Linear-Time Disk-Based Implicit Graph Search

RICHARD E. KORF

University of California, Los Angeles

Abstract. Many search algorithms are limited by the amount of memory available. Magnetic disk storage is over two orders of magnitude cheaper than semiconductor memory, and individual disks can hold up to a terabyte. We augment memory with magnetic disks to perform brute-force and heuristic searches that are orders of magnitude larger than any previous such searches. The main difficulty is detecting duplicate nodes, which is normally done with a hash table. Due to long disk latencies, however, randomly accessed hash tables are infeasible on disk, and are replaced by a mechanism we call delayed duplicate detection. In contrast to previous work, we perform delayed duplicate detection without sorting, which runs in time linear in the number of nodes in practice. Using this technique, we performed the first complete breadth-first searches of the 2×7, 3×5, 4×4, and 2×8 sliding-tile Puzzles, verifying the radius of the 4×4 puzzle and determining the radius of the others. We also performed the first complete breadth-first searches of the four-peg Towers of Hanoi problem with up to 22 discs, discovering a surprising anomaly regarding the radii of these problems. In addition, we performed the first heuristic searches of the four-peg Towers of Hanoi problem with up to 31 discs, verifying a conjectured optimal solution length to these problems. We also performed partial breadth-first searches of Rubik's Cube to depth ten in the face-turn metric, and depth eleven in the quarter-turn metric, confirming previous results.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*

General Terms: Algorithms

Additional Key Words and Phrases: External memory, magnetic disk storage, Towers of Hanoi, sliding-tile puzzles, Rubik's Cube, permutation encodings

ACM Reference Format:

Korf, R. E. 2008. Linear-time disk-based implicit graph search. J. ACM 55, 6, Article 26 (December 2008), 40 pages. DOI = 10.1145/1455248.1455250 http://doi.acm.org/10.1145/1455248.1455250

1. *Introduction and Overview*

1.1. INTRODUCTION. Best-first search algorithms, such as breadth-first search (BFS), Dijkstra's single-source shortest path algorithm [Dijkstra 1959], and the A*

This research was supported by the National Science Foundation under grants EIA-0113313 and IIS-0713178.

Author's address: Computer Science Department, University of California, Los Angeles, Los Angeles, CA 90095, e-mail: korf@cs.ucla.edu.

algorithm [Hart et al. 1968], are limited by the amount of memory available to store nodes in their Open and Closed lists. As a result, they typically exhaust memory in minutes on most machines. While more memory allows these algorithms to run longer, it remains the resource bottleneck, and is relatively expensive at about $100 per gigabyte.

In the past few years, however, there have been enormous increases in the capacity of magnetic disks, with little increase in their cost, resulting in dramatic reductions in the cost per byte. Currently one terabyte[1] disks are available at a cost of about $300, or about 30 cents per gigabyte. This is about two and a half orders of magnitude cheaper than semiconductor memory. This suggests the idea of using disk storage instead of memory to store nodes in a best-first search algorithm.

The main problem with this approach is that accessing an individual byte on magnetic disk incurs a latency of up to ten milliseconds. Thus, random-access data structures such as hash tables, which are often used to detect whether a particular state of a problem has already been generated, are completely impractical on disk.

The main contribution of this article is a set of efficient algorithms for performing very large searches that store nodes on magnetic disk, while avoiding the random-access latency of this medium. Instead of checking for duplicate states every time a node is generated, these checks are delayed, batched together, and then performed with purely sequential access to the disk. We refer to this method as *delayed duplicate detection*. In particular, we implement delayed duplicate detection in linear time, without the logarithmic penalty of sorting, and show how to combine it with a recent algorithm called frontier search. We also present new results using these techniques for sliding-tile puzzles, and the four-peg Towers of Hanoi problem, and confirm existing results for Rubik's Cube.

1.2. OVERVIEW. We begin by describing the memory requirements of various search algorithms, including depth-first search, best-first search, and a recent technique called *frontier search*. We then describe previous work on graph search using external memory.

The next section discusses delayed duplicate detection (DDD) in detail. For simplicity, we first describe the algorithm implemented entirely in memory, and then how to combine DDD with frontier search. Then we discuss DDD on disk, including previously known sorting-based implementations, a new more efficient implementation using hash functions, parallel DDD, interleaving node expansion with duplicate detection, best-first DDD, a technique called *breadth-first heuristic search*, and finally issues of reliability and interruptibility.

We also describe an efficient technique for mapping permutations of n elements in lexicographic order to unique integers in the range 0 to $n! - 1$ and vice-versa. These algorithms run in linear time, assuming that n is less than or equal to the word size of the machine, which is typically 64 bits, but use memory that is exponential in n.

We then describe our experimental results, in three different domains: Rubik's Cube, the sliding-tile puzzles, and the four-peg Towers of Hanoi problem. For Rubik's Cube, we performed breadth-first searches to depth 10 in the face-turn

[1] When referring to disk storage, gigabyte and terabyte refer to 10^9 and 10^{12} bytes respectively, while a gigabyte of semiconductor memory is 2^{30} bytes.

metric, and depth 11 in the quarter-turn metric, the largest depths we could reach with three terabytes of storage, confirming the results of previous searches.

For the sliding-time puzzles, we performed the first complete breadth-first searches of the Thirteen Puzzle, the Fourteen Puzzle, and the 4×4 and 2×8 Fifteen Puzzles, determining the number of states at each depth from a given initial state, and the maximum distance of any state from the initial state, which is the problem *radius*. These experiments confirm the problem radius of 80 moves for the 4×4 problem, and compute for the first time the radius of 108 moves for the Thirteen Puzzle, 84 moves for the Fourteen Puzzle, and 140 moves for the 2×8 Fifteen Puzzle.

The four-peg Towers of Hanoi problem is identical to the familiar three peg version, except for an additional peg, which makes the problem much more complex. In particular, there is a conjecture about the length of an optimal solution to this problem which dates to 1941, but it remains unproven. Using heuristic search, we were able to confirm this conjecture for the first time for all problems up to 31 discs.[2] In addition, by performing complete breadth-first searches of this problem for up to 22 discs, we discovered a surprising anomaly that disproves a previous conjecture. We believe that these experiments represent the largest best-first searches ever done, in one case expanding over 25 trillion[3] nodes in three CPU months.

Next we describe work on disk-based search that followed our work, including a technique called *structured duplicate detection*, followed by our conclusions.

Much of this work has previously appeared in conference and workshop papers, including [Korf 2003a, 2003b, 2004; Korf and Shultze 2005; Korf and Flner 2007].

2. *Memory Requirements of Search Algorithms*

Here we discuss the motivation behind this work, which is the memory requirements of search algorithms. We begin with depth-first search algorithms, then consider breadth-first search, and finally a recent technique called frontier search.

Search graphs can be specified either explicitly or implicitly. An explicit graph is described by listing all of its nodes and edges in a data structure large enough to hold the entire graph. An implicit graph is described by an initial node, and a set of operators that generate all the successor nodes of any given node. In this article we are concerned exclusively with very large graphs, which must be specified implicitly. To *generate* a node means to create an explicit data structure that represents the node, while to *expand* a node means to generate all of its children.

2.1. Depth-First Searches. We begin with depth-first search (DFS), an algorithm that has only minimal space requirements. DFS can be implemented with a last-in first-out stack. Nodes are taken from the top of the stack, expanded, and their children are placed on top of the stack. Alternatively, in a recursive implementation, when a node is generated, DFS is called recursively on each of its child nodes. The memory requirement of DFS, either in the explicit node stack or the call stack for the recursive implementation, is only linear in the maximum depth of search, and is generally inconsequential.

[2] To avoid confusion, we use the term "disk" to refer to a magnetic disk, and the term "disc" to refer to a Towers of Hanoi disc.
[3] We use "trillion" to represent 10^{12}, "billion" to represent 10^9, and "million" to represent 10^6.

DFS has several drawbacks however. In a graph with cycles, a pure depth-first search may not terminate. Even in a tree, the first solution found by DFS is not necessarily optimal. Both these deficiencies can be corrected with depth-first iterative-deepening (DFID) [Korf 1985], which performs a series of depth-first searches to successively greater depths, until a goal state is found. At that point, the current node stack or recursive call stack contains the solution path from the initial state to the goal state. DFID has the same linear space complexity as DFS.

Another drawback of DFS is not so easily rectified, however. In a graph with multiple paths to the same state, any depth-first search, including DFID, can generate far more nodes than there are states, due to its inability to detect most duplicate nodes that represent the same state. For example, consider a grid graph, where each node has four neighbors, to its North, South, East and West. An efficient DFS would keep track of the operator used to generate a node from its parent, and not apply its inverse when generating its children, reducing the branching factor from four to three. In a search to a radius r, DFS would generate $O(3^r)$ nodes. The vast majority of these are duplicates, however, since there are only $O(r^2)$ unique states within a radius of r. For example, applying North followed by East generates the same state as applying East followed by North. Thus, DFS can generate exponentially more nodes than there are distinct states in a problem space.

As another example, depth-first search is very inefficient on the Towers of Hanoi problem. In most states, all pegs are occupied by at least one disc. For any such state of the four-peg problem, there are six legal moves: the smallest disc can be moved to any of the other three pegs, the next smallest disc which is on top of a peg can be moved to two other pegs, and the third smallest disc on top of a peg can be moved to one other peg. If we rule out moving the same disc twice in a row, which is wasteful since the same effect can be accomplished with a single move, the branching factor is reduced to about 3.766. Any two moves which have no overlap in their source and destination pegs, such as from peg A to peg B and from peg C to peg D, are commutative and can be applied in either order with the same effect. If we only apply commutative operators in one order, the branching factor of any state with all four pegs occupied is reduced to about 3.258. To find an optimal solution to the 7-disc problem, we would have to search to depth 25, generating 3.258^{25} nodes in the worst case, which is over 6 trillion nodes. This is the largest problem we could solve with depth-first search in practice, in spite of the fact that the whole problem space only contains $4^7 = 16,384$ unique states.

2.2. BREADTH-FIRST SEARCH. The general solution to this problem is to store all the generated nodes, and compare newly generated nodes against the stored nodes to detect duplicates. For example, a breadth-first search maintains a Closed list of nodes that have been expanded, and an Open list of nodes that have been generated but not yet expanded. The Open list is managed as a first-in first-out queue. Nodes are removed from the head of the Open queue, expanded, marked as Closed, and their children are added to the tail of the Open queue. In addition, as each node is generated, it is added to a hash table if it is not already there, and is discarded if it is in the table. The drawback of this algorithm is the memory required to store the nodes. Typically, the node itself is stored in the hash table, the Open list consists of pointers through the hash table, and Closed nodes are indicated by a flag.

A complete breadth-first search using this algorithm eventually stores every problem state in memory. For example, the largest sliding-tile puzzle that could be searched this way with a gigabyte of memory is the 3×4 Eleven Puzzle, with $12!/2 = 239,500,800$ reachable states. The largest four-peg Towers of Hanoi problem that could be searched is the 14-disc problem with $4^{14} = 268,435,456$ states. Each of these problems requires almost four bytes to uniquely represent a state.

2.2.1. *BFS with Disk Queue and Bit Array.* In those cases where a complete or nearly complete breadth-first search is to be performed, and a path from the initial state to a goal state is not required, a more memory-efficient algorithm exists. The Open queue is stored on disk instead of memory. A queue is well-suited to disk because it is read sequentially from the head, and written sequentially to the tail.

The simplest implementation consists of a single file with two pointers, one at the head for reading, and one at the tail for writing. The drawback of this approach is that the elements that have already been read cannot be deleted. Alternatively, we can implement a queue on disk using two files. One file contains the head of the queue and is only read, while the other contains the tail of the queue and is only written. When the head file is exhausted, it is deleted, the tail file becomes the new head file, and a new tail file is created. By buffering reads and writes in memory, relatively large blocks of data can be efficiently read and written at once.

To detect duplicate nodes, we store one bit per state in memory. This requires mapping each state to a unique integer index. For example, in a legal state of the four-peg Towers of Hanoi problem, the discs on any peg must be in sorted order. Thus, we can uniquely represent a state by specifying the peg that each disc is on. With four pegs, this requires two bits per disc. Thus, any legal state of an n-disc problem can be uniquely represented by a bit string of $2n$ bits, and any such bit string represents a legal state. Given a state of the problem, we use the corresponding bit string as the index into our bit array. In Section 5, we will present a similar mapping function for permutation problems, such as the sliding-tile puzzles.

Initially, all bits in the array are set to zero, except for the element corresponding to the initial state. As each state is generated by the breadth-first search, we check its corresponding entry in the bit array. If it is equal to zero, meaning the state hasn't been generated before, we set the bit to one, and add the new state to the Open queue. Conversely, if the corresponding bit is already set to one, meaning the state has already been generated, we discard the state.

Using this method on the 2×7 Thirteen Puzzle would require about five gigabytes of memory for the bit array. The same method applied to the 17-disc four-peg Towers of Hanoi problem would require two gigabytes for the bit array.

2.3. BEST-FIRST SEARCH. Standard breadth-first search is a special case of an algorithm schema called *best-first search*. Other well-known examples of best-first search include Dijkstra's single-source shortest-path algorithm [Dijkstra 1959], and the A* algorithm [Hart et al. 1968]. These algorithms differ in the cost of a node, which determines the "best" node to expand next. For breadth-first search, the cost of a node is its depth. For Dijkstra's algorithm, the cost of a node n is $g(n)$, or the sum of the edge costs from the root to node n. For A*, the cost of a node n is $g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost of reaching a goal from node n. While breadth-first search manages the Open list as a first-in first-out queue, in general best-first searches manage the Open list as a priority queue in order to facilitate efficiently determining the best node to expand next. All of these algorithms suffer the same memory limitation as breadth-first search, since they store all nodes generated in their Open or Closed lists, and will exhaust the available memory in minutes.

2.4. FRONTIER SEARCH. Frontier search reduces the amount of memory required of best-first searches [Korf et al. 2005]. The main idea is to save only the Open list, which represents the frontier of the search, and not the Closed list. In other words, once a node is expanded, it is deleted from memory. The Closed list plays two roles in standard best-first search: detecting duplicate nodes, and reconstructing the solution path. Frontier search accomplishes both of these functions without a Closed list.

2.4.1. *Undirected Graphs.* For simplicity, we first assume that all operators can be applied in either direction. When a state is first generated, we store it in the Open list along with a *used operator bit* to indicate the operator that was used to generate it. For a sliding-tile puzzle, for example, we need four used-operator bits per state, one for each direction that a tile can be moved. We also need four used-operator bits for the four-peg Towers of Hanoi problem, to indicate the locations of the discs that were moved to reach each state. When a node is expanded, we don't apply the inverse of any used operator. This prevents the regeneration of Closed nodes. When another copy of an Open node is generated, the two copies are merged into one, and stored with the union of their used operator bits.

To regenerate the solution path, we can use bidirectional search, performing a frontier search from the initial state, and also from the goal state, until the two search frontiers meet. This gives us a state on a solution path that is approximately halfway between the initial and goal states. We then apply divide-and-conquer to find a solution path from the initial state to the intermediate state, and from the intermediate state to the goal state. An intermediate state can also be identified with a unidirectional search, which is simpler than bidirectional search.

Frontier search significantly decreases the space requirements of a best-first search in many problem spaces. For example, the size of the grid space described above grows as the square of the search radius, but the size of the frontier grows only linearly. For a complete breadth-first search, the space complexity is reduced from the size of the problem space to the width of the problem space, or the maximum number of states at any search depth. For the 4×4 Fifteen Puzzle, this reduction is a factor of over 13, and for the 2×8 Fifteen Puzzle it reduces space by a factor of over 20. For a complete breadth-first search of the 22-disc four-peg Towers of Hanoi problem, frontier search reduces the space required by a factor of over 63. For other problems, such as Rubik's Cube, it does not provide a large advantage.

Instead of storing used operator bits, one can store Closed nodes until all their children have been expanded [Bode and Hinz 1999; Zhou and Hansen 2003]. Newly generated nodes are compared to the Open and Closed lists to detect duplicates. In a problem with a small number of legal operators, frontier search may require less memory, since it doesn't store any Closed nodes, but stores only one bit per operator for each Open node. Conversely in a problem with a large branching factor but many duplicate nodes, frontier search may require more space. In general, however, frontier search will be faster since it never regenerates Closed nodes.

2.4.2. *Directed Graphs.* Frontier search is more complex on directed graphs. The basic problem is that expanding an Open node can regenerate a Closed node that is arbitrarily deep in the interior of the search. To deal with this problem, when a node is expanded we generate all its legal children, and also generate "dummy nodes" for each of its neighbors that could possibly regenerate it, with the operators from the dummy node to the expanded node marked as used. Then, when a duplicate of a dummy node is generated as a legal child, the used operators of the dummy copy are unioned with those of the legal child node. Note that this requires the ability to determine for each state what other states could generate it. Otherwise, there seems to be no alternative to saving all Closed nodes.

2.4.3. *Deleting Sterile Nodes.* A *fertile* node will generate at least one child when it is expanded, and a *sterile* will not generate any children. In frontier search, when all the operators of a node are used, it becomes sterile. Such a node is counted among the nodes at its depth, and then deleted, rather than storing it until the next iteration. This reduces the amount of storage required, and also saves some I/O time by not writing out sterile nodes to disk, and not reading them back in.

3. *Previous Work on Graph Search Using External Memory*

3.1. BRYAN. We believe that Jerry Bryan invented delayed duplicate detection to perform brute-force breadth-first searches of Rubik's Cube using external memory, primarily magnetic tape. His work was not published, but rather briefly described in a series of posts to the "Cube-Lovers" mailing list starting in December 1993.

Rather than checking for duplicates as soon as nodes are generated, they are simply appended to a list of nodes. At the end of each level of the breadth-first search, the newly-generated nodes are sorted by their state representation, bringing duplicate nodes to adjacent positions. These duplicates are then eliminated in a sequential scan. The sorted list of new nodes is also compared to the sorted list of nodes at the previous depth, deleting any additional duplicates.

The largest search Bryan performed using this technique was a partial breadth-first search of Rubik's cube to depth 11 using the quarter-turn metric. This will be described in more detail in Section 6.2. By making use of the symmetry of the cube, the number of states was reduced by a factor of 48, resulting in a search of over 1.3 billion nodes. This result was posted to Cube-Lovers in February 1995.

3.2. ROSCOE. Delayed duplicate detection first appears in the scientific literature in the context of breadth-first search for automatic verification [Roscoe 1994]. Roscoe compared newly-generated nodes to all previously-generated nodes to eliminate duplicates. His implementation didn't manipulate disk files, but worked in main memory, relying on virtual memory to handle the overflow to disk. His system was run on problem spaces with up to ten million nodes. If the problem space is significantly larger than the capacity of main memory, however, the asymptotic I/O complexity of this algorithm is at least the size of the problem space times the depth of the breadth-first search. For the 22-disk 4-peg Towers of Hanoi problem, for example, the depth of a complete breadth-first search is 394 moves.

3.3. STERN AND DILL. Stern and Dill [1998] also implemented a breadth-first search with DDD in a model checker for hardware verification. They maintain a disk file of all nodes generated. When memory becomes full, the disk file of existing nodes is linearly scanned to eliminate duplicates from memory, and the new nodes remaining in memory are appended to the disk file, and to a separate queue of Open nodes. They report results on problems with over a million states. Unfortunately, the I/O complexity of their algorithm also grows at least as fast as the size of the problem space times the search depth. In fact, its I/O complexity is worse if a complete level of the search won't fit in memory, since the complete Closed list is scanned every time memory becomes full, in addition to the end of each level of the search.

Extensions of this work have appeared in Penna et al. [2002] and Bao and Jones [2005]. Penna et al. [2002] reported solving a problem with about a billion states.

3.4. Explicit Graphs. There also exists a body of work on algorithms for explicitly represented graphs stored on disk, typically in the form of adjacency lists. For example, Munagala and Ranade [1999] give an algorithm for breadth-first search on undirected graphs that is similar to the sorting-based DDD algorithm described below. It generates the nodes at level k from the nodes at level $k-1$ as follows: it first generates the multiset of all neighbors of nodes at level $k-1$. It then sorts this multiset by state representation, and scans the sorted list to remove duplicates. Finally, it removes from this list all nodes that appear in the sorted lists of nodes at levels $k-1$ and $k-2$. Mehlhorn and Meyer [2002] improve on this algorithm by reducing the amount of scanning of the adjacency lists, and Meyer [2001] considers the case of undirected graphs of bounded degree. See Katriel and Meyer [2003] for an overview of this work. While none of these papers report implementations of their algorithms, Ajwani et al. [2006] implement and empirically compare the algorithms of Munagala and Ranade [1999] and Mehlhorn and Meyer [2002]. The focus of this body of work is on reducing the asymptotic number of I/O operations, and ignores computation costs.

4. *Delayed Duplicate Detection*

Here we describe our disk-based search algorithms and their implementation in detail. For the most part, we use the example of breadth-first search. For simplicity, we first describe DDD in memory with both Open and Closed lists. Next, we consider how to combine DDD with frontier search in memory. We then consider sorting-based DDD on disk, followed by a more efficient hash-based DDD algorithm. Parallelization of these algorithms is critical for performance and is discussed in detail. Next, we consider interleaving the two phases of node expansion and duplicate merging in order to reduce the amount of storage required. We then consider the more general best-first search with DDD, followed by an algorithm called breadth-first heuristic search. Finally, we consider issues of reliability and interruptibility, which are critical for algorithms that can run for months at a time.

4.1. DDD in Memory. For simplicity, we first describe breadth-first search with delayed duplicated detection running entirely in memory, storing both Open and Closed lists. This algorithm applies to both directed and undirected graphs, and doesn't require the ability to determine all the parents of a state. This is very similar to the algorithm briefly mentioned by Roscoe [1994], although his was augmented by virtual memory.

At the beginning of each iteration of a breadth-first search, the Open list contains those nodes at the current search depth, and the Closed list contains those nodes expanded at previous depths. We assume that both lists are in sorted order of state representation. Each Open node is expanded, and its children are appended to a list of child nodes at the next depth. This list is then sorted by state representation. By using an algorithm such as quicksort, this sort can be done in place, requiring no additional memory, and all the data accesses occur sequentially at the head and tail of the list, resulting in good cache performance. Next, the Closed list, Open list, and list of child nodes at the next depth are scanned simultaneously in order, deleting from the list of child nodes those that appear in the Open or Closed list. The deletion is done by maintaining two pointers into the child list, the first pointing to the first empty position, and the second pointing to the first element that hasn't been examined yet. If this element is not a duplicate, it is copied to the empty position indicated by the first pointer, and both pointers are advanced. If this element is a duplicate node, then just the second pointer is advanced. Finally, the sorted Open and Closed lists are merged into a single sorted Closed list, and the filtered and compacted list of child nodes becomes the new Open list. Merging the sorted Closed and Open lists will require additional space temporarily.

4.1.1. *Combining Frontier Search with DDD.* To save space and time in an undirected problem space, or one where all possible parents of a state can be determined, we combine frontier search with delayed duplicate detection. There are two cases to consider, depending on whether or not a pair of duplicate nodes can appear in consecutive levels of a breadth-first search. We begin by characterizing those graphs in which duplicate nodes can appear at successive levels.

THEOREM 1. *In an undirected problem-space graph, the same state can appear in consecutive breadth-first search levels if and only if there exist odd-length cycles.*

PROOF. To show that duplicate nodes at consecutive depths imply odd-length cycles, assume that a state x appears at two consecutive depths in a BFS of a graph. These two nodes must have a deepest common ancestor a. Thus, there is a path of length d from a to one copy of state x, and another path of length $d + 1$ from a to the other copy of state x. Combining these two paths together results in a cycle of length $2d + 1$, which is odd.

Next, we need to show that an odd-length cycle implies duplicate nodes at consecutive levels. Let d_i be the depth of state i, which is the shallowest depth at which it first appears in a breadth-first search. d_i will depend on the root node, which we assume is fixed for this argument. Given two adjacent nodes in a graph, their depths can differ by at most one, since each can be reached from the other in one move. Thus, in going from one node to an adjacent node, the depth can only increase by one, decrease by one, or stay the same. If we travel all the way around a cycle, the depth has to return to the depth of the node we start at. This means that the number of depth increases must equal the number of depth decreases. Assume that we have a cycle of odd length. In an odd-length cycle, the number of depth increases can only equal the number of depth decreases if there are at least two adjacent nodes x and y that are at the same depth $d_x = d_y$. When node x is first expanded, it will generate another copy of node y at depth $d_x + 1 = d_y + 1$. Thus, there will be two copies of node y at consecutive depths d_y and $d_y + 1$. Thus, in any breadth-first search of a graph with an odd-length cycle, there will be at least one pair of duplicate nodes at successive depths. □

4.1.2. *No Duplicates in Consecutive Levels of Search.* For example, consider the sliding-tile puzzle. In any cycle in the problem-space graph, the blank must return to its original position. If we color the squares in a sliding-tile puzzle in a checkerboard pattern, then every move of the blank goes from a square of one color to one of the opposite color. Thus, any tour of the blank that returns to the same position must be of even length. Therefore, any cycle in the problem space graph must be of even length. This means that a breadth-first search of this problem can never generate duplicate nodes at consecutive depths.

In that case, breadth-first frontier search with DDD works as follows in memory. Each iteration of the BFS starts with an Open list of nodes all at the same depth d, stored in a circular buffer. As each node at the head of the buffer is expanded, it is deleted and its children at depth $d + 1$ are appended to the end of the buffer. By using a circular buffer, the nodes at depth $d + 1$ can overwrite the expanded nodes at depth d. This is referred to as the expansion phase of the iteration. Once all the nodes at depth d have been expanded, the nodes at depth $d + 1$ are sorted by state representation, in the sort phase. This brings duplicate nodes to adjacent positions in the buffer. If an algorithm such as quicksort is used, the sorting is done in place, and all accesses to the buffer occur at only two positions, resulting in good cache performance. Finally, in the merge phase, in a single sequential scan through the buffer, duplicate nodes are removed after forming the union of their used operator bits, and empty positions are filled by following entries. This list becomes the Open list for the next iteration of the search. The used operators prevent generating nodes at depth $d - 1$, and the lack of odd-length cycles eliminates the need to compare the nodes at depth $d + 1$ to those at depth d for duplicates.

4.1.3. *Duplicates in Consecutive Levels of Search.* If the problem space graph contains odd-length cycles, then a breadth-first search will generate duplicate nodes at successive depths, and expanded nodes at one depth must be saved until they can be compared against newly generated nodes at the next depth to detect duplicates.

For example, in the Towers of Hanoi, moving the smallest disc from peg A to B to C and back to A produces a cycle of length three. If we rule out consecutive moves of the same disc, even the 2-disc 3-peg problem has odd-length cycles. For example, starting with two discs on peg A, the moves 1AB, 2AC, 1BA, 2CB, 1AC, 2BA, 1CA form a cycle of length seven, where the numbers indicate the disc size.

To see the problem with the simple breadth-first frontier DDD algorithm described above on a graph with odd-length cycles, consider the simple triangle graph in Figure 1. Assume that node A is generated first via operator w, causing it to be marked as used. In the next iteration, node A is expanded and deleted, generating node B with operator x marked as used, and node C with operator y marked as used. In the next iteration, node B is expanded and deleted, creating another copy of node C with operator z marked as used. These two copies of node C are not merged because we are in the middle of an iteration. Instead, the first copy of node C is expanded and deleted, generating another copy of node B, with operator z marked as used. At the end of this iteration, we have one copy each of nodes B and C, each with operator z marked as used. In the next iteration, both these nodes will be expanded and deleted, generating two copies of node A. In the following sort-merge phase, these two copies of node A will be merged into a single copy with operators x and y marked as used, but not operator w. In the next iteration, node A will be expanded, generating its neighbor via operator w, causing the search to

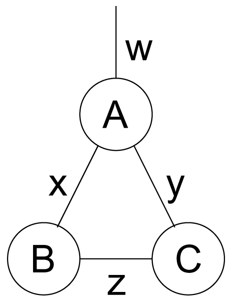

FIG. 1. Triangle Graph.

"leak" into the interior. Note that both pure frontier search, and pure DDD, work just fine on this example, but their combination does not.

The solution to this problem is when expanding nodes at depth d, we don't delete them immediately. Note that the list of nodes at depth d was left in sorted order by the sort and merge phases of the previous iteration. Once all the children at depth $d + 1$ are generated, they are sorted by state representation. Then, as we then scan this list sequentially to merge duplicate nodes, we simultaneously scan the sorted list of nodes at depth d. If a node at depth $d + 1$ also appears at depth d, it is simply deleted. Once this merge phase is completed, all the nodes at depth d can be deleted. Thus, in a graph with odd-length cycles, our algorithm must maintain up to two levels of the search in memory simultaneously.

4.1.4. *Advantages of DDD in Memory.* The primary reason for describing DDD in memory is pedagogical, allowing us to explain the DDD mechanism without the additional complexity of disk files. These algorithms are useful in their own right, however, because they often run faster than standard BFS. The reason is that standard BFS uses a hash table to detect duplicate nodes as soon as they are generated. Since a hash table is designed to spread its entries randomly, it has very poor cache performance, with most accesses missing the cache and going to main memory. The difference in access time between the first-level cache and main memory can be several orders of magnitude. Conversely, in our DDD algorithms, all memory accesses are sequential, resulting in excellent cache performance.

Another advantage of DDD in memory is that it can require less memory than a hash table, which requires more memory per element than simply the state description. If chaining is used to handle collisions, then space for pointers is needed, and if open-addressing is used instead, a significant fraction of the table must remain empty for good performance. DDD does not need memory to handle collisions, but does require additional memory to temporarily store duplicate nodes.

4.2. SORTING-BASED DDD ON DISK.

We now consider sorting-based delayed duplicate detection on disk. We believe that this is very similar to the algorithm implemented by Jerry Bryan on Rubik's Cube. Before beginning, however, a note about virtual memory is in order. Virtual memory is designed to allow a program to use a larger address space than the available physical memory, without the programmer worrying about which pages are in memory and which are on disk. Thus, in principle one could implement BFS with a hash table that was significantly larger than the physical memory, and leave it to the virtual memory system to handle the page faults. In practice, however, such a program would perform very poorly, since most hash-table accesses would result in a page fault. The DDD algorithms described above would perform better in a virtual memory system, since all the memory accesses are sequential. In general, however, the best performance is achieved by the programmer explicitly managing what data is in memory versus disk at any given time. Thus, our DDD algorithms directly control the creation, reading, writing, and deletion of disk files.

4.2.1. *Even-Length Cycles.* We begin with the simpler case of only even-length cycles, allowing nodes to be deleted as soon as they are expanded. Instead of storing the nodes in memory, we store them on disk. The simplest solution is for all nodes at a given depth to be stored in a single file. We include the search depth as part of the file name. During the expansion phase of each iteration, the nodes to be expanded at depth d are read sequentially, and the nodes generated at depth $d + 1$ are written sequentially. In practice, rather than reading and writing one node at a time, larger blocks are read into a buffer in memory, and written from a buffer in memory. Alternatively, this buffering can be left to the operating system.

Instead of storing all the nodes at a single depth in one file, we break them up into a sequence of separate files for several reasons. The first is that once a set of nodes has been read for the last time, if they are in a sequence of files, as soon as a file has been read completely, it can be deleted, freeing the corresponding disk space. If they are all in one file, the file can't be deleted until it has been read completely. A second reason is to facilitate sorting, discussed below. Each file name includes a sequence number, in addition to the search depth.

Once all the nodes at depth d have been expanded and deleted, the nodes at depth $d+1$ must be sorted. There are many different algorithms for sorting disk files. See, for example Garcia-Molina et al. [2000, pp. 42–48]. We describe here the one we implemented, which is a multi-way merge sort. As the child nodes are generated, they are written to a large buffer in memory. Once this buffer is full, the nodes are sorted internally using quicksort, duplicates within the buffer are eliminated, and the buffer is written to a disk file. This process continues until all nodes at depth $d + 1$ have been generated, sorted in groups, and written to disk.

Next, the individual sorted files must be merged into a single sorted sequence, eliminating duplicates between files. We maintain a separate buffer in memory for each file, initially filled with the first group of nodes in each file. The first node from each file is then used to initialize a heap data structure whose size is the number of files. At each step we remove the top node of the heap, which is the next node in the total order, and compare it to the last node from the top of the heap. If it is a duplicate node, it is discarded, and otherwise it is written to an output buffer. In either case, this node is replaced at the top of the heap with the next node from the input buffer that it originally came from, and percolated down to its correct position in the heap. When the output buffer becomes full, we write it out to disk. When an input buffer becomes empty, we refill it from its corresponding input file.

Since we can't delete an input file until it has been read completely into memory, the multi-way merge step can increase the amount of disk space needed temporarily by up to a factor of two, if each separate sorted sequence is in a single file. The solution to this problem is to divide each sorted sequence into multiple files, each no bigger than the size of the memory buffers for the multi-way merge. Then, as each individual file is read into memory, it can be immediately deleted from disk.

4.2.2. *Odd-Length Cycles.* In a problem space with odd-length cycles, we also have to delete nodes at depth $d + 1$ that previously appeared at depth d. This is also done during the multi-way merge. We add an additional memory buffer for the sequence of sorted files of nodes at depth d, which is initially filled from the first file in the sequence. A pointer into this buffer is maintained at the first node whose value is greater than or equal to the node at the top of the heap. When a node at the top of the heap also appears in this buffer, the node is deleted, rather than writing it to the output buffer. When the pointer in this buffer reaches the end of the buffer, the buffer is overwritten with the next file of nodes at depth d.

4.3. HASH-BASED AND DIRECT-ADDRESS DELAYED DUPLICATE DETECTION. The time-complexity of this approach is $O(n \log m)$, where n is the total number of nodes, and m is the number of nodes at a given search depth. In some of our experiments, m was in the hundreds of billions, and $\log m$ was almost 40. Sorting, however, is a more expensive operation than is needed to remove duplicates. Here we present a more efficient means of removing duplicates based on hashing.

Consider, for example, the task of eliminating all duplicates from a single very large file of nodes of the 30-disc four-peg Towers of Hanoi problem. As explained above, a state can be uniquely identified by two bits for the location of each disc, for a total of 60 bits. We divide the discs into the 16 largest, and the 14 smallest, for reasons we will explain below. First, we read each node, and write it to a file based on the position of the 16 largest discs. Thus, all nodes in any given file will have the 16 largest discs in identical positions, and any sets of duplicate nodes must be confined to the same file. Then, we read one file at a time into a hash table in memory, detect and merge any duplicate nodes in that file, and write out just one copy of each node to an output file. This allows us to merge duplicate nodes in linear time without sorting, using two orthogonal hash functions, one based on the largest discs, and the other based on the smallest discs in this case. When using hash-based DDD, there is no reason to merge all the nodes at a given depth into a single file. Rather, when nodes are first generated, they are written to separate files based on their largest discs, eliminating the first step described above.

4.3.1. *Reducing the Size of State Descriptions.* Another advantage of maintaining nodes in separate files is that it reduces the size of the state descriptions. For example, representing a complete state of the 30-disc four-peg Towers of Hanoi problem requires 60 bits, or most of an eight-byte integer. If all the states in a given file have their 16 largest disks on the same pegs, then the positions of those disks can be encoded in the file name, and each state in the file only has to specify the positions of the 14 smallest discs. This requires only 28 bits and fits in a four-byte integer, saving a factor of two in disk space.

In a frontier search, used operator bits are stored in addition to the description of each state. For the Towers of Hanoi problem, one used operator bit is needed for each peg, indicating whether the disc on top of that peg was moved to reach that state. Thus, the four-peg problem requires four used operator bits. These four bits, added to the 28 bits needed to represent the state, sum to a 32 bit single-precision integer. This is the reason for partitioning the discs of the 30-disc problem into the 16 largest and 14 smallest. For problems with fewer discs, the positions of the 14 smallest discs are stored with each state, and the positions of the remaining larger discs are encoded in the file names.

4.3.2. *Direct-Address Tables.* In some problems, we can save considerable amounts of memory by using a direct-address table instead of a hash table. In a direct-address table, the mapping from states to indices in the table is one-to-one, eliminating all collisions. For example, in the four-peg Towers of Hanoi problem, if we allocate an array with 4^{14} entries, then we can use the 28-bit string representing the positions of the 14 smallest disks as the index into this table.

A direct-address table saves memory because we don't store the state description in the table, since the index uniquely determines the state. In our example, we store only the four used operator bits in the table, and not the 28-bit state description. This allows us to pack two entries to a byte, and the entire table occupies only $4^{14}/2$ or 128 megabytes of memory. This saves at least an order of magnitude compared to a open-addressed hash table holding the same number of entries.

There is also a time savings, since by eliminating collisions, we eliminate the need to search subsequent locations when adding or probing for an element in the table.

4.3.3. *Merging Small vs. Large Files.* When we examine in more detail either the hashing or direct-address approaches to detecting and merging duplicate nodes, an interesting efficiency issue emerges. To merge the duplicates in a file, we must start with an empty table in memory. Then, each node in the file must be read in, stored in its appropriate place in the table, or merged with an existing entry. Finally, each nonempty entry must be written out to a file, and the table must be emptied in preparation for merging another file.

The easiest way to implement these last two steps is to linearly scan the table, write out each nonempty entry, and then empty that entry. An additional advantage of this method is that the table is accessed sequentially, resulting in excellent cache performance. With a file that contains only a small number of entries, however, most of the time will be spent scanning empty entries in the table.

An alternative approach is to implement these last two steps by re-reading each entry in the input file, writing out the corresponding entry in the table, and then emptying that entry. If the entire file fits in the input file buffer, then the file need not be re-read, but only the file buffer. This has a running time that is proportional to the number of entries in the original file, rather than the size of the table. A drawback of this approach is that the accesses to the table will be random, resulting in poor cache performance. With a large file that fills most of the table, this approach will be much less efficient than linearly scanning the table, since it requires re-reading the original file and randomly accessing the table.

The solution to this dilemma is to use both approaches, depending on the size of the file being merged. For small files, identifying the non-empty entries by re-reading the file buffer is chosen, whereas for large files a sequential scan of the table is used. The breakpoint between these two methods should be determined empirically. In our experiments with the four-peg Towers of Hanoi problem, we used a breakpoint of about seven million elements in a table of 128 megabytes.

4.4. I/O COMPLEXITY. Here, we consider the disk I/O complexity of delayed duplicate detection, measured by the number of nodes read from and written to disk. We assume here a complete breadth-first search of the problem space, with bidirectional operators. We also assume that we detect and remove duplicate nodes at each level of the search. The I/O complexity is a function of the number of unique states or vertices, and the number of operator instances or edges in the problem-space graph. The number of bidirectional edges is approximately the number of unique states times the average branching factor, divided by two. Since duplicates are detected at each level, no state is expanded more than once, and the number of node expansions is no greater than the number of unique states in the space. In practice, it will be less, since when all operators generating a state are used, the state is sterile and not expanded, and hence our analysis gives an upper bound. By using frontier search, no operator instance is applied in both directions, and hence the number of nodes generated is equal to the number of edges in the problem-space graph.

We first consider problem spaces with only even-length cycles, and then consider odd-length cycles. The I/O complexity is the same for both the hashing and direct-address algorithms, and the sorting-based algorithm, assuming a single multi-way merge is done in memory. We assume that the individual files of child nodes are small enough to be sorted entirely in memory, or hashed in memory.

When a node is first generated, it is written to either a sorted output file, or to a hash file. It is then read into memory, in either the multi-way merge or into a hash or direct address table. These operations are performed on every generated node. Thus, there is one write and one read for each edge of the problem-space graph.

In any of the merge algorithms, only one copy of each state is written to an output file. Each of these is read back into memory for expansion during the next iteration. Thus, for problem spaces with only even-length cycles, there will be one write and one read for each state or vertex of the problem-space graph, and one write and one read for each operator or edge of the problem-space graph.

For problems with odd length cycles, each expanded node must be read again during the multi-way merge or the hash or direct-address merge following its expansion. This adds an additional read for each vertex of the problem-space graph.

The sorting-based, hash-based and direct-addressed algorithms all have the same I/O complexity. The sorting-based algorithm has an $n \log n$ time complexity for sorting the individual files and the multi-way merge, however, while the hash-based and direct-addressed algorithms have only linear time complexity in practice. This assumes the existence of good hash functions that balance the size of the different files, and spread out the contents of each file in memory. In the worst case, an adversary could construct states that defeat any particular pair of hash functions.

4.5. Managing Very Large Numbers of Files. The simplest way to deal with nodes in multiple files is to maintain an array in memory with an entry for each possible file. Each entry contains at least the size of the corresponding expansion or merge file, or zero if it doesn't currently exist.

The problem with this approach is that for very large problems, the number of potential files can grow prohibitively large. For example, to keep track of the positions of 30 discs in the four-peg Towers of Hanoi problem, if each node encodes the position of 14 discs, plus four used operator bits, then the filename must encode the positions of the remaining 16 discs. This results in $4^{16} = 2^{32}$ possible files.

To deal with this many possible files, we store in memory a hash table with information on only the files that exist on disk at any given point in time.

4.6. Interleaving Expansion and Merging. So far, we have assumed that all the nodes at a given depth are expanded before any of the children at the next depth are merged to eliminate duplicates. This may require more disk space than necessary, for two reasons. One is that all duplicates nodes at the child depth will exist at the end of the expansion phase. The second is that for a graph with odd-length cycles, all the nodes at the parent depth must be saved throughout the expansion phase to eliminate all duplicate child nodes.

In some problems, however, an alternative approach can be used. For the Towers of Hanoi, for example, an individual file contains nodes for which the positions of the largest discs are identical. Similarly, for the sliding-tile puzzles, a individual file contains nodes for which the positions of some of the tiles, including the "blank tile", are identical. Thus, when expanding nodes in a given parent file, there are only a limited number of child files the children can belong to, and similarly a given child file can only receive nodes from a limited number of parent files.

For example, in a single move of a sliding-tile puzzle, the blank can only move to at most four other positions. Those tiles that are not adjacent to the blank cannot move at all, and those tiles adjacent to the blank can only move into the the blank position in a single move. Thus, if all the nodes in any given file are characterized by a common set of positions for any subset of the tiles including the blank, then neighboring nodes can only go to one of at most four other files.

In the Towers of Hanoi problem, the nodes in any given file are characterized by a common set of positions for a subset of the discs. Any state of the four-peg Towers of Hanoi problem has at most six legal moves as described above. Thus, the nodes in one file can generate neighboring nodes in at most six other files. We refer to any pair of files where a parent node in one file can possibly generate a child node in the other file as *neighboring files*.

We refer to files of nodes to be expanded at a given depth d as parent files, and files of nodes generated at depth $d + 1$ as child files. Once all the neighboring parent files of a given child file have been expanded, then the child file can be merged to eliminate duplicate nodes, since all of its child nodes will have been generated. At that point, the original child file containing duplicates can be deleted, saving disk space. In addition, for graphs with odd-length cycles, once a child file has been merged, then the corresponding parent file can be deleted if it has already been expanded, since it is no longer needed to detect duplicates in the child file.

To minimize the maximum amount of disk space needed, we interleave expansion with merging within an iteration. We merge a child file as soon as it becomes eligible for merging, with merges taking priority over expansions. The order in which files are expanded impacts when child files can be merged, and hence the maximum disk space required. Determining the optimal ordering is a difficult combinatorial problem. We determine this order heuristically by performing a breadth-first search over the space of files. For example, if the largest discs determine the file names for the Towers of Hanoi, we perform a breadth-first search of the space consisting of only the large discs, and then expand the corresponding files in the same order.

4.7. Best-First Frontier Search with DDD.

So far, we have used breadth-first search to illustrate delayed duplicate detection. We now turn our attention to best-first searches such as Dijkstra's algorithm [Dijkstra 1959] and the A* algorithm [Hart et al. 1968]. Recall that in a best-first search algorithm, each node has an associated cost, and at each step of the algorithm an Open node of lowest cost is expanded. Both frontier search and delayed duplicate detection apply to best-first search as well as breadth-first search.

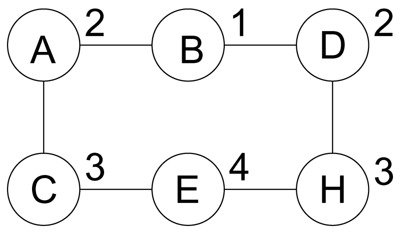

FIG. 2. Example of Best-First Frontier Search with DDD.

A difficulty occurs when we combine frontier search with DDD in a best-first search such as A*. For example, consider the example graph fragment shown below in Figure 2. Assume that node A is the start state, all edges have unit cost, the heuristic value of each node is written next to the node, and the cost function is the A* cost function of $f(n) = g(n) + h(n)$. In a best-first search, an iteration most naturally corresponds to the period of time during which all nodes of a given cost value are expanded, with duplicate detection occurring between such iterations. Here we assume that the cost of a child node is always greater than or equal to the cost of its parent, which is true of A* with a consistent heuristic function.

In this example, the first iteration expands nodes with the f value of the initial node A, which is 2. Node A is expanded and deleted, generating nodes B and C, with $f(B) = 1 + 1 = 2$, and $F(C) = 1 + 3 = 4$. In the same iteration, node B is expanded and deleted, generating node D with $f(D) = 2 + 2 = 4$. The next iteration expands all nodes with an f value of 4. Thus, node C is expanded and deleted, generating node E with $f(E) = 2 + 4 = 6$, and node D is expanded and deleted, generating node H with $f(H) = 3 + 3 = 6$. The next iteration expands those nodes with an f value of 6. Thus, node H is expanded and deleted, generating another copy of node E with $f(E) = 4 + 4 = 8$. The fact that this is a duplicate node is not detected, since we are in the middle of an iteration. Instead, the original copy of node E is expanded and deleted, regenerating node H with $f(H) = 3+3 = 6$. This copy of node H is expanded and deleted, regenerating node D with $f(D) = 4 + 2 = 6$, and this node is expanded and deleted, regenerating node B with $f(B) = 5 + 1 = 6$, and finally this node is expanded and deleted, regenerating node A with $f(A) = 6 + 2 = 8$. Thus, the search has "leaked" several moves into the interior of the search, regenerating Closed nodes that are no longer stored. On the next iteration, nodes E and A will be reexpanded.

The essential problem here is that in A*, an arbitrarily long chain of nodes can be generated during the same iteration, before duplicate detection is performed, allowing the search to leak into the interior of a frontier search. One solution to this problem is to perform duplicate detection more often. For example, an iteration may be limited to expanding all the nodes of a given cost that are present at the beginning of an iteration, but deferring the expansion of any children of the same cost to the next iteration. The drawback of this approach is that successive iterations that expand children of the same cost can expand fewer and fewer nodes between duplication-detection phases, resulting in large overheads.

Note that we can combine frontier search with DDD in Dijkstra's algorithm, as long as all edges have positive cost, and a merge phase is performed after all nodes of the same cost have been expanded. This guarantees that the frontier cannot advance by more than one edge in any one place before duplicates are detected, preventing the search from leaking back into the interior.

4.8. BREADTH-FIRST HEURISTIC SEARCH. A solution to this problem is *breadth-first heuristic search* (BFHS) [Zhou and Hansen 2006a]. Breadth-first heuristic search explores nodes in a search space in breadth-first order. Each iteration only expands nodes at a given depth. This can be followed by a duplicate detection phase. When combined with frontier search, this prevents the search from leaking into the interior.

Each search has an associated cost threshold, which is an upper bound on the cost of nodes expanded in that iteration. If the f cost of a generated node exceeds the cost threshold for the search, the node is simply deleted.

The cost threshold for a search can be determined in different ways. In some problems, such as the four-peg Towers of Hanoi, it is known that there exists a solution of a given cost, and we are only interested in solutions of lower cost. Thus, the cost threshold can be set to one move less than the known solution cost.

More generally, the cost threshold can be set using iterative deepening [Korf 1985], resulting in *breadth-first iterative-deepening-A** [Zhou and Hansen 2006a]. In this algorithm, the initial cost threshold is the heuristic value of the initial state, and the cost threshold for each succeeding search is the minimum cost among all nodes pruned on the previous search. Each search consists of a sequence of expansion and merge phases, each expanding nodes at a given depth.

4.9. PARALLEL DDD. Using magnetic disk storage for nodes ameliorates the space bottleneck of best-first search, allowing some searches to run for months as we will see. Thus, time becomes the limiting resource. Many modern workstations have multiple processors, and most modern processors have multiple cores, requiring parallel processing to maximize performance. Furthermore, these algorithms are disk I/O intensive, so even with a single processor, parallel processing allows some processes to run while others are blocked waiting for disk I/O. We focus here on a shared-memory model, since this is the standard model of a multi-processor workstation, or a multi-core processor. The parallel programming model we used is multi-threading, in particular Posix threads [Nichols et al. 1996].

The unit of work assigned to a thread is the expansion of the nodes in a file, or the merging of duplicate nodes in a file. We maintain two queues, one containing those files remaining to be expanded in the current iteration, and another with those files eligible to be merged, once all the files that could contribute children to them have been expanded. There is a small pool of threads that first look to the merge queue for work, and if the merge queue is empty, they take a file on the expansion queue. These common queues are protected by a mutual exclusion lock, allowing access by only a single thread at a time.

Each thread needs its own local memory. For expansion, each thread has an input file buffer for parent nodes, and a separate output buffer for each file that child nodes could be written to. For merging, in addition to an input and output file buffer, the main memory requirement is for either a hash or direct-address table. Since a given thread is either expanding or merging nodes at any given time, but not both, the same physical memory can be used for both the expansion output file buffers and the hash or direct-address table for merging. Memory is statically allocated, and divided evenly among the different threads.

The main problem in parallelizing these algorithms is avoiding contention between multiple threads writing the same memory locations. There is no contention between two threads merging separate files. Each merge thread reads a separate file of child nodes, and possibly a file of parent nodes if there are odd-length cycles, merges the duplicates using a separate table in memory, and writes a separate output file. In general, there is no contention between expansion threads and merge threads, since a child file is not merged until all neighboring parent files that could contribute children to it have already been expanded.

This leaves contention between expansion threads. In particular, two threads expanding different parent files that have neighboring child files in common could write to the same child file. Somewhat surprisingly, no coordination is required here. If two parent files with a common neighboring child file are expanded simultaneously, the children are temporarily stored in separate output buffers. When a output buffer becomes full, a call is made to the operating system to write the buffer to the corresponding file. We rely on the operating system to serialize these writes, since the order in which the child nodes appear in a file is immaterial.

All that remains is to keep track of the number of nodes in each child file. When a thread completes the expansion of a file and returns to the expansion and merge queues for more work, it adds the number of nodes it added to each child file to the total for that file. Since the work queues are protected by a mutual exclusion lock, these counters can only be updated by one thread at a time.

For simplicity, all threads complete work on one iteration, and node counts are generated and printed, before any work on the next iteration begins.

4.10. Reliability, Interruptibility, and Correctness. In computations that run for weeks or months, reliability and interruptibility are important considerations, since faults are more likely to occur, and restarting a computation from scratch after a fault can be prohibitively expensive. Unlike other large-scale computations, a breadth-first search cannot be easily decomposed into independent computations, which can fail and be restarted until they succeed. The primary faults that occur are power failures, system crashes, and I/O errors.

4.10.1. *Power Failures.* Power failures can last from a fraction of a second to minutes or hours. Even a minor power glitch will crash a computer. Fortunately, an uninterruptible power supply (UPS) provides a cheap and effective solution to this problem. A UPS contains a battery and a data output to the computer. The computer plugs into the UPS, which plugs into the power outlet. The battery is kept continuously charged. In the event of a temporary power glitch, power to the computer is maintained by the battery, and the computer operating system is notified. Depending on the power demands of the computer, and the size of the battery, a UPS can power the computer for tens of minutes.

To deal with longer power outages, our programs maintain their state in a set of files on magnetic disks. A UPS plays an important role in this case as well, however. Consider the merging of a child file to eliminate duplicates, for example. For robustness, the old file is not deleted until the new file is written to disk. That way, in theory, at least one of the files is always available. The problem is that the system call to write the new file to disk will return as soon as the data has been moved to kernel space in the operating system. As the result, the following call to delete the original file can complete before the new file has been committed to disk. If a power failure occurs at that point, both files can be lost.

A UPS also solves this problem. If a power outage lasts long enough to exhaust the battery, the UPS informs the computer that it will no longer be able to maintain power. The operating system then initiates a clean shutdown of the system, including flushing all file caches to disk, leaving the computation in a safe state.

Our programs are designed to be interruptible, and restarted in the middle of an iteration. Expansion and merge files are not deleted until they are completely expanded or merged. If a merge is interrupted, its output file can be deleted, and the file remerged from scratch. If an expansion is interrupted, it may have already written children to merge files. The file can be reexpanded from scratch, generating additional copies of children already generated, but they will be merged as duplicates, with no affect on the number of unique nodes at a given search depth.

4.10.2. *I/O Errors.* Many of our experiments early in this project were plagued with disk I/O errors. Disk specifications often cite "unrecoverable error" rates of one in 10^{13} to 10^{15} bits. For example, if writing a word to disk results in a parity error, there is no way for the system to recover this data. Since our experiments involved continuously reading and writing disk files for weeks or months, these rates become significant. In addition, disk driver bugs can cause additional errors.

The main problem with unrecoverable disk write errors is that they are not detected until the corresponding files are read. For example, if during an iteration that expands parent nodes at depth d a write error is made in an file of child nodes at depth $d + 1$ after merging duplicates, this file will not be read and the error not detected until the next iteration which expands nodes at depth $d + 1$. At that point, the parent files that generated those nodes may have been deleted from the disk.

One solution to this problem is to save files from multiple levels of the search at the same time. Then, if an error is detected during a given iteration, the search can be restarted from a previous iteration. The drawback of this approach is that it requires more disk storage. In particular, it increases the maximum amount of storage needed near the width of the search graph. This works as long as the disk errors are confined to user files. If an error occurs in system data, such as an I-node table, then the entire file system can be corrupted.

A more general solution to this problem is a Redundant Array of Inexpensive Disks (RAID) [Patterson et al. 1988]. For example, in a simple RAID, an extra disk holds the exclusive OR of the corresponding bits on the other disks. In the event of an unrecoverable error on any one disk, its data can be reconstructed from the others, often without interrupting the program. In the event of complete loss of a disk, that disk can be unplugged and replaced, and its data reconstructed from the other disks. Unfortunately, a RAID requires additional I/O time, as well as more disk space, to write and store the redundant data.

4.10.3. *Correctness.* In the absence of observed faults, how do we know our results are correct? There are two main sources of error: software bugs and transient faults, such as undetected I/O errors. There are also several reasons for confidence.

The first is that we ran the same searches on smaller problems, for which other techniques are feasible. There are a number of very different techniques for implementing a breadth-first search, for example. For small problems, we have implemented most of these methods, and compared their results on problems that are as large as possible for each method. If very different methods give exactly the same numbers of nodes at each depth, that is strong evidence that the results are correct.

Unfortunately, for the larger problems, only the most space and time-efficient techniques are feasible, usually leaving us with only one feasible implementation. In addition, the largest problems take so long to run that the probability of an undetected error increases, and it is not feasible to rerun them to check for transient errors. In these cases, our main source of confidence in our complete breadth-first search results is that the total number of unique nodes generated equals the total number of states in the problem space. In particular, given the nature of frontier search, and our bijective encoding of states as unique integers, it seems very unlikely that we would get the correct total number of states in the presence of errors.

Ultimately, confirmation will require other researchers reproducing our results. For example, a complete search of the Fourteen Puzzle was done independently by Blai Bonet, Julio Castillo, and Cesar Romero (B. Bonet, personal communication, 2007), with identical results.

5. *Linear-Time Lexicographic Permutation Indexing*

Before describing our experimental results, we describe a related result that was instrumental in our experiments. As mentioned above, there are several advantages to efficiently mapping between states and unique integers. One is that it optimally compresses the size of a state description. Another is that it allows duplicate detection using a direct-address table, which is much more compact than a hash table, since an index into a direct-address table uniquely identifies the state, eliminating the need to store a state description with each entry. We previously described a very simple such mapping for the four-peg Towers of Hanoi problem, which represents the peg that each disc is on with two bits, and a complete state with a bit string of length $2n$, where n is the number of discs. Since individual bits represent the positions of different discs, operators can be applied directly in this representation.

States of permutation problems, such as the sliding-tile puzzles or Rubik's Cube, for example, are more difficult to encode. An ideal mapping function is a bijection between permutations of n elements, and integers in the range 0 to $n! - 1$. We'd like to be able to compute this mapping in both directions as efficiently as possible. Myrvold and Ruskey [2001] provide such a mapping that runs in time linear in n in both directions, but their mapping is not lexicographic.

5.1. LEXICOGRAPHIC MAPPINGS. In a *lexicographic mapping*, ordering the permutations by their integer representations results in a lexicographic ordering. For permutations of three elements, for example, this mapping is: 012-0, 021-1, 102-2, 120-3, 201-4, 210-5. Myrvold and Ruskey [2001] claim that, "... it seems that a major breakthrough will be required to do that computation in linear time, if indeed it is possible at all." Here we provide a practical linear-time lexicographic mapping, by using $O(2^{n-1})$ space. For most applications, this is not a significant limitation. For the Fifteen Puzzle, for example, this only requires $2^{15} = 32,768$ additional words of memory.

In a lexicographic mapping, the high-order bits represent the positions of the first few elements in the permutation. For the Fifteen Puzzle, the high-order bits can be used to represent the position of the blank and tiles 1, 2, and 3, for example, with the remaining bits used to represent the positions of the remaining tiles. These high-order bits can be encoded in the file names, so that the actual nodes in the file encode only the positions of the remaining tiles. Another advantage of this scheme is that the number of neighbors of a given file is limited to the maximum branching factor of any node in the problem space, which is four in this case. The reason is that if all the nodes in a file have the blank and a subset of tiles in a particular set of positions, there are at most four other possible positions for these tiles that can be reached in a single move from any of these states. This limits the number of file buffers that are needed for child nodes when expanding nodes in a parent file. Furthermore, it allows interleaving of file expansion and file merging, since a child file can be merged as soon as all its possible parent files have been expanded, reducing the maximum amount of storage needed at any time.

Next we describe our mapping of permutations to integers, and then vice-versa.

5.2. MAPPING PERMUTATIONS TO INTEGERS. For clarity, we divide the mapping into two steps, first mapping the permutation to a sequence of digits in *factorial base*, and then mapping the factorial-base digits to an integer. A number in factorial base is of the form: $d_{n-1} \cdot (n-1)! + d_{n-2} \cdot (n-2)! + \cdots + d_2 \cdot 2! + d_1 \cdot 1!$ Digit d_i can range from 0 to i, inclusive. Any integer can be uniquely represented as a number in factorial base, and any integer in the range 0 to $n! - 1$ can be uniquely represented as an $n-1$ digit number in factorial base.

To map a permutation to a sequence of digits in factorial base, we subtract from each element of the permutation the number of original elements to its left that are less than it. For example, the mapping from permutations of three elements to factorial base digits is: 012-000, 021-010, 102-100, 120-110, 201-200, 210-210. Note that the rightmost factorial digit is always zero, and is dropped.

Then, given a permutation represented as a sequence of digits in factorial base, we perform the arithmetic operations indicated by the formula above to compute the actual integer value. For permutations of three elements, for example, this results in the desired values: 012-00-0, 021-01-1, 102-10-2, 120-11-3, 201-20-4, 210-21-5.

A straightforward implementation of this algorithm takes $O(n^2)$ time to compute the digits in factorial base. Here, we provide an $O(n)$ algorithm.

We scan the permutation from left to right, constructing a bit string of length n, indicating which elements of the permutation we've seen so far. Initially, the string is all zeros. As each permutation element is encountered, we use it as an index into the bit string, counting from the left starting with zero, and set the corresponding bit to one. When we encounter element k in the permutation, to determine its corresponding digit in factorial base, we need to subtract the number of elements less than k to its left, which is the number of ones in the first k bits of our bit string so far. We extract the first k bits by shifting the string to the right by $n-k$ positions. This can be done with a single instruction if the bit string fits in a single machine word, which is typically 64 bits long on modern machines. This reduces the problem to counting the number of ones in a bit string.

We solve this problem in constant time by using the bit string as an index into a precomputed table, containing the number of ones in the binary representation of the index. For example, the initial entries of this table are: 0-0, 1-1, 2-1, 3-2, 4-1, 5-2, 6-2, 7-3. The size of this table is $O(2^{n-1})$ where n is the number of permutation elements, since the largest possible permutation element is $n-1$, and hence the n-bit string will always be shifted right by at least one bit.

To map a sequence of factorial-base digits to an integer, we multiply each digit by the corresponding factorial, and add the results, in linear time. This gives us a linear-time algorithm for mapping permutations of n elements in lexicographic order to unique integers from zero to $n! - 1$, assuming that n is less than or equal to the word size of our machine. For permutations of more than 64 elements, the precomputed table would be too much too large to store in memory, but only a tiny fraction of such a problem space could be generated in practice anyway.

5.3. MAPPING INTEGERS TO PERMUTATIONS. We now consider the inverse mapping from integers in the range 0 to $n! - 1$, to permutations of n elements. Again, we first describe a quadratic-time algorithm, and then a linear-time algorithm.

First, the integer value is mapped to a sequence of digits in factorial base, using integer division and remaindering. For example, the rightmost digit is the remainder mod 2, for the next digit we divide the integer by 2! then take the remainder mod 3, next we divide by 3! and take the remainder mod 4, etc. In general, digit i counting from the right starting with one is $v/i! \bmod i + 1$, where v is the integer value.

Next, initialize an array indexed from 0 to $n - 1$ with the values 0 to $n - 1$, respectively. Take the leftmost digit in the factorial base number, use it as an index into the array, and the value stored there becomes the first element of the permutation. This will always equal the left-most digit. Then, "burp" the array by sliding each element in the array with a higher index to the adjacent position to its left with an index one less, filling in the position vacated, and reducing the length of the array by one. The next element of the permutation is the value stored in the array at the index corresponding to the next factorial-base digit to the right. Slide the following elements to the left by one again, and continue until the entire permutation is generated. This requires time quadratic in n, since each step requires linear time to burp the array.

We can also perform this mapping in linear time, using $O(n2^n)$ space. The main idea is to pre-compute and store each possible instance of the array described above. For each bit string of length n, we store a one-dimensional array of size n. Each bit string represents a set of permutation elements by setting the bits at the corresponding indices to one. The corresponding one-dimensional array contains the remaining permutation elements not in the set, in sorted order. For example, given the bit string 1010, representing elements 0 and 2 in a permutation of four elements, the corresponding array contains elements 1 and 3 in that order.

This set of 2^n arrays is computed once, and stored as a two-dimensional array, using the bit strings as the first set of indices. In practice, each array only needs to be as long as n minus the number of ones in its corresponding bit string, and we don't need an array for the empty bit string at all. Using such variable-length arrays complicates access to them, however.

To map an integer to a permutation, we first map it to a sequence of digits in factorial base, using integer division and remaindering as described above. The first element of the permutation is the left-most factorial-base digit. We initialize a bit string of length n to all zeros. Then, we set the bit whose index equals the first element to one, retrieve the array corresponding to this new bit string, and take the value at the index corresponding to the next factorial-base digit as the next element of the permutation. We set the bit corresponding to that element to one, and continue in this fashion until we have computed the entire permutation.

5.4. EXPERIMENTAL COMPARISON. We implemented both the quadratic and linear-time algorithms above in both directions, and tested them by mapping all permutations of up to 14 elements. The experiments were done on an IBM workstation with a two gigahertz AMD Opteron processor. Table I shows the results. The

TABLE I. Mapping between Integers and All Permutations of N Elements

n	P to I, $O(n^2)$	P to I, $O(n)$	Ratio	I to P, $O(n^2)$	I to P, $O(n)$	Ratio
11	16	3	5.333	21	19	1.105
12	223	37	6.027	286	244	1.172
13	3,280	511	6.419	4,074	3,454	1.180
14	52,437	7,513	6.980	62,378	50,623	1.232

first column gives the number of elements n in the permutation. The second column is the total running time in seconds to map all permutations of n elements to their integer representation in lexicographic order, using the quadratic-time algorithm described above. The third column is the corresponding time for the linear-time algorithm, and the fourth column is the ratio between the two. The fifth column gives the total time in seconds to map each integer from 0 to $n! - 1$ to its corresponding permutation using the quadratic-time algorithm described above, the sixth column is the corresponding time for the linear-time algorithm described above, and the last column is their ratio.

To map permutations of 14 elements to integers, the linear algorithm was almost seven times faster than the quadratic algorithm, and this ratio increases with increasing problem size. For mapping integers to permutations, however, the linear algorithm was only slightly faster than the quadratic algorithm. One reason may be that mapping an integer to a sequence of digits in factorial base requires integer division and remaindering, which are much more expensive than multiplication, and dominate the cost of the mapping. This suggests that for mapping integers to permutations, the linear-time algorithm may not be worth the extra space required.

5.5. More Recent Work. Since this work was originally published in Korf and Shultze [2005], there has been further work on efficiently mapping between integers and permutations in lexicographic order. In particular, Bonet [2008] presents an algorithm that runs in $O(n \log n)$ time, and exponential space, but significantly less space than our algorithm. It also allows trading off space for time. In his experiments mapping permutations of 16 elements to integers, his algorithm ran at approximately the same speed as our linear-time algorithm.

6. *Experimental Results*

Here we describe our main experimental results. They include partial breadth-first searches of Rubik's Cube up to depths 10 and 11, complete breadth-first searches of sliding-tile puzzles up to the 4×4 and 2×8 Fifteen Puzzles, complete breadth-first searches of the Four-Peg Towers of Hanoi problems with up to 22 discs, and heuristic searches to determine the optimal solution depths for all Four-Peg Towers of Hanoi Problems with up to 31 discs. For the sliding-tile puzzles and Towers of Hanoi problems, these searches produced new results as will be described below.

6.1. Hardware and Software Environment. All these experiments were done on an IBM Intellistation A Pro workstation with dual two-gigahertz 64-bit AMD Opteron Processors, and two gigabytes of memory, running various versions of Linux. We found CentOS Linux to be the most reliable. For disk storage on the sliding-tile puzzles and Towers of Hanoi experiments we used four external LaCie "Big Disk Extreme" units, plus a Firewire 800 (IEEE 1394b) interface card for each. Each unit packages two 250 gigabyte drives, with a maximum transfer rate of 100 megabytes per second. In addition, we used two 300-gigabyte and one 400-gigabyte Serial ATA (SATA) disks internal to the workstation.

Since this work began in 2002, there have been considerable advances in disk technology, and this setup is no longer state-of-the-art. For example, we currently have the same amount of storage in three one-terabyte internal SATA-II drives. Furthermore, our internal SATA drives have not produced any of the disk I/O errors that we experienced with the external LaCie Firewire drives. For one of our Rubik's Cube searches, we used the three one-terabyte internal SATA-II disks, plus one LaCie disk, for a total of 3.5 terabytes of storage. We also repeated our search of the 4×4 Fifteen Puzzle using just the three one-terabyte disks.

Multiple disks also provide an opportunity for I/O parallelism, by assigning different files to different disks. For example, with three equal-size disks, we assign each file to a disk by taking the file number modulo three.

6.2. PARTIAL BREADTH-FIRST SEARCHES OF RUBIK'S CUBE. Although these experiments were done last, we describe them first because they are the simplest. We performed two breadth-first searches of the standard Rubik's Cube, to the maximum depth we could reach with three terabytes of storage. There are two standard ways to count moves. In the *quarter-turn metric*, a primitive move is a 90-degree twist of any face in either direction. In the more common *face-turn metric*, a 180-degree twist also counts as a single move. The face-turn metric allows cycles of any length, while the quarter-turn metric only allows even-length cycles.

The standard Rubik's cube consists of eight corner cubies, with three faces each, and twelve edge cubies, with two faces each. Each corner cubie can be in one of eight different positions, and in one of three different orientations, for a total of 24 configurations. Each edge cubie can be in one of twelve different positions, and in one of two different orientations, for a total of 24 configurations. Thus, the state of any cubie can be stored in five bits. In a reachable state, the position and orientation of one corner cubie and one edge cubie are determined by the others, so we only store seven corner cubies and eleven edge cubies, for a total of 90 bits.

In the face-turn metric, we only need to keep track of which of the six faces have been rotated to reach each state, requiring six used-operator bits, since rotating the same face twice in a row never generates a new state. Thus, a state of the cube plus its used-operator bits can be stored in 96 bits in the face-turn metric, which fits in three 32-bit words. In the quarter-turn metric, we need to know which specific operators have been applied, requiring 12 used operator bits. Thus, we used six 16-bit words to store the state, and another for the used operator bits.

In either metric, twists of opposite faces are commutative, so we only allow opposite faces to be twisted consecutively in one order. In addition, in the quarter-turn metric, we don't allow two consecutive counter-clockwise twists or three consecutive clockwise twists, since these generate duplicate nodes. These optimizations reduce the number of nodes generated, reducing both the runtime and the storage needed.

In these experiments, we executed a breadth-first frontier search using just one thread, performed all expansions before any merges in each iteration, and stored the entire state with each node, rather than factoring out common elements and storing them in the file names. We hashed a state to a 32-bit word. In the face-turn metric, we exclusive-ORed the corresponding bits in the three 32-bit words, excluding the

TABLE II. PARTIAL BREADTH-FIRST SEARCHES OF RUBIK'S CUBE

	Face-Turn Metric		Quarter-Turn Metric	
Depth	Unique States	Nodes in Tree	Unique States	Nodes in Tree
1	18	18	12	12
2	243	243	114	114
3	3,240	3,240	1,068	1,068
4	43,239	43,254	10,011	10,011
5	574,908	577,368	93,840	93,840
6	7,618,438	7,706,988	878,880	879,624
7	100,803,036	102,876,480	8,221,632	8,245,296
8	1,332,343,288	1,373,243,544	76,843,595	77,288,598
9	17,596,479,795	18,330,699,168	717,789,576	724,477,008
10	232,248,063,316	244,686,773,808	6,701,836,858	6,791,000,856
11		3,266,193,870,720	62,549,615,248	63,656,530,320
12		43,598,688,377,184	583,570,100,997	596,694,646,092

used operator bits. In the quarter-turn metric, we appended pairs of 16-bit words together, then exclusive-ORed the three pairs. To determine the file to which we hash a state, we took this 32-bit value modulo the prime number of files, which was 4999 in the face-turn metric, and 1511 in the quarter-turn metric. When merging a file in memory, we took this same 32-bit value modulo the prime size of the hash table, which was 100,000,007 in both cases. The hash table included the complete state description, and was open addressed. These constant values were determined by the two gigabytes of memory we had available.

In the face-turn metric, the graph has odd-length cycles, and hence the nodes at each depth must be compared to nodes at the preceding depth to remove duplicates. This is not required in the quarter-turn metric. We generated and eliminated all duplicate nodes up to depth 10 in the face-turn metric, and up to depth 11 in the quarter-turn metric. No attempt was made to optimize the code for time or space, since these runs only took a few days each.

The results are shown in Table II. The first column gives the depth of the search, the next pair of columns correspond to the face-turn metric and the last pair of columns correspond to the quarter-turn metric. In each pair, the first column is the number of unique states in the problem-space graph, eliminating all duplicates, and the second column is the number of nodes in the problem-space tree, using only the duplicate elimination rules described above. The tree values are easily computed in time linear in the depth from simple recurrence relations.

Jerry Bryan was the first to compute the values in this table. He performed the quarter-turn search to depth 11 using sorting-based delayed duplicate detection on magnetic tape in 1995. By using symmetries of the cube to reduce the node generations by a factor of 48, his search generated about 1.3 billion nodes. The depth 12 quarter-turn search and the face-turn searches were done using a very different algorithm [Fiat et al. 1989], which only stores the nodes at half the search depth. This algorithm is specific to permutation groups, however, and is not very efficient for complete breadth-first searches. The reason is that its asymptotic time complexity is the square of the number of states at up to half the maximum search depth. For example, if the conjectured diameter of 20 face turns for the Rubik's Cube problem space is correct, then the algorithm of Fiat et al. [1989] would generate about 6.3×10^{23} nodes, which is more than four orders of magnitude larger than the total number of states in the problem space, which is about 4.3252×10^{19}.

Our experiments confirmed all of Bryan's results, except for the depth 12 quarter-turn search, which we didn't do since storing the states at depth 12 without symmetry reductions would require over 16 terabytes of storage in our representation.

6.3. Complete Breadth-First Searches of Sliding-Tile Puzzles.

The well-known sliding-tile puzzles consist of a rectangular frame containing square tiles, with one empty or "blank" position. The legal moves are to slide any tile horizontally or vertically adjacent to the blank position into that position.

We performed complete breadth-first searches of several such puzzles. A complete BFS can be used to construct pattern database heuristics [Culberson and Schaeffer 1998], and to determine the radius of a problem space.

6.3.1. *State of the Art without Delayed Duplicate Detection.* Without delayed duplicate detection, the most memory-efficient algorithm for a complete BFS uses a bit array in memory, with one bit per state, and a last-in-first-out queue on disk. This requires a one-to-one mapping between states and integers, and ideally a bijection. For the sliding-tile puzzles, only half of the possible permutations of the tiles are reachable from any given state [Johnson and Storey 1879]. Thus, for an $n \times m$ sliding-tile puzzle, the total number of reachable states is $(n \times m)!/2$. The largest sliding-tile puzzle that we could solve using this algorithm is the 2×6 or the 3×4 Eleven Puzzles. These problems have $12!/2 = 239,500,800$ reachable states, requiring almost 29 megabytes of memory. The next larger problem, the 2×7 Thirteen Puzzle, would require over five gigabytes of memory for this algorithm.

6.3.2. *Symmetry.* We used symmetry to reduce the number of node generations on some of these problems [Culberson and Schaeffer 1998]. For example, in the 4×4 Fifteen Puzzle, our initial state has the blank in the upper left-hand corner. Thus, the problem is symmetric about the main diagonal from upper left to lower right. For every state reached via a path traversed by the blank starting from the initial state, there is a mirror-image state reached by starting from the same initial state, but reflecting the path of the blank about the main diagonal.

If we reflect the initial state about the main diagonal, each tile off the main diagonal is mapped to another tile, and the tiles on the main diagonal are mapped to themselves. Given any state, we compute its mirror image by reflecting it about the main diagonal, then renumbering every tile by the tile it is mapped to when reflecting the initial state about the main diagonal. Note that the mirror image of some states is the same state, such as the initial state for example.

In the symmetric problems, each pair of mirror-image states is represented by a canonical member of the pair. To count the number of states at a given level, we must determine whether the mirror-image of each state is itself or another state. This symmetry reduces the number of nodes by about a factor of two where it applies, but the overall time reduction is less due to the computational overhead of computing mirror images. Two-fold symmetry was used for the 4×4 Fifteen Puzzle. It can also be used for the Fourteen Puzzle if the blank starts in the center.

6.3.3. *Implementation Details.* We performed complete breadth-first frontier searches with delayed duplicate detection, using all of the ideas presented above, including symmetry where applicable, on sliding-tile puzzles up to and including the Fifteen Puzzles. For the larger problems, the file names encoded the positions of the blank and first three physical tiles. For the Fifteen Puzzles, this leaves 12 remaining tiles whose positions must be stored with each node, for a total of $12!/2 =$

TABLE III. COMPLETE SEARCHES OF SLIDING-TILE PUZZLES

Size	Tiles	Radius	Total States	Max Width	Depth	Ratio
2×2	4	6	12	2	2	6.000
2×3	5	21	360	44	14	8.182
2×4	7	37	20,160	1,999	24	10.085
3×3	8	31	181,440	24,047	24	7.545
2×5	9	55	1,814,400	133,107	36	13.361
3×4	11	53	239,500,800	21,841,159	36	10.966
2×6	11	80	239,500,800	13,002,649	49	18.419
2×7	13	108	43,589,145,600	1,862,320,864	66	23.406
3×5	14	84	653,837,184,000	45,473,143,333	42	14.486
4×4	15	80	10,461,394,944,000	784,195,801,886	53	13.340
2×8	15	140	10,461,394,944,000	367,084,684,402	85	28.499

239, 500, 800 reachable states per file. Each of these can be stored in 28 bits, since $2^{28} = 268, 435, 456$. Since we also need one used-operator bit for each of the four possible moves (up, down, left, and right), each node just fits in a 32-bit word. File merging was done with a direct address table in memory, storing only the four used-operator bits in each entry, requiring 114 megabytes per table.

The code was implemented with parallel threads. Using two processors, the best performance was achieved with six threads on the 3×5 Fourteen Puzzle. This resulted in a speedup of slightly greater than a factor of two over a single thread. Even with a single processor, we would expect a speedup with multiple threads, since when a threads blocks waiting for disk I/O, other threads can still run.

6.3.4. *Results.* Our results are shown in Table III. The first column gives the dimensions of the problem, and the second the number of physical tiles n. Problems are normally named by the number of physical tiles, but note that there are two Eleven Puzzles and two Fifteen Puzzles. The third column gives the radius of the problem space, or the maximum distance of any state from an initial state with the blank in a corner. The fourth column gives the size of the problem space, or the total number of reachable states, which is $(n + 1)!/2$. The fifth column gives the width of the problem space, which is the maximum number of nodes at any depth of the search, and the sixth column shows the depth at which it reached this maximum width. The last column is the ratio of the size of the problem space to its width. The horizontal line separates the problems that could be completely searched without delayed duplicate detection from those which could not.

The width of the problem space gives an estimate of the maximum amount of disk space needed by the search, but is not exact for several reasons. One is that during the course of an iteration, there will be files of child nodes containing duplicates that haven't been merged yet. In addition, there is space consumed by I-nodes, filenames, and other data in the file system. On the other hand, the width also includes sterile nodes, which are not stored. For example, if we take the 4×4 Fifteen Puzzle, multiply the width by four bytes stored per node, and divide by two for symmetry, we get an estimate of about 1.568 terabytes of disk space. The maximum storage actually used was about 1.4 terabytes.

The ratio of the size of the problem space to the width of the space indicates the savings due to frontier search, since a standard BFS must store the entire problem space. This ranges up to a factor of 28.5 for the 2×8 Fifteen Puzzle.

	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15

7	14	5	4	3	2	9	1
15	6	13	12	11	10	8	

FIG. 3. 2×8 Fifteen Puzzle: Initial State and State at Maximum Distance of 140 Moves Away.

In comparing the results for the two Eleven Puzzles and two Fifteen Puzzles to each other, we note that in both cases the problem with the larger aspect ratio has both a larger radius and smaller width. This is due to the lower branching factor for the problem with the larger aspect ratio.

Using four Firewire disks and three internal SATA disks, the Fourteen Puzzle search took about 25 hours to run, The 4×4 Fifteen Puzzle took 28 days and 8 hours to run, and the 2×8 Fifteen Puzzle ran for about 63.5 days. Even though the two Fifteen Puzzles have the same number of states, the 4×4 problem is symmetric, while the 2×8 problem is not. We repeated the search of the 4×4 Fifteen Puzzle using the three internal SATA II one-terabyte disks, and this ran for only 17 days and 16 hours, using five parallel threads.

To our knowledge, the is the first time that a complete breadth-first search has been performed for the problems below the horizontal line in Table III, and possibly for some above the line as well. This establishes the radius for the Thirteen (108 moves), Fourteen (84 moves), and 2×8 Fifteen Puzzle (140 moves), and confirms the radius of 80 moves for the 4×4 Fifteen Puzzle. This latter value was first computed by Brungger et al. [1999], using more complex methods. For the 4×4 Fifteen Puzzle, there are 17 different states at a depth of 80 moves from an initial state with the blank in the corner, while for the 2×8 Fifteen Puzzle there is a unique state at the maximum distance of 140 moves from the initial state. Figure 3 shows the initial state of this problem and the unique state 140 moves away. Table IV shows the number of unique states at each depth of the 4×4 Fifteen Puzzle.

6.4. FOUR-PEG TOWERS OF HANOI. The well-known Towers of Hanoi Problem consists of a set of pegs, and a set of different-sized discs, initially stacked on one peg in decreasing order of size. The goal is to move all the discs from the initial peg to a different goal peg. The rules are that only the top disc on any peg can be moved at any one time, and that a larger disk can never be placed on top of a smaller disc. With three pegs and n discs, there is a guaranteed optimal recursive strategy that requires $2^n - 1$ moves.

As mentioned above, the problem is much more interesting with four or more pegs. In this case, there exists a general solution strategy for any number of discs, and a conjecture dating from 1941 that the strategy is optimal [Frame 1941; Stewart 1941], but this conjecture remains unproven. Thus, absent a proof, systematic search is the only way to verify this conjecture for a given number of discs.

We performed two different sets of experiments on the Four-Peg Towers of Hanoi problem. We first describe complete breadth-first searches starting with all discs on one peg, and then heuristic searches to verify the "presumed optimal" solution strategy for moving all discs from one peg to another.

TABLE IV. COMPLETE SEARCH OF FIFTEEN PUZZLE

Depth	States	Depth	States	Depth	States
0	1	27	116,238,056	54	777,302,007,562
1	2	28	204,900,019	55	742,946,121,222
2	4	29	357,071,928	56	683,025,093,505
3	10	30	613,926,161	57	603,043,436,904
4	24	31	1,042,022,040	58	509,897,148,964
5	54	32	1,742,855,397	59	412,039,723,036
6	107	33	2,873,077,198	60	317,373,604,363
7	212	34	4,660,800,459	61	232,306,415,924
8	446	35	7,439,530,828	62	161,303,043,901
9	946	36	11,668,443,776	63	105,730,020,222
10	1,948	37	17,976,412,262	64	65,450,375,310
11	3,938	38	27,171,347,953	65	37,942,606,582
12	7,808	39	40,271,406,380	66	20,696,691,144
13	15,544	40	58,469,060,820	67	10,460,286,822
14	30,821	41	83,099,401,368	68	4,961,671,731
15	60,842	42	115,516,106,664	69	2,144,789,574
16	119,000	43	156,935,291,234	70	868,923,831
17	231,844	44	208,207,973,510	71	311,901,840
18	447,342	45	269,527,755,972	72	104,859,366
19	859,744	46	340,163,141,928	73	29,592,634
20	1,637,383	47	418,170,132,006	74	7,766,947
21	3,098,270	48	500,252,508,256	75	1,508,596
22	5,802,411	49	581,813,416,256	76	272,198
23	10,783,780	50	657,076,739,307	77	26,638
24	19,826,318	51	719,872,287,190	78	3,406
25	36,142,146	52	763,865,196,269	79	70
26	65,135,623	53	784,195,801,886	80	17

Within a file, a node is represented by the positions of the 14 smallest discs, which occupy 28 bits. In addition, there are four used-operator bits identifying the discs that have been moved to reach the state by the pegs they are on top of. Thus, each node occupies 32 bits. The positions of the remaining larger discs are encoded in the file name. File merging is done with a direct address table in memory that stores only the four used-operator bits. Each such table is $4^{14}/2$ bytes, or 128 megabytes. This problem space contains odd-length cycles, so nodes at a given depth must be used to eliminate duplicates at the next depth.

If we start with all discs on one peg, then the three remaining pegs are symmetric, and can be permuted in six ways. Thus, each state is represented by a canonical state in which the three non-initial pegs are sorted by the largest disc on them. This symmetry saves almost a factor of six, or three factorial.

6.4.1. *Complete Breadth-First Searches.* We performed complete breadth-first searches of the Four-Peg Towers of Hanoi problem, starting with all discs on one peg, for all sizes up to and including 22 discs. These experiments used the full range of techniques described above. Our results are shown in Table V. The first column shows the number of discs. The second column gives the optimal number of moves needed to transfer all discs to another peg. The third column shows the problem radius starting with all discs on one peg. The fourth column gives the total number of states in the space, which is 4^n, where n is the number of disks. The fifth column shows the maximum width of the search space, and the sixth column the

TABLE V. COMPLETE SEARCHES OF FOUR-PEG TOWERS OF HANOI PROBLEMS

Discs	Optimal	Radius	Total States	Max Width	Depth	Ratio
1	1	1	4	3	1	1.333
2	3	3	16	6	2	2.666
3	5	5	64	30	4	2.133
4	9	9	256	72	7	3.555
5	13	13	1,024	282	10	3.631
6	17	17	4,096	918	14	4.462
7	25	25	16,284	2,568	19	6.341
8	33	33	65,536	9,060	25	7.234
9	41	41	262,144	31,638	32	8.286
10	49	49	1,048,576	109,890	41	9.542
11	65	65	4,194,304	335,292	52	12.509
12	81	81	16,777,216	1,174,230	64	14.288
13	97	97	67,108,864	4,145,196	78	16.190
14	113	113	268,435,456	14,368,482	94	18.682
15	129	130	1,073,741,824	48,286,104	111	22.237
16	161	161	4,294,967,296	162,989,898	134	26.349
17	193	193	17,179,869,184	572,584,122	160	30.004
18	225	225	68,719,476,736	1,994,549,634	188	34.454
19	257	257	274,877,906,944	6,948,258,804	218	39.561
20	289	294	1,099,511,627,776	23,513,260,170	251	46.761
21	321	341	4,398,046,511,104	77,536,421,184	286	56.722
22	385	394	17,592,186,044,416	278,269,428,090	333	63.220

depth at which this maximum width occurs. The last column is the total number of states divided by the maximum width, which is the memory savings of breadth-first frontier search over standard breadth-first search. Note that for 22 disks, frontier search saves a factor of more than 63 in memory over standard breadth-first search. This problem took 9 days and 18 hours to run, and required over 2.2 terabytes of storage. The horizontal line below 17 discs represents the largest size problem for which the presumed optimal had previously been verified [Bode and Hinz 1999].

6.4.1.1. A VERY SURPRISING ANOMALY. The astute reader will notice that the values in the second and third columns of Table V are the same in most cases, but differ for 15, 20, 21 and 22 discs. In other words, in most but not all cases, the optimal number of moves needed to transfer all the discs from one peg to another is the same as the radius of the problem space starting with all discs on one peg. With 15, 20, 21, and 22 discs, there are some states that are further away from the standard initial state than the standard goal state. This is never the case for the three-peg problem. We have no explanation for this surprising phenomenon, but conjecture that these values will differ for any number of discs greater than 19.

6.4.2. *Brute-Force Half-Depth Searches.* Next, we consider trying to verify or disprove the length of the "presumed optimal" solution for moving a given number of discs from one peg to another in the four-peg Towers of Hanoi problem. In this case, the symmetry between the standard initial and goal states provides an enormous reduction in the amount of search required [Hinz 1997]. Any solution must, at some point, move the largest disc from the initial peg to the goal peg. In order to do that, all the remaining discs must be distributed over the two remaining intermediate pegs. We refer to such a state as a "middle" state. Once we reach such a state, we move the largest disc from the initial peg to the goal peg. If we then

TABLE VI. BRUTE-FORCE HALF-DEPTH SEARCHES OF FOUR-PEG TOWERS OF HANOI PROBLEM

Discs	Optimal	Maximum States	Total States	Time
22	385	539,050,519	11,476,448,258	1:13:01
23	449	1,974,178,490	48,655,324,192	6:13:05
24	513	6,266,537,659	172,329,745,265	1:03:35:19
25	577	17,331,989,387	520,984,238,575	4:04:04:36

execute all the moves made to reach the middle state, but in reverse order, we will return all but the largest disc to the initial peg. However, if we make all the moves made to reach the middle state in reverse order, but interchange the initial and goal pegs in all those moves, we will move all but the largest disc to the goal peg, thus completing the solution. Thus, all we need to do to find the length of an optimal solution is to find a shortest path to a middle state, then double the length of this path and add one move of the largest disc. Thus, we only have to search to half the optimal solution depth. We refer to such a search as a "half-depth" search.

We performed brute-force breadth-first half-depth searches of the 22 through 25-disc problems. Our results are shown in Table VI below. The first column gives the number of discs, and the second the optimal solution length. The third column shows the maximum number of unique states at any depth, which occurs at half the solution depth, and the fourth column shows the total number of unique states generated. These numbers reflect the actual numbers of unique states generated and stored by the program, taking advantage of the six-fold symmetry described above. The last line gives the running time of the algorithm in the form of days:hours:minutes:seconds. The first line of this table illustrates the benefit of only searching to half the depth compared to a complete breadth-first search, reducing the running time from almost ten days to one hour and thirteen minutes.

6.4.3. *Heuristic Searches of Four-Peg Towers of Hanoi.*

To verify the optimal solution depth for larger problems, we turn to heuristic search to prune the space.

6.4.3.1. BREADTH-FIRST HEURISTIC SEARCH.

The search algorithm we used is breadth-first heuristic search (BFHS) [Zhou and Hansen 2006a]. As described above, BFHS expands nodes in breadth-first order, applies a heuristic function to each node generated, and deletes those nodes n for which $f(n) = g(n) + h(n)$ exceeds a given cost threshold. Since our goal is to verify or disprove the presumed optimal solution length for a given number of discs, we can set the cost threshold to one less than the presumed optimal solution length. If BFHS fails to find a solution at this depth or less, then the presumed optimal length is indeed optimal. In practice, we set the cost threshold to the presumed optimal solution depth, and verify that a solution is actually found at this depth. The reason is that failure to find a shorter solution could be due to software bugs that cause some nodes not to be generated, or errors in our heuristic function that cause too many nodes to be pruned. By reporting the actual solution found, we increase our confidence that it is indeed optimal.

6.4.3.2. PATTERN DATABASE HEURISTIC EVALUATION FUNCTIONS.

To implement any heuristic search, we need a heuristic evaluation function. The function we used is based on a large number of recent improvements to pattern database heuristics.

Pattern databases [Culberson and Schaeffer 1998] are lookup tables stored in memory, and are the most effective heuristics known for many problems, including the sliding-tile puzzles [Korf and Felner 2002], Rubik's Cube [Korf 1997], and the Towers of Hanoi [Korf and Felner 2007]. For example, for the Towers of Hanoi, a pattern database for 15 discs would contain a separate entry for each possible configuration of the 15 pattern discs, for a total of 4^{15} entries in the four-peg problem. Each entry contains the minimum number of moves needed to move all 15 pattern discs from their current peg to the goal peg, assuming there are no other discs in the problem. Since the maximum such value is 130 moves, each entry requires only one byte, and this table would occupy a gigabyte of memory. A pattern database is constructed by performing a single breadth-first search backward from the goal state with the pattern discs, and recording in the table the depth at which each different configuration of pattern discs is generated for the first time. For a problem with more than 15 discs, the pattern database entry for any 15 of the discs serves as a lower bound on the total number of moves needed to solve the problem from its current state, and hence a lower-bound or admissible heuristic function for the original problem. Given a particular state in a heuristic search, the configuration of 15 pattern discs is used to compute an index into the table, and the corresponding entry becomes the heuristic value for that state.

Given two or more admissible heuristics, their maximum is also an admissible heuristic. Thus, given a 15-disc pattern database, we can look up the positions of different and even overlapping sets of 15 discs in the pattern database to get multiple heuristic values, and then take their maximum value as an overall admissible heuristic for any state. Since the number of moves needed to move 15 different-sized discs to the goal peg, ignoring any other discs, is the same regardless of their absolute sizes, we can use the same pattern database for all these lookups.

If we choose multiple sets of discs that are nonoverlapping, then we can add the individual heuristic values in an admissible heuristic, called a disjoint additive pattern database heuristic [Korf and Felner 2002]. The reason is that the individual pattern databases only count moves of their pattern discs, and each move in the Towers of Hanoi only moves one disc. For example, in a 30-disc problem, we can divide the discs into the 15 largest and the 15 smallest, or any other disjoint partition, look up the positions of the corresponding discs in the same 15-disc pattern database, and then add together the two heuristic values. In general, this results in a much more accurate admissible heuristic than taking their maximum.

The more discs that are used in constructing a pattern database, the larger and hence more accurate the resulting heuristic values are. Unfortunately, more discs mean a larger pattern database that occupies more memory. One solution to this dilemma is a *compressed pattern database* [Felner et al. 2007]. For example, we showed above that we can perform a complete breadth-first search of the 22-disk four-peg Towers of Hanoi problem, using disk storage for the nodes. We can identify a subset of 15 of these discs, such as the 15 largest, allocate a one gigabyte pattern database in memory during this search, and initialize all the values to an empty value, such as minus one. As each state of the 22-disc problem is generated, we look up the configuration of the 15 largest discs in the pattern database, and if the corresponding entry in the database is empty, we store the current search depth in the entry. At the end of this search, each entry in the database, corresponding to a particular configuration of the 15 largest discs, will contain the minimum number of moves needed to move all 22 discs to their goal location, where the minimization is over all $4^7 = 16384$ configurations of the 7 smallest discs for which the 15 largest discs are in the corresponding configuration. We refer to this as a 22-disc database compressed to the size of a 15-disc pattern database. Its values are much larger and hence more accurate than a simple 15-disc database, because it represents moves of all 22 discs, while only occupying the memory of a 15-disc database.

As described above, when searching from a state with all discs on one peg to a state with all discs on another peg, we can take advantage of the symmetry between these standard initial and goal states to only search to half the solution depth, looking for a middle state where all but the largest disc are distributed over the two intermediate pegs. This is easily done in a brute-force search, simply by checking that the initial peg contains only the largest disc and the goal peg is empty. It is not so obvious how to do this in a heuristic search, since we don't know how the discs are distributed over the intermediate pegs in an optimal middle state. The naive approach of computing a heuristic to each possible middle state is prohibitively expensive, since there are 2^{n-1} different middle states with n discs.

The solution to this problem is called a *multiple-goal pattern database* [Korf and Felner 2007]. While a standard pattern database is constructed by a BFS starting from the goal state, a multiple-goal pattern database is constructed by initially seeding the BFS queue with all possible goal states. In this case, the BFS would start with all possible middle states in the search queue at depth zero. Thus, at depth one, the search will generate all states that are one move away from some middle state, at depth two all states that are two moves away from some middle state, etc. In other words, the multiple-goal pattern database will contain in each entry the minimum number of moves needed to map the elements in the pattern database to any one of the middle states.

Our actual heuristic for the four-peg Towers of Hanoi combines all these ideas. We describe it in detail for the largest problem we solved, the 31-disc problem. Since we are searching for a shortest path to a middle state, we only need to consider how to move 30 discs from one peg to two other intermediate pegs. We first perform a complete BFS of the 22-disc problem, initially seeding the queue with all possible middle states in which all 22 discs are distributed over the two intermediate pegs. During this search, we construct in memory a compressed pattern database using the configuration of the 15 largest discs as the index into the one-gigabyte table. We separately compute an eight-disc multiple-goal pattern database, initially seeding the breadth-first search with all states where the eight discs are distributed over the two intermediate pegs. This database only occupies 64 kilobytes of memory.

In our BFHS, we compute the heuristic for each state as follows. We first divide the 30 discs into the 22 largest and the eight smallest. We look up the configuration of the 15 largest discs in the compressed pattern database, and add the stored heuristic value to the stored heuristic value based on the eight smallest discs from the eight-disc database. Then we divide the discs into the 22 smallest and the eight largest. We look up the configuration of the 15 largest discs among the 22 smallest in the compressed pattern database, and add this value to the stored heuristic value based on the 8 largest discs from the eight-disc database. Finally, the maximum of these two sums is our overall heuristic value. For the smaller problems, we used the 22-disc database compressed to the size of a 15-disc database, and a separate database for the remaining $n - 22$ discs.

TABLE VII. HEURISTIC SEARCHES OF FOUR-PEG TOWERS OF HANOI PROBLEM

Discs	Optimal	Maximum States	Total States	Time
25	577	5,388,302	443,870,346	2:37
26	641	32,939,487	2,948,283,951	14:17
27	705	135,938,322	12,107,649,475	1:03:59
28	769	588,391,203	38,707,832,296	4:01:02
29	897	7,046,622,727	547,627,072,734	2:05:09:02
30	1025	49,804,167,328	4,357,730,168,514	17:07:37:47
31	1153	250,247,900,242	25,589,806,135,105	102:22:05:30

6.4.3.3. MINIMIZING THE NUMBER OF HEURISTIC CALCULATIONS. Computing the heuristic values from pattern databases can occupy a significant fraction of the time for a node generation, since it involves multiple random accesses to large tables, resulting in poor cache performance. With such heuristics, however, we can easily determine the maximum possible heuristic value $maxh$ for any state, simply by examining the databases. In breadth-first heuristic search, we compare the total cost of a node, $g(n) + h(n)$, to a cost threshold t, and only prune those nodes whose cost exceeds t. Pruning of a node n could not possibly occur unless $g(n)+maxh > t$. Thus, in early iterations of the breadth-first search when the $g(n)$ values of the nodes being expanded are too small to allow any pruning, we don't compute the heuristic values at all, and don't even load the pattern databases into memory. This speeds up the early iterations, and saves memory for file buffers and direct-address tables, allowing more threads to run in parallel. Note that this is a general technique that can be applied to any heuristic search algorithm that prunes based on a cost threshold, such as iterative-deepening-A* (IDA*) [Korf 1985] for example, using a heuristic where the maximum value can be determined a priori.

6.4.3.4. RESULTS. We ran BFHS, using the pattern database heuristic described above, searching for a middle state with all but the largest disc distributed over the two intermediate pegs, for up to 31 discs. If p is the presumed optimal solution depth for a given number of discs, the cost threshold was set to $(p - 1)/2$, to guarantee finding at least one solution, while searching for any shorter solutions. Five parallel threads were run on two processors.

Table VII shows our results for 25 through 31 discs. The first column shows the number of discs, and the second column gives the optimal solution length. The third column gives the maximum number of unique fertile nodes that were stored at any depth, and the fourth column shows the total number of unique fertile nodes that were expanded. These actual node numbers reflect the six-fold symmetry described above. The fifth column shows the running time in days:hours:minutes:seconds. Comparing the first line of this table to the last line of Table VI, both of which represent 25 discs, shows the benefit of the heuristic function, reducing the running time from 100 hours to two minutes and 37 seconds.

In all cases, the length of the shortest solution found equaled the presumed optimal solution length. The 31-disc problem ran for over three months, and required a maximum of two terabytes of storage. Unfortunately, an unrecoverable disk error occurred at depth 419. This required deleting one disk block containing 1024 nodes. Since there are over 196 billion nodes at that depth, there is a one in 191 million probability that one of the lost nodes was on a shorter path to the goal, and hence that our 31-disc result is incorrect.

Previous to our work, the largest problem for which the presumed optimal solution had been verified was 17 discs [Bode and Hinz 1999]. They used a brute-force half-depth search in memory, with the six-fold symmetry reduction described above, storing three adjacent levels of the search at one time.

7. *Subsequent Work*

Since this work first appeared [Korf 2003b], others have used magnetic disk storage to extend memory in search algorithms. We briefly discuss some of this work below.

7.1. STRUCTURED DUPLICATE DETECTION. The most extensive subsequent work is an alternative to DDD called *structured duplicate detection* (SDD) [Zhou and Hansen 2004, 2005, 2006b, 2007a, 2007b].

The key idea of SDD is that the states of many problem spaces can be partitioned into subsets so that all the neighbors of states in one subset can be found in only a small number of other subsets. For example, if we partition the states of a sliding-tile puzzle into subsets based on the position of the blank, then any neighbor of a state with the blank in one position can only have the blank in one of at most four other positions, and hence can belong to one of only four other subsets. These four subsets are known as the *duplicate detection scope* of the first subset.

The size of these duplicate detection scopes can be further limited using a technique called *edge partitioning*. In edge partitioning, nodes are only partially expanded, using only a subset of their operators at any given time. For example, in the sliding-tile puzzles, if only one of the four possible operators is applied to nodes in a given subset at one time, the corresponding children can only be in one subset.

SDD stores each subset of nodes in a separate file. When expanding the nodes from one file, it keeps in memory all the nodes from the expansion file's duplicate detection scope. As each node is generated, it is compared against the nodes in memory, and any duplicate nodes are detected immediately. When expanding nodes from a different subset requires loading into memory a different duplicate detection scope, nodes that are no longer needed in memory are written back to disk. The main advantage of structured duplicate detection is that multiple copies of a node never exist on disk at the same time, reducing both the amount of storage needed, and the I/O time needed to write and read the duplicate nodes to and from disk.

There are several disadvantages as well. One is that SDD is not completely general, but requires that the structure of limited duplicate detection scopes exist in the problem, and be identified. Zhou and Hansen [2006b] have implemented the algorithm for domain-independent planning, with this structure automatically extracted from the problem description, however. A second disadvantage is that for a given partition of the state space and/or operators into subsets, there must be enough memory to store the largest duplicate detection scope for any given subset. If not, then the partition must be refined into smaller subsets to reduce the size of the largest duplicate detection scope. Finally, in the worst case, the number of times each subset of nodes must be read into memory and written out to disk could be as large as the number of subsets that have the given subset in their duplicate detection scopes.

SDD may be more efficient than delayed duplicate detection for problems where the total amount of storage needed is only slightly larger then the amount of memory available. Whether it scales up to the size of searches described here is an open question, however, since it has not been implemented on problems of this scale.

It should be noted that we have taken advantage of limited duplicate detection scopes in some of our implementations of DDD in a couple ways. One is that when expanding a file, we only maintain output buffers for each of its neighboring files, allowing the individual buffers to be larger. Second, when interleaving expansion and merging in the same iteration, we wait until all the expansion files that can contribute children to a given merge file have been expanded before merging the given file. We did not use these techniques in our Rubik's Cube implementation, however, to demonstrate the generality of delayed duplicated detection.

7.2. EXTERNAL A*. Our original work focussed on breadth-first search. *External A** [Edelkamp et al. 2004] extends this work to the A* algorithm using disk storage. Since in A* states are expanded in best-first order of $g(n) + h(n)$, external A* groups together in the same files nodes that have the same g and h values. They apply their algorithm to the Fifteen Puzzle using the Manhattan distance evaluation function.

7.3. DISTRIBUTED MEMORY IMPLEMENTATION. Niewiadomski et al. [2006] combine frontier A* with DDD. They implement their algorithm on a cluster of distributed workstations, but using only semiconductor memory. The key idea in their algorithm is that nodes are distributed to different processors based on their state, so that duplicate nodes will be sent to the same processor, which can then detect duplicates. They apply their algorithm to multiple sequence alignment, and achieve essentially linear parallel speedup.

7.4. IMPLICIT OPEN LIST. A related but significantly different technique uses an *implicit open list* [Kunkle and Cooperman 2007]. This method is most useful for a complete BFS of a problem where the maximum width of the problem space is a large fraction of the size of the problem space. It relies on a function that maps every state to a unique integer index. It uses an array on disk with one entry for each state of the problem space, with each element storing the depth of its corresponding state from the initial state in the BFS. Initially, all the elements are set to "unknown". In each iteration of the BFS, the array is scanned, the states at the current search depth are generated from their indices, the states are expanded generating their children, and the depths of the children in the array are updated. Kunkle and Cooperman [2007] used this algorithm to perform a complete BFS of a large subspace of Rubik's Cube.

We improved this technique by reducing the amount of space needed to two bits for each state of the problem space, and made other changes to significantly reduce the amount of disk-I/O required [Korf 2008]. We used our algorithm, called *two-bit breadth-first search*, to perform the first complete breadth-first searches of several *pancake groups*. These are permutation groups formed by reversing the order of all the elements in an initial prefix of a permutation.

We also used our algorithm to perform a complete BFS of the subspace of Rubik's Cube defined by just the edge cubies, a problem space with about 981 billion states, confirming the results of a previous search of this space by Tomas Rokicki. Note that the Rubik's Cube searches we described in Section 6.2 are fundamentally different than this search, since the former are partial searches of the complete problem space, while this is a complete search of a subspace of the problem space, where a state is defined by only the edge cubies.

8. *Conclusions*

Delayed duplicate detection (DDD) is a technique for breadth-first and best-first search of graphs that are too large to fit in memory. DDD stores its nodes on magnetic disk, which typically has several orders of magnitude more capacity than main memory. The key challenge is detecting and merging duplicate nodes. The main idea is that rather than checking for duplicates as soon as a node is generated, these checks are batched together so that they can be performed efficiently using only sequential access to the disk. This technique was pioneered by Jerry Bryan and others using sorting to merge duplicate nodes. We present hash-based and direct-addressing DDD algorithms that in practice run in time linear in the number of nodes generated, rather than $n \log n$ for the sorting-based approaches.

We also show how to combine DDD with other techniques such as frontier search, and breadth-first heuristic search, and how to parallelize these algorithms. By significantly reducing the storage bottleneck, we are able to run our algorithms on problems that take weeks or months of computation, raising important issues of reliability, interruptibility, and correctness.

A separate contribution of this work is a linear-time algorithm for bijectively mapping between permutations and integers, assuming that the number of elements being permuted is less than or equal to the word size of the machine.

We have implemented our algorithms, and run experiments in three different domains. For Rubik's Cube, we performed partial breadth-search searches to depth 10 in the face-turn metric and depth 11 in the quarter-turn metric, confirming the number of unique nodes at each depth. For the sliding-tile puzzles, we performed the first complete breadth-first searches of the Thirteen, Fourteen, and two different Fifteen Puzzles, computing for the first time the problem radius for three of these problems, and confirming the radius for the 4×4 Fifteen Puzzle.

For the Four-Peg Towers of Hanoi problem, we performed the first complete breadth-first searches of the 17 through 22-disc problems. We discovered a very surprising anomaly, namely that for 15, 20, 21, and 22 discs, the problem radius from the standard initial state with all discs on one peg is greater than the optimal solution length to transfer all discs from one peg to another. We also performed heuristic searches on the 23 through 30-disc problems, verifying in each case the optimality of a conjectured presumed optimal solution. For the 31-disc problem, we verified this result with an error probability of about one in 191 million, due to an unrecoverable disc error.

ACKNOWLEDGMENTS. The idea of multiple-goal pattern databases, which was used in our heuristic searches of the four-peg Towers of Hanoi problem, is due to Ariel Felner. Thanks to Peter Schultze for enormous help in getting and keeping our disks running. Thanks to Satish Gupta and IBM for donating the workstation used in these experiments. Thanks also to two anonymous reviewers for their helpful comments.

REFERENCES

AJWANI, D., DEMENTIEV, R., AND MEYER, U. 2006. A computational study of external-memory bfs algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*. ACM, New York, 601–610.

BAO, T., AND JONES, M. 2005. Time-efficient model checking with magnetic disk. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, New York, 526–540.

BODE, J.-P., AND HINZ, A. 1999. Results and open problems on the Tower of Hanoi. In *Proceedings of the 30th Southeastern International Conference on Combinatorics, Graph Theory, and Computing*.

BONET, B. 2008. Efficient algorithms to rank and unrank permutations in lexicographic order. In *Proceedings of the 1st International Symposium on Search Techniques in AI and Robotics*. Chicago, IL.

BRUNGGER, A., MARZETTA, A., FUKUDA, K., AND NIEVERGELT, J. 1999. The parallel search bench ZRAM and its applications. *Ann. Oper. Res. 90*, 45–63.

CULBERSON, J., AND SCHAEFFER, J. 1998. Pattern databases. *Computat. Intell. 14*, 3, 318–334.

DIJKSTRA, E. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 269–271.

EDELKAMP, S., JABBAR, S., AND SCHROEDL, S. 2004. External A*. In *Proceedings of the 27th German Conference on Artificial Intelligence*. Lecture Notes in Artificial Intelligence, vol. 3238, Springer-Verlag, New York, 226–240.

FELNER, A., KORF, R. E., MESHULAM, R., AND HOLTE, R. C. 2007. Compressed pattern databases. *J. Artif. Intell. Res. 30* (Oct.), 213–247.

FIAT, A., MOSES, S., SHAMIR, A., SHIMSHONI, I., AND TARDOS, G. 1989. Planning and learning in permutation groups. In *Proceedings of the 30th ACM Foundations of Computer Science Conference (FOCS)*. ACM, New York, 274–279.

FRAME, J. 1941. Solution to advanced problem 3918. *Amer. Math. Monthly 48*, 216–217.

GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. 2000. *Database System Implementation*. Prentice-Hall, Englewood Cliffs, NJ.

HART, P., NILSSON, N., AND RAPHAEL, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cyber. SSC-4,* 2 (July), 100–107.

HINZ, A. M. 1997. The Tower of Hanoi. In *Algebras and Combinatorics: Proceedings of ICAC'97*. Springer-Verlag, New York, 277–289.

JOHNSON, W., AND STOREY, W. 1879. Notes on the 15 Puzzle. *Amer. J. Math. 2*, 397–404.

KATRIEL, I., AND MEYER, U. 2003. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*. Lecture Notes in Computer Science, vol. 2625. Springer-Verlag, New York, 62–84.

KORF, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27,* 1, 97–109.

KORF, R. 1997. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*. AAAI Press, Menlo Park, CA, 700–705.

KORF, R. 2003a. Breadth-first frontier search with delayed duplicate detection. In *Proceedings of the IJCAI03 Workshop on Model Checking and Artificial Intelligence*. http://www.ijcai.org, 87–92.

KORF, R. 2003b. Delayed duplicate detection: Extended abstract. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*. http://www.ijcai.org, 1539–1541.

KORF, R. 2004. Best-first frontier search with delayed duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2004)*. AAAI Press, Menlo Park, CA, 650–657.

KORF, R. 2008. Minimizing disk I/O in two-bit breadth-first search. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-2008)*. AAAI Press, Menlo Park, CA.

KORF, R., AND FELNER, A. 2002. Disjoint pattern database heuristics. *Artif. Intell. 134*, 1-2 (Jan), 9–22.

KORF, R., AND FELNER, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*. http://www.ijcai.org, 2334–2329.

KORF, R., AND SHULTZE, P. 2005. Large-scale, parallel breadth-first search. In *Proceedings of the 20th AAAI Conference on Artificial Intelligence*. AAAI Press, Menlo Park, CA, 1380–1385.

KORF, R., ZHANG, W., THAYER, I., AND HOHWALD, H. 2005. Frontier search. *JACM 52,* 5 (Sept.), 715–748.

KUNKLE, D., AND COOPERMAN, G. 2007. Twenty-six moves suffice for Rubik's cube. In *Proceedings of the 32nd International Symposium on Symbolic and Algebraic Computation (ISSAC'07)*. ACM, New York, 235–242.

MEHLHORN, K., AND MEYER, U. 2002. External memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*. Lecture Notes in Computer Science, vol. 2461. Springer-Verlag, New York, 723–735.

MEYER, U. 2001. External memory bfs on undirected graphs with bounded degree. In *Proceedings of the 12th Annual Symposium on Discrete Algorithms*. ACM-SIAM, New York, 87–88.

MUNAGALA, K., AND RANADE, A. 1999. I/O complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM, New York, 687–694.

MYRVOLD, W., AND RUSKEY, F. 2001. Ranking and unranking permutations in linear time. *Inf. Proc. Lett. 79*, 281–284.

NICHOLS, B., BUTLER, D., AND FARRELL, J. 1996. *Pthreads Programming*. O'Reilly.

NIEWIADOMSKI, R., AMARAL, J., AND HOLTE, R. 2006. Sequential and parallel algorithms for frontier a* with delayed duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2006)*. AAAI Press, Menlo Park, CA, 1039–1044.

PATTERSON, D., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks. In *SIGMOD*. ACM, New York, 109–116.

PENNA, G. D., INTRIGILA, B., TONCI, E., AND ZILLI, M. V. 2002. Exploiting transition locality in the disk based mur(phi) verifier. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*. Lecture Notes in Computer Science, vol. 2517. Springer-Verlag, New York, 202–219.

ROSCOE, A. 1994. Model-checking csp. In *A Classical Mind, Essays in Honour of CAR Hoare*, A. Roscoe, Ed. Prentice-Hall, Englewood Cliffs, NJ.

STERN, U., AND DILL, D. 1998. Using magnetic disk instead of main memory in the mur(phi) verifier. In *Proceedings of the 10th International Conference on Computer-Aided Verification*. 172–183.

STEWART, B. 1941. Solution to advanced problem 3918. *Amer. Math. Monthly 48*, 217–219.

ZHOU, R., AND HANSEN, E. 2003. Sparse-memory graph search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*. http://www.ijcai.org, 1259–1266.

ZHOU, R., AND HANSEN, E. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2004)*. AAAI Press, Menlo Park, CA, 683–688.

ZHOU, R., AND HANSEN, E. 2005. External memory pattern databases using structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2005)*. AAAI Press, Menlo Park, CA, 1398–1404.

ZHOU, R., AND HANSEN, E. 2006a. Breadth-first heuristic search. *Artif. Intell. 170*, 4-5, 385–408.

ZHOU, R., AND HANSEN, E. 2006b. Domain-independent structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2006)*. AAAI Press, Menlo Park, CA, 1082–1087.

ZHOU, R., AND HANSEN, E. 2007a. Edge partitioning in external-memory graph search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-07)*. http://www.ijcai.org, 2410–2416.

ZHOU, R., AND HANSEN, E. 2007b. Parallel structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2007)*. AAAI Press, Menlo Park, CA, 1217–1223.

RECEIVED JANUARY 2008; REVISED JUNE 2008 AND OCTOBER 2008; ACCEPTED OCTOBER 2008