Comparing Search Algorithms Using Sorting and Hashing on Disk and in Memory

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

Abstract

We compare sorting and hashing for implicit graph search using disk storage. We first describe efficient pipelined implementations of both algorithms, which reduce disk I/O. We then compare the two algorithms and find that hashing is faster, but that sorting requires less disk storage. We also compare disk-based with in-memory search, and surprisingly find that there is little or no time overhead associated with disk-based search. We present experimental results on the sliding-tile puzzles, Rubik's Cube, and the 4-peg Towers of Hanoi.

1 Introduction and Overview

Best-first graph searches, such as breadth-first and A* [Hart, Nilsson, and Raphael, 1968], store all generated nodes. As a result, they are memory-limited, often exhausting available memory in minutes. External storage offers much higher capacity at much lower cost than random-access memory. For example, magnetic disks with up to six terabytes cost about 50 dollars per terabyte. Several existing graph-search algorithms store nodes on disk, allowing them to run for days or weeks at a time. We use magnetic disks here, but since our interface to the disk is a standard file system, all these techniques apply to solid state drives as well, which are intermediate in cost between memory and magnetic disk.

Storage is used primarily to detect duplicate nodes, which represent the same state reached via different paths. This can be done with a hash table in memory, but a hash table is impractical on disk, since random access can take up to ten milliseconds. The fundamental problem is how to implement search on disk with only sequential access. For simplicity, we use breadth-first search here, but these techniques also apply to heuristic search, as we will explain.

We focus here on delayed duplicate detection (DDD), the most general disk-based search paradigm. Within this paradigm, there are two fundamentally different approaches. The most common one sorts a set of nodes by their state representation, bringing duplicate nodes to adjacent positions, allowing them to be merged in a linear scan. Alternatively, hashing has also been used to detect duplicate nodes. With one exception noted below, we are unaware of any experimental comparison of these two methods.

First, we briefly describe several approaches to disk-based search. We then describe DDD using both sorting and hashing in detail, and apply the general idea of pipelining to hash-based algorithms to reduce disk I/O. Previously, it has only been applied to sorting-based algorithms. We then show that hashing is faster than sorting. On the other hand, we show that sorting uses significantly less disk storage than hashing. Finally, we show that there is little or no time overhead associated with disk-based search, compared to in-memory search. We found this result surprising.

2 General Approaches to Disk-Based Search

Previous approaches to disk-based search include explicit graph search, two and four-bit breadth-first search, structured duplicate detection, and delayed duplicate detection (DDD).

2.1 Explicit vs. Implicit Graph Search

There is a significant literature on external-memory search of explicit graphs, where the entire graph is stored on disk (e.g. [Ajwani, Demetiev, and Meyer, 2006]). By contrast, we focus on combinatorial search, where the graph is defined implicitly, and adjacent nodes are generated by a successor function.

2.2 Two and Four-Bit Breadth-First Search

These algorithms represent the complete problem space by a single array on disk, using two bits [Robinson, Kunkle, and Cooperman, 2007; Kunkle and Cooperman, 2007; Korf, 2008b], or four bits [Sturtevant and Rutherford, 2013] per state. They require a one-to-one function from states to array indices, and enough space to store the entire problem space. They are not applicable to searches of larger spaces, and use much more space than frontier search (see below).

2.3 Structured Duplicate Detection

In structured duplicate detection (SDD) [Zhou and Hansen, 2004; 2006b; 2007a; 2007b], nodes are stored in separate files based on common state features. The features are chosen so that when expanding nodes in a given file, the children are contained in a small number of other files, which are temporarily stored in memory. SDD requires this structure in the problem space, and is not applicable to problem spaces without this structure, such as Rubik's Cube, for example.

2.4 Delayed Duplicate Detection (DDD)

In delayed duplicate detection (DDD), [Korf, 2004; Korf and Schultze, 2005; Korf, 2008a] duplicate checking is not done as each node is generated, but only periodically, such as after expanding all nodes of a given cost. We focus exclusively on DDD here, since it is the most general approach.

2.5 Frontier Search

Frontier search [Korf et al., 2005] can reduce the storage needed by orders of magnitude in some spaces. Rather than storing all generated nodes, only the nodes in one or two levels of the search are stored at any time. Generating a solution path requires only a small amount of additional work.

Frontier search can be implemented on undirected graphs by storing with each node "used operator bits" indicating the operators used to generate that node. When expanding a node, we don't apply the inverses of the used operators, preventing the generation of the parent of a node as one of its children.

3 Sorting-Based Delayed Duplicate Detection

We first consider the special case of problem spaces where all cycles have even length, such as the sliding-tile puzzles. Since the blank cannot move diagonally, to return to the same position it must move an even number of times. Thus, any set of duplicate nodes must be at the same depth, or separated by an even number of levels.

3.1 Problem Spaces with Only Even-Length Cycles

We begin each iteration with a single file of unique nodes at depth d. We read this file into memory in chunks, expand each node, and write the child nodes at depth $d+1$ to another file, without any duplicate checking. This is called the expansion phase. We then sort the file of child nodes by their state representation, bringing duplicate nodes to adjacent positions. Finally, we linearly scan the sorted file, merging adjacent duplicate nodes together, and unioning their used-operator bits. These unique nodes are written to a separate file at depth $d+1$. This is called the merge phase.

To sort a file of generated nodes, we can read in memory-size chunks, sort each chunk, and then write out the sorted chunks. Then we read all the sorted chunks in parallel, perform a multi-way merge using a heap in memory, and write out a file of unique nodes. For efficiency, chunks of generated nodes are sorted and merged during the expansion phase, before writing them to files to be merged in the merge phase.

3.2 Pipelining

Each iteration above has an expansion phase that writes sorted files of generated nodes, followed by a multi-way merge of the sorted files. During the merge phase, a sorted file of unique nodes is written to disk, and then read right back into memory in the next expansion phase. This is inefficient.

The solution is an idea called pipelining, which has been used in explicit [Ajwani, Demetiev, and Meyer, 2006] and implicit [Jabbar, 2008] sorting-based disk search, but not to our knowledge in hash-based disk algorithms. The alternating expansion and merge phases are combined into one phase, reducing disk I/O. Each iteration starts with a set of files of sorted nodes at depth d, including duplicates across files. These files are read in parallel, duplicates are merged in a multi-way merge, and each unique node is immediately expanded to generate its children at depth $d+1$. The child nodes are buffered in memory, the buffer is sorted and merged, and then written to disk. This results in a set of files of sorted nodes at depth $d+1$.

3.3 Breaking Single Files into Multiple Segments

Unfortunately, this algorithm can double the disk space required. In the multi-way merge, the files are read in parallel, and a file cannot be deleted until it has been read completely. Thus, in the worst case, we cannot delete any of the input files until the multi-way merge is complete, and all the files of nodes at the next depth have already been written out. This may require enough space for the largest two consecutive levels of the search, rather than just the largest level.

The solution is to write each file as a sequence of separate file segments. Then, as each segment is read into memory, it is deleted from disk. This reduces the maximum storage to approximately that required for the largest search level.

3.4 Problem Spaces with Arbitrary-Length Cycles

In the general case of problem spaces with both odd and even-length cycles, such as Rubik's Cube or the Towers of Hanoi, duplicate nodes can appear at consecutive depths, as can be seen from a triangle of three mutually adjacent nodes. Thus, in each iteration, we write to disk the unique nodes at depth d, and when merging nodes at depth $d+1$, we read the file of nodes at depth d, and delete any that also appear at depth $d+1$, using separate file segments as described above. This requires storing two levels of the search at one time.

3.5 Sorting and Merging Algorithm

Most of the time of this algorithm is spent sorting, so we optimized the sort. The fastest algorithm was our implementation of a hybrid between merge and insertion sort. At the highest levels, merge sort was used, while for 30 or fewer nodes, we used insertion sort. Merging duplicate nodes was done only during the top-level merge.

4 Hash-Based Delayed Duplicate Detection

An alternative to sorting to detect duplicate nodes is hashing [Korf, 2008a]. This relies on two orthogonal hash functions. The file hash function hashes states to different files, while the memory hash function hashes the states in a given file into memory locations. For simplicity, we begin with only even-length cycles and separate expansion and merge phases, then consider pipelining and finally arbitrary-length cycles.

4.1 Problem Spaces with only Even-Length Cycles

In the expansion phase, each generated node is hashed to a file using the file hash function, hashing duplicate nodes to the same file. In the merge phase, each file is processed separately. Each node is hashed to a location in memory, using the memory hash function. Duplicate nodes hash to the same location, where they are merged. The resulting unique nodes from the file are then written to a separate file. Each file must be small enough that its unique nodes fit into memory.

4.2 Pipelining

As with sorting, we can use pipelining to reduce the amount of disk I/O. Rather than separate expansion and merge phases, there is a single phase for each iteration. It starts with a set of generated nodes, including duplicates, in a set of files at depth d. After the nodes in a given file are hashed into memory, the hash table is scanned, the unique nodes are expanded, and their children are hashed to a set of files at depth $d + 1$.

4.3 Problem Spaces with Arbitrary-Length Cycles

In problem spaces with arbitrary-length cycles, the unique nodes at depth d are written to a file as they are expanded. Then, after hashing into memory the generated nodes at depth $d + 1$ in the next iteration, the corresponding file of unique nodes at depth d with the same file hash value is read, and those nodes are deleted from the hash table before expanding the unique nodes at depth $d + 1$.

4.4 Important Implementation Decisions

There are several choices that must be made in any hash-based disk search, including the number of files, the size of the hash tables, the hash functions, and the collision mechanism. These choices were based on numerous experiments.

Number of Files

Most search spaces start with one node, increase to a maximum size at some depth, and then decrease again. In sorting-based DDD, the number of files is easily varied over the course of the search, but this doesn't work well for hash-based DDD with arbitrary-length cycles, however.

With arbitrary-length cycles, we have to keep two sets of files at two successive depths, hashed with the same file hash function. If we change the number of files, we have to change the file hash function, then read and rehash the nodes at the previous depth to a new set of files. We implemented this scheme, but it didn't perform as well as using a constant number of files throughout the search, since there is little overhead associated with small files at the start and end of the search.

There must be enough files so that at the maximum size of the search space, each file can be hashed in memory. Since we rarely know the maximum size in advance, we can assume the search will occupy the entire disk. On our machine, this resulted in about 500 files. Surprisingly, using extra files usually improved performance, since smaller files allow smaller hash tables, resulting in better cache performance.

Size of Hash Tables

Our original hash-based algorithm [Korf, 2008a] used a single maximum size hash table for every file, based on the amount of memory. To avoid scanning a large sparse table when hashing small files, the input file buffer was saved, and once the nodes were hashed into memory and those at the previous depth deleted, the nodes in the input buffer were hashed again to find the occupied locations. Instead, we adjust the hash table size based on the size of each input file.

File and Memory Hash Functions

The simplest hash function takes the state description modulo a prime number. This produces a uniform distribution of states, and accommodates almost any size hash table. Computing the mod function with a non-constant modulus is relatively expensive, however, since it requires integer division. With a modulus known at compile time, this computation is optimized by the compiler and hence is much more efficient.

A simple alternative is to multiply two halves of the state description to mix the bits, and use the middle bits of this product as the hash value. This is efficient, but restricts the hash table sizes to powers of two.

Since we use a constant number of files known at compile time, we use the mod function for the file hash function, and the above product function for the memory hash function.

Hash Collisions

There are three standard ways to deal with different states hashing to the same location. One is to keep a linked list of items that hash to each location. This requires additional memory for pointers, and has poor locality, since successive elements can be far apart in memory. A second scheme is to rehash a colliding state to a different location. This requires additional hash functions, and also has poor locality. Our choice, called open addressing, places colliding entries in the next successive empty location. This provides excellent locality, but requires keeping some of the table empty to avoid long runs of occupied locations. Our hash table sizes were at least twice the number of nodes in each file.

5 Comparing Sorting and Hashing on Disk

How do sorting and hashing compare for disk-based search with DDD? The relevant measures are running time, amount of disk I/O, and the maximum storage needed.

5.1 Previous Work

The only previous performance comparison of sorting and hashing for disk search appears in one paragraph of an earlier paper [Korf, 2004]. We found that our hash-based algorithm was 40% faster than our sorting-based algorithm on the 20-disc 4-peg Towers of Hanoi. There are several important differences between our earlier work and this work. Previously we did not use pipelining, used only a single domain. and only considered running time. Previously we used quicksort, while we found merge sort to be faster. The hash functions we used were special-purpose rather than the general-purpose functions used here. Our earlier file hash function was based on the 6 largest discs, and our memory hash function was based on the 14 smallest discs. Since the state of the 6 largest discs was encoded in the file names, the 14 smallest discs and 4 used-operator bits could be stored in 32 bits, while we use 64 bits per state here. Thus, our earlier experiments required only half the disk I/O of our current work. Finally, our sorting-based algorithm took over 18 days to search the 20-disc problem 12 years ago, while our current algorithm takes just over one day on this problem.

5.2 Running Time

Sorting runs in $O(n \log n)$ time, where $O(n)$ is the number of nodes, while hashing runs in linear time. On the other hand, hashing has very poor cache performance compared to merge sort. Thus, their relative speed remains an empirical question.

5.3 Amount of Disk I/O

Let u be the number of unique states in a search, and g be the number of generated nodes, including duplicates. In some problems, such as the sliding-tile puzzle, u is the number of vertices in the graph, and g the number of edges. In other problems, however, g is less than the number of edges, due to move-pruning rules such as don't rotate the same Rubik's Cube face twice in a row, or don't move the same Towers of Hanoi disc twice in a row.

The total disk I/O of the original hash-based algorithm on problems with only even length cycles is $2g + 2u$, as each generated and unique node is written and read once. In problems with odd-length cycles, this increases to $2g + 3u$, since unique nodes are read again to delete them from the generated nodes at the next depth. Pipelining reduces these to $2g$ with only even-length cycles and $2g + 2u$ with odd-length cycles.

For the sorting-based algorithms, the g term is further reduced. As nodes are generated, they are buffered in memory. Once the buffer is full, the nodes are sorted, and duplicate nodes are merged before the buffer is written to disk. This reduces the disk I/O by the number of duplicates found within each buffer. In those iterations where all generated nodes fit in a single file, no duplicate nodes are written to disk.

The hash-based algorithm doesn't achieve this reduction. When expanding nodes, there is a separate buffer in memory for each file of generated nodes. While individual buffers could be hashed in memory to merge duplicates before they are written to disk, this is not done for two reasons. One is that the same nodes will have to be hashed again when the entire file is merged, resulting in extra running time.

The second reason is that this would remove very few duplicates, as we will see. In all our domains, a state is represented by a vector of variables, each representing the position of a sliding tile, the position of a Towers of Hanoi disc, or the position and orientation of an edge cubie in Rubik's Cube. In the sliding-tile puzzles and Towers of Hanoi, only one tile or disc is affected by each move, while in Rubik's Cube, one third of the edge cubies are affected by each move. Thus, except for moves that change the high-order bits of the state representation, most of the children of a node will be close to their parent and each other in a sorted order. Thus, when expanding a set of adjacent parents, most children will be close as well, resulting in a significant number of duplicate nodes. This doesn't happen with hashing, since the file hash function is designed to spread the children randomly among the files.

5.4 Maximum Storage Needed

For the same reason, the maximum amount of disk space needed by the sorting-based algorithm is significantly less than for the hash-based algorithm. For problem spaces with only even-length cycles, the hash-based algorithm stores the number of generated nodes at each depth, and the maximum storage needed is the maximum value of g_i, the number of generated nodes at depth i, times the space for a single node. The sorting-based algorithm stores these nodes minus any duplicates detected in memory. In problem spaces with arbitrary-length cycles, the maximum storage needed for the hash-based algorithm is the maximum value of $u_{i-1} + g_i$ over all depths i, with a smaller value for the sorting-based algorithm. Pipelining has no effect on this. Thus, the sorting-based algorithm requires both less disk I/O and less maximum storage than the hash-based algorithm, as we will see below. This is in contrast to our earlier claim [Korf, 2008a] that sorting and hashing have the same I/O complexity and by implication the same maximum storage requirements.

6 Experimental Results

DDD requires a problem space with groups of nodes that can be expanded in any order. For example, in a breadth-first search, nodes at the same depth can be expanded in any order. For simplicity and reproducibility, we use exhaustive breadth-first searches in our experiments. These results should also apply to heuristic searches as well. For example, breadth-first heuristic search [Zhou and Hansen, 2006a] is the most space-efficient heuristic search, and expands nodes in breadth-first order, pruning nodes when their $g + h$ values exceed a cost threshold for each iteration. As another example, A*[Hart, Nilsson, and Raphael, 1968] can expand nodes of the same cost in any order. Both exhaustive breadth-first and heuristic searches produce search spaces with similar shapes. The number of nodes as a function of depth increases exponentially initially, reaches a maximum, and then decreases. For breadth-first search, the decrease is due to exhausting unique nodes, whereas for heuristic search it is due to increased heuristic pruning with increased depth. Another advantage of exhaustive breadth-first search is that the total number of unique states is known, which greatly facilitates debugging.

We chose three standard combinatorial problems: sliding-tile puzzles, Rubik's Cube, and Towers of Hanoi. The sliding-tile puzzles have only even-length cycles, while the others two have odd-length cycles as well. The limiting resource for the sliding-tile and Towers of Hanoi problems is running time, while for Rubik's Cube it is disk capacity. The largest sliding-tile puzzle we searched is the 3x5 Fourteen Puzzle.

The 4-peg Towers of Hanoi adds an extra peg to the standard 3-peg version. It's an interesting search problem because the optimal solution is not known for an arbitrary number of discs. Our initial state has all discs on one peg. Permuting the three non-initial pegs results in equivalent states, and this symmetry reduces the search space by almost a factor of six. The largest problem we searched is the 21-disc version. While the Fifteen Puzzle and the 22-disc Towers of Hanoi have been searched exhaustively, to compare the running times of multiple different algorithms we used smaller-sized problems.

The 3x3x3 Rubik's Cube problem space is much too large to search exhaustively, so we used subspaces defined by the edge cubies. The largest of these that could be searched with 10 terabytes of disk space is based on 10 of the 12 edge cubies. The full 12 edge cubie subspace is four times larger.

Our machine is an Intel Xeon dual processor running at 3.3 gigahertz, with 48 gigabytes of memory. All our algorithms were implemented serially. We have five two-terabyte disks striped in a level-0 raid, which appears as a single 10-terabyte disk. In all our experiments, most of the time was spent merging duplicate nodes, rather than expanding nodes.

Table 1 summarizes our main experimental results. The

Rubik's Cube column refers to the 10 edge-cubie subspace. The first three data rows of the table show the maximum depth, the number of unique states in the problem space, and the total number of nodes generated in an exhaustive search of the space, including duplicates. For the Towers of Hanoi, the number of states includes all symmetric states, while the number of nodes doesn't count symmetric states.

Problem	Sliding Tile	Rubik's Cube	Tower Hanoi
Size	14 Puzzle	10 cubies	21 discs
Depth	84	13	341
States	6.54×10^{11}	2.45×10^{11}	4.40×10^{12}
Nodes	9.59×10^{11}	2.02×10^{12}	2.21×10^{12}
Sort-P	36:18:55	123:45:00	107:27:10
Hash-P	19:57:47	52:15:20	56:20:13
Hash-NP	29:02:21	81:46:49	61:26:55
S space	340 GB	6726 GB	283 GB
H space	524 GB	8919 GB	415 GB
Disk n/s	13.34×10^6	10.71×10^6	10.87×10^6
Size	13 Puzzle	8 cubies	19 discs
Depth	108	12	257
States	4.36×10^{10}	5.11×10^9	2.75×10^{11}
Nodes	5.92×10^{10}	4.05×10^{10}	1.38×10^{11}
In-Mem	1:19:00	1:03:47	3:05:57
Mem n/s	12.48×10^6	10.57×10^6	12.35×10^6

Table 1: Summary of Main Experimental Results

6.1 Running Time of Pipelined Algorithms

The 4th and 5th data rows of Table 1 compare the running times of pipelined versions of the sorting and hashing based algorithms. Times are shown in the form hours:minutes:seconds. On the Fourteen Puzzle, the hash-based algorithm, using 29 files, was 45% faster than the sorting-based algorithm. On the 10 edge-cubie subspace of Rubik's Cube, the hash-based algorithm, using 500 files, was 58% faster than the sorting-based algorithm. On the 21-disc 4-peg Towers of Hanoi problem, the hash-based algorithm, using 11 files, was 48% faster than the sorting-based algorithm. This is consistent with our results in [Korf, 2004], but for more efficient pipelined versions of the algorithms, using general-purpose hash functions, on two more domains.

6.2 Pipelined vs. Non-Pipelined Algorithms

To determine the effect of pipelining on running time, we compared the original hash-based algorithm with separate expansion and merge phases to the pipelined version. The 6th data row of Table 1 shows the running times of the original algorithm without pipelining. The pipelined algorithm is 31% faster on the Fourteen Puzzle, 36% faster on the Rubik's Cube, and 8% faster on the 21-disc Towers of Hanoi.

6.3 Maximum Disk Space Used

In all three domains, a state was encoded in eight bytes. The 7th and 8th data rows of the table show the maximum amount of disk space in gigabytes (10^9) used by the sorting and hash-based algorithms, respectively. These values are underestimates, since they only count space for the nodes themselves, and not the overhead of the file system. The sorting-based algorithm used 35% less on the Fourteen Puzzle, 25% less on Rubik's Cube, and 32% less on the Towers of Hanoi problem.

We also modified the hash-based algorithm to merge duplicate nodes before writing them to disk. On Rubik's Cube, which required the most disk space, this algorithm only reduced the maximum number of nodes stored by 0.3%, compared to the 25% reduction for the sorting-based algorithm. Furthermore, this algorithm took 79:10:13 to run, compared to 52:15:20 for the standard hash-based algorithm, or 52% slower. On the Fourteen puzzle, this algorithm only reduced the maximum storage needed by 0.8%, compared to 35% for the sorting-based algorithm, but took 31:07:28 to run compared to 19:57:47 for the standard version, or 56% slower. On the 21-disc Towers of Hanoi problem, this algorithm reduced the maximum storage needed by 1.16%, compared to 32% for the sorting-based algorithm, but took 80:21:56 to run, compared to 56:20:13 for the standard algorithm, or 43% slower. Thus, sorting-based disk search uses significantly less space than the hash-based algorithm. Furthermore, modifying the hash-based algorithm to merge duplicate nodes before writing them to disk does not lead to a significant space reduction, at the cost of a large penalty in running time.

7 Disk-Based vs. In-Memory Search

How fast is disk-based search compared to in-memory search? We ran them both on the largest size problems that fit in memory. These were the 2x7 Thirteen Puzzle, the Rubik's Cube subspace based on eight corner cubies, and the 19-disc 4-peg Towers of Hanoi problem. In all three domains, the disk-based algorithm was faster, but this is not a fair comparison, since the operating system buffers small files in memory and doesn't actually use the disk. A much better comparison is to run the best in-memory algorithms on the largest size problems that fit in memory, and compare their node generation rates with the corresponding rates for the largest problems solved on disk. The third row from the bottom of Table 1 shows the speed of our pipelined hash-based disk algorithm in each of our domains, in node generations per second. These values are the 3rd data row divided by by the 5th data row.

The standard in-memory frontier-search algorithm keeps all nodes in a single hash table. For problems with only even-length cycles, each iteration starts with the unique nodes at depth d. The table is scanned linearly, each node at depth d is expanded, its children at depth $d+1$ are hashed into the table, merging duplicate nodes, and the parent node is deleted. For problems with arbitrary-length cycles, the algorithm starts each iteration with the unique nodes at depths $d-1$ and d. Each iteration linearly scans the table, deleting nodes at depth $d-1$, expanding nodes at depth d, and storing new nodes generated at depth $d+1$ or merging them with duplicates, if they don't also appear at depth d. This leaves the nodes at depth d and $d+1$ in the table at the end of the iteration. Note that deleting a node from an open-addressed hash table leaves a hole which will terminate the search for any node that follows it, if its target location precedes the hole. Thus, any such states must be moved back in the table. This algorithm took 2:04:10 on the Thirteen Puzzle, 1:35:36 on the

Rubik's Cube subspace of 8 edge cubies, and 3:48:53 on the 19-disc Towers of Hanoi problem.

Our fastest in-memory frontier-search algorithm uses an explicit open list in addition to the hash table. Each iteration starts with the set of unique nodes at depth d in the open list. Each node is expanded, and the children at depth $d+1$ are hashed into the hash table, merging duplicate nodes. Then, for problems with only even-length cycles, the hash table is scanned linearly, the unique nodes at depth $d+1$ are moved to the open list, overwriting the previous nodes, and the table is cleared. For problems with arbitrary-length cycles, after all the nodes in the open list are expanded and their children hashed into the table, each node in the open list is hashed into the table, and if a match is found, that node is marked for deletion. It isn't immediately deleted since that would leave a hole in the table. Finally, the hash table is scanned, the entries not marked for deletion are moved to the open list, overwriting the previous entries, and the hash table is cleared.

This algorithm avoids the cost of deleting individual nodes from an open-addressed hash table. It also allows variable-size hash tables, since nodes are rehashed every iteration, avoiding scanning a sparse table in the early and late iterations. Furthermore, for problems with arbitrary-length cycles, it uses less memory than the standard algorithm. Let m be the maximum number of nodes at any depth. Assume our hash tables are set to twice the size of the number of nodes stored, to avoid long runs of occupied locations. The standard algorithm keeps two levels of the search in the hash table at once, requiring a hash table of size $4m$. The open-list algorithm only stores one level of the search in the hash table ($2m$), plus the open list (m), for a total size of $3m$. This open-list algorithm is our fastest in-memory algorithm in all three domains.

We also implemented DDD using sorting in memory, but it is slower than hashing, and requires more memory, since duplicate nodes are stored before they are merged, whereas hashing never stores duplicate nodes.

The bottom part of Table 1 shows the maximum search depth, the number of unique states, and the number of nodes generated by a complete breadth-first search for smaller problems. For the Towers of Hanoi, the states include all symmetric states, but the nodes generated only count their canonical representatives. It also shows the running times of our fastest in-memory algorithm, and its speed in node generations per second. Comparing the two node generation rates, our pipelined hash-based disk algorithm is slightly faster than our best in-memory algorithm on the sliding-tile puzzles and Rubik's Cube, and only 12% slower on the Towers of Hanoi. We found it quite surprising that storing nodes on disk instead of memory does not impose any significant time overhead.

It may be tempting to claim that disk-based search is faster than in-memory search on some problems, but it's not. Given enough memory, we can simulate any disk-based algorithm by storing the files in memory. For small problems, the operating system does exactly this by buffering files in memory.

8 Summary and Conclusions

The dominant paradigm for external-memory search uses sorting to merge duplicate nodes. A popular text on heuristic search [Edelkamp and Schroedl, 2012] devotes an entire chapter to this approach, but only one page to hashing.

We first describe efficient implementations of both sorting and hash-based algorithms, using pipelining to reduce disk I/O. To our knowledge, pipelining has not been used before in hash-based disk search. Our pipelined hash-based algorithm is 8% faster on the 21-disc 4-peg Towers of Hanoi problem, 31% faster on the Fourteen Puzzle, and 36% faster on the subspace of Rubik's Cube defined by ten edge cubies, compared to the non-pipelined hash-based algorithm. We also provide guidance on several implementation choices for the hash-based algorithm, including hash functions, collision resolution, the size of hash tables, and the number of files. Unlike the original hash-based algorithm [Korf, 2008a], we use hash tables whose sizes are adapted to the file size. We found that adapting the number of files to the number of nodes at a given depth is not worthwhile, since with arbitrary-length cycles it requires rehashing existing nodes. Furthermore, we found that the minimum number of files doesn't produce the best performance, since more files result in smaller files, allowing smaller hash tables with better cache performance.

We then show that hashing is significantly faster than sorting. Our pipelined hash-based algorithm is 45% faster on the Fourteen Puzzle, 48% on the Towers of Hanoi, and 58% on Rubik's Cube, compared to our pipelined sorting-based algorithm. We previously reported a speedup of 41% [Korf, 2004] on the Towers of Hanoi, using less efficient algorithms without pipelining, with special-purpose hash functions, on a slower machine and a smaller problem size.

We also show that the sorting-based algorithm requires less disk I/O and uses less storage than the hash-based approach. The savings were 35% on the Fourteen Puzzle, 25% on Rubik's Cube, and 32% on the Towers of Hanoi. Thus, if a search is limited by the available disk storage rather than running time, sorting may be a better choice.

Finally, we compare disk-based to in-memory search. In all three domains, our disk-based search was faster than the standard in-memory search using only a hash table. Our fastest in-memory frontier-search algorithm uses an explicit open list in addition to a hash table. On the Fourteen Puzzle and 10 edge-cubie Rubik's Cube, the node generation rate of our pipelined hash-based disk search was slightly faster than our in-memory open-list algorithm on the Thirteen Puzzle and 8 edge-cubie Rubik's Cube, respectively. On the Towers of Hanoi, our disk-based algorithm on the 21-disc problem was about 12% slower than our best in-memory algorithm on the 19-disc problem. Thus, our disk-based search allows us to solve problems several orders of magnitude larger than our best in-memory algorithm, with little or no time overhead in terms of nodes per second. The main drawback is the additional complexity of the code.

In a heuristic search, running time includes node generation, heuristic evaluation, and duplicate detection. In our brute-force searches of these simple domains, duplicate detection took most of the time. In heuristic searches of domains with more expensive node generation and evaluation, the relative running times of hashing and sorting should be closer together. For the same reason, the running times of in-memory and disk-based algorithms should be closer as well.

References

[Ajwani, Demetiev, and Meyer, 2006] Ajwani, D.; Demetiev, R.; and Meyer, U. 2006. A computational study of external-memory bfs algorithms. In *Proceedings of the Symposium on Discrete Algorithms (SODA)*, 601–610.

[Edelkamp and Schroedl, 2012] Edelkamp, S., and Schroedl, S. 2012. *Heuristic Search: Theory and Applications*. Morgan Kaufmann.

[Hart, Nilsson, and Raphael, 1968] 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.

[Jabbar, 2008] Jabbar, S. 2008. *External Memory Algorithms for State Space Exploration in Model Checking and Planning*. Ph.D. Dissertation, University of Dortmund, Dortmund, Germany.

[Korf and Schultze, 2005] Korf, R., and Schultze, P. 2005. Large-scale, parallel breadth-first search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2005)*, 1380–1385.

[Korf et al., 2005] Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *J.A.C.M.* 52(5):715–748.

[Korf, 2004] Korf, R. 2004. Best-first frontier search with delayed duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2004)*, 650–657.

[Korf, 2008a] Korf, R. 2008a. Linear-time disk-based implicit graph search. *J.A.C.M.* 55(6):26–1 to 26–40.

[Korf, 2008b] Korf, R. 2008b. Minimizing disk i/o in two-bit breadth-first search. In *Proceedings of the Nationsl Conference on Artificial Intelligence (AAAI-2008)*.

[Kunkle and Cooperman, 2007] Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for rubik's cube. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC-07)*, 235–242.

[Robinson, Kunkle, and Cooperman, 2007] Robinson, E.; Kunkle, D.; and Cooperman, G. 2007. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO-07)*, 78–87.

[Sturtevant and Rutherford, 2013] Sturtevant, N., and Rutherford, M. 2013. Minimizing writes in parallel external memory search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2013)*, 666–673.

[Zhou and Hansen, 2004] 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)*, 683–688.

[Zhou and Hansen, 2006a] Zhou, R., and Hansen, E. 2006a. Breadth-first heuristic search. *Artificial Intelligence* 170(4-5):385–408.

[Zhou and Hansen, 2006b] Zhou, R., and Hansen, E. 2006b. Domain-independent structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2006)*, 1082–1087.

[Zhou and Hansen, 2007a] 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)*, 2410–2416.

[Zhou and Hansen, 2007b] Zhou, R., and Hansen, E. 2007b. Parallel structured duplicate detection. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2007)*, 1217–1223.