Large-Scale Parallel Breadth-First Search

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

Abstract

Recently, best-first search algorithms have been introduced that store their nodes on disk, to avoid their inherent memory limitation. We introduce several improvements to the best of these, including parallel processing, to reduce their storage and time requirements. We also present a linear-time algorithm for bijectively mapping permutations to integers in lexicographic order. We use breadth-first searches of sliding-tile puzzles as testbeds. On the 3x5 Fourteen Puzzle, we reduce both the storage and time needed by a factor of 3.5 on two processors. We also performed the first complete breadth-first search of the 4x4 Fifteen Puzzle, with over 10^{13} states.

Introduction

Breadth-first search is a basic search algorithm. It is used in model checking, to show that certain states are reachable or unreachable, and to determine the radius of a problem space, or the longest shortest path from any given state. It is also used to compute pattern-database heuristics (Culberson & Schaeffer 1998). *Breadth-first heuristic search* (Zhou & Hansen 2004a) is a space-efficient version of A*(Hart, Nilsson, & Raphael 1968) for problems with unit edge costs. It implements A* as a series of breadth-first search iterations, with each iteration generating all nodes whose costs donot exceed a threshold for that iteration. Other methods for extending disk-based breadth-first search to A* have also been implemented (Korf 2004; Edelkamp, Jabbar, & Schroedl 2004), and the techniques described in this paper apply to such heuristic searches as well.

Breadth-first search is often much more efficient than depth-first search, because the latter can't detect duplicate nodes representing the same state, and generates all paths to a given state. For example, with a branching factor of 2.13, a depth-first search of the Fifteen Puzzle, the $4x4$ sliding-tile puzzle, to the average solution depth of 53 moves would generate 2.5×10^{17} nodes, whereas the entire problem space only contains 10^{13} unique states.

Our goal is to increase the size of feasible searches. The primary limitation of best-first search is the memory needed to store nodes, in order to detect duplicate nodes. Several recent advances have addressed this problem.

Previous Work

Frontier Search

Frontier search (Korf 1999; Korf & Zhang 2000; Korf *et al.* 2005) stores the Open list of generated nodes, but not the Closed list of expanded nodes. This reduces the memory required for breadth-first search from the size of the problem space to the *width* of the problem space, or the maximum number of nodes at any depth. For the Fifteen Puzzle, for example, this reduces storage by a factor of over 13.

Disk Storage

The next advance is storing nodes on magnetic disks (Roscoe 1994; Stern & Dill 1998; Korf 2003; 2004; Zhou & Hansen 2004b; Edelkamp, Jabbar, & Schroedl 2004). Disks costs less than $1 per gigabyte, compared to $200 per gigabyte for memory. Disks must be accessed sequentially, however, since disk latency is 10^5 times memory latency.

There exists a body of work on algorithms for graphs stored explicitly on disk, which focuses on asymptotic I/O complexity. See (Katriel & Meyer 2003), for example. By contrast, we are interested in search algorithms for very large implicit graphs defined by a root node and a successor function, which can't be explicitly stored even on disk.

Sorting-Based DDD In the first of these algorithms (Roscoe 1994; Korf 2003; 2004), each level of a breadth-first search starts with a file containing the nodes at the current depth. All these nodes are expanded, and their children are written to another file, without any duplicate checking. Next, the file of child nodes is sorted by their state representation, bringing duplicate nodes together. A single pass of the sorted file then merges any duplicate nodes. We refer to this as *sorting-based DDD*, for delayed duplicate detection.

Hash-based DDD To avoid the time complexity of sorting, *hash-based DDD* (Korf 2004) uses two orthogonal hash functions, and alternates expansion with merging.

In the expansion phase, we expand all the nodes at a given depth, and write the child nodes to different files, based on the value of a first hash function. Any duplicate nodes will be mapped to the same file. Frontier search guarantees that all children nodes are either one level deeper than their parents, in the case of the sliding-tile puzzles, or possibly at the same depth in general.

In the merge phase, we process each file, hashing its nodes into memory using a second hash function, thus detecting any duplicate nodes. Finally, we write one copy of each child node back to disk to begin the next iteration. This algorithm was described in (Korf 2004), but only implemented for the 4-peg Towers of Hanoi problem, where ideal hash functions are trivial to compute.

Structured Duplicate Detection *Structured duplicate detection* (Zhou & Hansen 2004b) detects duplicates as soon as they are generated. All nodes must be divisible into subsets, such that the children of nodes in one subset fall into a small number of subsets. In a sliding-tile puzzle, for example, subsets may be based on the blank position. The children of nodes with one blank position can have at most four other blank positions. Furthermore, all the states in any subset, plus all its child subsets, must fit in memory simultaneously. When expanding nodes in one subset, the children are looked up in the corresponding child subsets in memory. When expanding nodes in another subset, currently resident subsets may have to be swapped out to disk to make room for new parent and child subsets in memory.

Symmetry

For some sliding-tile puzzles, symmetry can reduce the space needed by a factor of two (Culberson & Schaeffer 1998). For the Fifteen Puzzle, for example, starting with the blank in a corner, every state has a mirror state computed by reflecting the puzzle about a diagonal passing through the initial blank position, and renumbering the tiles using the same transformation. Some states equal their mirror reflections. The reduction in time is less than a factor of two, due to the overhead of computing the mirror states.

Overview of Paper

This work is based on hash-based DDD (Korf 2004), the most promising algorithm for large-scale problems. Hash-based DDD is faster than sorting-based DDD, and preferable to structured duplicate detection for several reasons. The first is that it doesn't require the subset structure described above. A second reason is that it requires relatively little memory, whereas structured duplicate detection must be able to hold a parent subset and all its child subsets in memory simultaneously. Hashed-based DDD is easily parallelized, and can be interrupted and resumed. Finally, hash-based DDD only reads and writes each node at most twice.

We describe a number of improvements designed to reduce both the storage needed and the running time of hash-based DDD. These include efficient state encoding for permutation problems, interleaving expansion and merging, not storing nodes that have no children, parallel processing, and fault tolerance. On the $3x5$ Fourteen Puzzle, we reduce both the storage needed and the running time by a factor of 3.5, using two processors. We also completed a breadth-first search of the Fifteen Puzzle, with over 10^{13} states.

Efficient Permutation Encoding

Problems such as the sliding-tile puzzles and Rubik's Cube are *permutation problems*, in that a state represents a permutation of elements. The simplest representation of a permutation is to list the position of each element. For example, a Fifteen Puzzle state can be represented as a 16-digit hexadecimal number, where each digit represents the position of one tile or the blank. This occupies 64 bits of storage.

A more efficient encoding saves storage and reduces I/O time. Ideally, we would map a permutation of n elements to a unique integer from zero to $n! - 1$. For the Fifteen Puzzle, this requires only 45 bits. For example, we could map each permutation to its index in a lexicographic ordering of all such permutations. For permutations of three elements, this mapping is: 012-0, 021-1, 102-2, 120-3, 201-4, 210-5.

An algorithm for this mapping starts with a sequence of positions, and maps it to a number in *factorial base*, 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 $i - 1$, resulting in a unique representation of each integer. Given a permutation as a sequence of digits in factorial base, we perform the indicated arithmetic operations to compute the actual integer value.

To map a permutation to a sequence of factorial digits, we subtract from each element 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. By reducing these factorial digits to an integer, we obtain the desired values: 012-000-0, 021-010-1, 102-100-2, 120-110-3, 201-200-4, 210-210-5.

This algorithm takes $O(n^2)$ time to compute the digits in factorial base. Next 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 element of the permutation is encountered, we use it as an index into the bit string and set the corresponding bit to one. When we encounter element k in the permutation, to determine the number of elements less than k to its left, we need to know the number of ones in the first k bits of our bit string. We extract the first k bits by right shifting the string by $n - k$. This reduces the problem to: given a bit string, count the number of one bits in it.

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 each 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. Such a table for the Fifteen Puzzle would contain $32,768$ entries.

This gives us a linear-time algorithm for mapping permutations of n elements in lexicographic order to unique integers from zero to $n! - 1$. We implemented both the quadratic and linear algorithms above, and tested them by mapping all permutations of up to 14 elements. For 14 elements, the linear algorithm was seven times faster, and this ratio increases with increasing problem size, as expected.

This mapping is also used in heuristic searches of permutation problems using pattern databases (Culberson & Schaeffer 1998). By mapping each permutation of the pattern to a unique integer, the permutation doesn't have to be stored with the heuristic value, and each location corresponds to a

valid permutation, making efficient use of memory.

To expand a node, we need to regenerate the original permutation from its integer encoding. This can be done in linear time as well, but requires more memory, and is not significantly faster than the quadratic algorithm. The reason is that mapping the integer to a permutation requires integer division and remaindering, which is much more expensive than multiplication, dominating the cost of the mapping.

There exist other algorithms for mapping between permutations and integers in linear time and linear space (Myrvold & Ruskey 2001), but not in lexicographic order. In fact, (Myrvold & 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." Our algorithm runs in linear time, but uses $O(2^n)$ space. The space is for a table that is only computed once for a given value of n.

Perfect Hashing

As described above, hash-based DDD makes use of two hash functions. When a node is expanded, its children are written to a particular file based on the first hash value. For the Fifteen Puzzle, we map the positions of the blank and tiles 1, 2, and 3, to a unique integer in the range zero to $16 \cdot 15 \cdot 14 \cdot 13 - 1 = 43,679$. This value forms part of the name of the file. The diagonal symmetry mentioned above reduces the actual number of files to $21,852$.

Since all nodes in any one file have the blank and first three tiles in the same positions, we only have to specify the positions of the remaining twelve tiles. Since only half the initial states of a sliding-tile puzzle are solvable, the positions of the last two tiles are determined by the positions of the other tiles. Thus, we only specify the positions of ten tiles, by mapping their positions to a unique integer from zero to $12!/2 - 1 = 239,500,799$, requiring 28 bits.

To avoid regenerating expanded nodes, frontier search stores with each node its *used operators*, which lead to neighboring nodes that have already been expanded. Since a sliding-tile puzzle state has at most four operators, moving a tile up, down, left, or right, we need four used-operator bits. Thus, a Fifteen-Puzzle state can be stored in $28 + 4 = 32$ bits, which is half the storage needed without this encoding.

Since the states in a file are already encoded in a 28-bit integer, this is used as the second hash value. To merge the duplicate nodes in a given file, we set up a hash table in memory with $239,500,800$ 4-bit locations, initialized to all zeros. We then read each node from the file, map it to its unique location in the hash table, and OR its used-operator bits to those already stored in the table, if any, thus taking the union of used-operator bits of duplicate nodes. Finally, we write a single copy of each node, along with its used-operator bits, to a merged file. A perfect hash function that maps each state to a unique value saves a great deal of memory, since we don't have to store the state in the table, nor use empty locations or pointers to handle collisions.

After merging the nodes in one file, we need to zero the hash table. We could zero every entry in sequential order, but this is expensive if there are only a small number of non-zero entries. Alternatively, we can scan the input buffer, and only zero those entries that were set to a non-zero value. Zeroing the states in the order they appear in the input buffer may lead to poor cache performance, however. Our solution to this dilemma is that if only a small number of table entries were set, we explicitly zero those entries, and otherwise we sequentially zero the entire table. In our table of 239 million elements, the break-even point is about 2.5 million entries.

Interleaving Expansion and Merging

In our previous hash-based DDD algorithm (Korf 2004), all parent files at a given depth being expanded before any child files at the next depth were merged. The disadvantage of this approach is that at the end of the expansion phase, all nodes generated at the next level are stored on disk, including their duplicates. The storage required is thus proportional to the maximum number of nodes generated at any depth.

If we merge child files as soon as possible, however, we only have to store approximately the maximum number of unique states at any level. In order to merge each child file only once, we defer merging it until all the parent files that could contribute to it have been expanded. At that point, the child file is placed on a queue for merging. If any files are eligible for merging, they take priority over expanding files.

To minimize the time that a child file exists, when we expand a parent file, we'd like to expand other neighbors of its children as soon as possible. As a heuristic for this, we expand parent files in the order in which states in that file would first be generated in a breadth-first search.

For the Fourteen Puzzle, with symmetry, the maximum number of nodes generated at any depth is 3.2×10^{10}, while the maximum number of unique states is 2.3×10^{10}, saving 30%. For the Fifteen Puzzle, with symmetry, the maximum number of nodes generated is 5.9×10^{11}, while the maximum number of unique nodes is 3.9×10^{11}, saving 34%.

Not Storing Sterile Nodes

A *fertile* node has children that first appear at the next search depth, whereas all the neighbors of a *sterile* node appear at the previous depth. In our algorithm, sterile nodes are detected during merging when all their used-operator bits are set. Rather than writing sterile nodes to a file to be expanded in the next iteration, we simply count and delete these nodes.

For the Fourteen Puzzle, this reduces the maximum number of stored nodes from 2.3×10^{10} to 1.9×10^{10} , a 16% savings. For the Fifteen Puzzle, this reduces the number of nodes stored from 3.9×10^{11} to 3.4×10^{11}, a 12% savings.

A second advantage is that we save some time by not writing sterile nodes to disk, and not reading them back in, particularly in the latter stages of the search, where most nodes are sterile. For both the Fourteen and Fifteen Puzzles, this reduces the total I/O by about 5%.

Multi-Threading

Paradoxically, even on a single processor, multi-threading is important to maximize the performance of disk-based algorithms. The reason is that a single-threaded implementation will run until it has to read from or write to disk. At that point it will block until the I/O operation has completed. The

operating system will use the CPU briefly to set up the I/O transfer, but then the CPU will be idle until the I/O completes. Furthermore, many workstations are available with dual processors for a small additional price. For very large searches, machines with many processors will be required.

Hash-based DDD is ideally suited to multi-threading. Within an iteration, most file expansions and merges can be done independently. If we simultaneously expand two parent files that have a child file in common, the two expansions will interleave their output to the child file. While this is acceptable, we avoid it for simplicity.

To implement our parallel algorithm, we use the parallel primitives of POSIX threads (Nichols, Butler, & Farrell 1996). All threads share the same data space, and mutual exclusion is used to temporarily lock data structures.

Our algorithm maintains a work queue, which contains parent files waiting to be expanded, and child files waiting to be merged. At the start of each iteration, the queue is initialized to contain all parent files. Once all the neighbors of a child file have been expanded, it is placed at the head of the queue to be merged. To minimize the maximum storage needed, file merging takes precedence over file expansion.

Each thread works as follows. It first locks the work queue. If there is a child file to merge, it unlocks the queue, merges the file, and returns to the queue for more work.

If there are no child files to merge, it considers the first parent file in the queue. Two parent files conflict if they can generate nodes that hash to the same child file. It checks whether the first parent file conflicts with any other file currently being expanded. If so, it scans the queue for a parent file with no conflicts. It swaps the position of that file with the one at the head of the queue, grabs the non-conflicting file, unlocks the queue, and expands the file. For each child file it generates, it checks to see if all of its parents have been expanded. If so, it puts the child file at the head of the queue for expansion, and then returns to the queue for more work.

If there is no more work in the queue, any idle threads wait for the current iteration to complete. At the end of each iteration, various node counts are printed, and the work queue is initialized to contain all parent files for the next iteration.

Each thread needs its own hash table for merging, which takes about 114 megabytes, and space to buffer its file I/O. Our program worked best with relatively small I/O buffers, a total of only 20 megabytes per thread.

External Disk Storage

At four bytes per node, a complete search of the Fifteen Puzzle requires a maximum of 1.4 terabytes of storage.[1] The largest single disks currently available hold 400 gigabytes, but only a few will fit inside a typical workstation, leading us to consider external disk storage. There are many choices on the market, varying in cost per byte, maximum transfer rate, and reliability. To allow others to reproduce our results, we chose the least expensive solution. We purchased four LaCie "Big Disk Extreme" units, plus a Firewire 800 (IEEE 1394b) interface card for each. Each unit packages two 250 gigabyte drives striped together non-redundantly, with a maximum transfer rate of 88 megabytes per second. The cost of each unit plus the card was less than a dollar per gigabyte. By plugging each disk into its own card on the PCI bus, we can potentially multiply the total bandwidth by the number of disks. In addition to the Firewire disks, we also have two 300 gigabyte, and one 400 gigabyte serial ATA (SATA) disks inside our workstation.

Since hash-based DDD uses a large number of different files, the simplest way to use multiple disks is to partition the files among the different disks. This also gives the best performance, since the overhead of striping the data is avoided, and multiple threads can access files simultaneously if they are on different disks. This configuration was used for the Fourteen Puzzle experiments reported below.

[1] When referring to disk storage, gigabyte and terabyte refer to 10^9 and 10^{12} bytes respectively, rather than 2^{30} and 2^{40} as in the case of memory.

Fault Tolerance

Using disk storage, a breadth-first search can run for weeks. Unlike other large-scale computations, a breadth-first search cannot be easily decomposed into independent computations, which can fail and then simply be restarted until they succeed. Rather, it must be tolerant of memory losses, due to system crashes or power failures, and loss of disk data.

Loss of Memory

The simplest solution to memory loss is to keep all the nodes of one iteration until the next iteration completes. This allows restarting from the last completed iteration. This requires twice the disk space, however, since two complete levels of the search must be stored at once.

In fact, our program is interruptible with no storage overhead. When interrupted, the file system will contain parent files at the previous depth waiting to be expanded, child files at the current depth waiting to be merged, and child files at the current depth that have already been merged.

If a parent file is being expanded when the program is interrupted, it will still exist, but may have output some of its children to child files. The resumed program expands all the nodes in the parent file, creating additional copies of any child nodes already output, which will eventually be merged as duplicates. Duplicate nodes due to reexpanding the same nodes could be detected, since they have identical used-operator bits. In any case, the number of unique states will not be affected. If a child file is being merged when the program is interrupted, that file will still exist, and a partially merged output file may also exist. In that case, we delete the output file, and remerge the child file.

We never delete an input file until after the output files it generates have been written. Data written to disks is cached in memory on the disk controller, however, and even a blocking write call returns before the data has been magnetically committed to disk. As a result, during a power failure, cached data was lost before it was committed to disk, but after the input files that generated it had already been deleted.

The solution to this problem is an uninterrupted power supply (UPS), with a battery sufficient to power the computer long enough for a clean shutdown, flushing all file buffers, in the event of a power failure.

number of parallel threads	1	2	3	4	5	6	7	8	9	10
time in hours and minutes	52:13	28:45	26:08	25:11	24:52	24:50	24:59	25:11	25:13	25:30

Table 1: Fourteen Puzzle Runtimes vs. Number of Parallel Threads with Two Processors

Unrecoverable Disk Errors

Several attempts to complete the Fifteen Puzzle search on disk configurations optimized for speed failed, due to transient disk errors. Disk manufacturers specify "non-recoverable" read error rates between one in 10^{13} and one in 10^{15} bits. While single-bit errors are routinely corrected by error correcting codes, uncorrectable multiple-bit errors within a parity block do occur. If such an error occurs in a user file, that file cannot be read, but if it occurs in certain critical data, the entire file system can be corrupted. Most people are not aware of this failure mode of disks, because it occurs so rarely. For example, at an error rate of one in 10^{14} bits, we would expect such an error on the central file server in our department about once every 5 years. The complete Fifteen Puzzle search reads and writes a total of 8×10^{14} bits, however, and we saw these errors almost weekly.

The solution to this problem is a RAID, or Redundant Array of Inexpensive Disks (Patterson, Gibson, & Katz 1988). 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, without even interrupting the program. In the case of complete loss of a disk, the bad disk can be unplugged and replaced, and its data reconstructed from the other disks, again without interrupting the program.

For our successful Fifteen Puzzle search, we used a level-5 redundant software RAID composed of four external Firewire disks and two internal SATA disks. This increased the running time of our program by almost 50%, compared to a non-redundant disk array, due to the redundant output, and the CPU cycles needed to compute this output. It completely eliminated our disk error problem, however.

Experiments

Fourteen Puzzle

Previously, the largest sliding-tile puzzled searched completely breadth-first was the Fourteen Puzzle (Korf 2004). Using sorting-based DDD with symmetry, it required 259 gigabytes of storage at eight bytes per node, and almost 18 days on a 440 megahertz Sun Ultra-10 workstation. On an IBM Intellistation A Pro workstation with dual two-gigahertz, 64-bit AMD Opteron processors, two gigabytes of memory, and a single Firewire disk, it took 88 hours.

We ran the program described here on the Fourteen Puzzle with three Firewire disks, and two internal SATA disks, varying the number of parallel threads. The maximum amount of storage used was 75 gigabytes. The file hash function was based on the positions of the blank and first two tiles. Table 1 shows the results, with number of threads on top, and times in hours and minutes on the bottom.

With one thread, our hash-based DDD program ran for over 52 hours, a factor of 1.7 faster than our sorting-based DDD program, using a factor of 3.5 less storage. With six threads on two processors, our program took less than 25 hours to run, a parallel speedup of 2.1. Increasing the number of threads beyond six increased the running time on two processors, presumably due to coordination overhead.

With 5 disks and two processors, our program is CPU-bound. Changing the number of disks slightly doesn't significantly affect performance, but increasing the number of processors should improve it. Most analyses of disk-based algorithms assume they are I/O bound, however, ignoring CPU time and only counting disk I/O.

Fifteen Puzzle

Our main goal was a complete breadth-first search of the Fifteen Puzzle. We learned all the reliability lessons described above the hard way, as the program failed several times due to unrecoverable disk errors, until we diagnosed that problem, and once due to a power failure. Using six disks in a level-5 software RAID, and a UPS, we eventually completed the search in 28 days and 8 hours, using a maximum of 1.4 terabytes of disk storage. Since the RAID generated more I/O, and consumed CPU cycles, the best performance was achieved with three parallel threads on two processors.

Our results confirmed that the radius of the problem space, starting with the blank in a corner, is 80 moves, which was first determined by (Brungger *et al.* 1999) using a more complex method. We also found that there are exactly 17 states at depth 80, more than was previously known. Table 2 shows the number of unique states at each depth. The fact that the total number of states found is exactly $16!/2$ gives us additional confidence that the search is correct.

Conclusions

We presented a linear-time algorithm for bijectively mapping permutations to integers in lexicographic order. On permutations of 14 elements, our algorithm is seven times faster than an existing quadratic algorithm. We improved our disk-based search algorithm (Korf 2004), by interleaving expansion and merging, not storing sterile nodes, and introducing multi-threading, which improves its performance even on a single processor. On the Fourteen Puzzle, these improvements reduce both the storage needed and the running time by a factor of 3.5 on two processors, compared to the previous state of the art. Contrary to the usual assumption in the literature of disk-based algorithms, our program is CPU-bound rather than I/O-bound, even on two processors. We learned the hard way that a program running for a month, reading and writing a total of 3.5 terabytes of data per day, must be fault tolerant. Rare unrecoverable disk errors can be solved by a redundant array of inexpensive disks. Power failures require an algorithm that can be interrupted and resumed, plus a backup power supply to allow a clean system shutdown, flushing all file buffers. A complete search of

the Fifteen Puzzle required 28 days and 8 hours, and 1.4 terabytes of storage. To our knowledge, this is the largest best-first search ever completed.

Acknowledgements

This research was supported by NSF grant No. EIA-0113313, and by IBM, which donated the workstation. Thanks to Satish Gupta of IBM, and Eddie Kohler and Yuval Tamir of UCLA, for their support and help with this work.

References

Brungger, A.; Marzetta, A.; Fukuda, K.; and Nievergelt, J. 1999. The parallel search bench ZRAM and its applications. *Annals of Operations Research* 90:45–63.

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

Edelkamp, S.; Jabbar, S.; and Schroedl, S. 2004. External A*. In *Proceedings of the German Conference on Artificial Intelligence*, 226–240.

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.

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.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the Association for Computing Machinery (JACM)* ,to appear.

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

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

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79:281–284.

Nichols, B.; Butler, D.; and Farrell, J. 1996. *Pthreads Programming*. O'Reilly.

Patterson, D.; Gibson, G.; and Katz, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 109–116.

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

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.

Zhou, R., and Hansen, E. 2004a. Breadth-first heuristic search. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, 92–100.

Zhou, R., and Hansen, E. 2004b. Structured duplicate detection in external-memory graph search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-2004)*, 683–688.

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

Table 2: States as a Function of Depth for Fifteen Puzzle