combination yields a larger reduction in the solution cost in the 48-Puzzle than the one obtained with either plain ABULB or ITSA*.

In Chapter 6:

- We used the multiple sequence alignment (MSA) problem in molecular biology as an additional benchmark domain for ABULB. We discussed the minor modifications needed for applying ABULB to this domain.
- Our empirical results showed that, on our MSA test problems, both ABULB 1.0 and ABULB 2.0 scale up to larger problems than Anytime A*, another anytime heuristic search algorithm based on WA*. On problems that Anytime A* could solve, both variants of ABULB yielded a larger decrease in solution cost. Finally, ABULB 2.0 reduced the solution cost more quickly than ABULB 1.0.

7.2 Lessons learned and future work

In this section,we discuss three possible directions for future work.

7.2.1 Generalization of MSC-KWA* and beam search

In Chapter 3, we (unintentionally) came close to re-inventing beam search by combining the ideas of commitment and diversity and applying them to WA*. Indeed, it was this observation (together with the good performance of MSC-KWA*) that piqued our interest in beam search and eventually led to our work on BULB and ABULB. We now believe that beam search is at least as good as (and probably better than) WA* as a starting point for scaling up heuristic search. We hope that this research will spark some renewed interest in beam search.

Furthermore, in Chapter 3, we discussed one important difference between beam search and MSC-KWA*, namely the fact that the latter uses memory to store nodes in a reserve list while beam search does not. Thus, MSC-KWA* can later return to a discarded node, while beam search needs to find a new path to pruned nodes before expanding them. In this respect, the commitment (to a set of candidate nodes) is stronger in beam search than it is in MSC-KWA* given the same commitment level C. In fact, beam search and MSC-KWA* sit at two extremes on a spectrum of methods that use reserve lists of varying lengths R (namely, $R = 0$ for beam search and $R = \infty$ for MSC-KWA*). One explanation for the better scaling of beam search in the Rubik's Cube domain may be that MSC-KWA* runs out of memory because of the reserve list. Thus, a smaller value of R may improve scaling. However, this cannot be the whole explanation given the results summarized in Table 2 in the Towers of Hanoi domain. Making R a parameter of the search is one way to control the *strength* of the commitment.

Another dimension along which to contrast beam search and MSC-KWA* is their greediness (or equivalently their value for W). While MSC-KWA* can explicitly vary this value, beam search orders the set of candidate nodes according to h-values only (corresponding to $W = 1$). In contrast to the breadth-first-based version of beam search we used in this research, other variants exist in which nodes are ordered by increasing f-values (like in A*), corresponding to $W = 0.5$.

Yet another dimension of variation is the relation between the values of C and K. In breadth-first-based beam search, $C = K$, while MSC-KWA* only imposes that K be smaller than or equal to C.

In conclusion, we have discussed four components of a search strategy (namely, its greediness (W), its diversity (K), and its level (C) and strength (R) of commitment) which can be varied to cover a large number of variants of beam search and MSC-KWA*. The study of a general framework for this class of approaches promises to be an interesting direction for future work.

7.2.2 Application of neighborhood search to the shortest-path problem

Another lesson learned from this research is that local search in the space of solution paths is a promising avenue of research in anytime heuristic search. We have only scratched the surface with ITSA* and ABULB. For example, it makes sense to take advantage of existing techniques well-known to the Operations Research community. We discuss a few relevant issues.

First, we have used ABULB to select restart solutions when ITSA* reaches a local minimum. Another, standard approach in neighborhood search consists of randomizing the search. One possible way to randomize heuristic search is simply to add random noise to the h-values. This is expected to work well (even with noise of a small amplitude) in domains where many nodes have the same f-values due to the fact that both g- and h-values are integral and the number of distinct h-values is relatively small.

Second, the neighborhood searched by ITSA* is essentially constructed by a breadth-first search around a given solution path whose depth is limited only by the available memory. Other neighborhoods can of course be defined. One could perhaps use a uniform-cost search instead of a breadth-first search by taking into account edge costs instead of simply counting edges. The search literature contains at least one other type of neighborhood. [136] describes a post-processing method that improves on a sub-optimal solution path by repeatedly selecting a sub-path of it and running A* to find the shortest path between its start and end points. The main disadvantage of such methods is that they require the setting of parameters such as the location of the starting state and length of these sub-paths.

Third, it is interesting to consider the following trade-off. The larger the neighborhood, the more likely it is to contain a local optimum of high quality, but also the more computationally expensive it is to search. In our context, the size of the neighborhood is defined by the amount of available memory. In the extreme case where memory is unlimited, ITSA* reduces to A* and may take exponential time to search the neighborhood. In the intermediate case, the question arises whether all available memory should be allocated to ITSA*. It is an empirical question whether the improvement in solution cost is faster with a single execution of ITSA* until memory runs out, as opposed to repeated executions of ITSA* with limited memory. If the latter setup exhibits a faster cost reduction, then part of the available memory could be used for other purposes such as, for example, more informed memory-based heuristics.

Fourth, another research issue is the choice of the strategy used for searching the neighborhood. Neighborhood search is sometimes called iterative-improvement search [144, 1], since solution quality is improved (that is, solution cost is reduced) by iteratively jumping from one path to another one in the neighborhood. This general paradigm leaves open the question of how to select the next solution path within the neighborhood. ITSA* follows a best-improvement [1] or steepest-descent [130] strategy, since it uses a variant of A* to select the solution of lowest cost within the neighborhood. Alternatively, a first-improvement strategy would stop searching the neighborhood when any solution is found with a lower cost than the current solution.

Finally, the choices of (re)start solutions and neighborhoods may interfere with each other. [130] defines the *strength* of a neighborhood (of a given size) as the average quality of the local optima that are reached when searching the neighborhood. However, the quality of the attainable local optima may depend on the starting solution. An interesting direction for research is to study empirically how the strength of a neighborhood correlates with the average quality of the starting solutions. Ideally, a strong neighborhood would be weakly correlated with the quality of the starting solutions. One can thus compare the strength of different neighborhood structures using multiple methods for selecting restarts.

7.2.3 Domain-specific extensions

In Chapter 6, we have used the Multiple Sequence Alignment problem as an additional benchmark domain for ABULB. Our goal in this dissertation was not to improve the state of the art in this domain. Nevertheless, this is a worthwhile goal for the future. The first step in this direction requires the study of the relevant tools already developed within the bioinformatics community. This is no small undertaking given the vast array of existing algorithms for this problem. However, someone familiar with this tool box should be able to take advantage of it and to compound its power with ideas from heuristic search. The use of heuristics to guide the search could likely be applied with benefit in the context of other formalizations of this problem. For example, existing specialized algorithms for this problem build the alignment incrementally by adding one sequence at a time to an already completed sub-alignment. This process is in contrast to the way heuristic search builds an alignment, namely column by column while considering all sequences simultaneously (recall that each state in our search space represents a partial alignment of all sequences up to a certain prefix of the sequences and that each action consumes either one or zero letter in each sequence). Nevertheless, it may be possible to use heuristic search in a different space in which each state represents a complete alignment of a sub-set of sequences and each action adds one full sequence to the alignment under construction.

Finally, symbolic planning is another application area where heuristic search has had great success [14, 13, 12, 66, 159, 92, 116, 93]. Applications of heuristic search to symbolic planning have used variants of both WA* and beam search. We believe that it will be advantageous in terms of scaling to adapt our beam-search-based ideas to these tasks.

APPENDIX A

FORMAL PROOFS FOR FALCONS[*]

A.1 Introduction

This Appendix contains the formal proofs for the theoretical results pertaining to FALCONS stated in Section 2.6.2. The proofs are preceded by some definitions, notation, and assumptions.

A.2 Definitions

S, s_{start}, s_{goal}, $succ(s)$, $pred(s)$, $c(s, s')$, $gd(s)$, $sd(s)$, $g(s)$, $h(s)$, $h^*(s)$, $g^*(s)$, $f(s)$, and $f^*(s)$ are defined in Section 2.2. Furthermore:

D1 *G-values are admissible* iff $0 \leq g(s) \leq sd(s)$ for all states s.

D2 *H-values are admissible* iff $0 \leq h(s) \leq gd(s)$ for all states s.

D3 *G-values are consistent* iff $g(s_{start}) = 0$ and $0 \leq g(s') \leq g(s) + c(s, s')$ for all states s with $s \neq s_{start}$ and $s' \in succ(s)$, that is, if they satisfy the triangle inequality.

D4 *H-values are consistent* iff $h(s_{goal}) = 0$ and $0 \leq h(s) \leq c(s, s') + h(s')$ for all states s with $s \neq s_{goal}$ and $s' \in succ(s)$, that is, if they satisfy the triangle inequality.

D5 The state space S is *safely explorable* iff the goal distances of all states are finite

A.3 Notation

Superscripts of f-, g-, and h-values. In the following proofs, $f^t(s)$ (resp. $g^t(s)$ and $h^t(s)$) refers to the f-value (resp. g-value and h-value) of state s before the $t + 1$ value update, that is, before Step 3 (Figure 88) of iteration $t + 1$. Thus, $g^0(s)$ and $h^0(s)$ are the initial g- and h-values of state s before Step 3 of iteration 1.

[*]This Appendix is a modified version of [50].

```
1. $s := s_{start}$
2. $s' := \arg\min_{s'' \in succ(s)} f(s'')$,
   where $f(s'') := \max(g(s'') + h(s''), h(s_{start}))$                                  [F-CALC]
   Break ties in favor of a successor $s'$ with the smallest value of $c(s, s') + h(s')$     [TB]
   Break remaining ties arbitrarily (but systematically[†])                                 [TB2]
3. $g(s)$ := if $(s = s_{start})$ then $g(s)$                                                [G-UPDATE]
             else $\max(g(s),$
                      $\min_{s'' \in pred(s)}(g(s'') + c(s'', s)),$
                      $\max_{s'' \in succ(s)}(g(s'') - c(s, s'')))$
   $h(s)$ := if $(s = s_{goal})$ then $h(s)$                                                 [H-UPDATE]
             else $\max(h(s),$
                      $\min_{s'' \in succ(s)}(c(s, s'') + h(s'')),$
                      $\max_{s'' \in pred(s)}(h(s'') - c(s'', s)))$
4. If $(s = s_{goal})$ then stop successfully
5. $s := s'$
6. Go to Line 2
```

Figure 88: The FALCONS algorithm

Subscripts of state variables. s_t refers to the current state before Step 5 (Figure 88) of iteration $t + 1$. Thus, $s_0 = s_{start}$.

A.4 Assumptions

Our results hold under the following assumptions:

A1 The state space S is finite.

A2 The state space S is safely explorable.

A3 All actions costs are positive.

A4 The initial g- and h- values are admissible.

A5 The initial g-values are consistent.

Assumption A5 is only used for results pertaining to the use of FALCONS without G-UPDATE. Furthermore, A5 implies the part of A4 that pertains to the g-values, since the consistency of the

[†]Systematic tie-breaking is defined in Section 2.3.

g-values implies their admissibility. In practice, most admissible heuristic values are also consistent. Indeed, all of the heuristic values described in Section 2.7.1 are consistent.

A.5 Proofs

We first prove some lemmata pertaining to properties of the g-, h-, and f-values that are guaranteed to hold during the execution of FALCONS. Then, we prove that each trial of FALCONS is guaranteed to terminate (Theorems 1 and 2), that each run of FALCONS is also guaranteed to terminate, that is, FALCONS always converges to a unique path (Lemma 6 and Corollary 6), and finally that the path FALCONS converges to at the end of each run is a minimum-cost path (Theorems 3 and 4). When appropriate, the following lemmata and theorems are accompanied by corollaries that extend the results to FALCONS without the G-UPDATE rule.

Lemma 1

1. *Under assumptions A1-4, FALCONS cannot decrease the g-values.*

2. *Under assumptions A1-4, FALCONS cannot decrease the h-values.*

Proof:

Since only Step 3 of FALCONS modifies the heuristic values, we need only consider that step. Let $t \in \{1, 2, 3, \dots\}$ be the number of the current iteration. Let s be any state in S.

1. Proof for G-UPDATE

Case (i): $s = s_t$

If $s = s_{start}$ then $g^{t+1}(s) \overset{G-UPDATE}{=} g^t(s)$, else

$$g^{t+1}(s) \overset{G-UPDATE}{=} max \left\{ \begin{array}{l} g^t(s), \\ \min_{s'' \in pred(s)}(g^t(s'') + c(s'', s)), \\ \max_{s'' \in succ(s)}(g^t(s'') - c(s, s'')) \end{array} \right\} \overset{\text{def. of max}}{\geq} g^t(s).$$

Case (ii): $s \neq s_t$

In this case, $g(s)$ is not updated, and thus $g^{t+1}(s) = g^t(s)$.

Therefore in both cases, $\forall t \in \{1, 2, 3, \dots\}, s \in S$: $g^{t+1}(s) \geq g^t(s)$.

2. Proof for H-UPDATE

Case (i): $s = s_t$

If $s = s_{goal}$ then $h^{t+1}(s) \overset{H-UPDATE}{=} h^t(s)$, else

$$h^{t+1}(s) \overset{H-UPDATE}{=} max \left\{ \begin{array}{l} h^t(s), \\ \min_{s'' \in succ(s)}(c(s,s'') + h^t(s'')), \\ \max_{s'' \in pred(s)}(h^t(s'') - c(s'',s)) \end{array} \right\} \overset{\text{def. of max}}{\geq} h^t(s).$$

Case (ii): $s \neq s_t$

In this case, $h(s)$ is not updated, and thus $h^{t+1}(s) = h^t(s)$.

Therefore in both cases, $\forall t \in \{1, 2, 3, \ldots\}, s \in S$: $h^{t+1}(s) \geq h^t(s)$. ∎

Corollary 1

1. *Under assumptions A1-5, FALCONS without G-UPDATE cannot decrease the g-values.*

2. *Under assumptions A1-5, FALCONS without G-UPDATE cannot decrease the h-values.*

Proof:

1. Since G-UPDATE is the only place in FALCONS where the g-values are updated, FALCONS without G-UPDATE never modifies the g-values and thus cannot increase them.

2. The proof is the same as that for Lemma 1(2). ∎

Let us now formally define the start distance $sd(s)$ and goal distance $gd(s)$ of state s:

$$sd(s) := \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in pred(s)}(sd(s') + c(s',s)) & \text{otherwise} \end{cases} \quad (1)$$

$$gd(s) := \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in succ(s)}(c(s,s') + gd(s')) & \text{otherwise} \end{cases} \quad (2)$$

Lemma 2

1. *Under assumptions A1-4, the g-values remain admissible during the execution of FALCONS.*

2. *Under assumptions A1-4, the h-values remain admissible during the execution of FALCONS.*

Proof:

1. Proof by induction on t.

At $t = 0$, assumption A4 guarantees that $\forall s \in S$: $g^0(s)$ is admissible.

Assume that the induction hypothesis holds at the beginning of iteration t:

$$\forall s \in S, g^t(s) \; is \; admissible \tag{3}$$

Let us prove that $\forall s \in S, g^{t+1}(s)$ is admissible as well. Let s be any state in S.

If $s \neq s_t$, then $g(s)$ is not modified during iteration t. Therefore, $g^{t+1}(s) = g^t(s)$, which is admissible by Equation 3. If $s = s_t$, then $g(s)$ is only modified by G-UPDATE (Step 3 of FALCONS). Now, if $s = s_{start}$, then $g^{t+1}(s) \overset{G-UPDATE}{=} g^t(s)$, which is admissible by Equation 3. Therefore, we need only consider the situation where $s = s_t \neq s_{start}$, for which it holds that:

$$g^{t+1}(s) \overset{G-UPDATE}{=} max \left\{ \begin{array}{l} g^t(s), \\ \min_{s'' \in pred(s)}(g^t(s'') + c(s'', s)), \\ \max_{s'' \in succ(s)}(g^t(s'') - c(s, s'')) \end{array} \right\}$$

We distinguish 3 cases, depending on which of the 3 arguments of *max* is the largest.

$$\text{Let } sp \;:=\; \arg \min_{s'' \in pred(s)} (g^t(s'') + c(s'', s)). \tag{4}$$

$$\text{Let } ss \;:=\; \arg \max_{s'' \in succ(s)} (g^t(s'') - c(s, s'')). \tag{5}$$

Case (i): $g^{t+1}(s) = g^t(s)$

Then, by Equation 3, $g^{t+1}(s)$ is admissible.

Case (ii):

$$g^{t+1}(s) = g^t(sp) + c(sp, s) \tag{6}$$

Proof by contradiction.

$$\text{Let } ssd := \arg \min_{s'' \in pred(s)} (sd(s'') + c(s'', s)). \tag{7}$$

$$\text{Thus, } sd(s) = sd(ssd) + c(ssd, s). \tag{8}$$

(Note that Equation 8 implies that $ssd \neq s$.) Now, assume $g^{t+1}(s) > sd(s)$.

This, combined with Equations 6 and 8, yields

$$g^t(sp) + c(sp, s) = g^{t+1}(s) > sd(s) = sd(ssd) + c(ssd, s). \tag{9}$$

But, since $g^t(ssd)$ is admissible by Equation 3, $sd(ssd) + c(ssd, s) \geq g^t(ssd) + c(ssd, s)$, which, combined with Equation 9, implies: $g^t(sp) + c(sp, s) > g^t(ssd) + c(ssd, s)$. The latter contradicts Equation 4. Therefore, $g^{t+1}(s) \leq sd(s)$, i.e. $g^{t+1}(s)$ is admissible.

Case (iii):

$$g^{t+1}(s) = g^t(ss) - c(s, ss) \tag{10}$$

(Note that Equation 10 implies that $ss \neq s$. Otherwise, $g(s) = g(ss)$ would strictly decrease between t and $t+1$, since $c(s, ss) \overset{A3}{>} 0$.) From $g^t(ss) \overset{\text{Equation 3}}{\leq} sd(ss)$ and $sd(ss) \overset{\text{def. of } sd}{\leq} sd(s) + c(s, ss)$, we obtain $g^t(ss) \leq sd(s) + c(s, ss)$, or equivalently $g^t(ss) - c(s, ss) \leq sd(s)$, which, combined with Equation 10, yields $g^{t+1}(s) \leq sd(s)$. Therefore, $g^{t+1}(s)$ is admissible.

In conclusion, $g^{t+1}(s)$ is admissible in all cases.

2. Proof by induction on t.

At $t = 0$, assumption A4 guarantees that $\forall s \in S$: $h^0(s)$ is admissible.

Assume that the induction hypothesis holds at the beginning of iteration t:

$$\forall s \in S\text{: } h^t(s) \textit{ is admissible} \tag{11}$$

Let us prove that $\forall s \in S$: $h^{t+1}(s)$ is admissible as well. Let s be any state in S.

If $s \neq s_t$, then $h(s)$ is not modified during iteration t. Therefore, $h^{t+1}(s) = h^t(s)$, which is admissible by Equation 11. If $s = s_t$, then $h(s)$ is only modified by H-UPDATE (Step 3 of FALCONS). Now, if $s = s_{goal}$, then $h^{t+1}(s) \overset{H-UPDATE}{=} h^t(s)$, which is admissible by Equation 11. Therefore, we need only consider the situation where $s = s_t \neq s_{goal}$, for which it holds that:

$$h^{t+1}(s) \overset{H-UPDATE}{=} max \left\{ \begin{array}{l} h^t(s), \\ \min_{s'' \in succ(s)}(c(s, s'') + h^t(s'')), \\ \max_{s'' \in pred(s)}(h^t(s'') - c(s'', s)) \end{array} \right\}$$

We distinguish 3 cases, depending on which of the 3 arguments of *max* is the largest.

$$\text{Let } ss \quad := \quad \arg \min_{s'' \in succ(s)} (c(s, s'') + h^t(s'')). \tag{12}$$

$$\text{Let } sp \quad := \quad \arg \max_{s'' \in pred(s)} (h^t(s'') - c(s'', s)). \tag{13}$$

Case (i): $h^{t+1}(s) = h^t(s)$

Then, by Equation 11, $h^{t+1}(s)$ is admissible.

Case (ii):

$$h^{t+1}(s) = c(s, ss) + h^t(ss) \tag{14}$$

Proof by contradiction.

$$\text{Let } sgd := \arg \min_{s'' \in succ(s)} (c(s, s'') + gd(s'')). \tag{15}$$

$$\text{Thus, } gd(s) = c(s, sgd) + gd(sgd). \tag{16}$$

(Note that Equation 16 implies that $sgd \neq s$.) Now, assume $h^{t+1}(s) > gd(s)$.

This, combined with Equations 14 and 16, yields

$$c(s, ss) + h^t(ss) = h^{t+1}(s) > gd(s) = c(s, sgd) + gd(sgd). \tag{17}$$

But, since $h^t(sgd)$ is admissible by Equation 11, $c(s, sgd) + gd(sgd) \geq c(s, sgd) + h^t(sgd)$, which, combined with Equation 17, implies $c(s, ss) + h^t(ss) > c(s, sgd) + h^t(sgd)$. The latter contradicts Equation 12. Therefore, $h^{t+1}(s) \leq gd(s)$, i.e. $h^{t+1}(s)$ is admissible.

Case (iii):

$$h^{t+1}(s) = h^t(sp) - c(sp, s) \tag{18}$$

(Note that Equation 18 implies that $sp \neq s$. Otherwise, $h(s) = h(sp)$ would strictly decrease between t and $t + 1$, since $c(sp, s) \overset{A3}{>} 0$.) From $h^t(sp) \overset{\text{Equation 11}}{\leq} gd(sp)$ and $gd(sp) \overset{\text{def. of } gd}{\leq} c(sp, s) + gd(s)$, we obtain $h^t(sp) \leq c(sp, s) + gd(s)$, or equivalently $h^t(sp) - c(sp, s) \leq gd(s)$, which, combined with Equation 18, yields $h^{t+1}(s) \leq gd(s)$. Therefore, $h^{t+1}(s)$ is admissible.

In conclusion, $h^{t+1}(s)$ is admissible in all cases. ■

Corollary 2

1. *Under assumptions A1-5, the g-values remain admissible during the execution of FALCONS without G-UPDATE.*

2. *Under assumptions A1-5, the h-values remain admissible during the execution of FALCONS without G-UPDATE.*

Proof:

1. Since G-UPDATE in Step 3 of FALCONS is the only step that modifies the g-values, FALCONS without G-UPDATE does not modify the g-values, and the g-values thus remain admissible.

2. Since G-UPDATE does not have any effect on the h-values, its absence in FALCONS does not make a difference in whether the h-values remain admissible. Therefore, this proof is the same as that for Lemma 2(2). ■

Lemma 3 *Under assumptions A1-4, a trial of FALCONS could only run forever if, from some time on, it repeatedly moved along a finite cyclic path without modifying any of the g- and h-values in the cycle.*

Proof: Consider the h-values. Lemma 1(2) guarantees that, on every transition, the h-value of the current state s can only increase or stay the same. In addition, Lemma 2(2) provides an upper bound on $h(s)$, namely $gd(s)$ (which is finite, by A2). This means that the maximum number of strict increases of $h(s)$ is finite. This reasoning holds for all states s in S. And since S is finite (A1), we infer that the maximum total number (over S) of strict increases of h-values by H-UPDATE is finite. The same reasoning applies to G-UPDATE for the g-values. In conclusion, there is a maximum, finite number of strict increases possible for both the g- and h-values. Therefore, if FALCONS never terminates, there must be a point in time, say T, after which no g- nor h-values are modified. Now, we prove that from some time T_1 on ($T_1 \geq T$), it must be the case that FALCONS repeatedly moves along a cycle. Let s^1 denote the first state to be visited twice after time T (s^1 must exist, by A1). Let T_1 (resp. T_2) denote the instant in time at which s^1 is reached for the first (resp. second) time after time T. By definition, $T_2 \geq T_1 \geq T$. Let C be the sequence of states (starting with s^1) traveled through in the time interval [T_1,T_2). From time T_1 on, the cycle C is repeatedly followed by FALCONS. The reason for this is that no values in the state space changes after time T (and therefore after time T_1) and systematic tie-breaking (TB2) ensures that FALCONS will thereafter always choose the same successor at every decision point. ■

Corollary 3 *Under assumptions A1-5, a trial of FALCONS without G-UPDATE could only run forever if, from some time on, it repeatedly moved along a finite cyclic path without modifying any of the g- and h-values in the cycle.*

Proof: This proof is identical to that of Lemma 3, except that the finite number of strict increases of the g-values (namely zero) directly follows from the fact that FALCONS without G-UPDATE never modifies the g-values. ■

Lemma 4 *Under assumptions A1-4, assume that FALCONS makes a transition from a state s_t to a state s_{t+1} without modifying the g- and h-values of s_t. Let $s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h^t(s'))$. Then, s'' is such that:*

1. $h^t(s'') \leq h^t(s_t) - c(s_t, s'')$,
2. $g^t(s'') \leq g^t(s_t) + c(s_t, s'')$,
3. $f^t(s'') \leq f^t(s_t)$,
4. $f^t(s_{t+1}) \leq f^t(s'')$, *and*
5. *if* $f^t(s_t) \leq f^t(s_{start})$, *then* $f^t(s_{t+1}) \leq f^t(s_{start})$.

Proof:

First, note that $s_t \neq s_{goal}$. Otherwise, FALCONS would stop in s_t.

Second, note that $s'' \neq s_t$. If that was not the case, it would hold that $h^{t+1}(s_t) \overset{H-UPDATE}{\geq} \min_{s' \in succ(s_t)}(c(s_t, s') + h^t(s')) \overset{Equation\ 19}{=} c(s_t, s_t) + h^t(s_t)$, which would imply that $h^{t+1}(s_t) -$

$h^t(s_t) \geq c(s_t, s_t) \overset{A3}{>} 0$ and contradict our assumption that $h^{t+1}(s_t) = h^t(s_t)$. Thus, $s'' \neq s_t$.

$$\text{Let } s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h^t(s')). \tag{19}$$

1. Proof by contradiction.

Assume $h^t(s_t) < c(s_t, s'') + h^t(s'')$. This, together with Equation 19, implies that $h^t(s_t) < \min_{s' \in succ(s_t)}(c(s_t, s') + h^t(s')) \overset{H-UPDATE}{\leq} h^{t+1}(s_t)$, which contradicts $h^{t+1}(s_t) = h^t(s_t)$. Therefore, s'' must satisfy $h^t(s_t) \geq c(s_t, s'') + h^t(s'')$, or equivalently $h^t(s'') \leq h^t(s_t) - c(s_t, s'')$.

2. Case (i): $s_t = s_{start}$

First, note that

$$\forall t \in \{1, 2, 3, \ldots\}\colon g^t(s_{start}) = 0 \tag{20}$$

follows from the fact that initially admissible g-values (A4) remain admissible (Lemma 2(1)). $g^t(s'') \overset{Lemma\ 2(1)}{\leq} sd(s'')$ and $sd(s'') \overset{\text{def. of } sd}{\leq} sd(s_t) + c(s_t, s'') = c(s_t, s'')$ imply that $g^t(s'') \leq c(s_t, s'')$ or equivalently $g^t(s'') \leq 0 + c(s_t, s'') \overset{Equation\ 20}{=} g^t(s_{start}) + c(s_t, s'') = g^t(s_t) + c(s_t, s'')$.

Case (ii): $s_t \neq s_{start}$ (Proof by contradiction)

Assume $g^t(s_t) < g^t(s'') - c(s_t, s'')$.

This implies that $g^t(s_t) < \max_{s' \in succ(s_t)}(g^t(s') - c(s_t, s')) \overset{G-UPDATE}{\leq} g^{t+1}(s_t)$, which contradicts the assumption that $g^{t+1}(s_t) = g^t(s_t)$. Thus, $g^t(s_t) \geq g^t(s'') - c(s_t, s'')$ or equivalently $g^t(s'') \leq g^t(s_t) + c(s_t, s'')$.

3. From Results 1 and 2 above, we have $g^t(s'') + h^t(s'') \leq g^t(s_t) + c(s_t, s'') + h^t(s_t) - c(s_t, s'')$ or equivalently $g^t(s'') + h^t(s'') \leq g^t(s_t) + h^t(s_t)$. Thus, $max(h^t(s_{start}), g^t(s'') + h^t(s'')) \leq max(h^t(s_{start}), g^t(s_t) + h^t(s_t))$ which, by F-CALC, is equivalent to $f^t(s'') \leq f^t(s_t)$.

4. Since FALCONS chooses s_{t+1} as the next state, it must hold that $f^t(s_{t+1}) \leq f^t(s'')$.

5. Assume $f^t(s_t) \leq f^t(s_{start})$. This, together with Result 3 above, implies that

$$f^t(s'') \leq f^t(s_{start}). \tag{21}$$

Since FALCONS chooses to move to s_{t+1}, it must be the case that $f^t(s_{t+1}) \leq f^t(s'')$, which, together with Equation 21, yields $f^t(s_{t+1}) \leq f^t(s_{start})$. ■

Corollary 4 *Under assumptions A1-5, assume that FALCONS without G-UPDATE makes a transition from a state s_t to a state s_{t+1} without modifying the g- and h-values of s_t. Let $s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h^t(s'))$. Then, s'' is such that:*

1. $h^t(s'') \leq h^t(s_t) - c(s_t, s'')$,

2. $g^t(s'') \leq g^t(s_t) + c(s_t, s'')$,

3. $f^t(s'') \leq f^t(s_t)$,

4. $f^t(s_{t+1}) \leq f^t(s'')$, *and*

5. *if* $f^t(s_t) \leq f^t(s_{start})$, *then* $f^t(s_{t+1}) \leq f^t(s_{start})$.

Proof:

For the same reasons as in the proof for Lemma 4, $s_t \neq s_{goal}$ and $s'' \neq s_t$. In addition, the g-values are never modified.

1. This proof is the same as that for Lemma 4(1).

2. Case (i): $s_t = s_{start}$

This proof is the same as that for Lemma 4(2), except that Equation 20 is now true because of A4 and the absence of G-UPDATE, and that we use Corollary 2(1) instead of Lemma 2(1).

Case (ii): $s_t \neq s_{start}$

$g^t(s'') \leq g^t(s_t) + c(s_t, s'')$ directly follows from A5 and the definition of the consistency of the g-values.

3. This proof is the same as that for Lemma 4(3) above.

4. This proof is the same as that for Lemma 4(4) above.

5. This proof is the same as that for Lemma 4(5) above. ■

Lemma 5 *Under assumptions A1-4, assume that FALCONS follows a path P starting in any state s^1 without modifying the g- and h-values of any state on P. If $f(s^1) \leq f(s_{start})$, then for all states s on P,*

$$f(s) \leq f(s_{start}). \tag{22}$$

Proof: Proof by induction on the distance of s from s^1 on P.

If $s = s^1$, then $f(s) = f(s^1) \leq f(s_{start})$. So, Equation 22 trivially holds for s^1.

Assume that s is any state on P but the last one. Then, s has a successor s' on P. Lemma 4(5) directly allows us to infer that, if Equation 22 holds for s, then it also holds for s'. ■

Corollary 5 *Under assumptions A1-5, assume that FALCONS without G-UPDATE follows a path P starting in any state s^1 without modifying the g- and h-values of any state on P. If $f(s^1) \leq f(s_{start})$, then for all states s on P, s on P,*

$$f(s) \leq f(s_{start}). \tag{23}$$

Proof: The proof is the same as that for Lemma 5, except that it uses Corollary 4(5) instead of Lemma 4(5). ■

Lemma 6 *Under assumptions A1-4, at all times t during the execution of FALCONS, $f^t(s_{start}) = h^t(s_{start})$.*

Proof:

$f^t(s_{start}) \overset{F-CALC}{=} max(g^t(s_{start}) + h^t(s_{start}), h^t(s_{start})) \overset{A4+Lemma\ 2(1)}{=} max(0 + h^t(s_{start}), h^t(s_{start})) = h^t(s_{start})$. ■

Corollary 6 *Under assumptions A1-5, at all times t during the execution of FALCONS without G-UPDATE, $f^t(s_{start}) = h^t(s_{start})$.*

Proof:

The proof is the same as that for Lemma 6, except that it uses Corollary 2(1) instead of Lemma 2(1). ■

Theorem 4 (Termination 1) *Under assumptions A1-4, each trial of FALCONS is guaranteed to terminate.*

Proof: Proof by contradiction.

If FALCONS cycles forever then there exists a finite cyclic path P along which the g- and h-values do not change from some time T on (Lemma 3). In the following, we can drop the superscripts on the h- and f-values since they do not change after time T.

We distinguish two cases. Either all states on P have f-values smaller than or equal to $f(s_{start})$, or all states on P have f values greater than $f(s_{start})$. These are the only two possible cases. Indeed, if there is at least one state s^1 on P such that $f(s^1) \leq f(s_{start})$, then all states following s^1 on P will also have an f-value smaller than or equal to $f(s_{start})$ (by Lemma 5). But, since P is cyclic, every state s on P follows s^1 and therefore satisfies $f(s) \leq (s_{start})$.

Case (i): For all states s_t on the cycle, $f(s_t) > f(s_{start})$.

In this case, it must hold that for all successors s' of all states in the cycle, $f(s') > f(s_{start})$. Otherwise, FALCONS would choose as next state a successor with $f(s') \leq f(s_{start})$ and thus leave the cycle.

Let s_t be any state on this cycle, s_{t+1} be the successor of s on the cycle, and $s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h(s'))$.

By Lemma 4(3&4), $f(s_{t+1}) \leq f(s_t)$, i.e. the f-values cannot increase along a transition. Therefore they cannot decrease either because otherwise they would have to increase again before the end of the cycle. So the f-values of all states on the cycle are the same and in particular $f(s_{t+1}) = f(s_t)$ which, combined with Lemma 4(3&4) yields $f(s_{t+1}) = f(s'') = f(s_t)$. Since FALCONS chooses s_{t+1} as the next state, $c(s_t, s_{t+1}) + h(s_{t+1}) \overset{TB}{\leq} c(s_t, s'') + h(s'')$. By definition of s'' and H-UPDATE (since $h(s_t)$ does not change), $c(s_t, s'') + h(s'') \leq h(s_t)$ Combining the two previous inequalities yields $c(s_t, s_{t+1}) + h(s_{t+1}) \leq h(s_t)$. Since $c(s_t, s_{t+1}) \overset{A3}{>} 0$, it follows that $h(s_{t+1}) < h(s_t)$. This means that the h-value strictly decreases along this and therefore all transitions on the cycle, which is impossible.

Case (ii): For all states s_t on the cycle, $f(s_t) \leq f(s_{start})$.

Let s_t be any state on the cycle. Let s_{t+1} be the successor of s on the cycle and $s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h(s'))$.

Now, $f(s'') \overset{Lemma\ 4(3)}{\leq} f(s_t)$ and $f(s_t) \leq f(s_{start})$ imply that $f(s'') \leq f(s_{start}) \overset{Lemma\ 6}{=} h(s_{start})$, which, combined with $f(s'') \overset{F-CALC}{\geq} h(s_{start})$, yields

$$f(s'') = h(s_{start}). \tag{24}$$

Furthermore, $f(s_{t+1}) \overset{Lemma\ 4(5)}{\leq} f(s_{start}) \overset{Lemma\ 6}{=} h(s_{start})$ implies, together with $f(s_{t+1}) \overset{F-CALC}{\geq} h(s_{start})$, that $f(s_{t+1}) = h(s_{start})$. Combining this equation with Equation 24 yields $f(s'') = f(s_{t+1})$. Now, $c(s_t, s_{t+1}) + h(s_{t+1}) \overset{TB}{\leq} c(s_t, s'') + h(s'')$. In addition, since $h(s_t)$ does not change after the update, we know that $c(s_t, s'') + h(s'') \leq h(s_t)$. Chaining the two together, we get $c(s_t, s_{t+1}) + h(s_{t+1}) \leq h(s_t)$ or equivalently $h(s_{t+1}) \leq h(s_t) - c(s_t, s_{t+1}) < h(s_t)$, since $c(s_t, s_{t+1}) \overset{A3}{>} 0$. This means that the h-value strictly decreases along this and therefore all transitions in the cycle, which is impossible. ■

Theorem 5 (Termination 2) *Under assumptions A1-5, each trial of FALCONS without G-UPDATE is guaranteed to terminate.*

Proof: The proof for this theorem is the same as that for Theorem 4 except that it uses the corollaries instead of the lemmata with the corresponding numbers. ■

Lemma 7 (Convergence) *Under assumptions A1-4, assume FALCONS is reset to s_{start} at the end of each trial and the g- and h-values are kept from each trial to the next. Then, from some time on, FALCONS will always follow the same path.*

Proof:

Theorem 4 above has established that each trial of FALCONS will always terminate. We now assume that FALCONS is reset into s_{start} at the end of each trial. We can follow a reasoning similar to that used in the proof of Lemma 3 to establish that from some time T on, no g- and h-value will change any longer. This is because these values can only increase or remain unchanged (Lemma 1) and remain admissible (Lemma 2). Therefore, the g- and h-values are bounded from above by finite values (by A2) and cannot increase forever. Now, let t_1 denote the first trial that starts after time T and let P denote the path followed during t_1. Since no g- nor h-value changes during t_1, the next trial, say t_2, will start with the same heuristic values. And since remaining ties are broken systematically (TB2), FALCONS, starting at the same state s_{start}, will necessarily follow the same path P during t_2. The same reasoning holds for all subsequent trials. Therefore, from trial t_1 on, FALCONS will always follow the same path P. It has therefore converged to P. ■

Corollary 7 *Under assumptions A1-5, assume FALCONS without G-UPDATE is reset to s_{start} at the end of each trial and the g- and h-values are kept from each trial to the next. Then, from some time on, FALCONS without G-UPDATE will always follow the same path.*

Proof:

The proof is identical to that for Lemma 7 except that it uses Theorem 5 instead of Theorem 4 and the corollaries corresponding to the lemmata. ■

Theorem 6 (Convergence to a shortest path 1) *Under assumptions A1-4, FALCONS converges to a shortest path.*

Proof:

(In this proof, the time superscript of the f- and h-values are omitted for ease of reading.) Assume that FALCONS has converged to a path P from s_{start} to s_{goal} (Lemma 7).

Since the first state in P is s_{start} and its f-value is trivially less than or equal to $f(s_{start})$, we can use Lemma 5 to infer that, for all states s_t on P, $f(s_t) \leq f(s_{start})$. Let us consider any state s_t on P, s_{t+1} its successor on P, and let $s'' := argmin_{s' \in succ(s_t)}(c(s_t, s') + h(s'))$.

By combining Lemma 4(3) with $f(s_t) \leq f(s_{start})$, we get $f(s'') \leq f(s_t) \leq f(s_{start}) \overset{Lemma\ 6}{=} h(s_{start})$, which, combined with $f(s'') \overset{F-CALC}{\geq} h(s_{start})$ yields $f(s'') = h(s_{start})$. Similarly, $f(s_{t+1}) \overset{Lemma\ 4(5)}{\leq} f(s_{start}) \overset{Lemma\ 6}{=} h(s_{start})$ and $f(s_{t+1}) \overset{F-CALC}{\geq} h(s_{start})$ yield $f(s_{t+1}) = h(s_{start})$. Thus, $f(s'') = f(s_{t+1})$. Since the chosen successor is s_{t+1}, it must be the case (by TB) that $h(s_{t+1})+c(s_t, s_{t+1}) \leq h(s'')+c(s_t, s'')$. According to Lemma 4(1), $h(s'')+c(s_t, s'') \leq h(s_t)$. Combining the last two inequalities, we get $h(s_{t+1}) + c(s_t, s_{t+1}) \leq h(s_t)$, or equivalently

$$h(s_t) - h(s_{t+1}) \geq c(s_t, s_{t+1}). \tag{25}$$

Adding up the instances of Equation 25 for each transition on P yields $h(s_{start}) - h(s_{goal}) \geq cost_P(s_{start}, s_{goal})$, where $cost_P(s_{start}, s_{goal})$ denotes the total cost of path P. Since $h(s_{goal}) \overset{A4+Lemma\ 2(2)}{=} 0$, we infer $h(s_{start}) \geq cost_P(s_{start}, s_{goal})$. Now, the definition of the goal distance implies that $cost_P(s_{start}, s_{goal}) \geq gd(s_{start})$. Finally, admissibility of h means that $gd(s_{start}) \geq h(s_{start})$. Chaining the last three inequalities, we get $gd(s_{start}) \geq h(s_{start}) \geq cost_P(s_{start}, s_{goal}) \geq gd(s_{start})$ and conclude that $cost_P(s_{start}, s_{goal}) = gd(s_{start})$. Therefore, P is a minimum-cost path from s_{start} to s_{goal}, which means that FALCONS has converged to a shortest path. ■

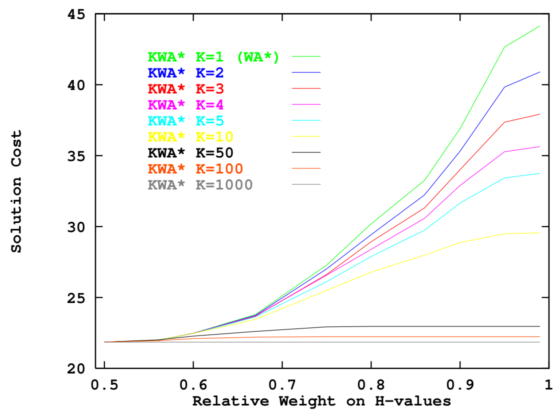
Theorem 7 (Convergence to a shortest path 2) *Under assumptions A1-5, FALCONS without G-UPDATE converges to a shortest path.*

Proof: The proof is the same as that for Theorem 6 except that the corollaries are used instead of the corresponding lemmata. ■

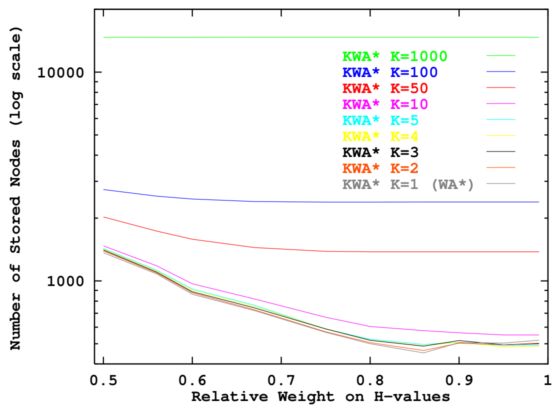
APPENDIX B

EMPIRICAL EVALUATION OF VARIANTS OF WA* IN THE N-PUZZLE

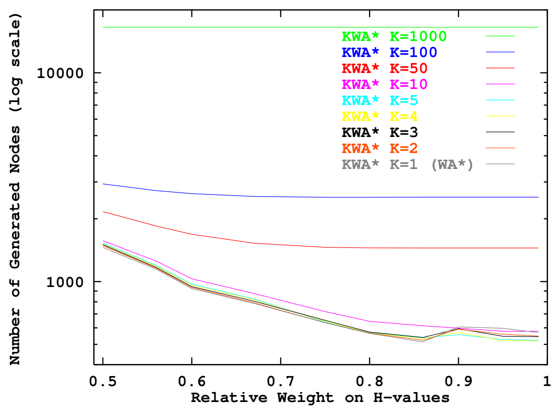
B.1 Empirical evaluation of KWA* in the N-Puzzle



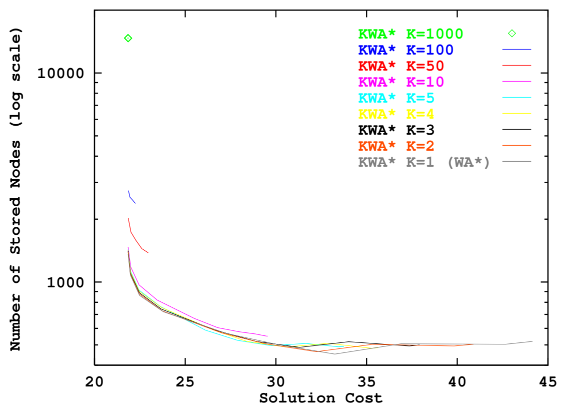
a) Solution cost versus W



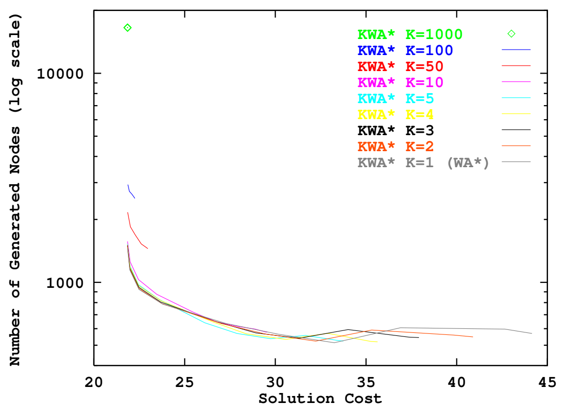
b) Memory usage versus W



c) Search effort versus W

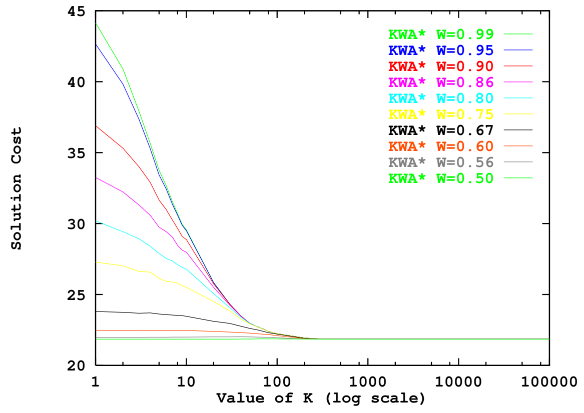


d) Memory usage versus solution cost

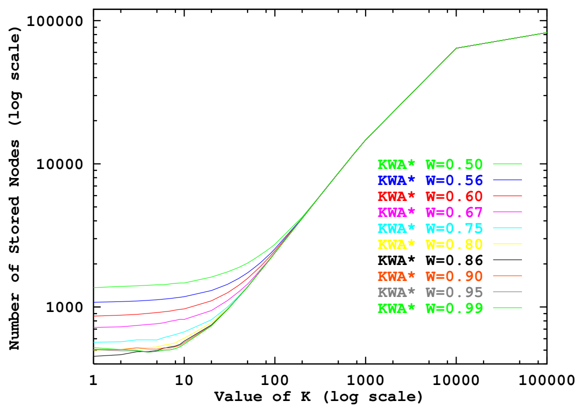


e) Search effort versus solution cost

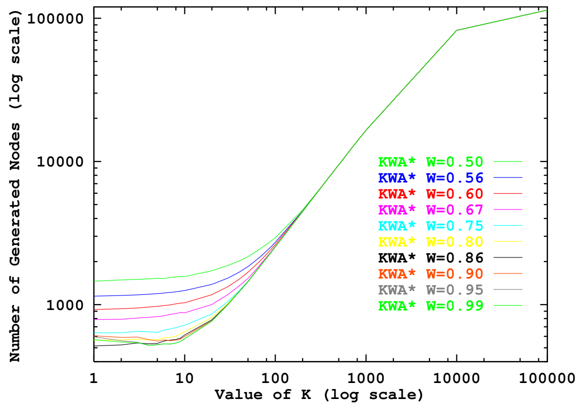
Figure 89: Performance of KWA* in the 8-Puzzle with varying W



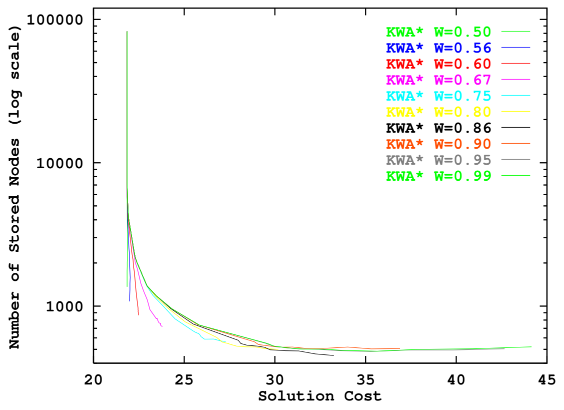
a) Solution cost versus K



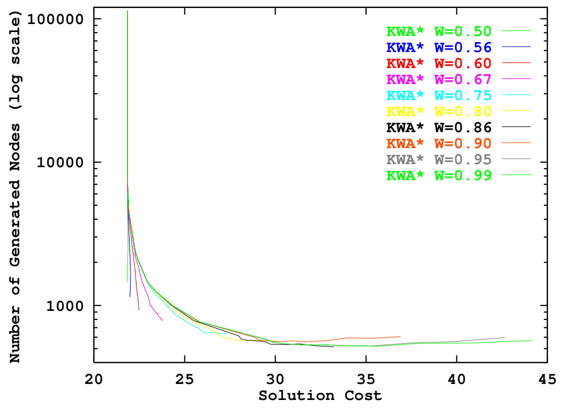
b) Memory usage versus K



c) Search effort versus K

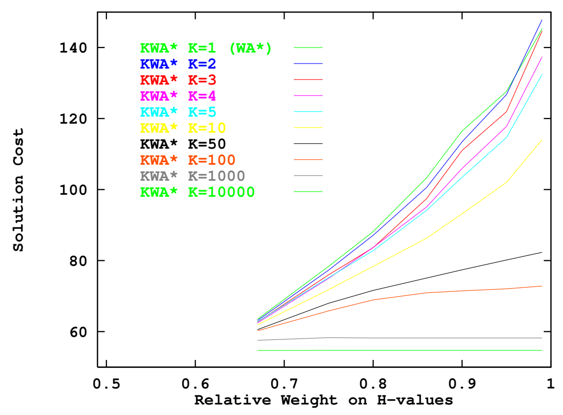


d) Memory usage versus solution cost

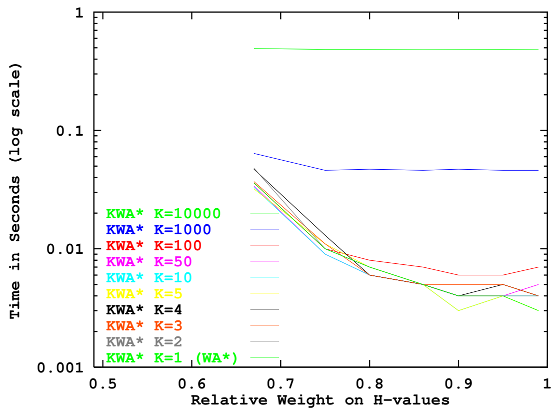


e) Search effort versus solution cost

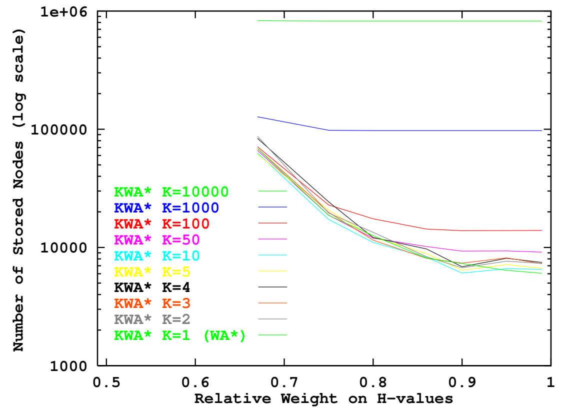
Figure 90: Performance of KWA* in the 8-Puzzle with varying K



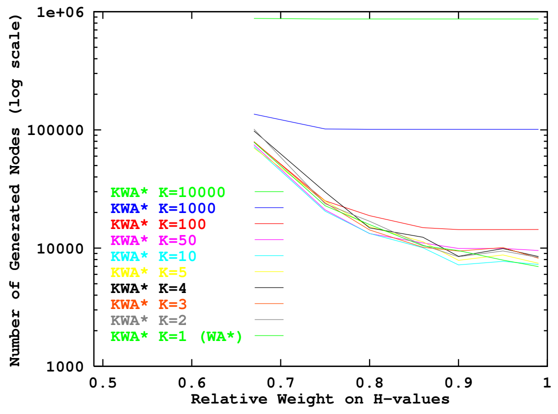
a) Solution cost versus W



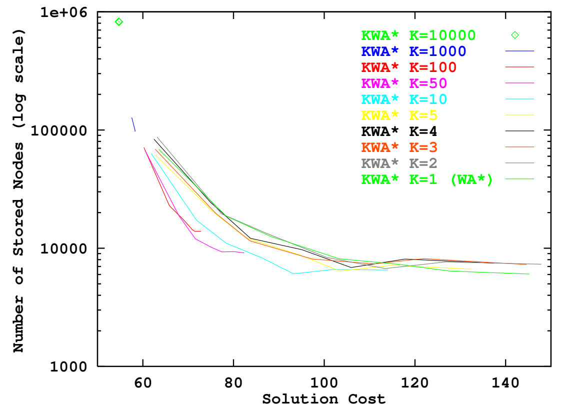
b) Runtime versus W



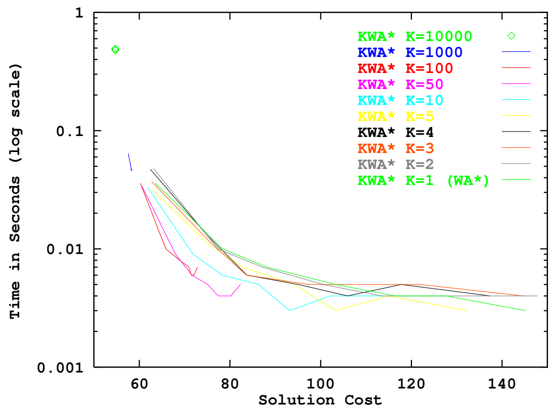
c) Memory usage versus W



d) Search effort versus W

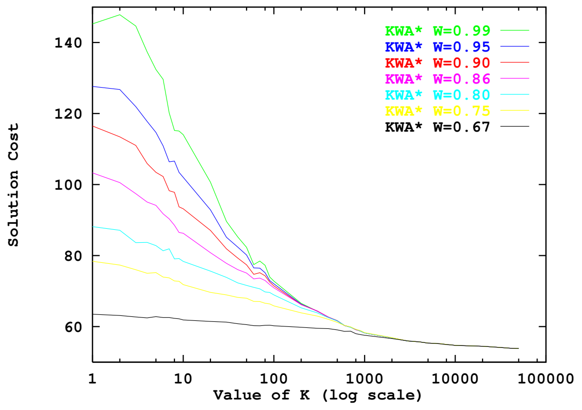


e) Memory usage versus solution cost

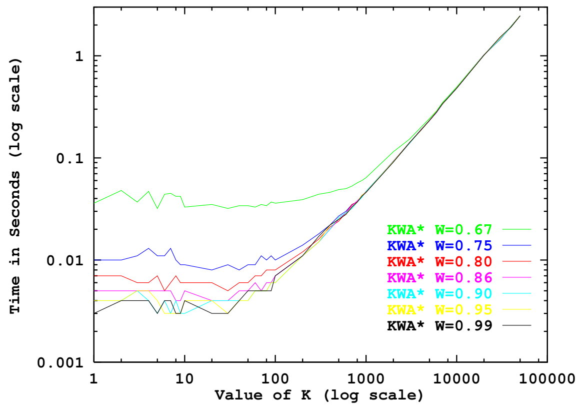


f) Runtime versus solution cost

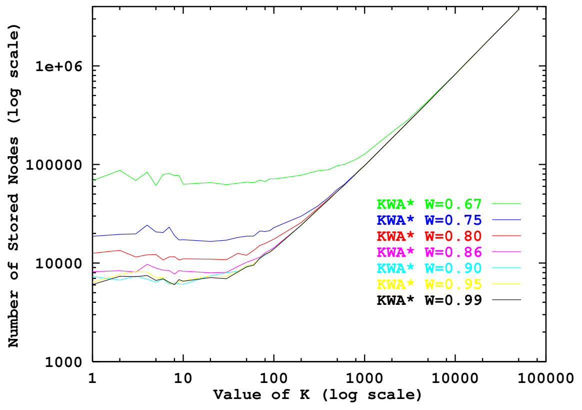
Figure 91: Performance of KWA* in the 15-Puzzle with varying W



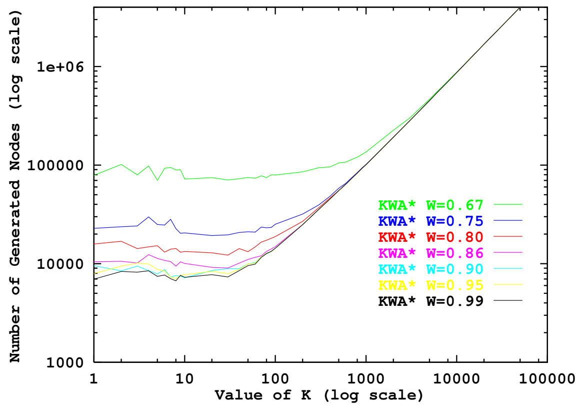
a) Solution cost versus K



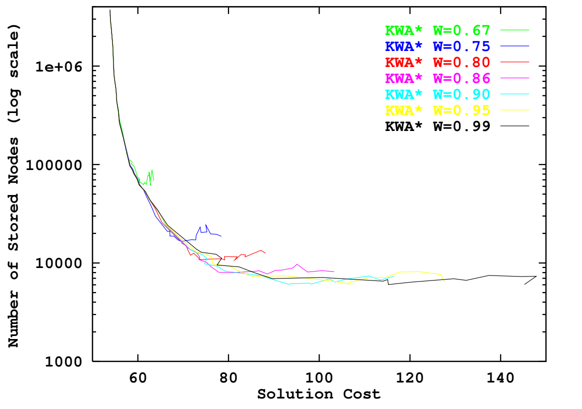
b) Runtime versus K



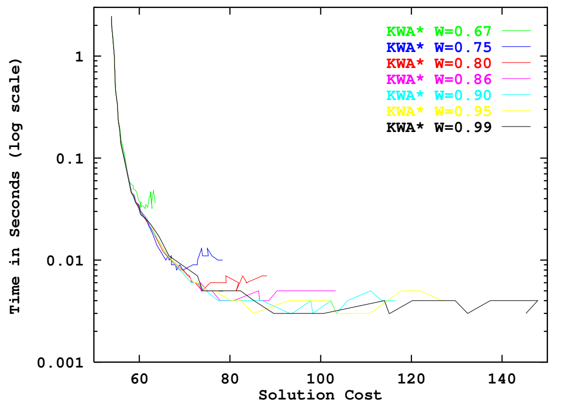
c) Memory usage versus K



d) Search effort versus K

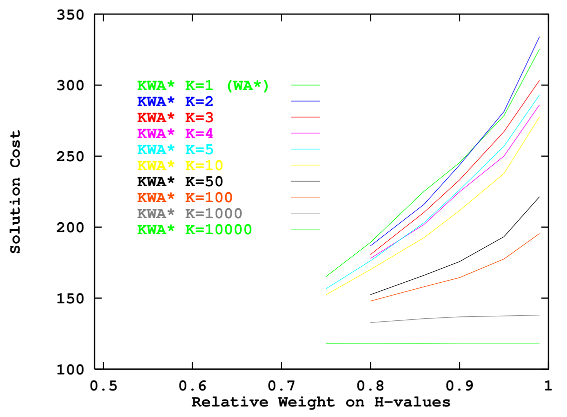


e) Memory usage versus solution cost

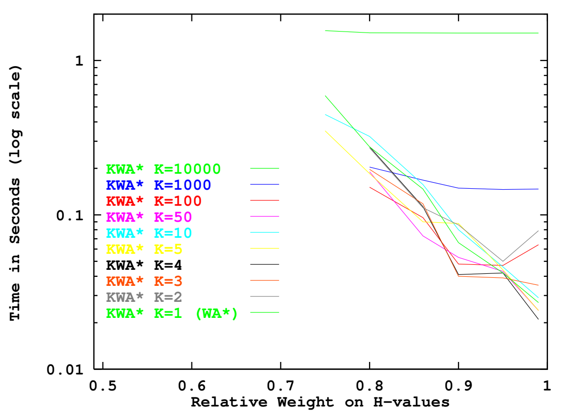


f) Runtime versus solution cost

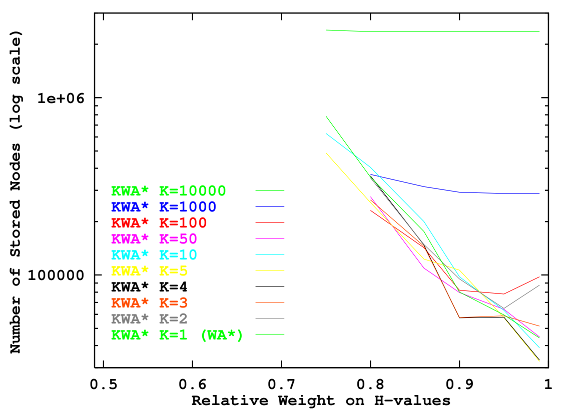
Figure 92: Performance of KWA* in the 15-Puzzle with varying K



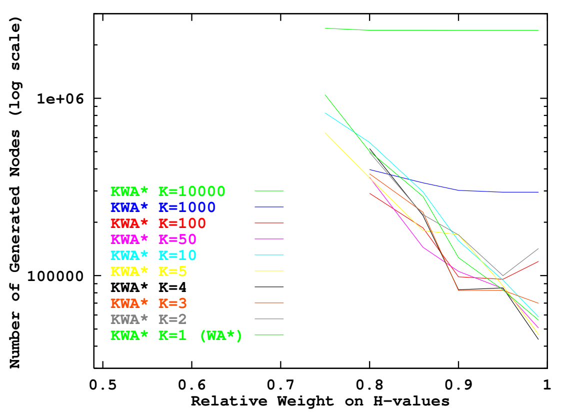
a) Solution cost versus W



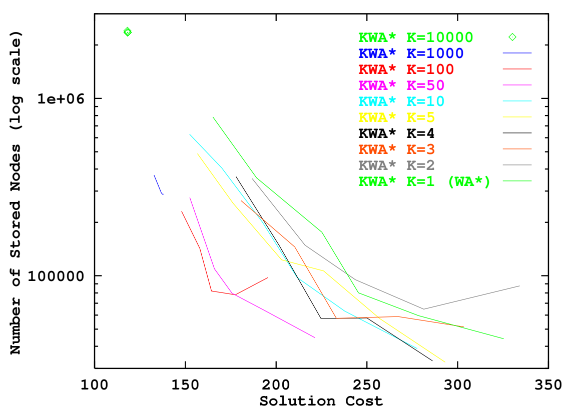
b) Runtime versus W



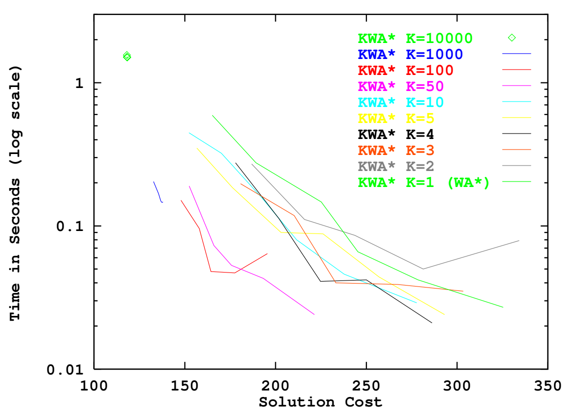
c) Memory usage versus W



d) Search effort versus W

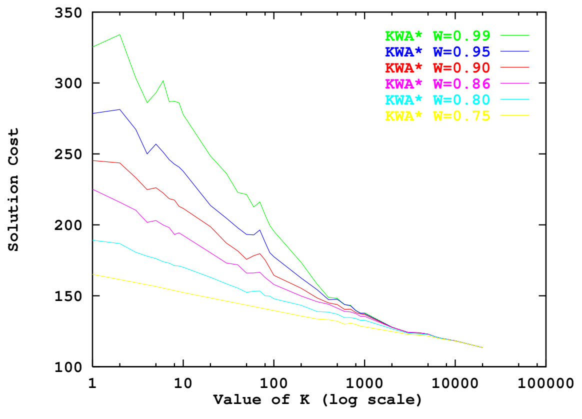


e) Memory usage versus solution cost

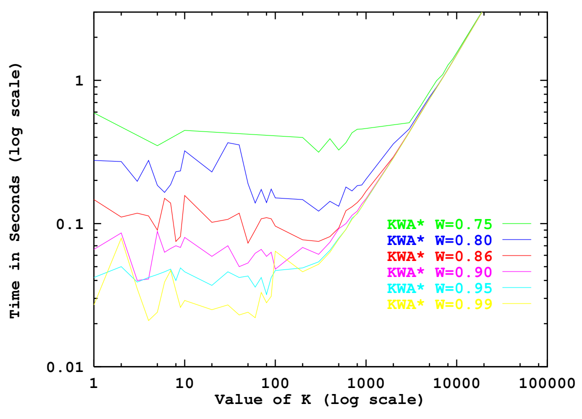


f) Runtime versus solution cost

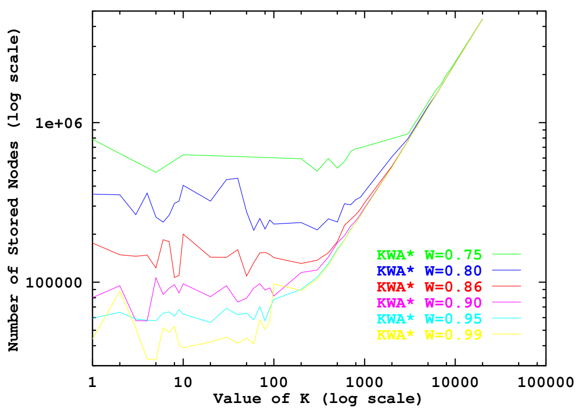
Figure 93: Performance of KWA* in the 24-Puzzle with varying W



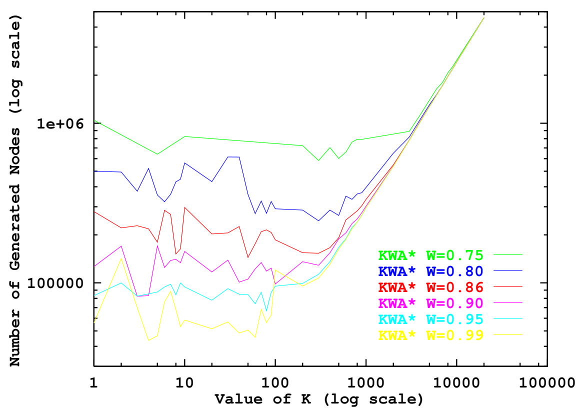
a) Solution cost versus K



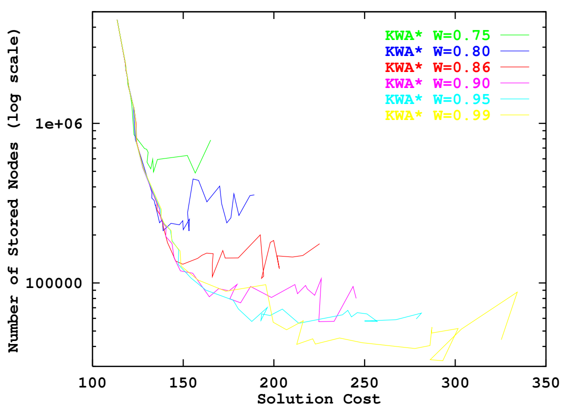
b) Runtime versus K



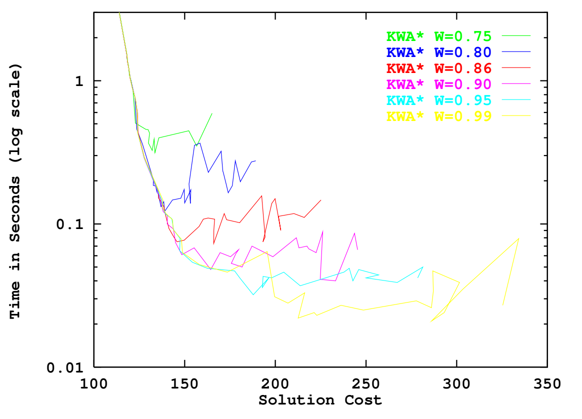
c) Memory usage versus K



d) Search effort versus K



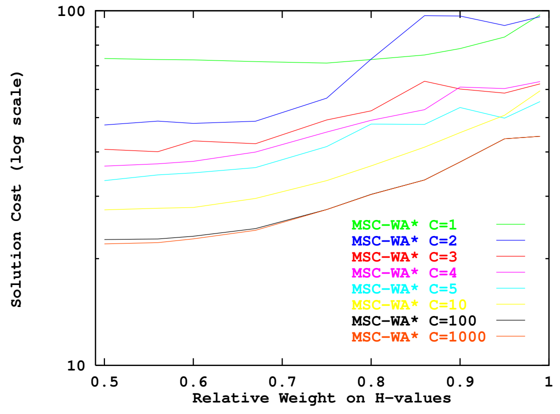
e) Memory usage versus solution cost



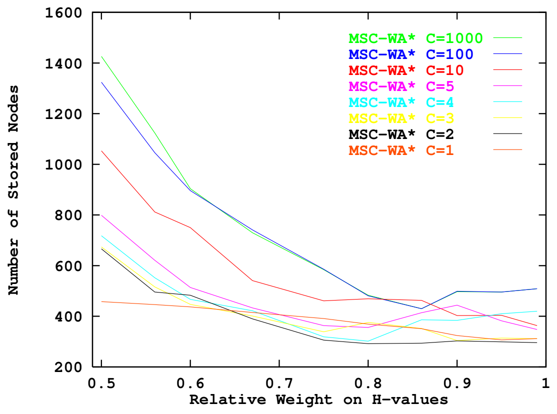
f) Runtime versus solution cost

Figure 94: Performance of KWA* in the 24-Puzzle with varying K

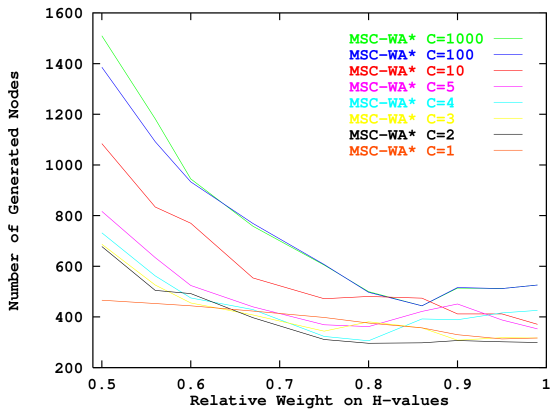
B.2 Empirical evaluation of MSC-WA in the N-Puzzle*



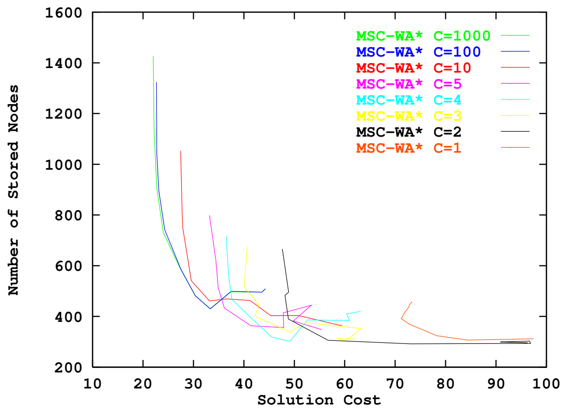
a) Solution cost versus W



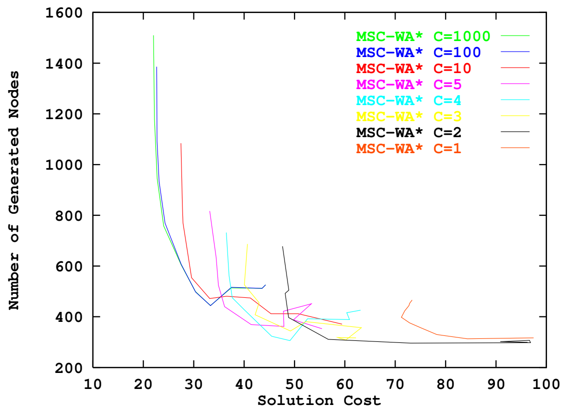
b) Memory usage versus W



c) Search effort versus W

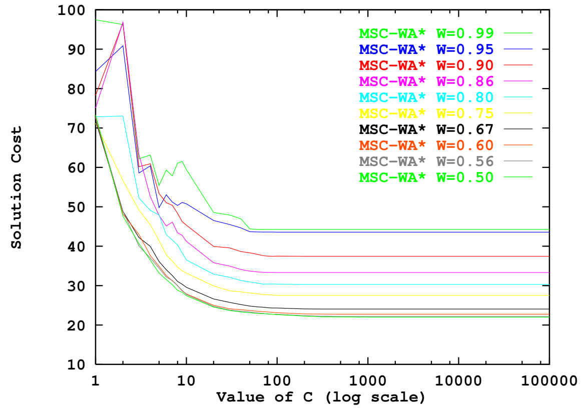


d) Memory usage versus solution cost

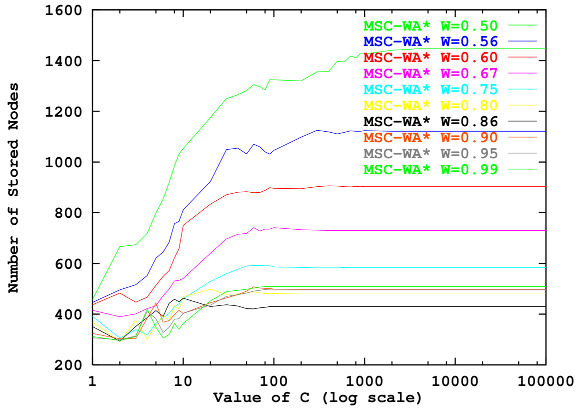


e) Search effort versus solution cost

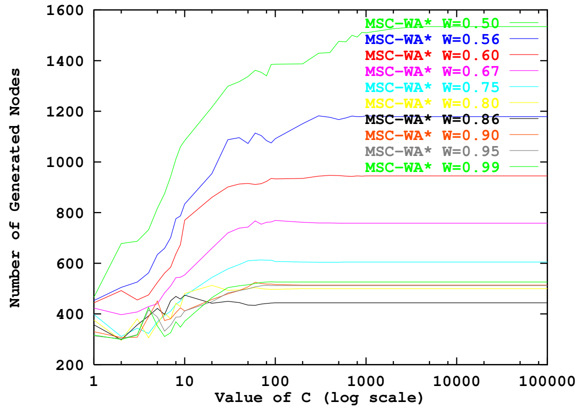
Figure 95: Performance of MSC-WA* in the 8-Puzzle with varying W



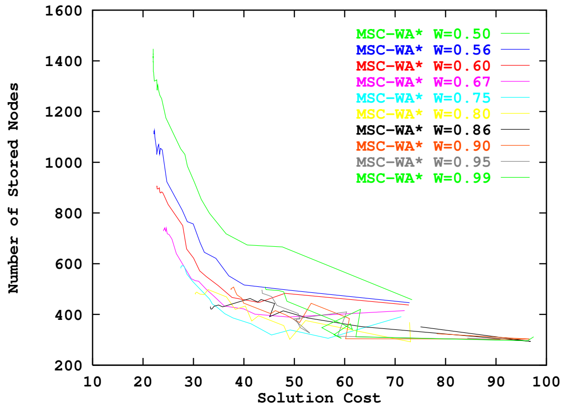
a) Solution cost versus C

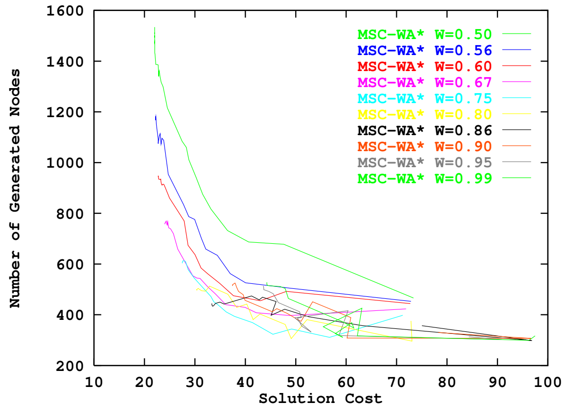


b) Memory usage versus C



c) Search effort versus C

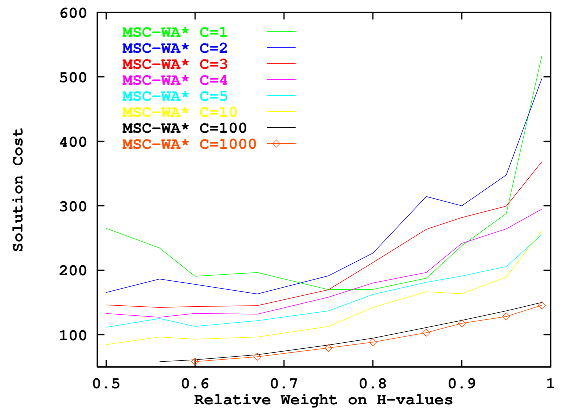




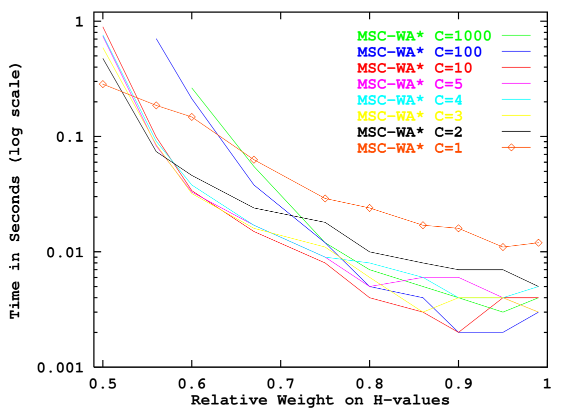
d) Memory usage versus solution cost

e) Search effort versus solution cost

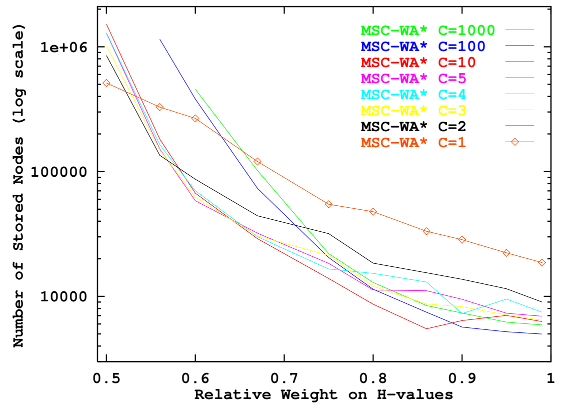
Figure 96: Performance of MSC-WA* in the 8-Puzzle with varying C



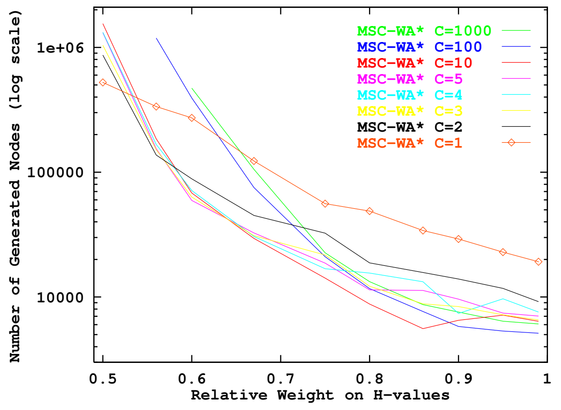
a) Solution cost versus W



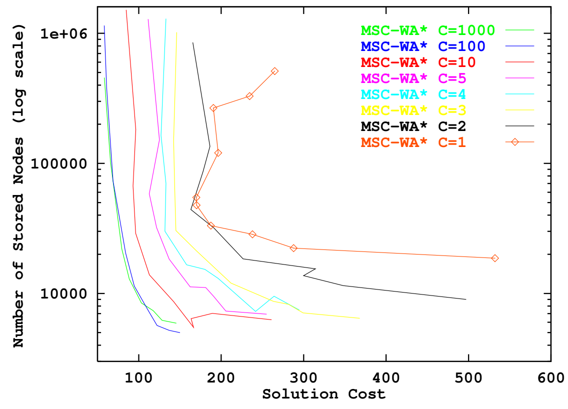
b) Runtime versus W



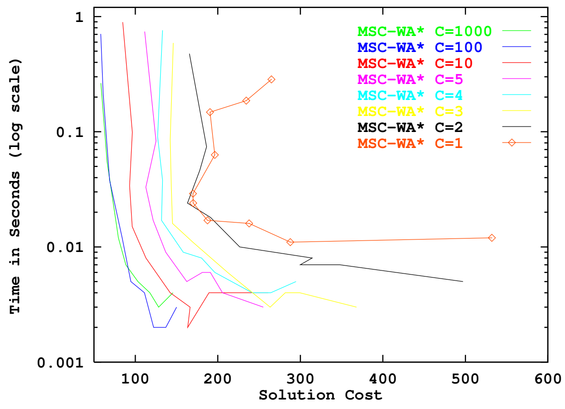
c) Memory usage versus W



d) Search effort versus W

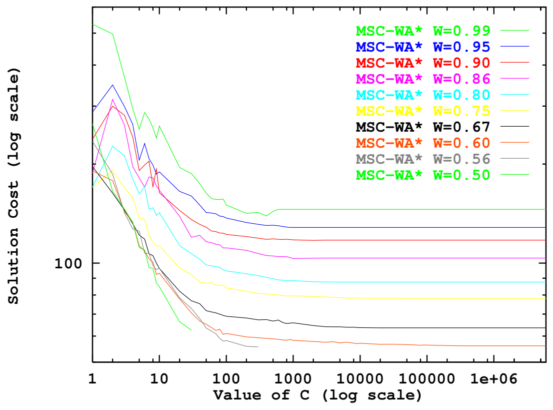


e) Memory usage versus solution cost

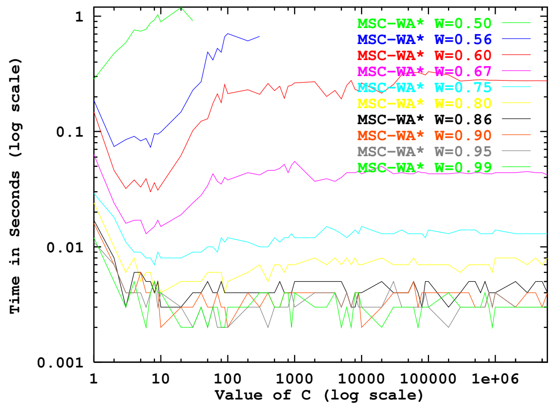


f) Runtime versus solution cost

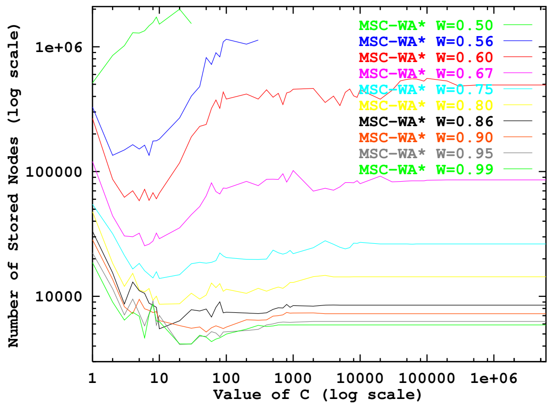
Figure 97: Performance of MSC-WA* in the 15-Puzzle with varying W



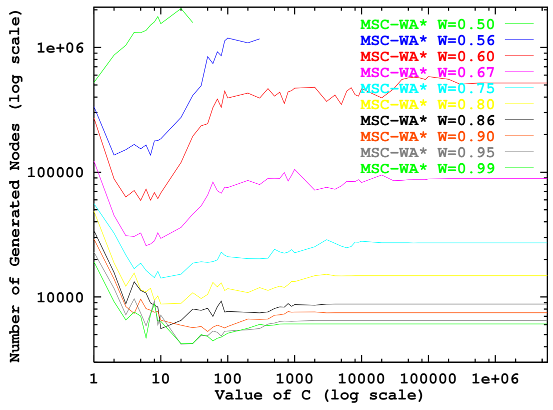
a) Solution cost versus C



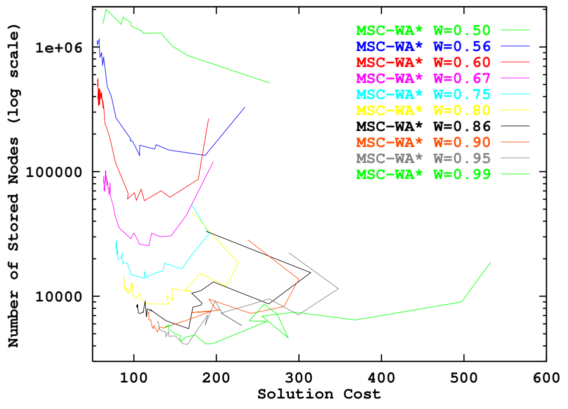
b) Runtime versus C



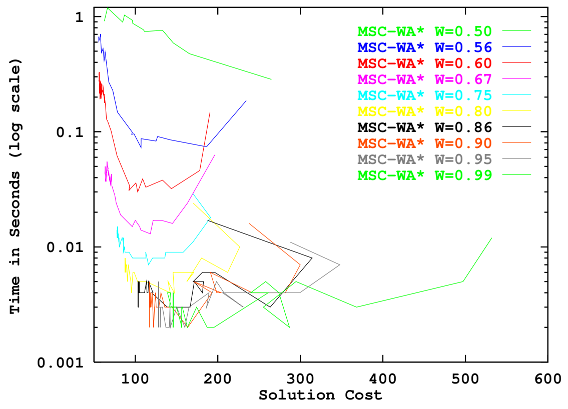
c) Memory usage versus C



d) Search effort versus C

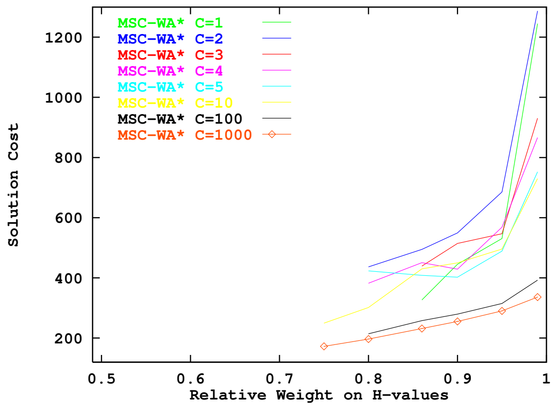


e) Memory usage versus solution cost

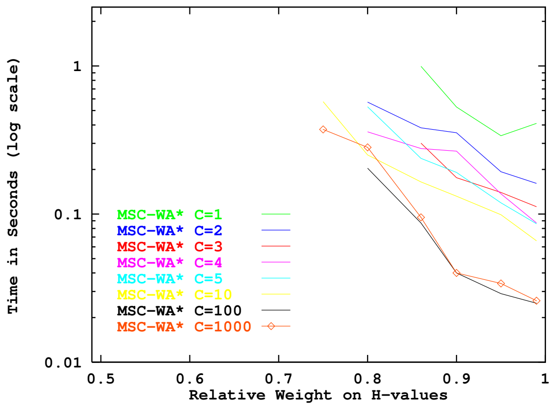


f) Runtime versus solution cost

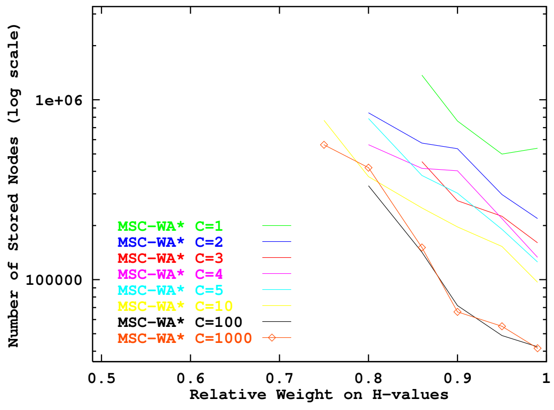
Figure 98: Performance of MSC-WA* in the 15-Puzzle with varying C



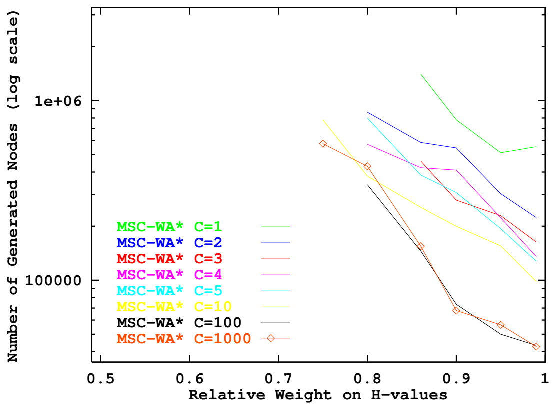
a) Solution cost versus W



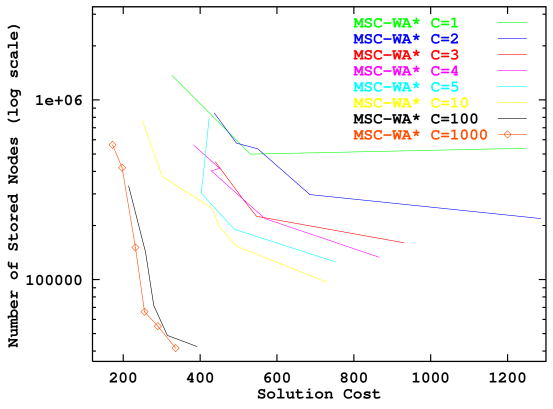
b) Runtime versus W



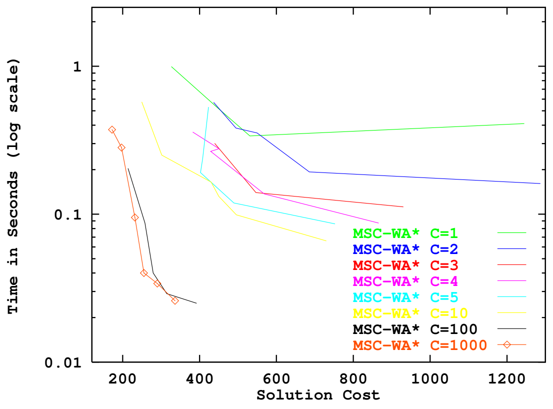
c) Memory usage versus W



d) Search effort versus W

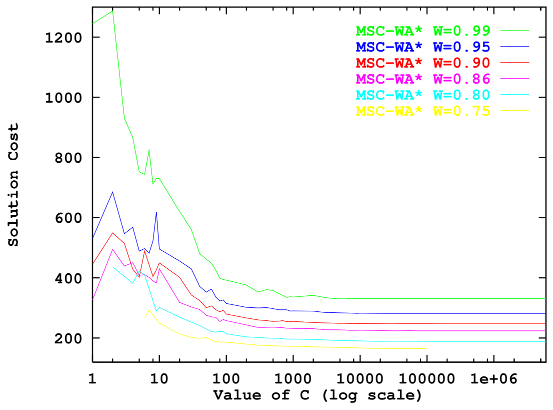


e) Memory usage versus solution cost

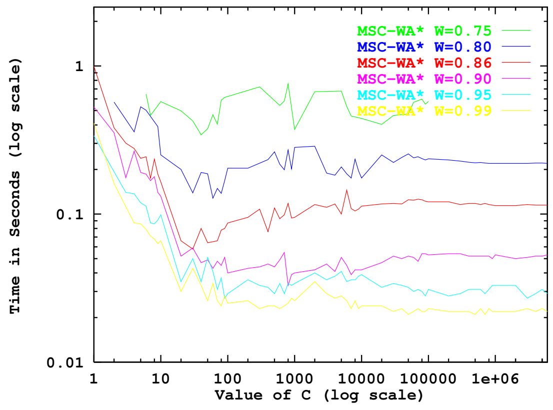


f) Runtime versus solution cost

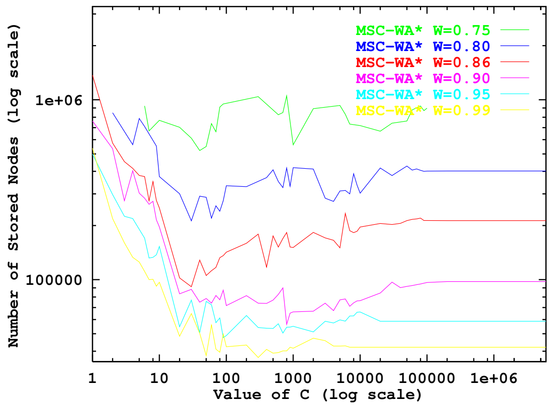
Figure 99: Performance of MSC-WA* in the 24-Puzzle with varying W



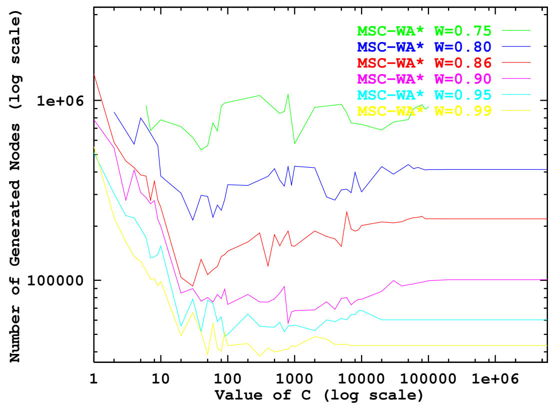
a) Solution cost versus C



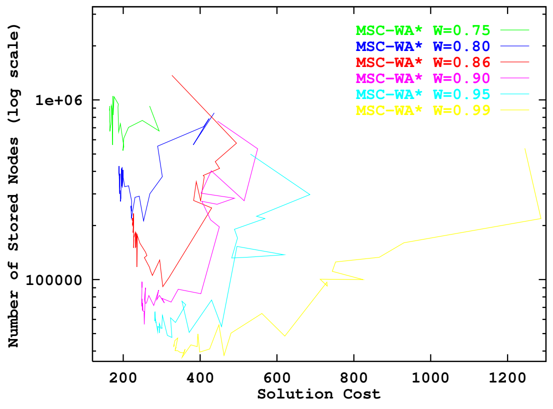
b) Runtime versus C



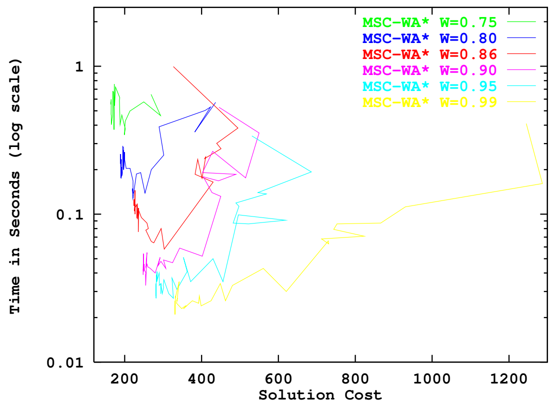
c) Memory usage versus C



d) Search effort versus C



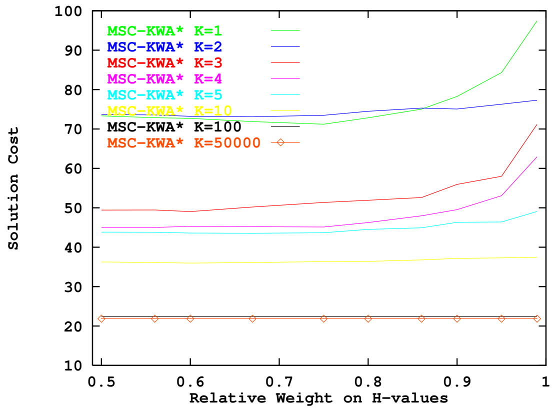
e) Memory usage versus solution cost



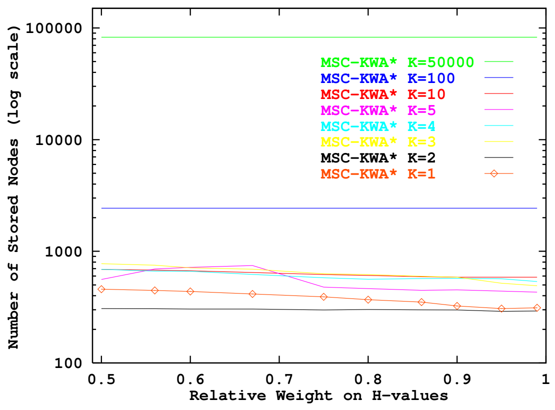
f) Runtime versus solution cost

Figure 100: Performance of MSC-WA* in the 24-Puzzle with varying C

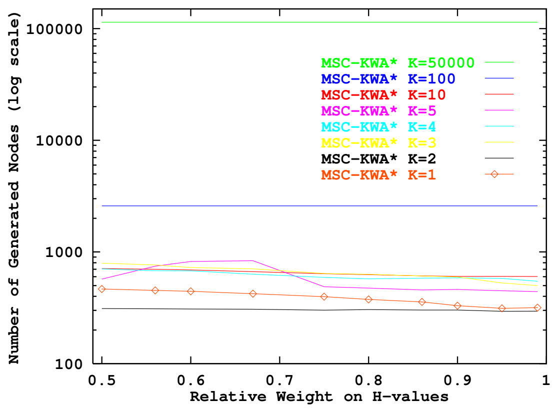
B.3 Empirical evaluation of MSC-KWA in the N-Puzzle*



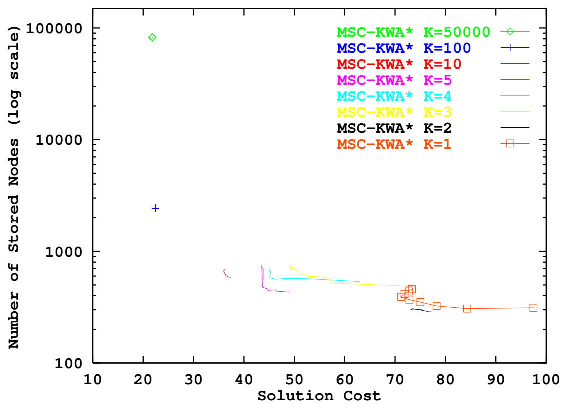
a) Solution cost versus W



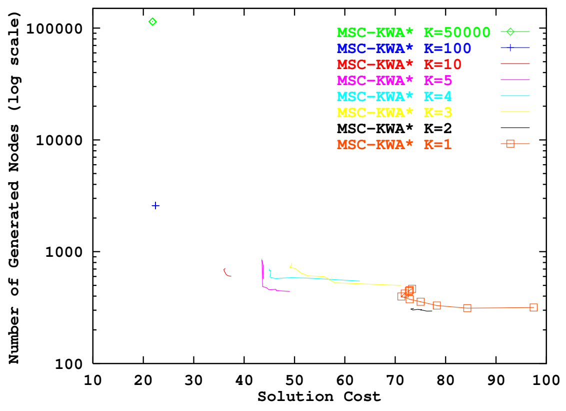
b) Memory usage versus W



c) Search effort versus W

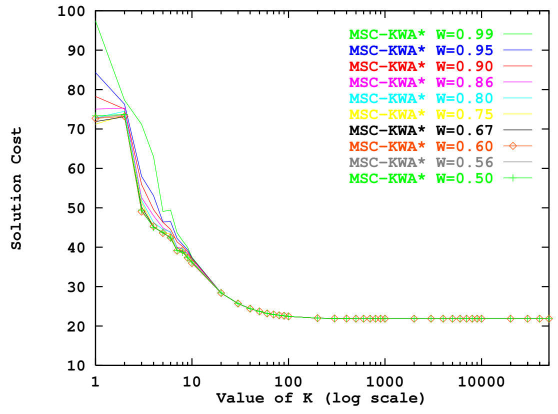


d) Memory usage versus solution cost

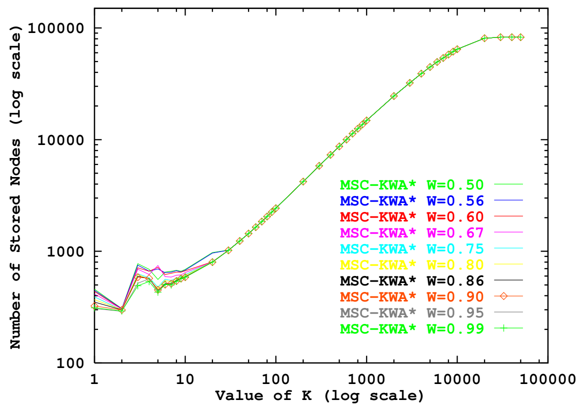


e) Search effort versus solution cost

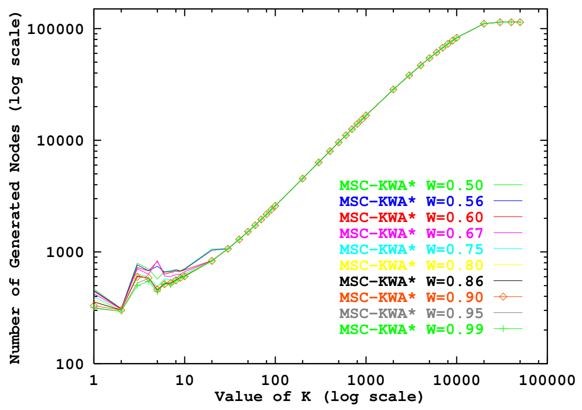
Figure 101: Performance of MSC-KWA* in the 8-Puzzle with varying W



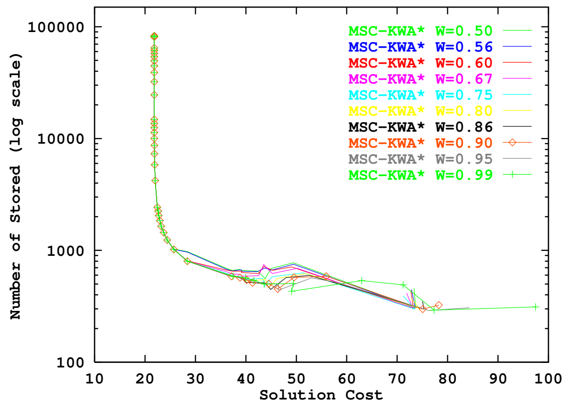
a) Solution cost versus K



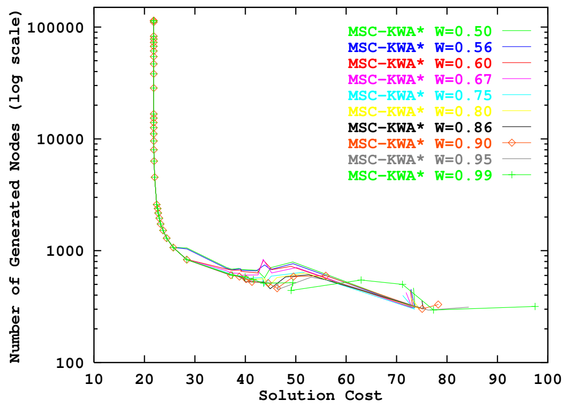
b) Memory usage versus K



c) Search effort versus K

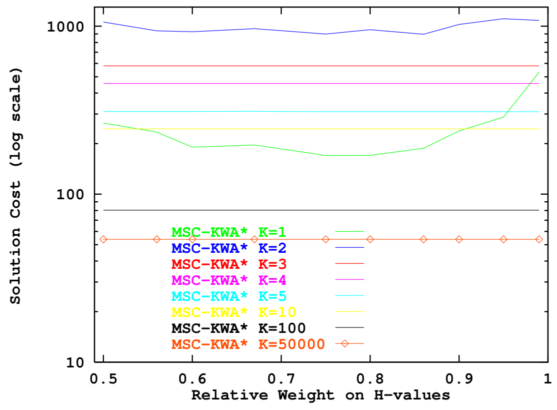


d) Memory usage versus solution cost

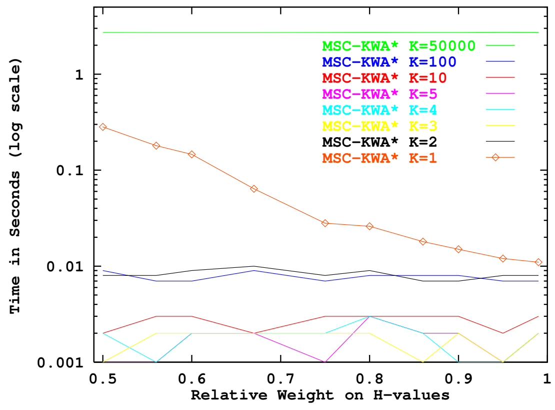


e) Search effort versus solution cost

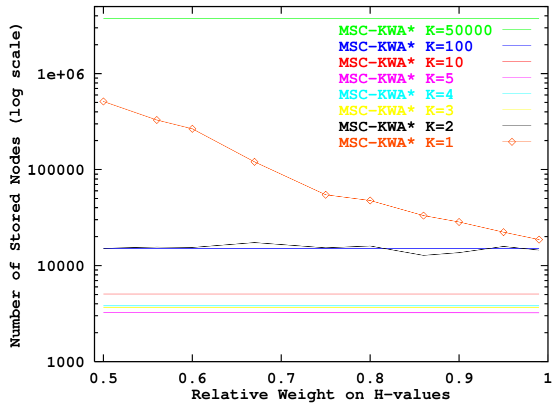
Figure 102: Performance of MSC-KWA* in the 8-Puzzle with varying K



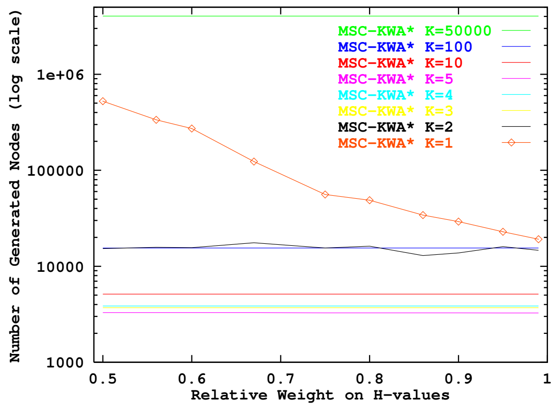
a) Solution cost versus W



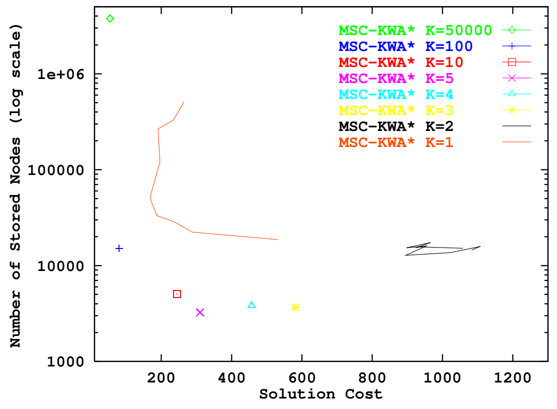
b) Runtime versus W



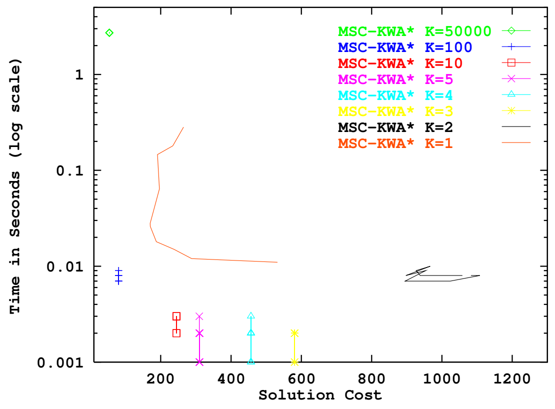
c) Memory usage versus W



d) Search effort versus W

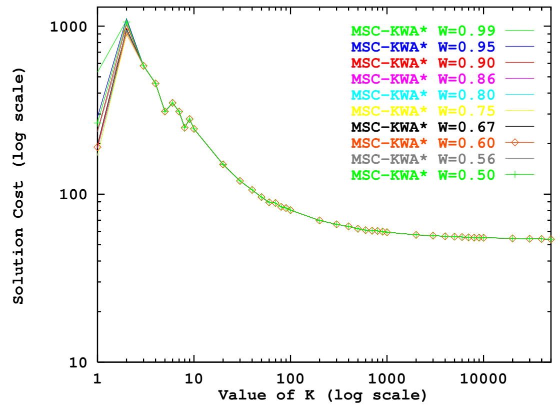


e) Memory usage versus solution cost

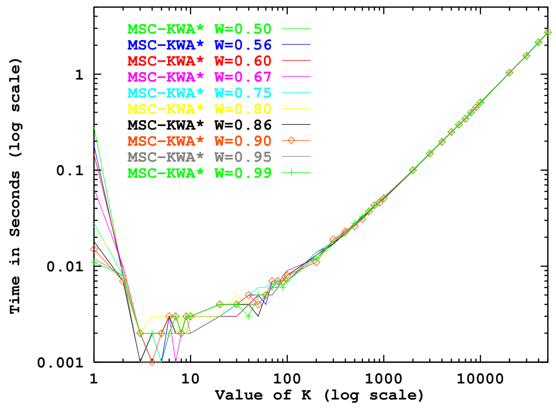


f) Runtime versus solution cost

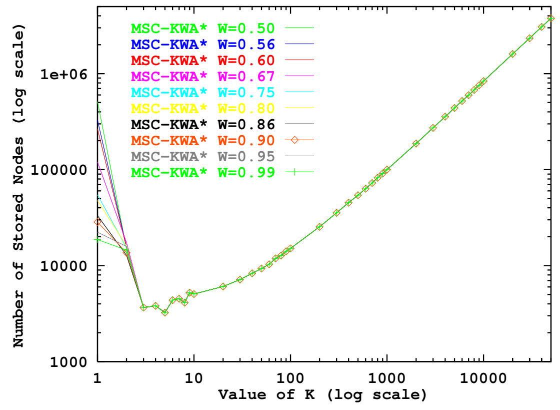
Figure 103: Performance of MSC-KWA* in the 15-Puzzle with varying W



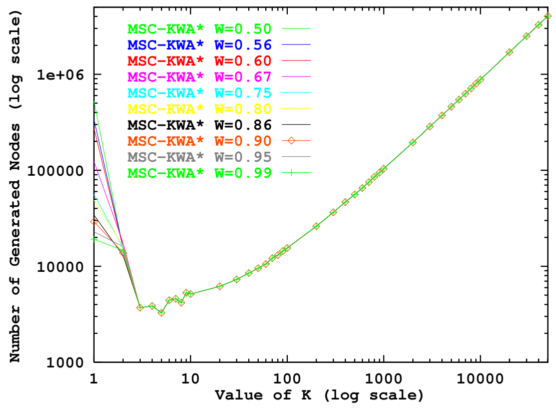
a) Solution cost versus K



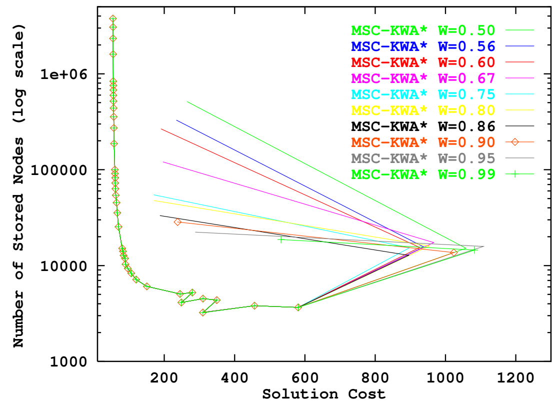
b) Runtime versus K



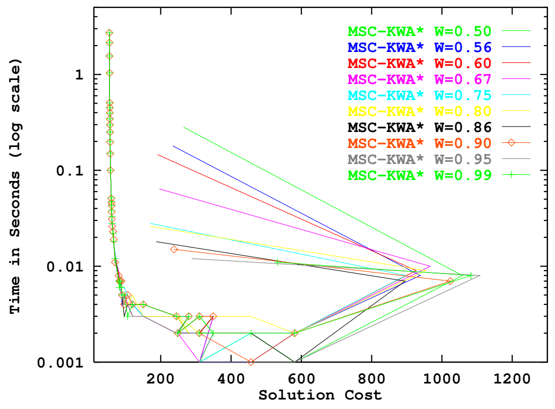
c) Memory usage versus K



d) Search effort versus K

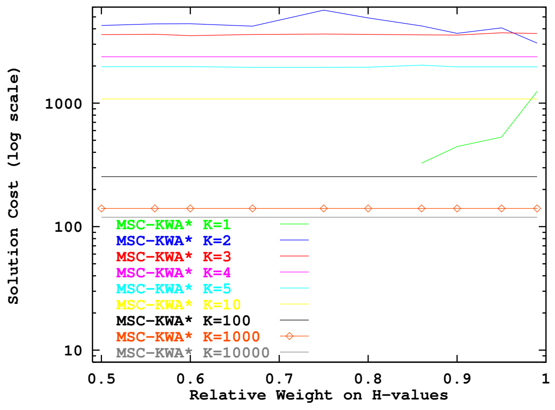


e) Memory usage versus solution cost

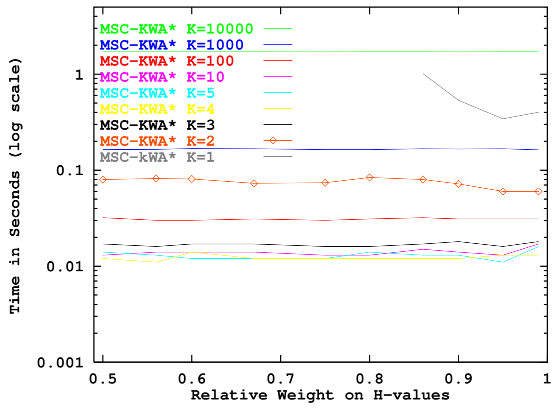


f) Runtime versus solution cost

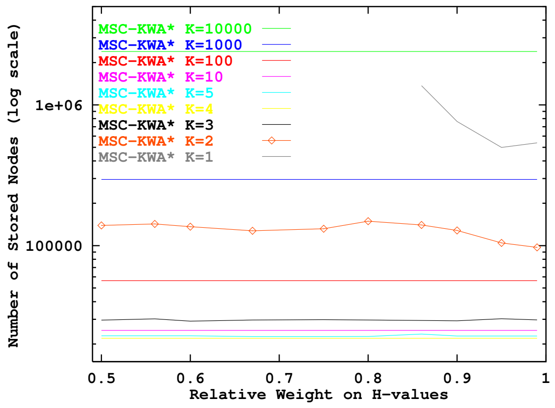
Figure 104: Performance of MSC-KWA* in the 15-Puzzle with varying K



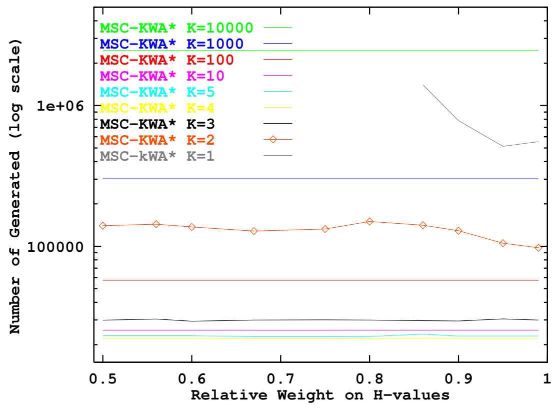
a) Solution cost versus W



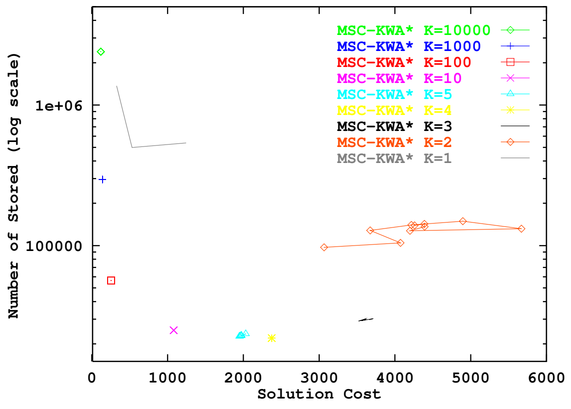
b) Runtime versus W



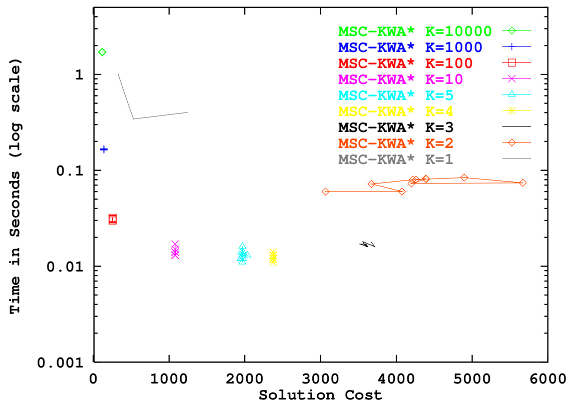
c) Memory usage versus W



d) Search effort versus W

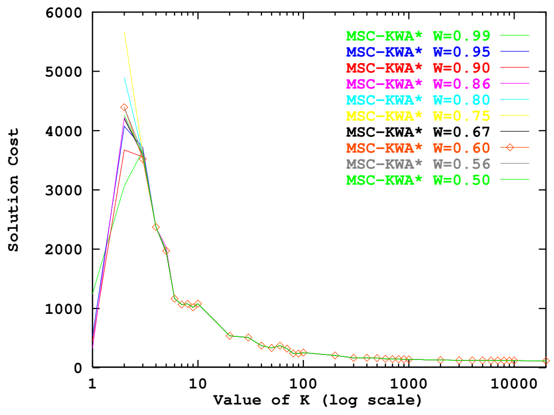


e) Memory usage versus solution cost

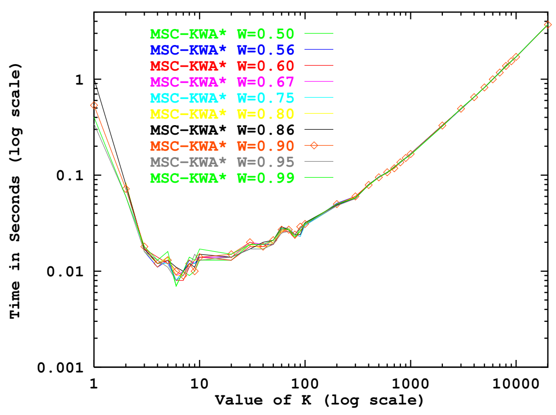


f) Runtime versus solution cost

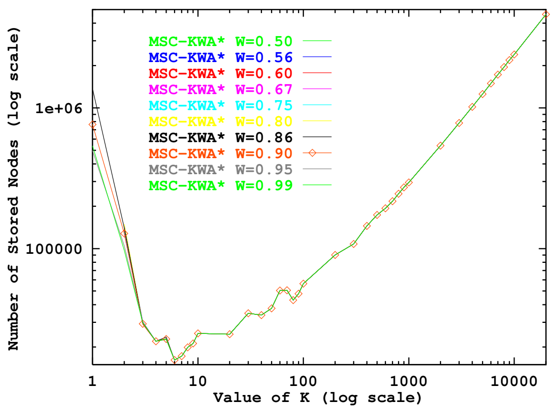
Figure 105: Performance of MSC-KWA* in the 24-Puzzle with varying W



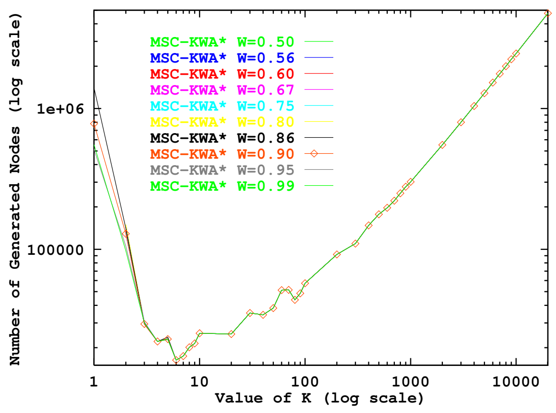
a) Solution cost versus K



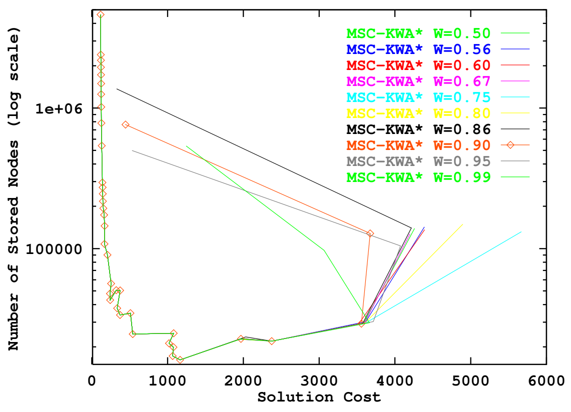
b) Runtime versus K



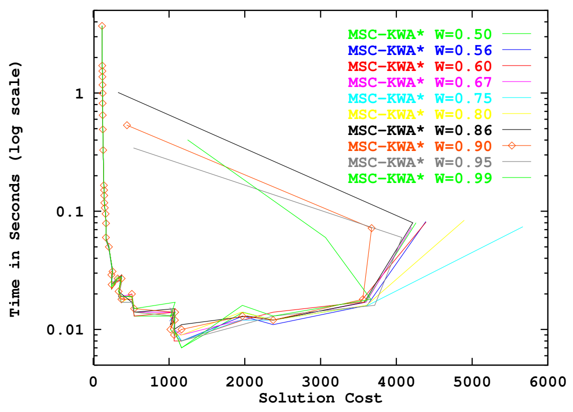
c) Memory usage versus K



d) Search effort versus K

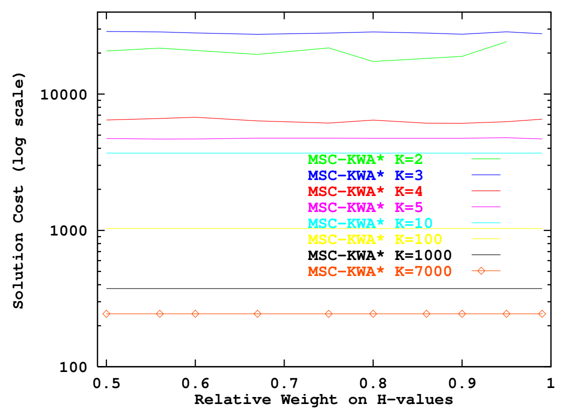


e) Memory usage versus solution cost

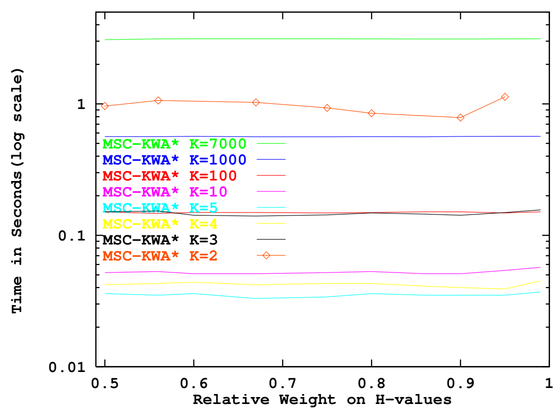


f) Runtime versus solution cost

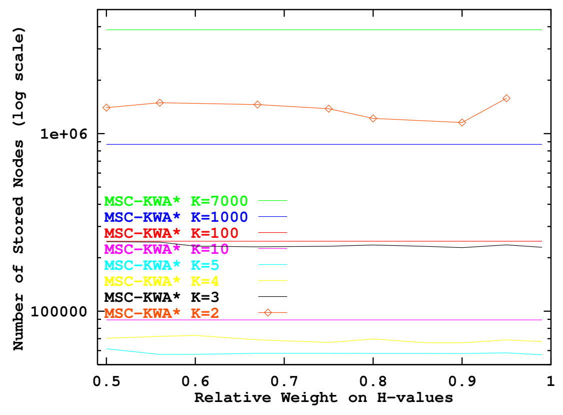
Figure 106: Performance of MSC-KWA* in the 24-Puzzle with varying K



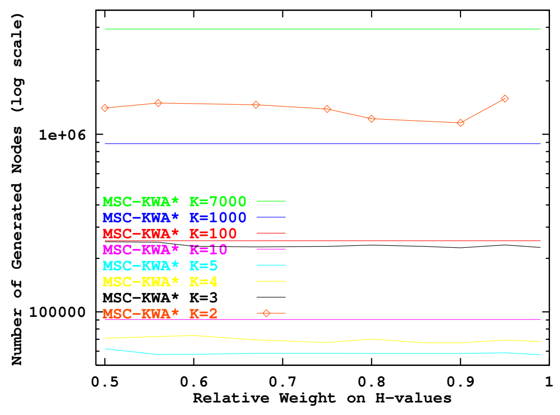
a) Solution cost versus W



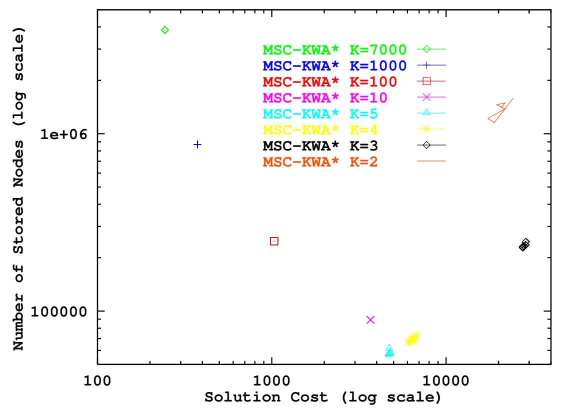
b) Runtime versus W



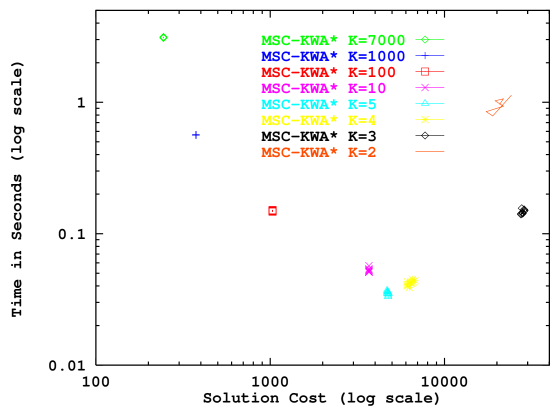
c) Memory usage versus W



d) Search effort versus W

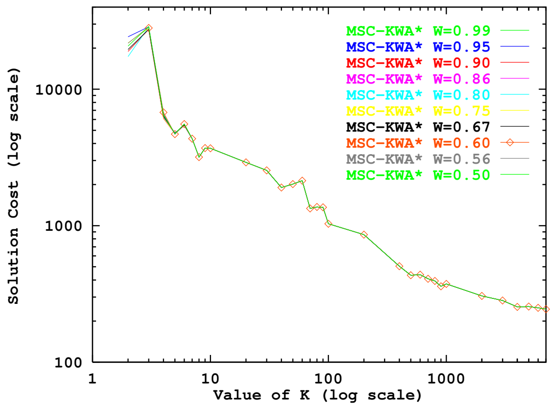


e) Memory usage versus solution cost

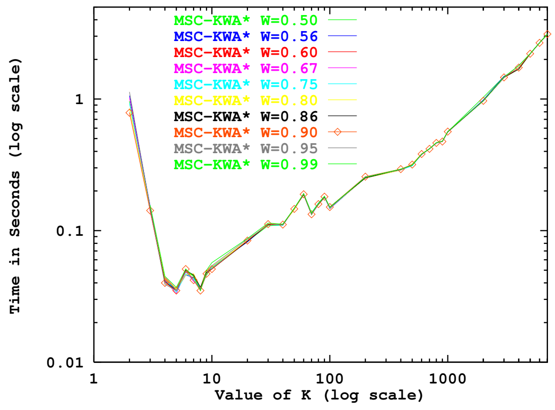


f) Runtime versus solution cost

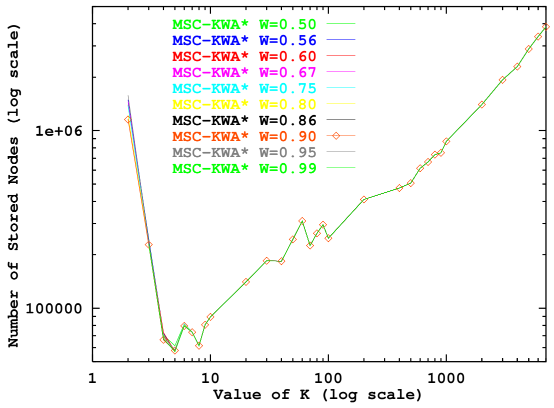
Figure 107: Performance of MSC-KWA* in the 35-Puzzle with varying W



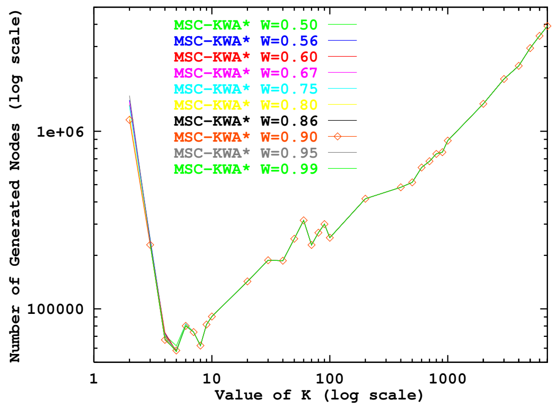
a) Solution cost versus K



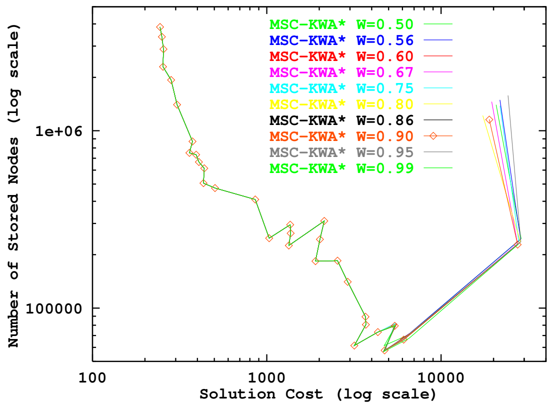
b) Runtime versus K



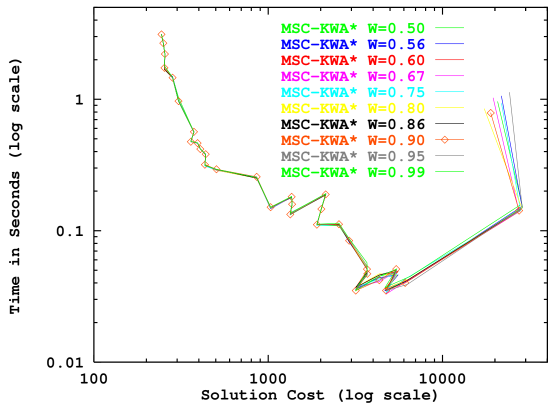
c) Memory usage versus K



d) Search effort versus K

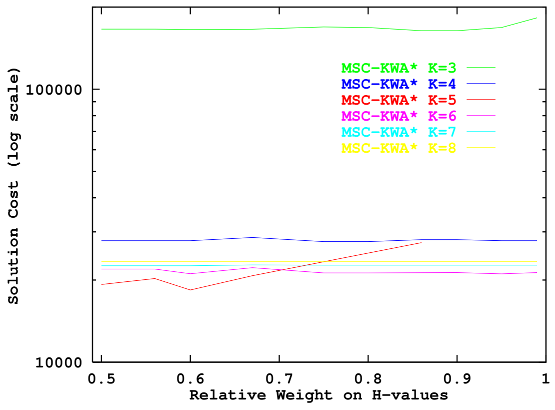


e) Memory usage versus solution cost

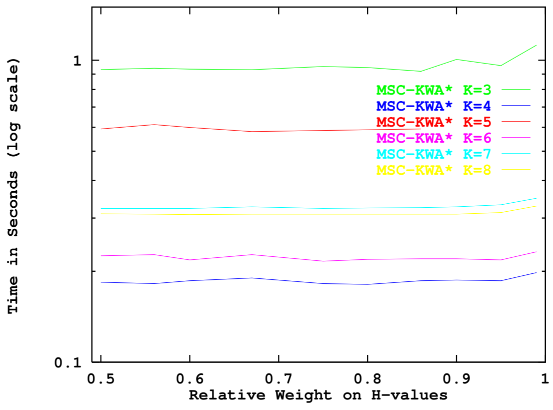


f) Runtime versus solution cost

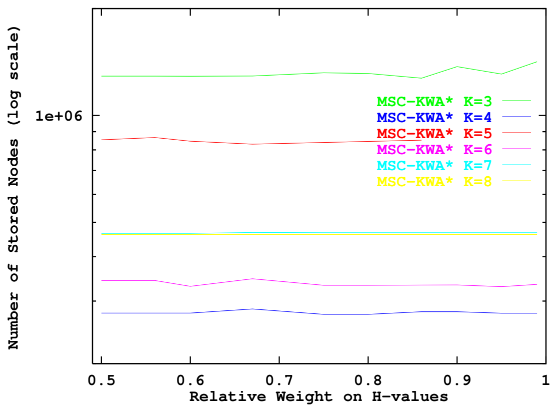
Figure 108: Performance of MSC-KWA* in the 35-Puzzle with varying K



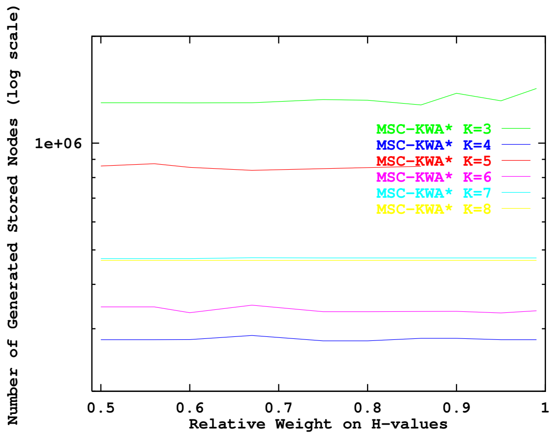
a) Solution cost versus W



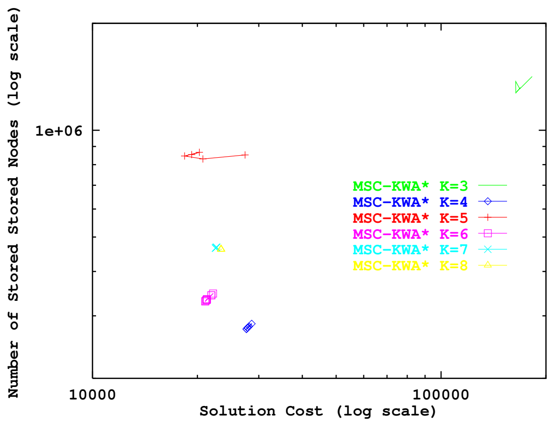
b) Runtime versus W



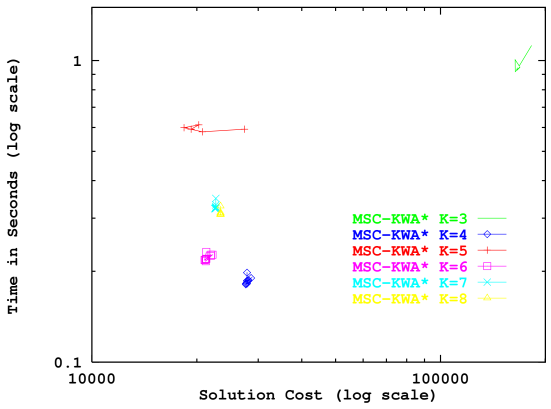
c) Memory usage versus W



d) Search effort versus W

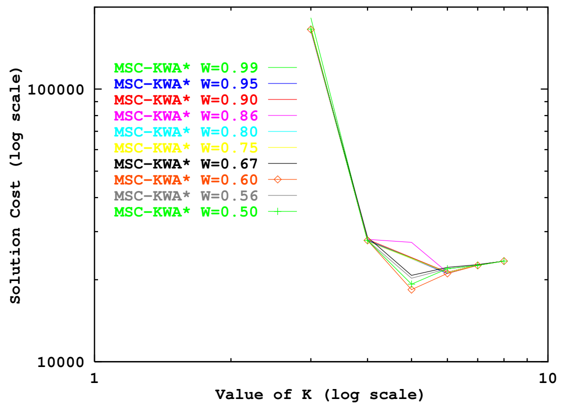


e) Memory usage versus solution cost

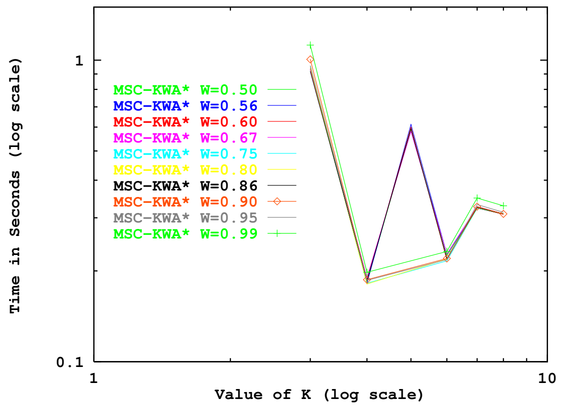


f) Runtime versus solution cost

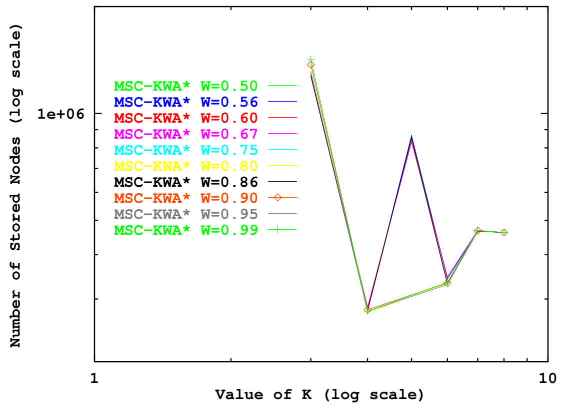
Figure 109: Performance of MSC-KWA* in the 48-Puzzle with varying W



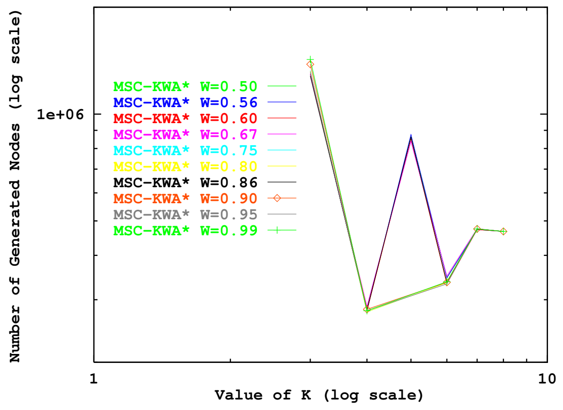
a) Solution cost versus K



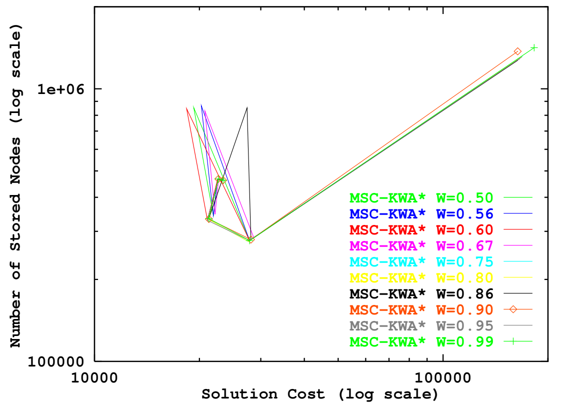
b) Runtime versus K



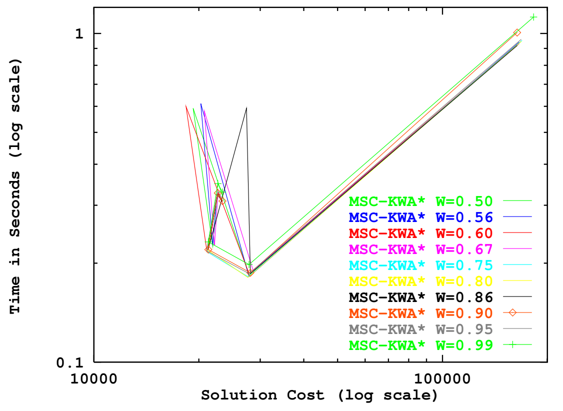
c) Memory usage versus K



d) Search effort versus K



e) Memory usage versus solution cost



f) Runtime versus solution cost

Figure 110: Performance of MSC-KWA* in the 48-Puzzle with varying K

REFERENCES

[1] AARTS, E. and LENSTRA, J., *Local Search in Combinatorial Optimization*. West Sussex, England: John Wiley & Sons, 1997.

[2] ALTSCHUL, S., GISH, W., MILLER, W., MYERS, E., and LIPMAN, D., "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[3] ALTSCHUL, S., MADDEN, T., SCHAFFER, A., ZHANG, J., ZHANG, Z., MILLER, W., and LIPMAN, D., "Gapped blast and psi-blast: A new generation of protein database search programs.," *Nucleic Acids Research*, vol. 25, pp. 3389–3402, 1997.

[4] ANANTHARAMAN, T. and BISIANI, R., "Hardware accelerators for speech recognition algorithms," in *Proceedings of the 13th International Symposium on Computer Architecture*, vol. 14 (2), pp. 216–223, IEEE, June 1986.

[5] ANDERSON, J. and LEBIERE, C., *The Atomic Components of Thought*. Mahwah, New Jersey: Lawrence Earlbaum, 1998.

[6] BARTO, A., BRADTKE, S., and SINGH, S., "Learning to act using real-time dynamic programming," *Artificial Intelligence*, vol. 73, no. 1, pp. 81–138, 1995.

[7] BISIANI, R., "Beam search," in *Encyclopedia of Artificial Intelligence* (SHAPIRO, S., ed.), pp. 56–58, New York : Wiley & Sons, 1987.

[8] BOARDMAN, J., GARRETT, C., and ROBSON, G., "A recursive algorithm for the optimal solution of a complex allocation problem using a dynamic programming formulation," *The Computer Journal*, vol. 29, pp. 182–186, Apr. 1986.

[9] BOARDMAN, J. and ROBSON, G., "Towards a problem-solving methodology for coping with increasing complexity: An engineering approach," *The Computer Journal*, vol. 29, pp. 161–166, Apr. 1986.

[10] BODDY, M., "Anytime problem solving using dynamic programming," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 738–743, 1991.

[11] BODDY, M. and DEAN, T., "Solving time-dependent planning problems," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 979–984, 1989.

[12] BONET, B. and GEFFNER, H., "Heuristic search planner 2.0," *AI Magazine*, vol. 22, no. 3, pp. 77–80, 2001.

[13] BONET, B. and GEFFNER, H., "Planning as heuristic search," *Artificial Intelligence*, vol. 129, no. 1–2, pp. 5–33, 2001. Special Issue on Heuristic Search.

[14] BONET, B., LOERINCS, G., and GEFFNER, H., "A robust and fast action selection mechanism," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 714–719, 1997.

[15] BRIGHT, J., KASIF, S., and STILLER, L., "Exploiting algebraic structure in parallel state space search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1341–1346, 1994.

[16] BULITKO, V., "Learning in real time search: A unifying framework." In preparation.

[17] BULITKO, V., "Learning for adaptive real-time search." Published in the Computing Research Repository (CoRR) as cs.AI/0407016, online at http://www.acm.org/repository, July 6, 2004.

[18] CARRILLO, H. and LIPMAN, D., "The multiple sequence alignment problem in biology," *SIAM Journal on Applied Mathematics*, vol. 48, pp. 1073–1082, October 1988.

[19] CHAKRABARTI, P., GHOSE, S., ACHARYA, A., and DE SARKAR, S., "Heuristic search in restricted memory," *Artificial Intelligence*, vol. 41, pp. 197–221, Dec. 1989.

[20] CHU, L.-C. and WAH, B., "Band search: An efficient alternative to guided best-first search," in *Proceedings of the International Conference on Tools for Artificial Intelligence*, pp. 154–161, IEEE Computer Society Press, Nov. 1992.

[21] CHU, L.-C. and WAH, B., "Solution of constrained optimization problems in limited time," in *IEEE Workshop on Imprecise Computation*, Dec. 1992.

[22] CULBERSON, J. and SCHAEFFER, J., "Searching with pattern databases," in *Proceedings of the Eleventh Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-96)*, vol. 1081 of *LNAI*, (Berlin), pp. 402–416, Springer-Verlag, 1996.

[23] CULBERSON, J. and SCHAEFFER, J., "Pattern databases," *Computational Intelligence*, vol. 14, no. 4, pp. 318–334, 1998.

[24] DAVIS, H., BRAMANTI-GREGOR, A., and WANG, J., "The advantages of using depth and breadth components in heuristic search," in *Methodologies for Intelligent Systems 3*, pp. 19–28, 1988.

[25] DAYHOFF, M., SCHWARTZ, R., and ORCUTT, B., "A model of evolutionary change in proteins," in *Atlas of Protein Structure* (DAYHOFF, M., ed.), vol. 5(Suppl. 3), pp. 345–352, Silver Spring, Md.: National Biomedical Research Foundation, 1978.

[26] DE LIEFVOORT, A. V., "An iterative algorithm for the Reve's puzzle," *The Computer Journal*, vol. 35, no. 1, pp. 91–92, 1992.

[27] DEAN, T. and BODDY, M., "An analysis of time-dependent planning," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 49–54, 1988.

[28] DIETTERICH, T. and MICHALSKI, R., "Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods," *Artificial Intelligence*, vol. 16, pp. 257–294, 1981.

[29] DIJKSTRA, E., "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[30] DORAN, J. and MICHIE, D., "Experiments with the Graph Traverser program," *Proceedings of the Royal Society of London*, vol. 294, Series A, pp. 235–259, 1966.

[31] DORST, L., MANDHYAN, I., and TROVATO, K., "The geometrical representations of path planning problems," *Robotics and Autonomous Systems*, vol. 7, pp. 181–195, 1991.

[32] DUNKEL, O., "Editorial note concerning advanced problem 3918," *American Mathematical Monthly*, vol. 48, p. 219, 1941.

[33] DURBIN, R., EDDY, S., KROGH, A., and MITCHISON, G., *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[34] EDELKAMP, S. and ECKERLE, J., "New strategies in real-time heuristic search," in *Proceedings of the AAAI-97 Workshop on On-Line Search*, pp. 30–35, AAAI Press, 1997.

[35] EDELKAMP, S. and KORF, R., "The branching factor of regular search spaces," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 299–304, 1998.

[36] FELNER, A., KORF, R., and HANAN, S., "Additive pattern databases." Accepted for publication in the Journal of Artificial Intelligence Research (JAIR). July 2004. To appear.

[37] FELNER, A., KRAUS, S., and KORF, R., "KBFS: K-best-first search," *Annals of Mathematics and Artificial Intelligence*, vol. 39, pp. 19–39, 2003.

[38] FELNER, A., MESHULAM, R., HOLTE, R., and KORF, R., "Compressing pattern databases," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 638–643, 2004.

[39] FELNER, A., STERN, R., BEN-YAIR, A., KRAUS, S., and NETANYAHU, N., "PHA*: Finding the shortest path with A* in unknown physical environments," *Journal of Artificial Intelligence Research (JAIR)*, vol. 21, pp. 631–670, 2004.

[40] FENG, D.-F. and DOOLITTLE, R., "Progressive sequence alignment as a prerequisite to correct phylogenetic trees.," *Journal of Molecular Evolution*, vol. 25, pp. 351–360, 1987.

[41] FIAT, A., MOSES, S., SHAMIR, A., SHIMSHONI, I., and TARDOS, G., "Planning and learning in permutation groups.," *Proceedings of the 30th A.C.M. Foundations of Computer Science Conference (FOCS)*, pp. 274–279, 1989.

[42] FINKELSTEIN, L. and MARKOVITCH, S., "A selective macro-learning algorithm and its application to the $N \times N$ sliding-tile puzzle," *Journal of Artificial Intelligence Research*, vol. 8, pp. 223–263, 1998.

[43] FIREBAUGH, M., *Artificial Intelligence: A Knowledge-Based Approach*. Boston: Boyd & Fraser, 1988.

[44] FOX, M., *Constraint-directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Carnegie-Mellon University, Pittsburgh, 1983.

[45] FRAME, J., "Solution to advanced problem 3918," *American Mathematical Monthly*, vol. 48, pp. 216–217, 1941.

[46] FURCY, D., "eFALCONS: Speeding up the convergence of real-time search even more," Tech. Rep. GIT-COGSCI-2001/04, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2001.

[47] FURCY, D., "Limited discrepancy beam search," Tech. Rep. GIT-COGSCI-2004/02, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), USA, 2004.

[48] FURCY, D., "Scaling up weighted A* with commitment and diversity," Tech. Rep. GIT-COGSCI-2004/01, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), USA, 2004.

[49] FURCY, D. and KOENIG, S., "Speeding up the convergence of real-time search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 891–897, 2000.

[50] FURCY, D. and KOENIG, S., "Speeding up the convergence of real-time search: Empirical setup and proofs," Tech. Rep. GIT-COGSCI-2000/01, College of Computing, Georgia Institute of Technology, Atlanta (Georgia), 2000.

[51] FURCY, D. and KOENIG, S., "Combining two fast-learning real-time search algorithms yields even faster learning," in *Proceedings of the European Conference on Planning (ECP)*, 2001.

[52] GASCHNIG, J., "Performance measurement and analysis of certain search algorithms," Technical Report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University, 1979. Ph. D. Thesis.

[53] GEFFNER, H. and BONET, B., "Solving large POMDPs by real-time dynamic programming," tech. rep., Departamento de Computación, Universidad Simón Bolivar, Caracas (Venezuela), 1998.

[54] GINSBERG, M. and HARVEY, W., "Iterative broadening," *Artificial Intelligence*, vol. 55, pp. 367–383, June 1992.

[55] GOTOH, O., "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[56] HANSEN, E. and ZILBERSTEIN, S., "Anytime heuristic search: Preliminary report," in *Proceedings of the AAAI Fall Symposium on Flexible Computation in Intelligent Systems; Results, Issues and Opportunities*, pp. 55–59, 1996.

[57] HANSEN, E., ZILBERSTEIN, S., and DANILCHENKO, V., "Anytime heuristic search: First results," Tech. Rep. CMPSCI 97–50, Department of Computer Science, University of Massachusetts, Amherst (Massachusetts), September 1997.

[58] HANSSON, O., MAYER, A., and YUNG, M., "Criticizing solutions to relaxed models yields powerful admissible heuristics," *Information Sciences*, vol. 63, pp. 207–227, 1992.

[59] HART, P., NILSSON, N., and RAPHAEL, B., "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4, pp. 100–107, July 1968.

[60] HART, P., NILSSON, N., and RAPHAEL, B., "Correction to "A formal basis for the heuristic determination of minimum cost paths"," *SIGART Newsletter*, vol. 37, pp. 28–29, 1972.

[61] HARVEY, W. and GINSBERG, M., "Limited discrepancy search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 607–615, 1995.

[62] HAYES, P., "A note on the Towers of Hanoi problem," *The Computer Journal*, vol. 20, no. 3, pp. 282–285, 1977.

[63] HERNÁDVÖLGYI, I., "Searching for macro operators with automatically generated heuristics," in *Proceedings of the 14th Canadian Conference on Artificial Intelligence (AI-2001)*, pp. 194–203, 2001.

[64] HINZ, A., "The Tower of Hanoi," in *Algebras and Combinatorics, An International Congress, ICAC'97* (SHUM, K.-P., TAFT, E., and WAN, Z.-X., eds.), pp. 277–289, Hong Kong: Springer, 1999.

[65] HOEBEL, L. and ZILBERSTEIN, S., eds., *Proceedings of the AAAI Workshop on Building Resource-Bounded Reasoning Systems.* AAAI Press, 1997.

[66] HOFFMANN, J. and NEBEL, B., "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.

[67] HOHWALD, H., THAYER, I., and KORF, R., "Comparing best-first search and dynamic programming for optimal multiple sequence alignment," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1239–1245, 2003.

[68] HOLTE, R., DRUMMOND, C., PEREZ, M., ZIMMER, R., and MACDONALD, A., "Searching with abstractions: A unifying framework and new high-performance algorithm," in *Proceedings of the Canadian Conference on Artificial Intelligence*, pp. 263–270, 1994.

[69] HOLTE, R., NEWTON, J., FELNER, A., MESHULAM, R., and FURCY, D., "Multiple pattern databases," in *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 122–131, 2004.

[70] HOLTE, R., PEREZ, M., ZIMMER, R., and MACDONALD, A., "Hierarchical A*: Searching abstraction hierarchies efficiently," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 530–535, 1996.

[71] HORVITZ, E., "Reasoning about beliefs and actions under computational resource constraints," in *Proceedings of the AAAI Workshop on Uncertainty in Artificial Intelligence*, July 1987.

[72] HORVITZ, H. and ZILBERSTEIN, S., eds., *Proceedings of the AAAI Fall Symposium on Flexible Computation in Intelligent Systems: Results, Issues, and Opportunities.* AAAI Press, 1996.

[73] IBARAKI, T., "Depth-*m* search in branch-and-bound algorithms," *International Journal of Computer and Information Sciences*, vol. 7, no. 4, pp. 315–343, 1978.

[74] IKEDA, T. and IMAI, H., "Fast A* algorithms for multiple sequence alignment," in *Proceedings of the Genome Informatics Workshop IV*, pp. 90–99, Universal Academy Press, 1994.

[75] IKEDA, T. and IMAI, H., "Enhanced A* algorithms for multiple alignments: Optimal alignments for several sequences and k-opt approximate alignments for large cases," *Theoretical Computer Science*, vol. 210, pp. 341–374, Jan. 1999.

[76] ISHIDA, T., "Moving target search with intelligence," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 525–532, 1992.

[77] ISHIDA, T., "Two is not always better than one: Experiences in real-time bidirectional search," in *Proceedings of the International Conference on Multi-Agent Systems*, pp. 185–192, 1995.

[78] ISHIDA, T., "Real-time bidirectional search: Coordinated problem solving in uncertain situations," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, pp. 617–628, June 1996.

[79] ISHIDA, T., *Real-Time Search for Learning Autonomous Agents*. Kluwer Academic Publishers, 1997.

[80] ISHIDA, T., "Real-time search for autonomous agents and multiagent systems," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 2, pp. 139–167, 1998.

[81] ISHIDA, T. and KORF, R., "Moving target search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 204–210, 1991.

[82] ISHIDA, T. and KORF, R., "Moving target search: A realtime search for changing goals," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 6, pp. 606–619, 1995.

[83] ISHIDA, T. and SHIMBO, M., "Improving the learning efficiencies of real-time search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 305–310, 1996.

[84] JOHNSON, W. W. and STORY, W. E., "Notes on the "15" puzzle," *American Journal of Mathematics*, vol. 2, pp. 397–404, 1879.

[85] KAELBLING, L., LITTMAN, M., and MOORE, A., "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[86] KAINDL, H. and KHORSAND, A., "Memory-bounded bidirectional search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1359–1364, 1994.

[87] KITAMURA, Y., TERANISHI, K., and TATSUMI, S., "Organizational strategies for multiagent real-time search," in *Proceedings of the First International Conference on Multi–Agent Systems*, pp. 150–156, 1995.

[88] KITAMURA, Y., YOKOO, M., MIYAJI, T., and TATSUMI, S., "Multi-state commitment search," in *Proceedings of the International Conference on Tools for Artificial Intelligence*, pp. 431–439, 1998.

[89] KNIGHT, K., "Are many reactive agents better than a few deliberative ones?," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 432–437, 1993.

[90] KOENIG, S., "Exploring unknown environments with real-time search or reinforcement learning," in *Proceedings of the Neural Information Processing Systems*, pp. 1003–1009, 1999.

[91] KOENIG, S., "Agent-centered search," *Artificial Intelligence Magazine*, vol. 22, no. 4, pp. 109–131, 2001.

[92] KOENIG, S., FURCY, D., and BAUER, C., "Heuristic search-based replanning," in *International Conference on Artificial Intelligence Planning & Scheduling (AIPS)*, pp. 310–317, 2002.

[93] KOENIG, S., LIKHACHEV, M., and FURCY, D., "Lifelong Planning A*," *Artificial Intelligence*, vol. 155, pp. 93–146, May 2004.

[94] KOENIG, S. and SIMMONS, R., "Solving robot navigation problems with initial pose uncertainty using real-time heuristic search," in *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pp. 154–153, 1998.

[95] KORF, R., "Towards a model of representation changes," *Artificial Intelligence*, vol. 14, pp. 41–78, 1980.

[96] KORF, R., "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, pp. 97–109, 1985.

[97] KORF, R., "Macro-operators: A weak method for learning," *Artificial Intelligence*, vol. 26, pp. 35–77, 1985.

[98] KORF, R., "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990.

[99] KORF, R., "Linear-space best-first search," *Artificial Intelligence*, vol. 62, no. 1, pp. 41–78, 1993.

[100] KORF, R., "Improved limited discrepancy search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 286–291, 1996.

[101] KORF, R., "Finding optimal solutions to Rubik's cube using pattern databases," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 700–705, 1997.

[102] KORF, R., "Divide-and-conquer bidirectional search: First results," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1184–1189, 1999.

[103] KORF, R., "Delayed duplicate detection: Extended abstract," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1539–1541, 2003.

[104] KORF, R., "Best-first frontier search with delayed duplicate detection," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 650–657, 2004.

[105] KORF, R. and FELNER, A., "Disjoint pattern database heuristics," *Artificial Intelligence*, vol. 134, no. 1, pp. 9–22, 2002.

[106] KORF, R. and REID, M., "Complexity analysis of admissible heuristic search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 305–310, 1998.

[107] KORF, R., REID, M., and EDELKAMP, S., "Time complexity of iterative-deepening-A^*," *Artificial Intelligence*, vol. 129, no. 1–2, pp. 199–218, 2001.

[108] KORF, R. and TAYLOR, L., "Finding optimal solutions to the twenty-four puzzle," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1202–1207, 1996.

[109] KORF, R. and ZHANG, W., "Divide-and-conquer frontier search applied to optimal sequence alignment," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 910–916, 2000.

[110] KUMAR, V., "Branch-and-bound search," in *Encyclopedia of Artificial Intelligence* (SHAPIRO, S. C., ed.), pp. 1000–1004, New York : Wiley, 2nd ed., 1990.

[111] LAWLER, E., LENSTRA, J., KAN, A. R., and SHMOYS, D., eds., *The Traveling Salesman Problem.* John Wiley and sons, 1985.

[112] LAWLER, E. and WOOD, D., "Branch-and-bound methods: A survey," *Operations Research*, vol. 14, no. 4, pp. 699–719, 1966.

[113] LERMEN, M. and REINERT, K., "The practical use of the A* algorithm for exact multiple sequence alignment," *Journal of Computational Biology*, vol. 7, no. 5, pp. 655–671, 2000.

[114] LIKHACHEV, M., GORDON, G., and THRUN, S., "ARA*: Formal analysis," Tech. Rep. CS-03-148, Carnegie Mellon University, Pittsburgh, PA, 2003.

[115] LIKHACHEV, M., GORDON, G., and THRUN, S., "ARA*: Anytime A* with provable bounds on sub-optimality," in *Proceedings of Advances in Neural Information Processing Systems 16 (NIPS)*, 2004.

[116] LIU, Y., KOENIG, S., and FURCY, D., "Speeding up the calculation of heuristics for heuristic search-based planning," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 484–491, 2002.

[117] LU, X.-M., "An iterative solution for the 4-peg Towers of Hanoi," *The Computer Journal*, vol. 32, no. 2, pp. 187–189, 1989.

[118] LUCAS), N. C. . E., "La Tour d'Hanoi, jeu de calcul," *Science et Nature*, vol. 1, no. 8, pp. 127–128, 1884.

[119] MATSUBARA, S. and ISHIDA, T., "Real-time planning by interleaving real-time search with subgoaling," in *Proceedings of the International Conference on Artificial Intelligence Planning Systems*, pp. 122–127, 1994.

[120] MCNAUGHTON, M., LU, P., SCHAEFFER, J., and SZAFRON, D., "Memory-efficient A* heuristics for multiple sequence alignment," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 737–743, 2002.

[121] MÉRÕ, L., "A heuristic search algorithm with modifiable estimate," *Artificial Intelligence*, vol. 23, no. 1, pp. 13–27, 1984.

[122] MESEGUER, P., "Interleaved depth-first search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1382–1387, 1997.

[123] MESEGUER, P. and WALSH, T., "Interleaved and discrepancy based search," in *Proceedings of the European Conference on Artificial Intelligence*, pp. 239–243, 1998.

[124] MIURA, T. and ISHIDA, T., "Stochastic node caching for memory-bounded search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 450–456, 1998.

[125] NEEDLEMAN, S. and WUNSCH, C., "A general method applicable to the search for similarities in the amino acid sequence of two proteins.," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1970.

[126] NEWELL, A., "Harpy, production systems, and human cognition," in *Perception and Production of Fluent Speech* (COLE, R. A., ed.), pp. 289–380, Hillsdale, N.J.: Lawrence Erlbaum, 1980.

[127] NEWMAN-WOLFE, R., "Observations on multi-peg Towers of Hanoi," Tech. Rep. TR 187, Dept. of Computer Science, University of Rochester, NY, July 1986.

[128] NILSSON, N., *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.

[129] NORVIG, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP*. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers, 1992.

[130] PAPADIMITRIOU, C. and STEIGLITZ, K., *Combinatorial Optimization: Algorithms and Complexity*. Mineola, New York: Dover Publications, 1998.

[131] PEARL, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1985.

[132] POHL, I., "First results on the effect of error in heuristic search," in *Machine Intelligence* (MELTZER, B. and MICHIE, D., eds.), vol. 5, pp. 219–236, American Elsevier, New York, 1970.

[133] PRIEDITIS, A., "Machine discovery of effective admissible heuristics," *Machine Learning*, vol. 12, no. 1–3, pp. 117–141, 1993.

[134] PROVOST, F., "Iterative weakening: Optimal and near-optimal policies for the selection of search bias," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 749–755, 1993.

[135] RAO, V., KUMAR, V., and KORF, R., "Depth-first versus best-first search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 434–440, 1991.

[136] RATNER, D. and POHL, I., "Joint and LPA*: Combination of approximation and search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 173–177, 1986.

[137] RATNER, D. and WARMUTH, M., "Finding a shortest solution for the $n \times n$ extension of the 15-*Puzzle* is intractable," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 168–172, 1986.

[138] REINEFELD, A., "Complete solution of the eight-puzzle and the benefit of node ordering in IDA*," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 248–253, 1993.

[139] RIBEIRO, C. and HANSEN, P., eds., *Essays and Surveys in Metaheuristics*. Kluwer Academic Publishers, 2001.

[140] RICH, E. and KNIGHT, K., *Artificial Intelligence*. New York: McGraw-Hill, Inc., second ed., 1991.

[141] ROHL, J. and GEDEON, T., "The Reve's puzzle.," *The Computer Journal*, vol. 29, no. 2, pp. 187–188, 1986.

[142] ROMEIN, J., PLAAT, A., BAL, H., and SCHAEFFER, J., "Transposition table driven work scheduling in distributed search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 725–731, 1999.

[143] RUSSELL, S., "Efficient memory-bounded search methods," in *Proceedings of the European Conference on Artificial Intelligence*, pp. 1–5, 1992.

[144] RUSSELL, S. and NORVIG, P., *Artificial Intelligence – A Modern Approach.* Prentice Hall, first ed., 1995.

[145] SARKAR, U., "On the design of a constructive algorithm to solve the multi-peg towers of Hanoi problem," *Theoretical Computer Science*, vol. 237, no. 1–2, pp. 407–421, 2000.

[146] SCHAAL, S. and ATKESON, C., "Robot juggling: An implementation of memory-based learning," *Control Systems Magazine*, vol. 14, 1994.

[147] SCHOFIELD, P., "Complete solution of the 'eight-puzzle'," in *Machine Intelligence 1* (COLLINS, N. and MICHIE, D., eds.), pp. 125–133, Edinburgh: Oliver and Boyd, 1967.

[148] SELMAN, B., LEVESQUE, H., and MITCHELL, D., "A new method for solving hard satisfiability problems," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 440–446, 1992.

[149] SHANG, Y., FROMHERZ, M., ZHANG, Y., and CRAWFORD, L., "Constraint-based routing for ad-hoc networks," in *Proceedings of the International Conference on Information Technology: Research and Education (ITRE)*, pp. 306–310, IEEE, 2003.

[150] SHELL, P., QUIROGA, G., HERNANDEZ-RUBIO, J., ENCINAS, E., GARCIA, J., and BERBIELA, J., "CRESUS: An integrated expert system for cash management," in *Proceedings of the IAAI-92 Conference on Innovative Applications of Artificial Intelligence* (SCOTT, A. and KLAHR, P., eds.), pp. 151–170, 1992.

[151] SHELL, P., RUBIO, J., and BARRO, G., "Improving search through diversity," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1323–1328, 1994.

[152] SHIMBO, M. and ISHIDA, T., "Towards Real-Time search with inadmissible heuristics," in *Proceedings of the European Conference on Artificial Intelligence*, pp. 609–613, 2000.

[153] SHIMBO, M. and ISHIDA, T., "Controlling the learning process of real-time heuristic search," *Artificial Intelligence*, vol. 146, no. 1, pp. 1–41, 2003.

[154] SHUE, L.-Y., LI, S.-T., and ZAMANI, R., "An intelligent heuristic algorithm for project scheduling problems," in *Proceedings of the Thirty-Second Annual Meeting of the Decision Sciences Institute*, 2001.

[155] SHUE, L.-Y. and ZAMANI, R., "An admissible heuristic search algorithm," in *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93)*, vol. 689 of *LNAI*, pp. 69–75, Springer Verlag, 1993.

[156] SHUE, L.-Y. and ZAMANI, R., "A heuristic search algorithm with learning capability," in *ACME Transactions (Association for Chinese Management Educators)*, pp. 233–236, 1993.

[157] SIMON, H., "The functional equivalence of problem solving skills," *Cognitive Psychology*, vol. 7, pp. 268–288, 1975.

[158] SPOUGE, J. L., "Speeding up dynamic programming algorithms for finding optimal lattice paths.," *SIAM Journal of Applied Math*, vol. 49, pp. 1552–1566, Oct. 1989.

[159] SRIVASTAVA, B., NGUYEN, X., KAMBHAMPATI, S., DO, M., NAMBIAR, U., NIE, Z., NIGENDA, R., and ZIMMERMAN, T., "ALTALT Combining graphplan and heuristic state search," *AI Magazine*, vol. 22, no. 3, pp. 88–90, 2001.

[160] STEWART, B., "Solution to advanced problem 3918," *American Mathematical Monthly*, vol. 48, pp. 217–219, 1941.

[161] SUTTON, R. and BARTO, A., *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[162] TAYLOR, L. and KORF, R., "Pruning duplicate nodes in depth-first search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 756–761, 1993.

[163] THOMPSON, J., HIGGINS, D., and GIBSON, T., "CLUSTALW: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice.," *Nucleic Acids Research*, vol. 22, pp. 4673–4680, 1994.

[164] THORPE, P. E., "A hybrid learning real-time search algorithm," Master's thesis, University of California Los Angeles, 1994.

[165] THRUN, S., "The role of exploration in learning control with neural networks," in *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches* (WHITE, D. and SOFGE, D., eds.), pp. 527–559, Van Nostrand Reinhold, 1992.

[166] VOSS, S., MARTELLO, S., OSMAN, I., and ROUCAIROL, C., eds., *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic, 1999.

[167] WAH, B. and CHU, L.-C., "TCGD: A time-constrained approximate guided depth-first search algorithm," in *Proceedings of the International Computer Symposium*, (Tsing Hua University, Hsinchu, Taiwan, R.O.C.), pp. 507–516, Dec. 1990.

[168] WALSH, T., "Iteration strikes back—At the cyclic Towers of Hanoi," *Information Processing Letters*, vol. 16, no. 2, pp. 91–93, 1983.

[169] WALSH, T., "Depth-bounded discrepancy search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1388–1393, 1997.

[170] WINSTON, P., *Artificial Intelligence*. Addison-Wesley, Reading, MA, third ed., 1992.

[171] YOKOO, M. and KITAMURA, Y., "Multiagent real-time A* with selection: Introducing competition in cooperative search," in *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, pp. 409–416, 1996.

[172] YOSHIZUMI, T., MIURA, T., and ISHIDA, T., "A* with partial expansion for large branching factor problems," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 923–929, 2000.

[173] ZHANG, W., "Complete anytime beam search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 425–430, 1998.

[174] ZHANG, W., *State-Space Search: Algorithms, Complexity, Extensions, and Aplications*. Springer-Verlag, New York, 1999.

[175] ZHANG, W., "Depth-first branch-and-bound versus local search: A case study," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 930–936, 2000.

[176] ZHANG, W., "Iterative state-space reduction for flexible computation," *Artificial Intelligence*, vol. 126, no. 1–2, pp. 109–138, 2001.

[177] ZHOU, R. and HANSEN, E., "Memory-bounded A* graph search," in *Fifteenth International FLAIRS Conference (FLAIRS-02)*, 2002.

[178] ZHOU, R. and HANSEN, E., "Multiple sequence alignment using Anytime A*," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 975–976, 2002. Student abstract.

[179] ZHOU, R. and HANSEN, E., "Sparse-memory graph search," in *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1259–1266, 2003.

[180] ZHOU, R. and HANSEN, E., "Breadth-first heuristic search," in *Proceedings of the International Conference on Automated Planning and Scheduling*, pp. 92–100, 2004.

[181] ZHOU, R. and HANSEN, E., "Space-efficient memory-based heuristics," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 677–682, 2004.

[182] ZHOU, R. and HANSEN, E., "Structured duplicate detection in external-memory graph search," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 683–688, 2004.

[183] ZILBERSTEIN, S., "Using anytime algorithms in intelligent systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.

[184] ZILBERSTEIN, S. and RUSSELL, S., "Optimal composition of real-time systems," *Artificial Intelligence*, vol. 82, no. 1–2, pp. 181–213, 1996.