A*+IDA*: A Simple Hybrid Search Algorithm

Zhaoxing Bu and **Richard E. Korf**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{zbu, korf}@cs.ucla.edu

Abstract

We present a simple combination of A* and IDA*, which we call A*+IDA*. It runs A* until memory is almost exhausted, then runs IDA* below each frontier node without duplicate checking. It is widely believed that this algorithm is called MREC, but MREC is just IDA* with a transposition table. A*+IDA* is the first algorithm to run significantly faster than IDA* on the 24-Puzzle, by a factor of almost 5. A complex algorithm called dual search was reported to significantly outperform IDA* on the 24-Puzzle, but the original version does not. We made improvements to dual search and our version combined with A*+IDA* outperforms IDA* by a factor of 6.7 on the 24-Puzzle. Our disk-based A*+IDA* shows further improvement on several hard 24-Puzzle instances. We also found optimal solutions to a subset of random 27 and 29-Puzzle problems. A*+IDA* does not outperform IDA* on Rubik's Cube, for reasons we explain.

1 Introduction and Overview

A* ([Hart *et al.*, 1968]) stores all the nodes it generates in the Open or Closed lists, and hence is severely space-limited in practice. Iterative-Deepening-A* (IDA*) ([Korf, 1985]) requires space that is only linear in the maximum search depth, and is often faster than A*, because it incurs less overhead per node. The main overhead of IDA* is not the initial iterations, but rather re-expansions during the same iteration of the same states via different paths, called duplicate nodes, in graphs with cycles. This is because IDA* is a depth-first search, and doesn't store most previously generated nodes.

Almost as soon as IDA* was introduced, researchers realized that since it uses so little memory, there should be a way to use more memory to speed it up by detecting some duplicate nodes and not re-expanding them. A large number of different algorithms were proposed over the years, including MREC ([Sen and Bagchi, 1989]), MA* ([Chakrabarti *et al.*, 1989]), SMA* ([Russell, 1992; Kaindl and Khorsand, 1994]), IDA* with a transposition table ([Reinefeld and Marsland, 1994]), BIDA* ([Manzini, 1995]), DBIDA* ([Eckerle and Schuierer, 1995]), and breath-first heuristic search ([Zhou and Hansen, 2006]). Most of these algorithms demonstrated a reduction in node generations compared to IDA*, but either reported small or no improvments in running time, or failed to report running times at all. None of these algorithms was shown to be faster than IDA* on the 24-Puzzle.

We propose a very simple combination of A* and IDA*. We run A* until memory is almost exhausted, then execute IDA* starting from the frontier nodes, which are the nodes on Open at the end of the A* phase. We define an iteration of A*+IDA*'s IDA* phase as a series of calls of IDA* on the frontier nodes with the same cost bound. This corresponds to one single iteration of pure IDA* called on the root node. In each iteration of A*+IDA*, we execute IDA* from each frontier node whose f-cost does not exceed the bound, and store with each such node an updated h-value based on the search tree generated below it. These are pure IDA* iterations, with no duplicate-node checking. For each iteration, the frontier nodes of lowest f-cost are searched in increasing order of their stored h-values. This has two main advantages over IDA*. The first is the elimination of duplicate nodes during the A* phase, reducing the number of frontier nodes for the IDA* phase, and the second is the ordering of the frontier nodes during the IDA* phase, which results in finding a solution very early in the last iteration. Overall, A*+IDA* is almost five times faster than an efficient implementation of IDA* on the 24-Puzzle with a 6-6-6-6 Pattern Database.

The only algorithm in the literature that claims to be faster than IDA* on the 24-Puzzle is DIDA* ([Felner *et al.*, 2005; Zahavi *et al.*, 2006; Felner *et al.*, 2010]), based on a complex algorithm called dual search that is only applicable to permutation problems. Based on code provided to us by the authors, we discovered that the version of IDA* they compared to is not the most efficient version, and that their DIDA* code contained a performance bug. We further optimized dual search, and combined it with A*+IDA*, achieving a factor of 6.7 speedup over an efficient version of IDA* on the 24-Puzzle.

We implemented a disk-based version of the A* phase, and showed that this version is faster only on extremely hard 24-Puzzle instances. We also experimented with Rubik's Cube, and found that A*+IDA* is not faster than IDA*, because there are fewer duplicate nodes, and the node ordering is less effective on the final iteration. We optimally solved a subset of random instances of the 4x7 and 5x6 Puzzles. The results show that we will need at least months to solve hard problems optimally. We also show how to use A*+IDA* to quickly find a (potentially suboptimal) solution for such hard problems.

2 Related Work

How did this simple algorithm escape notice for so long? The answer seems to be that most researchers believe it was discovered in 1989 as the MREC algorithm ([Sen and Bagchi, 1989]). For example, Korf [1993][1] described MREC as running A* until memory is full, then running iterations of IDA* below the frontier nodes. Several other researchers repeated this erroneous characterization of MREC ([Edelkamp and Schrödl, 2000; Felner, 2005; Stern *et al.*, 2010; Akagi *et al.*, 2010; Schütt *et al.*, 2013]).

In fact, MREC is IDA*, except that it stores in memory the first nodes generated, until memory is almost full. As each node is generated both before and after this point, it is checked against the stored nodes, and updated if a lower-cost path to the node is found. This algorithm was rediscovered, and referred to as IDA* with a transposition table (IDA*+TT) in [Reinefeld and Marsland, 1994]. Transposition tables ([Slate and Atkin, 1983]) were first used in two-player games to cache search states and reduce duplicate nodes. [Akagi *et al.*, 2010] also implemented IDA*+TT, but did not apply it to the sliding-tile puzzles.

There are four main differences between A*+IDA* and MREC (IDA*+TT). First, A*+IDA* stores the first nodes generated by A*, while MREC stores the first nodes generated by IDA*. These sets can differ significantly among the nodes of largest f-cost that are stored. Second, A*+IDA* starts each iteration of IDA* from the Open nodes of the A* search, while MREC starts each iteration from the start state. Third, during the IDA* phase, A*+IDA* orders the frontier nodes of equal f-cost in increasing order of h-value. Finally, A*+IDA*'s IDA* phase does not check for duplicate nodes, while MREC always does duplicate checking. This saves significant overhead since accessing a large hash table requires an expensive access to main memory. Transposition tables can also use various replacement strategies to dynamically modify the nodes stored, but this adds additional overhead.

The MREC paper [Sen and Bagchi, 1989] shows no speedup on the 15-Puzzle. [Reinefeld and Marsland, 1994] reported a node generation reduction of 54%, and a speedup of 37% compared to IDA* on the 15-Puzzle. As we will see below, however, such speedups can be very sensitive to the efficiency of the implementation of IDA* they are compared to. We use Korf's implementation of IDA* on the sliding-tile puzzles, which has been widely disseminated, and is generally considered quite efficient.

The first appearance of a version of A*+IDA* in the literature is in [Korf, 1993], which he mistakenly described as MREC. He also claims that MREC in [Sen and Bagchi, 1989] did not check for duplicate nodes during the IDA* search on the 15-Puzzle, contrary to the pseudo-code in the paper. He implemented a version of A*+IDA*, but with duplicate node checking during the IDA* phase, and no mention of ordering Open nodes. He reported a 41% reduction in node generations, but a 64% increase in running time compared to IDA*, presumably due to checking for duplicate nodes during IDA*.

[1]We refer to Korf here in the third person because he is only one of the authors of this paper.

MA* ([Chakrabarti *et al.*, 1989]) and SMA* ([Russell, 1992; Kaindl and Khorsand, 1994]) run A* until memory is full, then contract the worst Open nodes, replacing them by their parents, to free up memory to expand the best Open nodes. The overhead of these algorithms is considerable and they don't outperform IDA* on the sliding-tile puzzles.

Eckerle and Schuierer [1995] proposed Dynamic Balanced IDA* (DBIDA*), a modification of A*+IDA* that dynamically adjusts the frontier nodes by adding new frontier nodes with a large subtree below them and removing existing frontier nodes that have a small subtree below them. DBIDA* is very complex as it utilizes multiple sets and priority queues to adjust the frontier. The authors reported a speedup of less than a factor of two, but this was for finding all optimal solutions in the last iteration, and hence is not directly comparable to finding a single optimal solution.

Bidirectional A*-IDA* (BAI, [Kaindl *et al.*, 1995]) first runs A* from one direction, then runs IDA* with duplicate detection from the other direction. Auer and Kaindl [2004] further presented several other bidirectional search algorithms based on BAI and BS* ([Kwa, 1989]) and showed that their algorithm is 7 times faster than IDA*. Manzini [1995] presented BIDA* and showed that it is almost 8 times faster than IDA*. BIDA* first builds a perimeter of nodes all at the same distance d from the goal state, then runs IDA* from the start state. For each node s generated, BIDA* computes the heuristic to the perimeter nodes p, and expands those for which $g(s) + h(s, p) + d$ does not exceed the current search bound. These works used Manhattan Distance (MD) heuristic and tested on the 15-Puzzle. However, these algorithms do not beat IDA* on the 24-Puzzle, for these reasons: 1) IDA* with duplicate detection is two to three times slower than IDA* without duplicate detection on our machine. 2) These algorithms use the MD heuristic so the heuristic between any two states can be easily calculated. However, this is not easy with Pattern Databases. 3) The search space of the 24-Puzzle is $7.4 * 10^{11}$ times larger than that of the 15-Puzzle, making any A* or BS* variations unable to solve random problems. Furthermore, the chance that a node generated in BAI's IDA* phase was already generated in its A* phase is very small. 4) As shown in [Barker and Korf, 2015], bidirectional algorithms with the 6-6-6-6 PDB on the 24-Puzzle would generate more nodes than unidirectional search algorithms.

The above work was done before Pattern Database heuristics (PDBs, [Culberson and Schaeffer, 1998]) enabled optimal solutions to the 24-Puzzle [Korf and Felner, 2002]. While Korf and Felner were aware of these previous works, they chose standard IDA* as their search algorithm.

Dual IDA* (DIDA*) ([Felner *et al.*, 2005; Zahavi *et al.*, 2006; Felner *et al.*, 2010]) was believed to be the state-of-the-art solver for the 24-Puzzle and reduced the nodes generated by IDA* by a factor of 9.3. However, there are three considerations here: 1) Dual search is very complicated and hard to implement. 2) For each state generated in DIDA*, there is a corresponding dual state generated, so the total number of states generated should be doubled. 3) Korf's 24-Puzzle solver ([Korf and Felner, 2002]) has two PDB lookups for each new node, while the version of IDA* used for comparison does eight PDB lookups for each node. Compared to

Korf's more efficient IDA* code and Zahavi *et al.*'s code, DIDA* is only 8.5% faster than IDA*.

Lookahead-A* (AL*) ([Stern *et al.*, 2010; Bu *et al.*, 2014]) is another A* variation which performs a DFS with bound $f(parent) + k$ for each node generated to increase its heuristic value and hence reduce the memory requirement. We found that on our machine, $k = 6$ is the smallest k that enables us to solve all of Korf' 50 test cases ([Korf and Felner, 2002]). When $k = 6$, AL* is also faster than IDA* on the 24-Puzzle, but slower than A*+IDA*. However, to use AL* in practice, a lot of time would be needed to find the correct k, as the k-value is not known in advance and a small k will lead AL* to run out of memory before finding a solution.

External IDA* ([Edelkamp and Schroedl, 2011; Edelkamp, 2016]) first runs disk-based A* then runs IDA* on the frontier nodes, which appears similar to our disk-based A*+IDA*, described later. They did not indicate whether or not duplicate detection is performed during the IDA* phase, nor whether any node ordering of the Open nodes is done. They only reported data for two instances of the 24-Puzzle, did not compare it to IDA*, and did not report any running times.

Forward Perimeter Search (FPS) ([Schütt *et al.*, 2013]) builds a perimeter from the start state, then runs Breadth-first IDA* (BF-IDA*) ([Zhou and Hansen, 2006]) from the perimeter to find a solution. A perimeter is a set of nodes that have the same or similar distance from the start state. During search, this perimeter is also dynamically adjusted. There are two issues here: 1) The experiment was performed on a cluster with 32 computers and no running time for FPS on a single computer was provided. 2) Schütt *et al.* [2013] mistakenly compared the number of nodes expanded in FPS with the number of nodes generated in [Korf and Felner, 2002].

3 A*+IDA*

We use a hash table to store nodes, and A*'s Open list is implemented as in PSVN ([Hernádvölgyi and Holte, 1999; Burch *et al.*, 2014]), which uses a three-dimensional vector (C++'s STL vector). The first dimension corresponds to the f-value, the second dimension corresponds to the g-value, and the third dimension stores indices to all nodes that have the same f and g-values. Adding or removing a node from Open takes $O(1)$ time. To the best of our knowledge, this is the most efficient implementation of A* for the 24-Puzzle.

After A* runs out of memory, we run a series of iterations of IDA* on frontier nodes without duplicate detection. The first bound is set to be the smallest f-value among all frontier nodes. In each iteration of A*+IDA*, we perform an iteration of IDA* on each frontier node (in increasing order of stored h-values) whose f-value equals the bound. After each IDA* iteration on a frontier node s, we increase the stored $h(s)$-value to the difference between the minimum f-value of all the nodes generated but not expanded below s and $g(s)$.

4 Experimental Results and Analysis

All experiments were performed on a 3.33GHz Intel Xeon X5680 CPU with 96 GB of RAM. We used the same heuristic function (6-6-6-6 disjoint PDBs with reflection) and the same 50 test cases as in [Korf and Felner, 2002]. We ran

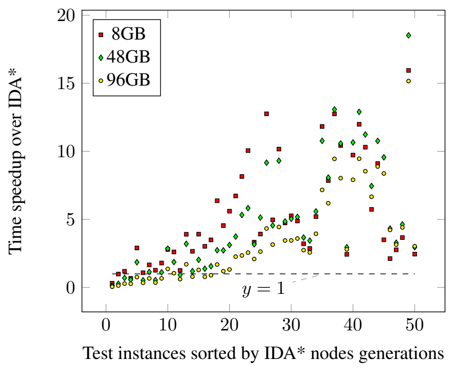

Figure 1: A*+IDA*'s speedup over IDA*.

Algorithm	Stored nodes	Total nodes	Last iteration	Time(s)
IDA*	0	18,044,623,983,548	9,861,089,635,700	572,845
Cached-IDA*	33,149,796,158	10,416,963,057,231	5,703,252,119,005	446,293
A*+IDA*(8GB)	4,565,459,317	4,552,871,977,112	454,939,266,782	150,387
A*+IDA*(48GB)	30,491,473,996	3,143,297,938,874	24,670,542,049	116,963
A*+IDA*(96GB)	56,833,924,109	2,839,875,132,068	24,133,797,786	129,871
ratio		5.74	400	4.9

Table 1: Cumulative data for A*+IDA*.

A*+IDA* with 8GB, 48GB, and 96GB memory, and the number of nodes stored were 96,855,259, 774,842,075, and 1,549,684,150, respectively.

We also implemented the original MREC/IDA*+TT ([Sen and Bagchi, 1989; Reinefeld and Marsland, 1994]). However, because of the high cost of duplicate detection for each node generated during search, it is two to three times slower than IDA*, so we omit the results here.

For comparison purposes, we also include an algorithm we call Cached-IDA*, which uses a transposition table (TT) to store the first nodes generated by IDA*. Before the TT is full, Cached-IDA* behaves the same as MREC/IDA*+TT. After TT is full, Cached-IDA* only does duplicate checking for children if the parent is stored in TT. As a result, when Cached-IDA* generates a node c that is not in TT and TT is full, Cached-IDA* turns off duplicate checking for the search tree below c. Cached-IDA* generates more nodes than the original MREC/IDA*+TT, but is much faster as it avoids duplicate detection for most nodes. Our Cached-IDA* stored the same number of nodes as A*+IDA* with 48GB memory.

We present the speedups of A*+IDA* over IDA* in run time on all 50 test cases in Figure 1. We sort the 50 test cases according to the number of nodes generated by IDA*, so the left-most test case is the easiest and the right-most test case is the hardest. The squares, diamonds, and circles correspond to A*+IDA* with 8GB, 48GB, and 96GB memory respectively. The scattered nature of this plot is due to variation in when the first optimal solution is found on the last iteration.

We present the cumulative data of the 50 test cases in Table 1, where the first column is the algorithm, the second column is the total number of nodes stored, the third column is the total number of nodes generated, the fourth column is the nodes generated in the last iteration, and the last column is total time in seconds for all 50 problems. The first five rows correspond to different algorithms and the last row is the nodes or time ratio of IDA* over A*+IDA* with 48GB memory, which had the best performance in time.

Over all 50 problems, A*+IDA*'s speedup over IDA* is 3.8 (8GB), 4.9 (48GB), and 4.4 (96GB) respectively. A* solved 5, 15, and 19 problems respectively. A*+IDA* is slower than IDA* on the easy problems due to A*'s expensive memory and duplicate detection operations. On the medium and hard problems, A*+IDA* is consistently faster than IDA*. A*+IDA* (8GB) is also faster than the 48GB/96GB version on easy problems as its A* phase is much shorter than that of the 48GB/96GB version. Increasing the memory from 8GB to 48GB reduced the total number of nodes generated by 31.0%, while increasing the memory from 48GB to 96GB only reduced the nodes by an additional 9.7%.

In Table 1, the overall nodes ratio for A*+IDA* (48GB) is 5.74, which is higher than the speedup ratio of 4.9. There are two reasons for this. First, A* is about 7 times slower than IDA* in terms of nodes generated per second. However, this is a minor reason as the nodes generated in A* are less than 1% of the total generated nodes in A*+IDA*. The major reason is that the IDA* phase in A*+IDA* is always slower (in terms of nodes generated per second) than pure IDA* due to the overhead for each separate IDA* call. This overhead mainly comes from fetching the start node from a large hash table, which usually causes cache misses. This is also the reason why A*+IDA* (96GB) is slower than A*+IDA* (48GB). The former generated fewer nodes, but as the frontier nodes doubled, this overhead for all IDA* runs also doubled.

The speedup comes from three factors: fewer IDA* iterations, fewer duplicates, and early goal termination. A*+IDA* typically performs only two to five iterations in its IDA* phase, while pure IDA* typically performs around 10 iterations. However, this is a minor reason as the last two iterations dominate the search. A*+IDA* (48GB) reduced the nodes generated in all but the last iteration by 2.62 times over IDA* and 1.51 times over Cached-IDA*. By starting from the frontier nodes instead of the start state, we do not need to re-generate the nodes above the frontier nodes and the search tree generated by A*+IDA*'s IDA* phase is much smaller than that of IDA*. Our data shows that on average, each separate IDA* call in A*+IDA* (48GB) only generated 190 nodes. This number is dominated by the hard test cases and for most test cases, this number is below 50. The better duplicate pruning effect of A*+IDA* over Cached-IDA* also suggests that expanding and storing nodes in a best-first order is better than depth-first order, as the latter can be heavily biased to the left part of the search tree.

The last iteration of IDA* is usually the most time-consuming iteration. In Table 1, 54.6% of IDA* nodes were generated in the last iteration. A*+IDA* reduced the nodes generated in the last iteration by 400 times, making it less than 1% of the total nodes generated. We present the detailed last iteration comparison in Figure 2, where the y-axis is the number of nodes generated in IDA*'s last iteration over A*+IDA*'s. We see that the 8GB version usually leads to more nodes generated, but there is not much difference between the 48GB and 96GB version. For the 48GB/96GB version, A*+IDA* reduced the number of nodes in the last iteration by more than three orders of magnitude on most test

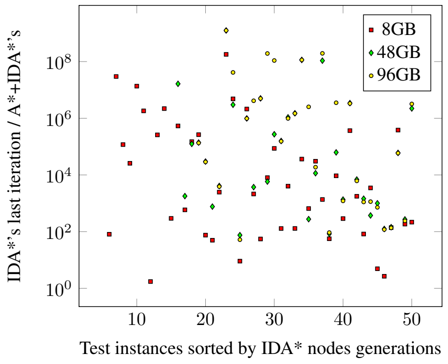

Figure 2: A*+IDA*'s last iteration vs. IDA*'s.

Memory	A*+IDA*	Cached-IDA*+node ordering
8GB	453,763,507,684	557,116,891,535
48GB	24,636,552,525	320,412,253,835
96GB	24,133,797,786	40,276,211,512

Table 2: Nodes generated in the last iteration in A*+IDA* and Cached-IDA* with node ordering.

cases. This result shows that by expanding frontier nodes of equal f-values in increasing order of h-values, the goal state was found very early in the final iteration. It also suggests that finding the optimal solution is much easier than verifying optimality. Expanding nodes according to the f and h-values to significantly reduce the number of nodes generated in the last iteration was first done in [Powley and Korf, 1989]. As noted there, the benefit of node ordering may be mitigated by the extra cost for node ordering. This may be the reason why node ordering was not commonly used in IDA* in the past. However, we show that effective node ordering can be easily achieved by utilizing A*+IDA*'s frontier list.

We further applied the above node ordering on Cached-IDA*'s last iteration. We scanned the transposition table, selected the nodes that do not have all their children stored in the transposition table, pruned the nodes whose f-values exceed the bound, then sorted the nodes in increasing order of stored h-values and expanded them. Cached-IDA* with node ordering is then similar to our A*+IDA* as it now starts from the frontier nodes instead of the start state. We present the results for the 31 test cases that A* (96GB) did not solve in Table 2. We only list the number of nodes generated in the last iteration. IDA* generated 9,789,829,523,428 nodes in its last iteration on these 31 test cases. We see that A*+IDA* consistently generated fewer nodes than Cached-IDA* with node ordering. The most significant difference is under the 48GB setting where Cached-IDA* with node ordering generated 13 times more nodes than A*+IDA*. This is largely due to one test case where Cached IDA* with node ordering generated 280,591,231,353 nodes, proving that even with node ordering, Cached-IDA* can sometimes find the goal state very late in the last iteration.

5 Combining Dual Search with A*+IDA*

We further applied dual search ([Zahavi *et al.*, 2008]) to the IDA* phase of our A*+IDA*. Dual search (DS) is very com-

Algorithm	Total nodes	Time (s)	Speedup
IDA*	18,044,623,983,548	572,845	1
Original DIDA*	3,876,071,228,494	523,948	1.1
Our DIDA*	3,684,978,748,642	259,506	2.2
Original A*+IDA*	3,143,297,938,874	116,963	4.9
A*+IDA* w/ DS	858,888,100,914	85,405	6.7

Table 3: A*+IDA* combined with dual search.

plicated and we do not have the space to explain it here. This section does not require readers to understand DS.

Zahavi *et al.* [2008] compared their DIDA* (IDA* with DS) code to an inefficient IDA* implementation which does 8 PDB lookups for each node. However, an efficient IDA* implementation only does two PDB lookups for each node. We compared their DIDA* code with Korf's efficient IDA* code ([Korf and Felner, 2002]) and found that their DIDA* code is only 8.5% faster than Korf's efficient IDA* code.

The original DIDA* code in [Zahavi *et al.*, 2008] performs 8 PDB lookups for a new child state and 8 PDB lookups for the child's dual state, a total of 16 PDB lookups. We optimized the DIDA* code to perform only 4.435 PDB lookups in total for each state and its corresponding dual state. We also discovered an error in the original Dual PDB lookup and Dual search implementation in [Felner *et al.*, 2005; Zahavi *et al.*, 2006], resulting in 32% of the total generated nodes' regular PDB lookups and reflection PDB lookups always returning the same value. We omit the details of our optimizations here due to space limitations.

We present the results in Table 3, where the first column is the algorithm, the second column is the total number of nodes generated, the third column is time, and the last column is the speedup over IDA*. We used the same 50 test cases as before. Zahavi *et al.* [2006] used bidirectional pathmax (BPMX, [Felner *et al.*, 2005]) in their DIDA*. BPMX, which is used with inconsistent heuristics, propagates higher heuristic values from a child node to its parent and siblings, thus reduces node expansions. BPMX reduced by about 2% the number of nodes generated during search. To be consistent with IDA*, we did not use BPMX in Table 3.

From Table 3 we see the original DIDA* is only 1.1 times faster than Korf's efficient IDA* code. After our optimization and bug fix, DIDA* is 2.2 times faster than Korf's efficient IDA* code. After we combined dual search with A*+IDA*, the speedup increases from 4.9 to 6.7. Contrary to the original dual search paper, we treat a state and its dual state as two distinct states. Therefore, the number of nodes for the original DIDA* is roughly two times that in [Zahavi *et al.*, 2008]. The number is only roughly because we did not use BPMX.

6 Disk-based A*+IDA*

We then applied delayed duplicate-detection (DDD, [Korf, 2004]) to A*+IDA*. Our implementation is similar to External IDA* ([Edelkamp and Schroedl, 2011; Edelkamp, 2016]). However, they did not indicate whether or not duplicate detection was performed during the IDA* phase, nor whether any node ordering of the frontier nodes was done. They reported data for only two instances of the 24-Puzzle, did not compare it to IDA*, and did not report any running times.

State	Length
s1: 23 1 12 6 16 2 20 10 21 18 14 13 17 19 22 0 15 24 3 7 4 8 5 9 11	113
s2: 0 19 3 14 17 13 21 4 10 22 12 11 9 15 24 18 7 2 6 8 5 1 23 16 20	108
s3: 3 5 0 15 4 13 22 2 21 23 14 7 16 17 10 11 1 8 12 24 9 20 19 18 6	104

Table 4: Three hard test cases in Table 5.

Algo. & Test case	Total nodes	Time (s)	Speedup
A*+IDA*(48GB), s1	1,343,383,770,705	44,602	1
A*+IDA*(Disk), s1	816,569,465,508	41,085	1.09
A*+IDA*(48GB), s2	5,011,462,601,471	165,972	1
A*+IDA*(Disk), s2	1,620,137,788,282	76,547	2.17
A*+IDA*(48GB), s3	3,039,454,921,549	100,722	1
A*+IDA*(Disk), s3	2,043,687,639,368	88,428	1.14

Table 5: Disk-based A*+IDA* vs. 48GB RAM-based A*+IDA*.

We first ran disk-based A* with DDD up to 10.2 billion expansions, then we removed the duplicates, leaving the unique frontier nodes, then ran IDA* on the frontier nodes. Common disk-based A* assumes consistent heuristics ([Edelkamp and Schroedl, 2011]). However, the 6-6-6-6 PDB for the 24-Puzzle is an inconsistent heuristic and pathmax cannot turn it into a consistent heuristic ([Holte, 2010]). Therefore, re-expanding the same state in A* cannot be avoided. Our results show that disk-based A*+IDA* is only faster than RAM-based A*+IDA* on extremely hard problems. The reason is the same as why the 96GB A*+IDA* version is slower than the 48GB version: 1) more time is spent on the A* phase and 2) with more frontier nodes, the overhead for initiating IDA* on the frontier nodes increases.

We randomly generated 150 test cases and found two (s2 and s3) that are harder than the most difficult test case (s1) we used before. We present these three test cases and their solution length in Table 4 and the results in Table 5, where the first column is the algorithm and test case, the second column is the total number of nodes generated in A*+IDA*, the third column is time, and the last column is the speedup of disk-based version over 48GB RAM-based version. Disk-based A*+IDA* is faster than RAM-based A*+IDA* on all three problems. On s2, the disk-based version reduced the nodes by a factor of three and achieved a speedup of 2.17. On s3, disk-based A*+IDA* reduced the nodes by one third and search time by 12%. The disk-based version performed better on s2 because it reduced the number of nodes generated by 2,479 billion in the last iteration.

7 A*+IDA* on Rubik's Cube

We randomly generated 100 test cases for Rubik's Cube. We used the maximum PDB value of the 8 corner cubies and 9 edge cubies. We present the results in Table 6, where the first column is the algorithm, the second column is the number of nodes stored by A*+IDA*'s A* phase, the third column is the total number of nodes generated, the fourth column is the nodes generated in the last iteration, and the last column is time. Our A*+IDA* stored the same number of nodes (774,842,075) as A*+IDA* (48GB) on the 24-Puzzle, and required 32GB for A* and 38GB for the PDBs.

A*+IDA* is 1% slower than IDA* and only reduced the nodes generated in IDA* by 23%. There are two reasons for this: 1) With the move pruning used in [Korf, 1997], there are few duplicate nodes generated in IDA*. Depth-first search only generates 2.1%, 3.1%, 4.2%, 5.4%, 6.6%, and 8.0% duplicate nodes at depth 7, 8, 9, 10, 11, and 12 respectively (`http://www.cube20.org` and [Korf, 1997]). Most of the frontier nodes stored by A*+IDA* are at depth 7 to 12. Therefore, there will not be much gain in duplicate detection as there do not exist many duplicates. 2) A*+IDA* generated more nodes than IDA* in the last iteration on 19 out of 82 test cases that A* did not solve, with the highest ratio as 39. The reason is that the frontier nodes that are on the optimal paths sometimes have a perfect heuristic value and hence a small g-value, causing them to be expanded very late in the last iteration. For IDA*, the expansion order is based on the order of operators. The lesson here is that it is hard to get a good node ordering for Rubik's Cube.

As we mentioned before, there is an overhead with each IDA* run, so A*+IDA*'s IDA* phase generated fewer nodes per second than pure IDA*. The time spent in the A*-phase, lower speed in the IDA*-phase, rare duplicate nodes, and less powerful early termination in the last iteration make A*+IDA* not faster than IDA* on Rubik's Cube.

8 Parallel A*+IDA* on 27 and 29-Puzzle

We implemented a multi-threaded A*+IDA* and tested it on the 4x7 (27) and 5x6 (29) sliding-tile puzzles. We first ran a single-threaded A*, then ran multiple IDA*s in parallel with each thread given a different frontier node. Unlike the 24-Puzzle, 27 and 29-Puzzles cannot use reflections for PDB lookups. For the 27-Puzzle, we built two 7-7-7-6 PDBs (and took the maximum) whose sparse mapping ([Felner *et al.*, 2007]) required 26GB in total. For the 29-Puzzle, we built two 7-7-7-7-1 PDBs and two 6-6-6-6-5 PDBs (and took the maximum) whose sparse mapping required 64GB in total.

We randomly generated 10 instances for each puzzle. To date, we have found the optimal solution for 5 and 4 instances for the 27 and the 29-Puzzle respectively. To our best knowledge, these are the first random instances of a sliding-tile puzzle larger than the 24-Puzzle to be solved optimally. For the remaining instances, we have found solutions but not yet verified their optimality. The solutions we found range from 108 to 149 for the 27-Puzzle and 117 to 152 for the 29-Puzzle. The average (not guaranteed to be optimal) lengths for the solutions we found are 123.6 and 129.4 respectively. The average heuristic values for one million random states are 81.6, 100.3, and 107.6 for the 24, 27, and 29-Puzzles respectively.

For the sliding-tile puzzles, A*+IDA* can be easily converted to an approximation algorithm that enables us to quickly find a solution. For example, we can only run IDA* on the top 20% of the frontier nodes (sorted by h-values) in each iteration. If no solution is found, we go to the next iteration. If a solution is found, we decrease the bound to verify the optimality or find a better solution. We used this approximation algorithm to quickly find solutions for the 10 test cases for the 27 and 29-Puzzles. Given a large bound, most of nodes generated in the IDA* phase have high h-values. For example, the last 5% of the frontier nodes would generate more than two orders of magnitude more nodes than the first 5% of the frontier nodes. Therefore, even expanding the first half of the frontier nodes would generate only a small portion of the total nodes generated. The optimal solutions are always generated very early in the last iteration, so the first solution found by this algorithm is usually the optimal one. However, our analysis suggests that it would take months to years to verify the optimality of the hardest 27 and 29-Puzzle test cases we generated, so we cannot claim to have optimally solved these instances.

Algo.	A* stored	Total nodes	Last iteration	Time (s)
IDA*	0	918,896,332,771	576,086,877,811	323,083
A*+IDA*	69,909,543,136	710,571,584,688	381,865,691,696	325,950

Table 6: A*+IDA* vs. IDA* on Rubik's Cube.

9 Discussion and Conclusions

We present a simple combination of A* and IDA*, called A*+IDA*, to make use of additional memory. It is widely believed that this algorithm was discovered in 1989 and called MREC, but in fact MREC is just IDA* with a transposition table. While there are several combinations of A* and IDA* in the literature, to the best of our knowledge, none of them forego duplicate detection during IDA* and order the frontier nodes of equal f-cost by increasing h-values at the same time. A*+IDA* is almost 5 times faster than the most efficient implementation of IDA* on the 24-Puzzle, and the first algorithm to significantly outperform IDA* on this problem. We combined A*+IDA* with our improved dual search and found that it is 6.7 times faster than IDA*. We also show that using a disk-based A* search in A*+IDA* speeds up search on several very hard 24-Puzzle instances. We also found optimal solutions to a subset of random 27 and 29-Puzzle problems and we show how to use A*+IDA* to quickly find a solution on sliding-tile puzzles. A*+IDA* is not faster than IDA* on Rubik's Cube, due to many fewer duplicate nodes and less effective node ordering on the last iteration, but not significantly slower either.

Our combination provides a general way to combine A* with DFS algorithms. We also implemented A*+RBFS and observed similar speedups as A*+IDA* on the 24-Puzzle. In general, if A* solves a problem, there is no overhead in using A*+IDA*. If not, then compared to pure IDA*, the overhead of A*+IDA* is likely to be more than compensated for by the duplicate pruning in its A* phase, and the improved node ordering of the last iteration in its IDA* phase.

Another way to use memory to speedup IDA* is to build larger PDBs. We built 8-8-7-1, 7-7-7-3, 7-7-6-4, and 7-7-5-5 PDBs and found that none of them is better than the standard 6-6-6-6 PDB for the 24-Puzzle. The reason is that they all produce more small heuristic values than the 6-6-6-6 PDB (see [Holte *et al.*, 2004] for a more detailed discussion on the effect of small heuristic values).

Acknowledgements

We thank Robert Holte for providing `PSVN` and Ariel Felner for providing the DIDA* implementation on the 24-Puzzle.

References

[Akagi *et al.*, 2010] Yuima Akagi, Akihiro Kishimoto, and Alex Fukunaga. On transposition tables for single-agent search and planning: Summary of results. In *Third Annual Symposium on Combinatorial Search*, 2010.

[Auer and Kaindl, 2004] Andreas Auer and Hermann Kaindl. A case study of revisiting best-first vs. depth-first search. In *ECAI*, volume 16, page 141, 2004.

[Barker and Korf, 2015] Joseph K Barker and Richard E Korf. Limitations of front-to-end bidirectional heuristic search. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[Bu *et al.*, 2014] Zhaoxing Bu, Roni Stern, Ariel Felner, and Robert Craig Holte. A* with lookahead re-evaluated. In *Seventh Annual Symposium on Combinatorial Search*, 2014.

[Burch *et al.*, 2014] Neil Burch, Robert Holte, and Broderick Arneson. Psvn manual (june 20, 2014). 2014.

[Chakrabarti *et al.*, 1989] Partha Pratim Chakrabarti, Sujoy Ghose, Arup Acharya, and SC De Sarkar. Heuristic search in restricted memory. *Artificial intelligence*, 41(2):197–221, 1989.

[Culberson and Schaeffer, 1998] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Eckerle and Schuierer, 1995] Jürgen Eckerle and Sven Schuierer. Efficient memory-limited graph search. In *Annual Conference on Artificial Intelligence*, pages 101–112. Springer, 1995.

[Edelkamp and Schrödl, 2000] Stefan Edelkamp and Stefan Schrödl. Localizing a*. In *AAAI/IAAI*, pages 885–890, 2000.

[Edelkamp and Schroedl, 2011] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.

[Edelkamp, 2016] Stefan Edelkamp. External-memory state space search. In *Algorithm Engineering*, pages 185–225. Springer, 2016.

[Felner *et al.*, 2005] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual lookups in pattern databases. In *Proceedings of the 19th IJCAI*, pages 103–108, 2005.

[Felner *et al.*, 2007] Ariel Felner, Richard E Korf, Ram Meshulam, and Robert C Holte. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30:213–247, 2007.

[Felner *et al.*, 2010] Ariel Felner, Carsten Moldenhauer, Nathan R Sturtevant, and Jonathan Schaeffer. Single-frontier bidirectional search. In *AAAI*, 2010.

[Felner, 2005] Ariel Felner. Finding optimal solutions to the graph partitioning problem with heuristic search. *Annals of Mathematics and Artificial Intelligence*, 45(3-4):293–322, 2005.

[Hart *et al.*, 1968] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[Hernádvölgyi and Holte, 1999] István T Hernádvölgyi and Robert C Holte. Psvn: A vector representation for production systems. 1999.

[Holte *et al.*, 2004] Robert C Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *ICAPS*, pages 122–131, 2004.

[Holte, 2010] Robert C Holte. Common misconceptions concerning heuristic search. In *Third Annual Symposium on Combinatorial Search*, 2010.

[Kaindl and Khorsand, 1994] Hermann Kaindl and Aliasghar Khorsand. Memory-bounded bidirectional search. In *AAAI*, pages 1359–1364, 1994.

[Kaindl *et al.*, 1995] Hermann Kaindl, Gerhard Kainz, Angelika Leeb, and Harald Smetana. How to use limited memory in heuristic search. In *IJCAI*, pages 236–242, 1995.

[Korf and Felner, 2002] Richard E Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134(1):9–22, 2002.

[Korf, 1985] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[Korf, 1993] Richard E Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.

[Korf, 1997] Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

[Korf, 2004] Richard E Korf. Best-first frontier search with delayed duplicate detection. In *AAAI*, volume 4, pages 650–657, 2004.

[Kwa, 1989] James BH Kwa. Bs*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38(1):95–109, 1989.

[Manzini, 1995] Giovanni Manzini. Bida*: an improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.

[Powley and Korf, 1989] Curt Powley and Richard E Korf. Single-agent parallel window search: A summary of results. In *IJCAI*, pages 36–41, 1989.

[Reinefeld and Marsland, 1994] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[Russell, 1992] Stuart Russell. Efficient memory-bounded search methods. *ECAI-1992, Vienna, Austria*, 1992.

[Schütt *et al.*, 2013] Thorsten Schütt, Robert Döbbelin, and Alexander Reinefeld. Forward perimeter search with controlled use of memory. In *Proceedings of IJCAI*, pages 659–665. AAAI Press, 2013.

[Sen and Bagchi, 1989] Anup K Sen and Amitava Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.

[Slate and Atkin, 1983] David J Slate and Lawrence R Atkin. Chess 4.5—the northwestern university chess program. In *Chess skill in Man and Machine*, pages 82–118. Springer, 1983.

[Stern *et al.*, 2010] Roni Stern, Tamar Kulberis, Ariel Felner, and Robert Holte. Using lookaheads with optimal best-first search. In *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence*, 2010.

[Zahavi *et al.*, 2006] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1076, 2006.

[Zahavi *et al.*, 2008] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4-5):514–540, 2008.

[Zhou and Hansen, 2006] Rong Zhou and Eric A Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4):385–408, 2006.