Best-First Frontier Search with Delayed Duplicate Detection

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

Best-first search is limited by the memory needed to store the Open and Closed lists, primarily to detect duplicate nodes. Magnetic disks provide vastly more storage, but random access of a disk is extremely slow. Instead of checking generated nodes immediately against existing nodes in a hash table, delayed duplicate detection (DDD) appends them to a file, then periodically removes the duplicate nodes using only sequential disk accesses. Frontier search saves storage in a best-first search by storing only the Open list and not the Closed list. The main contributions of this paper are to provide a scalable implementation of DDD, to combine it with frontier search, and to extend it to more general best-first searches such as A*. We illustrate these ideas by performing complete breadth-first searches of sliding-tile puzzles up to the 3x5 Fourteen Puzzle. For the 4-peg Towers of Hanoi problem, we perform complete searches with up to 20 disks, searching a space of over a trillion nodes, and discover a surprising anomaly concerning the problem-space diameter of the 15 and 20-disk problems. We also verify the presumed optimal solution lengths for up to 24 disks. In addition, we implement A* with DDD on the Fifteen Puzzle. Finally, we present a scalable implementation of DDD based on hashing rather than sorting.

Introduction

Best-First Search

Best-first search (BFS) is an algorithm schema that includes as special cases breadth-first search, uniform-cost search or Dijkstra's algorithm (Dijkstra 1959), and A* (Hart, Nilsson, & Raphael 1968). BFS maintains a Closed list of nodes that have been expanded, and an Open list of nodes that have been generated but not yet expanded. Each cycle of the algorithm chooses an Open node of lowest cost, expands it, generating and evaluating its children, and moves it to Closed. Each child node is checked to see if the same state already appears on the Open or Closed lists. If so, only a single copy reached via a shortest path from the start is saved.

If the cost of a node is its depth, then BFS becomes breadth-first search. If the cost of a node n is $g(n)$, the sum of the edge costs from the start to node n, then BFS becomes uniform-cost search or Dijkstra's algorithm (Dijkstra 1959).

If the cost of a node is $f(n) = g(n) + h(n)$, where $h(n)$ is a heuristic estimate of the cost from n to a goal, then BFS becomes A* (Hart, Nilsson, & Raphael 1968). Normally, BFS terminates when a goal node is expanded, but breadth-first search can terminate when a goal node is generated.

The Problem

BFS stores every state it generates in either the Open or the Closed list. For problems where node expansion is relatively efficient, such as the ones discussed in this paper, BFS typically exhausts the available memory in a matter of minutes, terminating the algorithm. In some problems, this limitation can be avoided by depth-first searches (DFS) such as depth-first branch-and-bound (DFBnB), depth-first iterative-deepening (DFID), or iterative-deepening-A* (IDA*)(Korf 1985), which require space that is only linear in the search depth. Since DFS can't detect most duplicate nodes, however, it can generate exponentially more nodes than BFS. For example, in a rectangular-grid problem space, breadth-first search generates $O(r^2)$ nodes within a radius of r, while DFS generates $O(3^r)$ nodes, since each node has four neighbors, one of which is its parent.

To detect all duplicates, we must store large numbers of nodes. Disks with hundreds of gigabytes cost less than a dollar per gigabyte, a hundred times cheaper than memory. Furthermore, a 32-bit machine can only directly address four gigabytes of memory, but unlimited disk space. Disks cannot directly replace memory, however, since disk latency is about 5 milliseconds, or 100,000 times longer than memory latency. A disk behaves more like a sequential device such as a magnetic tape, with large capacity and high bandwidth, but only if it is accessed sequentially. To quickly detect duplicate states, however, nodes are usually stored in a hash table, which is designed for random access. Virtual memory doesn't help, because of the random access pattern.

Prior Work

Delayed Duplicate Detection

A solution to this problem, which we call *delayed duplicate detection* (DDD), was developed by (Roscoe 1994) in the context of breadth-first search for automatic verification. Rather than checking for duplicates as soon as a node is generated, it is simply appended to the Open list. At the end of

each level of the search, the nodes on Open are sorted by their state representation, bringing duplicate nodes to adjacent positions, where they are eliminated. The sorted Open list is then merged with a sorted Closed list, deleting the closed nodes from Open, and producing a new sorted Closed list. Roscoe's implementation doesn't manipulate disk files, but works 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 20-disk 4-peg Towers of Hanoi problem, for example, the depth of a complete breadth-first search is 294 moves.

(Stern & Dill 1998) also implemented a breadth-first search with DDD in their model checker for hardware verification. They maintain a disk file of all states generated. When memory becomes full, the disk file of existing states is linearly scanned to eliminate duplicates from memory, and the new states remaining in memory are appended to the disk file, and to a separate queue of Open nodes. They report results on problems with just 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 complexity is worse if a complete level of the search won't fit in memory, since in that case, the complete Closed list is scanned every time memory becomes full, in addition to the end of each level of the search.

Frontier Search

Both of these schemes store all nodes generated in the search. Frontier search (Korf 1999; Korf & Zhang 2000) saves memory in any best-first search by storing only the Open list, and not the Closed list. In a breadth-first search, its space complexity is proportional to the maximum number of nodes at any depth, or the width of the problem space, rather than the total number of nodes in the space. In our experiments, these differed by as much as a factor of 46.

(Bode & Hinz 1999) implemented a form of breadth-first frontier search in memory for the 4-peg Towers of Hanoi, verifying the presumed optimal solution lengths for up to 17 disks. They save the nodes at the current depth, the prior depth, and the following depth. By contrast, our algorithm only stores two levels of the search at a time for the Towers of Hanoi, and one level for the sliding-tile puzzles.

Graph Algorithms using External Storage

There exists a body of work in the theory community on graph algorithms using external storage. See (Katriel & Meyer 2003) for a recent and comprehensive treatment of this area. That work is focussed on explicit graphs, where all nodes and edges are stored on disk, and analyzing the asymptotic complexity of the algorithms, without implementations or experimental results. By contrast, we are concerned with implicit graphs defined by a successor function, and the performance of practical implementations.

Overview

We first describe best-first search (BFS) with DDD in memory, and then consider using disk storage. We then show how to combine frontier search with DDD for breadth-first and uniform-cost search. The disk I/O complexity of our breadth-first frontier search with DDD scales linearly with the size of the state space. Unfortunately, DDD and frontier search can't be directly combined in A*. Using breadth-first frontier search with DDD, we compute the maximum optimal solution lengths for all sliding-tile puzzles up to the 3x5 Fourteen Puzzle, and confirm the presumed optimal solution lengths for the 4-peg Towers of Hanoi problem with up to 24 disks. By exhaustive search of the 4-peg Towers of Hanoi problems with up to 20 disks, we discover a surprising anomaly in the problem space diameter for 15 and 20 disks. The largest of these searches generate almost 4000 times more nodes than the previous state of the art. We solve a standard set of 100 random 15-puzzle instances using A* with DDD. Finally, we describe a scalable implementation of DDD based on hashing rather than sorting. Early versions of this work appeared in (Korf 2003b) and (Korf 2003a).

Delayed Duplicate Detection in Memory

At first we assume that all nodes fit in memory, and that the cost of a child is never less than the cost of its parent. Define $bestf$ as the lowest $f = g + h$ cost among all Open nodes at any given time. BFS will expand all Open nodes of cost $bestf$, then increment $bestf$ to the next higher cost of any Open node, expand those nodes, etc.

BFS with DDD stores open and closed nodes intermixed in a single node list. Each iteration of our algorithm linearly scans the node list, expanding all nodes of cost $bestf$, and skipping nodes of lower or higher cost. For example, an iteration of breadth-first search will expand all nodes at a given depth. When a node is expanded, its children are generated, evaluated, and appended to the end of the node list, without checking for duplicates. If a child node has the same cost as its parent, the child will be appended to the end of the list, and expanded during the same iteration as the parent, when the scan reaches the child node.

Normally, duplicate detection is performed after each iteration. We sort the nodes by their state representation, using quicksort, since it sorts in place without additional memory. Sorting brings to adjacent locations any duplicate nodes representing the same state. Then, in a linear scan of the sorted node list, we remove all duplicate nodes, saving one copy of minimum g value. During this scan, we also reset $bestf$ to the lowest cost of those nodes not yet expanded. We then start a new iteration, with the new value of $bestf$. Alternatively, duplicate elimination could be performed whenever memory becomes full.

If there are many unique node values, then a single iteration may expand all open nodes in a range of cost values. An appropriate range can be determined during the preceding duplicate scan. In that case, we terminate when we complete the first iteration that chooses a goal node for expansion.

If our cost function is non-monotonic, meaning that a child may have a lower cost than its parent, then we mark

open nodes of cost less than $bestf$, so they are expanded when they are reached during the same iteration. Alternatively, after expanding a node, we can recursively expand all its descendents of cost less than or equal to $bestf$, appending them all to the end of the node list.

Recovering the Solution Path

Once we find an optimal solution, we may want the actual solution path as well. Normally, this is done by storing with each node a pointer to its parent, and then following these pointers from the goal state back to the start state. Sorting the nodes moves them in memory, however, invalidating the pointers. Instead, with each node we store the operator that generated it from its parent, along a lowest-cost path to the node. Alternatively, if operators are invertible, we could store the operator that generates the parent from the child. Once we find a solution, we sort the node list again, but in decreasing order of g value, rather than by state representation. We then scan the sorted list from the beginning, until we reach the goal node. We generate its parent state using the stored operator, and continue scanning the node list from the same point until we reach the parent node. When we reach the parent, we generate the grandparent state, etc, continuing until we reach the start state. If every edge has a positive cost, a parent node will always follow its children in the resorted node list, and hence we can reconstruct the entire solution path in one scan of the node list.

Advantages of DDD in Memory

Even when storing all nodes in memory, there are several advantages of BFS with DDD over standard BFS. One is that DDD requires less memory per node, since it doesn't need pointers for a hash table or Open list. For the Fifteen Puzzle, for example, BFS with DDD needs less than half the memory per node of standard BFS.

A second advantage of BFS with DDD is that it may run faster than standard BFS, due to improved cache performance. A hash table spreads nodes uniformly over memory. By contrast, BFS with DDD scans the node list linearly, appending nodes to the end. Quicksort only accesses nodes at both ends of a list, and the purge phase linearly scans the sorted node list. Each of these steps exhibit good locality.

BFS with DDD using Disk Storage

The main advantage of DDD, however, is that it allows us to store the node list on disk. We first give an overview of the algorithm. Think of the node list as a single file. We read this file sequentially, expanding nodes of cost $bestf$, and appending the children to the end of the file. When we've read the entire file, including nodes added during the current iteration, we sort it. Algorithms for sorting disk files are well-known (Garcia-Molina, Ullman, & Widom 2000). Finally, we sequentially scan the sorted file to remove duplicate nodes, and begin a new iteration.

To implement this efficiently, we start with a sorted node list in an input file, initially containing only the start node. We read this file sequentially, expanding those nodes of cost $bestf$. If a child has cost $bestf$ or less, it is recursively expanded, as are grandchildren of the same or lower cost, etc. In any case, all newly generated nodes are appended to a buffer in memory. When this buffer is full, the nodes are sorted by their state representation, duplicate nodes are deleted, saving one copy of lowest g value, and the remaining nodes are written to an output file. We continue reading the input file, producing multiple output files. Each of these are sorted, and contain no duplicates within them, but duplicates will appear across different files.

We then merge all the sorted output files, together with the sorted input file, into one sorted file without duplicates, in a single multi-way merge. We initialize a heap with the first node in each file. At each cycle, the top node of the heap is removed, written to a new output file, and replaced in the heap by the next node from the file that it came from. During this process, duplicate nodes come together at the top of the heap, and are merged at that time, keeping a copy of lowest g value. This process continues until all the output files, and the input file, have been exhausted. Then the next iteration of the search begins with the new output file as the input file. The virtue of a multi-way merge is that each node is only read once, and only one copy of each node is written once, in each sort-merge phase.

The algorithm alternates expansion and sort-merge phases, until an optimal solution is found. If we want the solution path itself, then the nodes are resorted into a single file in decreasing order of g value. This file is then scanned linearly, looking for the goal node, then its parent, grandparent, etc, until the start node is reached.

The sort-merge algorithm described above could require up to twice as much disk space as necessary. When we read a node into memory, we no longer need it on disk, since it is stored until it is written to the new input file. However, most file systems don't allow deleting elements from a file as they are read. If the nodes in each output file and the old input file are uniformly distributed over the state space, we may not be able to delete any of these files until the entire sort-merge phase is almost complete, at which point all of these nodes, minus their duplicates, will have been written to the new input file. To avoid this problem, each "file" described above is actually broken up into a sequence of smaller file segments, so that we can delete an entire file segment as soon as it is read into memory.

The I/O complexity of this algorithm is on the order of the size of the search space times the number of expansion iterations. For a breadth-first search, this is no worse than the size of the space times its depth. It can be reduced by performing the initial expansion iterations without sorting and merging, until memory is full.

Frontier Search

Frontier search (Korf 1999; Korf & Zhang 2000) is an independent way of implementing best-first search that saves memory by storing only the Open list and not the Closed list. We first describe the algorithm, and then show how to combine it with DDD. We assume here that all operators are invertible, i.e. the problem-space graph is undirected, but the algorithm applies to directed graphs as well.

Frontier search stores only the Open list, representing the frontier or boundary of the search, and deletes a node once it is expanded. The trick is to keep the search from "leaking" back into the expanded or interior region of the space. With each node, we store a *used* bit for each operator, to indicate whether the neighboring state reached by that operator has been expanded yet. For example, in the sliding-tile puzzles we need four bits for each state, one for each direction in which a tile could move. For the Towers of Hanoi, we need one bit for each peg, indicating the location of the last disk moved. When we expand a parent node, we only generate those children that are reached via unused operators, and delete the parent node from memory. In each child node, the operator that generates its parent is marked as used. We also look up each child state in the Open list. If a duplicate state is found, only the copy reached by a shortest path is kept, and any operator marked as used in any copy is marked as used in the retained copy. Since the Closed list is not stored, reconstructing the solution path requires some additional work, described in (Korf 1999; Korf & Zhang 2000).

The memory required by frontier search is proportional to the maximum size of the Open list, or the *width* of the search, rather than the size of the entire space. For example, the width of a rectangular grid space grows only linearly with the search radius, while the entire space grows quadratically. As another example, the maximum number of states at any depth of the n-disk, 3-peg Towers of Hanoi space is only 2^n, while the entire space contains 3^n states.

Combining Frontier Search with DDD

Since frontier search saves memory in best-first search, and DDD allows disk storage to be used, to maximize the size of problems we can solve we'd like to combine the two algorithms. While this combination is straightforward for breadth-first search and uniform-cost search, they cannot be easily combined in the case of A*. We begin with breadth-first frontier search with DDD.

Breadth-First Frontier Search with DDD

In breadth-first frontier search with DDD, we merge duplicates at the end of each iteration, when all nodes at a given depth have been expanded. In the simplest case, the input file contains all nodes at depth d, and the output files contain nodes at depth $d+1$, since the used operator bits prevent generating nodes at depth $d-1$. We delete the input file when it's exhausted, and sort and merge the output files together.

Consider the square graph at left in Figure 1, where node A is the start node. Node A is expanded and then deleted, generating node B with operator w marked as used, and node C, with operator x marked as used. In the next iteration, node B is expanded and deleted, generating node D with operator y marked as used, and node C is expanded and deleted, generating another copy of node D with operator z marked as used. In the sort-merge phase, these two copies of node D are merged into a single copy, with operators y and z both marked as used. Since node D has no more neighbors, the search would terminate at this point.

The I/O complexity of this algorithm is linear in the size of the search space. The first time a node is generated, it is stored until memory is full, then sorted and merged with any duplicates in memory, and written to a disk file. Once all the nodes at the current depth are expanded, a merge phase begins which will read the node back into memory, merge it with any duplicates from other files, and write it out to a single file. Finally, in the next expansion phase, it will be read back into memory, expanded and deleted. Thus each node is read twice and written twice. A given state can be represented by as many duplicate nodes as the branching factor of the problem, one from each of its neighbors. Since we assume a constant branching factor, the I/O complexity is still linear in the number of states in the problem space.

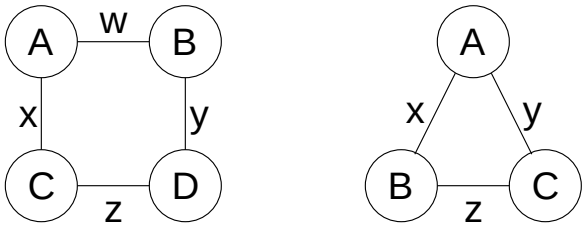

Figure 1: Example Graphs

This algorithm works fine in a graph where the lengths of all paths between any pair of nodes have the same parity. For example, for any pair of nodes in a sliding-tile puzzle, either all paths between them have odd length, or all have even length. In a triangle graph, however, such as that shown at right in Figure 1, there is a path of length one and of length two between any pair of nodes. The Towers of Hanoi problem space graphs contain many such triangles.

Consider our simple algorithm on the triangle in Figure 1, assuming that A is the start node. 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 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 if node A had any other neighbors, it would be reexpanded.

This problem does not occur without DDD, since as soon as the second copy of node C is generated, it would be immediately merged with the first copy, with both operators y and z marked as used. When we then tried to expand node C, the search would terminate.

To fix this problem, when expanding nodes at depth d, instead of deleting them, we write them to their own output

file. At the end of the iteration, we sort and purge the nodes at depth d together with the nodes at depth $d+1$, creating a single new input file with nodes at both depths. In the next iteration, as we scan the new input file, we expand and save the nodes at depth $d+1$, but delete the nodes at depth d. Essentially, we save two levels of the search at a time, rather than just one level, which is similar to a version of frontier search developed by (Zhou & Hansen 2003). This roughly doubles the amount of storage needed, but the I/O complexity is still linear in the size of the search space, since each node is now read three times and written twice.

Frontier-A* with DDD

Even though both frontier search and DDD apply to A* individually, there is no obvious way to combine both methods with A*, without the search "leaking" back into the interior. In the example below, we adopt the constraints of the sliding-tile puzzles with the Manhattan distance heuristic. In particular, all edges have unit cost, all paths between any pair of nodes have the same parity, and the heuristic value of a child is either one greater or one less than that of its parent.

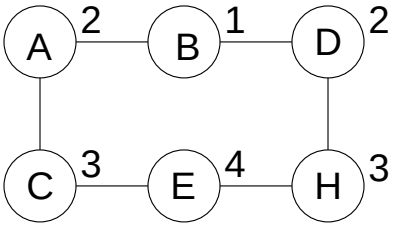

Figure 2: Failure of Frontier A* with DDD

Consider the graph in Figure 2, where node A is the start state, and the numbers denote the heuristic values of the states. For the first iteration, $bestf = h(A) = 2$. Node A is expanded, generating nodes B and C, with $f(B) = 1+1 = 2$ and $f(C) = 1+3 = 4$. Since $f(B) = 2$, it is also expanded in the first iteration, generating node D with $f(D) = 2+2 = 4$. The next iteration, with $bestf = 4$, expands nodes C and D, generating nodes E and H with $f(E) = 2+4 = 6$ and $f(H) = 3+3 = 6$. In the next iteration, with $bestf = 6$, node E is expanded, generating another copy of node H with $f(H) = 3+3 = 6$. This duplicate node is not immediately detected, since we are in the middle of an iteration. The new copy of node H is expanded in the same iteration, generating another copy of node D, with $f(D) = 4+2 = 6$. This copy of node D is also expanded in this iteration, generating another copy of node B with $f(B) = 5+1 = 6$, which is also expanded in this iteration, generating another copy of node A.

This last iteration reexpanded nodes that were first expanded several iterations previously. By extending the length of the cycle, we can construct examples where frontier-A* with DDD would "leak" arbitrarily far into the interior of the search. Thus, when applying DDD to A*, we cannot use frontier search, but have to store all closed nodes. This is not a major drawback, however, in problems where the number of nodes grows exponentially with cost, as in the sliding-tile puzzles with the Manhattan distance heuristic, for example. In such cases, the size of the closed list is often small compared to the size of the open list.

Uniform-Cost Frontier Search with DDD

Fortunately, we can combine frontier search with DDD in uniform-cost search or Dijkstra's algorithm, if all edges have positive cost, and a sort-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.

Experiments

We implemented breadth-first frontier search with DDD on sliding-tile puzzles and the Towers of Hanoi, and A* with DDD, but without frontier search, on the Fifteen Puzzle. In addition to finding optimal solutions and determining the diameter of a problem space, breadth-first search is also used in hardware and software verification, and for computing pattern database heuristics (Culberson & Schaeffer 1998).

Complete Searches of Sliding-Tile Puzzles

The sliding-tile puzzle consists of a rectangular frame filled with square tiles except for one blank position. Any tile horizontally or vertically adjacent to the blank can be slid into that position. The task is to rearrange the tiles from some initial configuration to a particular goal configuration.

(Schofield 1967) first performed a complete breadth-first of the Eight Puzzle. We've extended this to all sliding-tile puzzles up to the 3×5 Fourteen Puzzle. The first column of Table 1 gives the x and y dimensions of the puzzle, and the second column shows the number of moves needed to reach all solvable states, which is the worst-case optimal solution length. The blank started in the center for the 3×5 puzzle, and in a corner in the other cases. The third column gives the number of solvable states, which is $(xy)!/2$, and the fourth column gives the width of the problem space, which is the maximum number of nodes at any depth. The horizontal line represents the previous state of the art.

Size	Moves	Total States	Width of Space
2×2	6	12	2
2×3	21	360	44
2×4	37	20,160	1,999
3×3	31	181,440	24,047
2×5	55	1,814,400	133,107
2×6	80	239,500,800	13,002,649
3×4	53	239,500,800	21,841,159
2×7	108	43,589,145,600	1,862,320,864
3×5	84	653,837,184,000	45,136,428,198

Table 1: Complete Searches of Sliding-Tile Puzzles

Previously, the most efficient way to implement a complete breadth-first search was to allocate an array with one bit per state. A bit is set if and only if the corresponding state has already been generated. We implement the breadth-first

search with a FIFO queue of nodes, either in memory or on disk, and use the bit array to reject duplicate nodes. We refer to this as a bit-state breadth-first search.

The 3×4 and 2×6 Eleven Puzzles were the largest we could search completely without DDD. We implemented the bit-state breadth-first search, a breadth-first frontier search, and a breadth-first frontier search with DDD. Bit-state breadth-first search required 15.25 minutes, breadth-first frontier search required 19.75 minutes, and breadth-first frontier search with DDD required 9.25 minutes, on a 440 Megahertz Sun Ultra 10 workstation. This shows that DDD can be useful even for problems that fit in memory.

To solve the 3×5 Fourteen Puzzle, we used symmetry by placing the blank at the center of the initial state, and rotating every state 180 degrees. This reduced the actual storage requirements by almost a factor of two. At 8 bytes per state, this problem required over 200 gigabytes of disk storage, and ran for almost 18 days on the above machine. Using this same symmetry, the bit-state breadth-first search would require over 38 gigabytes of memory for the bit array, plus over 168 gigabytes of disk space for the FIFO queue, since it must store a complete state representation for half the nodes in the width of the space. Frontier search without DDD would require over 168 gigabytes of memory.

Linear-space depth-first searches (DFS), which explore the problem-space tree, become increasingly inefficient with depth, since they can't detect most duplicate nodes. For example, in the Fourteen Puzzle graph, the number of states at each depth increases up to depth 51, then decreases to a single state at depth 84. The tree, however, continues to grow exponentially with a branching factor of just over two. Thus, a DFS to depth 84 would generate over 10^{25} nodes. Furthermore, DFS cannot detect the maximum search depth in a problem space.

Complete Searches of 4-Peg Towers of Hanoi

The Towers of Hanoi problem consist of at least three pegs, and a set of different-size disks initially stacked on one peg. The task is to move all disks from the initial peg to a goal peg, subject to the constraints that only the top disk on any peg can move at one time, and a larger disk cannot be stacked on top of a smaller disk. For the well-known 3-peg Towers of Hanoi puzzle, there is a simple algorithm that provably generates a shortest solution, which is of length 2^{n-1} moves, where nn is the number of disks.

The 4-peg Towers of Hanoi problem (Hinz 1999), is much more interesting. There is a deterministic algorithm to move all disks from one peg to another, and a conjecture that it is optimal (Frame 1941; Stewart 1941; Dunkel 1941), but the conjecture is unproven. Thus, systematic search is the only method guaranteed to find optimal solutions, or to verify the conjecture for a given number of disks.

We begin with exhaustive searches of these problem spaces, starting with all disks stacked on one peg. The three remaining pegs are symmetric. Given a particular state, we can generate up to five more equivalent states by permuting these three pegs. We represent each state by a canonical state in which the non-initial pegs are sorted by the size of their largest disk. This contributes almost a factor of six in performance (Bode & Hinz 1999).

Using this symmetry, we implemented breadth-first frontier search with DDD. Since the problem space contains path lengths of unequal parity between pairs of states, we save two levels at a time. We exhaustively searched all problems with up to 20 disks, and the results are shown in Table 2.

The first column gives the number of disks n, and the second column gives the minimum depth by which all legal states are generated, starting with all disks on one peg. The third column gives the number of states in the space, which is 4^n. The last column gives the width of the space, or the maximum number of states at any depth. The actual numbers of nodes expanded are approximately one-sixth the numbers shown, due to symmetry. The 20-disk problem took over 18 days, using the algorithm described above.

The line under the 16-disk problem represents the state-of-the-art without DDD. In particular, using the bit-state breadth-first search described above, a 16-disk bit array requires $4^{16}/8$ or half a gigabyte of memory. The FIFO queue is stored on disk.

Disks	Depth	States	Width
1	1	4	3
2	3	16	6
3	5	64	30
4	9	256	72
5	13	1024	282
6	17	4,096	918
7	25	16,384	2,568
8	33	65,536	9,060
9	41	262,144	31,638
10	49	1,048,576	109,890
11	65	4,194,304	335,292
12	81	16,777,216	1,174,230
13	97	67,108,864	4,145,196
14	113	268,435,456	14,368,482
15	130	1,073,741,824	48,286,104
16	161	4,294,967,296	162,989,898
17	193	17,179,869,184	572,584,122
18	225	68,719,476,736	1,994,549,634
19	257	274,877,906,944	6,948,258,804
20	294	1,099,511,627,776	23,513,260,170

Table 2: Complete Searches of 4-Peg Towers of Hanoi

Half-Depth Searches of 4-Peg Towers of Hanoi

In the standard problem of moving all disks from one peg to another, the initial and goal states are symmetric. To solve this problem, we must move the largest disk from the initial peg to the goal peg at some point, which requires distributing the remaining disks among the other two pegs. Once we find such a state, if we apply the same moves used to reach that state in the opposite order, but interchange the initial and goal pegs, we will complete the solution. Thus, we only have to search to half the solution depth, to the first state that distributes all but the largest disk on the two intermediate pegs (Bode & Hinz 1999). Using this symmetry pre-

cludes using most heuristic functions, since we don't know the actual distribution of the disks on the intermediate pegs.

Using both symmetries, we implemented breadth-first frontier search with DDD, and verified the presumed optimal solutions for all problems with up to 24 disks. The first column of Table 3 gives the number of disks, and the second column shows the optimal solution length. The third column gives the total number of states up to half the optimal solution depth, rounded up. The fourth column gives the number of states at that depth. Here we show the actual numbers of states generated, taking advantage of the problem symmetries. The 24-disk problem took almost 19 days to solve with a gigabyte of memory, and used almost 130 gigabytes of disk space. The largest problem that had been solved previously is the 17-disk problem (Bode & Hinz 1999), indicated by the horizontal line in the table. Concurrently with this work, (Felner *et al.* 2004) solved the 18-disk problem using frontier-A* with a pattern-database heuristic. Their algorithm applies to arbitrary initial and goal states.

Disks	Moves	States	Width
1	1	1	1
2	3	3	1
3	5	6	3
4	9	18	7
5	13	45	16
6	17	105	36
7	25	434	137
8	33	1,244	293
9	41	3,335	562
10	49	8,469	1,780
11	65	39,631	6,640
12	81	139,624	21,187
13	97	371,053	50,777
14	113	933,286	86,944
15	129	2,407,813	256,579
16	161	11,682,622	965,931
17	193	45,800,496	3,278,853
18	225	145,656,379	10,004,107
19	257	379,925,343	24,317,501
20	289	957,046,489	45,544,033
21	321	2,394,213,892	133,985,287
22	385	12,024,012,134	547,563,876
23	449	50,661,772,967	2,006,448,775
24	513	178,693,859,657	6,364,114,392

Table 3: Half-Depth Searches of 4-Peg Towers of Hanoi

The branching factor of the problem-space tree for this problem is about 3.766, assuming that we never move the same disk twice in a row. To find an optimal solution to the 9-disk problem, we would have to search to depth 20. $3.766^{20} \approx 3.29 \times 10^{11}$. Thus, the 9-disk problem is the largest that could be solved with a depth-first search, due to the proliferation of undetected duplicate nodes.

A Very Surprising Anomaly

An incredibly astute reader might notice that in almost every case, the maximum search depth from a state with all disks on a single peg, shown in Table 2, is the same as the optimal number of moves needed to transfer all disks to another peg, shown in Table 3. For 15 disks, however, the optimal solution length is 129 moves, but there are 588 states at depth 130, starting with all disks on a single peg. For 20 disks, there are a total of $11,243,652$ states beyond the optimal solution length of 289 moves, up to a depth of 294. This result for 15 disks has been confirmed by about a half-dozen different implementations of breadth-first search, and the result for 20 disks has been confirmed by two different implementations. We have no explanation for this surprising anomaly, and leave it for future work.

A* with DDD on the Fifteen Puzzle

We also implemented A* with DDD on the Fifteen Puzzle, with the Manhattan distance heuristic, but without frontier search. We used the set of 100 random solvable problem instances in (Korf 1985). With a gigabyte of memory, standard A* can store 35 million nodes, and can solve 83 of these problems. Frontier-A* without DDD uses the same amount of memory per node as standard A*, but only stores about half as many nodes. It can solve 93 of the 100 problems and runs about 38% slower than standard A*. A* with DDD can store 70 million nodes in the same amount of memory, and can solve 87 of these problems, but runs about three times slower than standard A*. Using disk storage, A* with DDD easily solves all the problems, and runs about five times slower than standard A*. To our knowledge, this is first time all these problems have been solved by A* with less than a gigabyte of memory. IDA* (Korf 1985) is still the preferred algorithm for this problem, however, since it uses only linear memory, and is very simple to implement.

Duplicate Elimination Without Sorting

A large fraction of the CPU time of these algorithms is due to sorting. Sorting is not necessary to eliminate duplicate nodes, however, as shown by (Stern & Dill 1998). Here we present a duplicate elimination algorithm, based on hashing, which scales linearly with problem size.

The algorithm uses two hash functions that ideally are orthogonal, meaning that they hash most nodes to different values. As we expand each node in a breadth-first frontier search, we apply the first hash function to its children, and append each child to an output file dedicated to nodes with the same hash value. At the end of the expansion phase, any duplicate nodes occur only within the same file.

In the second phase, we iterate through each of the output files. For each file, we hash each node to a location in memory using the second hash function. Any duplicate nodes will hash to the same location and are merged. As we compact each file, we write its unique nodes to a new output file of nodes at the next depth. The next expansion phase then iterates through each of these files.

We implemented this method for the 4-peg Towers of Hanoi problem, and repeated the complete search experiments shown in Table 2. For this problem, we maintain two sets of files, one for the nodes at the current depth, and the other for the nodes at the next depth. For the second hash

function we used the positions of the 14 smallest disks, and for the first hash function we used the positions of the larger disks. Thus, each file contains only states with the same configuration of the large disks. An advantage of this approach is that we never have to explicitly store the positions of the large disks, since it is uniquely determined by the file containing the states, and only have to store the positions of the 14 smallest disks, plus the used-operator bits.

The CPU time of our hashing-based implementation is less than half that of our sorting-based implementation. This is also true of total elapsed time for up to 17 disks, since only 15 pairs of files are needed, due to symmetry. For larger numbers of disks, however, the elapsed time is more than half the time of our sorting-based implementation. This appears to be due to reading and writing a large number of files, many of which contain relatively few nodes. For 20 disks, our current hashing-based implementation takes 11 days to run, compared to over 18 days for the sorting-based implementation, a speedup of over 41%.

Summary and Conclusions

We presented an implementation of delayed duplicate detection (DDD) that scales up to large problems. For breadth-first search and uniform-cost search, we showed how to combine DDD with frontier search, which saves only the Open list, but DDD cannot be easily combined with frontier search for A*. We implemented breadth-first frontier search with DDD on sliding-tile puzzles and the 4-peg Towers of Hanoi. We performed complete breadth-first searches on all sliding-tile problems up to the 3×5 Fourteen Puzzle. We also performed complete searches of the 4-peg Towers of Hanoi problem with up to 20 disks, discovering a surprising anomaly in the 15 and 20-disk problems. In addition, we verified the presumed optimal solution lengths for the 4-peg Towers of Hanoi problems with up to 24 disks. These searches expand over three orders of magnitude more nodes than the previous state of the art. We also implemented A* with DDD, but without frontier search, solving all of a standard set of 100 Fifteen Puzzle instances. Finally, we presented an implementation of DDD that relies on hashing rather than sorting, which runs over 40% faster than our sorting-based implementation.

Acknowledgments

This research was supported by NSF under grant No. EIA-0113313, by NASA and JPL under contract No. 1229784, and by the State of California MICRO grant No. 01-044. Thanks to Jianming He for independently verifying the 15-disk anomaly, and to Ariel Felner for helpful discussions concerning this research.

References

Bode, J.-P., and Hinz, A. 1999. Results and open problems on the tower of hanoi. In *Proceedings of the Thirtieth Southeastern International Conference on Combinatoris, Graph Theory, and Computing*.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dijkstra, E. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Dunkel, O. 1941. Editorial noe concerning advanced problem 3918. *American Mathematical Monthly* 48:219.

Felner, A.; Meshulam, R.; Holte, R.; and Korf, R. 2004. Compressing pattern databases. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-04)*, 1184–1189.

Frame, J. 1941. Solution to advanced problem 3918. *American Mathematical Monthly* 48:216–217.

Garcia-Molina, H.; Ullman, J. D.; and Widom, J. 2000. *Database System Implementation*. Upper Saddle River, N.J.: Prentice-Hall.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Hinz, A. M. 1999. The tower of hanoi. In *Algebras and Combinatorics: Proceedings of ICAC'97*, 277–289. Hong Kong: Springer-Verlag.

Katriel, I., and Meyer, U. 2003. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies, LNCS 2625*. Springer-Verlag. 62–84.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2000)*, 910–916.

Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Korf, R. 1999. Divide-and-conquer bidirectional search: First results. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1184–1189.

Korf, R. 2003a. Breadth-first frontier search with delayed duplicate detection. In *Proceedings of the IJCAI03 Workshop on Model Checking and Artificial Intelligence*, 87–92.

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

Roscoe, A. 1994. Model-checking csp. In Roscoe, A., ed., *A Classical Mind, Essays in Honour of CAR Hoare*. Prentice-Hall.

Schofield, P. 1967. Complete solution of the eight puzzle. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 3*. New York: American Elsevier. 125–133.

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. *American Mathematical 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)*, 1259–1266.