Delayed Duplicate Detection: Extended Abstract

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 nodes in order to detect duplicates. Disks can greatly expand the amount of storage available, but randomly accessing a disk is impractical. Rather than checking newly-generated nodes as soon as they are generated, we append them to a disk file, then sort the file, and finally scan the sorted file in one pass to detect and remove duplicate nodes. This also speeds up such searches that fit entirely in memory, by improving cache performance. We implement this idea for breadth-first search, performing the first complete searches of the 2 x 7 sliding-tile puzzle, and the 18-disk, 4-peg Towers of Hanoi puzzle.

1 Introduction: The Problem

Best-first search algorithms, such as breadth-first search (BFS), Dijkstra's algorithm [Dijkstra, 1959], and A* [Hart, Nilsson, *k* Raphael, 1968], store every node that is generated, in either the Open list or the Closed list. One reason for doing this is to detect duplicate nodes, and avoid expanding a state more than once. As a result, these algorithms are limited by the available memory.

In some problems, this memory limitation can be avoided by depth-first searches (DFS) such as depth-first iterative-deepening (DFID) or iterative-deepening-A* (IDA*) [Korf, 1985], but DFS can generate exponentially more nodes than BFS. For example, in a rectangular-grid problem space, BFS will generate $O(r^2)$ nodes within a radius of r, while DFID will generate $O(3^r)$ nodes. While there are ways of detecting some duplicate nodes in a DFS[Taylor *k* Korf, 1993], they do not apply to all problem spaces.

2 Frontier Search

An algorithm called *frontier* searc/i[Korf, 1999; Korf *k* Zhang, 2000] saves only the Open list of nodes at the frontier of the search, and not the Closed list of nodes that have been expanded, thus saving some memory. With each node, it stores a *used-operator bit* for each operator, to indicate whether the neighboring state reached by that operator has already been generated. For example, the sliding-tile puzzles require four such bits for each state, one for each direction in which a tile could move. When a parent node is expanded, only children that are reached via unused operators are generated, and the parent node is deleted from memory. In each child node, the operator that generates the parent is marked as used. When duplicate states are found in the Open list, only one copy is kept, and any operator that is 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 additional work. See [Korf, 1999; Korf *k* Zhang, 2000] for two ways to do this.

The memory required by frontier search is proportional to the maximum size of the Open list, or the *width* of the problem space, rather than the size of the space. For example, in the grid space mentioned above, the width of the space grows only linearly with the search radius, while the entire space grows quadratically. As another example, the width of the n-disk, 3-peg Towers of Hanoi space is only 2^n states, while the entire space contains 3^n states. Frontier search is still limited by the memory required to store the Open list, however.

3 Delayed Duplicate Detection (DDD)

Hard disks with hundreds of gigabytes of storage are available for less than a dollar per gigabyte, which is over a hundred times cheaper than memory. Because of high latency, however, a disk behaves more like a sequential device, such as a magnetic-tape drive, with large capacity and high bandwidth, but only if it is accessed sequentially. To quickly detect duplicate states in a search algorithm, however, nodes are usually stored in a hash table, which is designed to be accessed randomly.

Our solution to this problem is rather than checking each newly-generated node for duplicates as soon as it is generated, we append each node to a disk file containing previously generated nodes. At some point later we sort the file, thereby bringing together nodes representing the same state. Then we scan the sorted list of nodes in one pass, merging any duplicate nodes.

3.1 Breadth-First Frontier Search

As a simple example, we describe breadth-first frontier search with delayed duplicate detection. BFS is normally implemented with a FIFO queue, which we implement with two disks files, an input file and an output file. Initially, the input file contains the initial state. As we read each node in the input file, we expand it, and write its children to the output file, with no duplicate checking. When the input file is exhausted, we delete it. At that point, the output file contains all the nodes at the next depth, including any duplicate nodes. We then sort the nodes in the output file by their state representations, which brings together any duplicate nodes representing the same state. Next, we linearly scan the output file, merging duplicate nodes and ORing their used operator bits, and write one copy of each state to a new input file, deleting the output file when we're done. This completes one level of the breadth-first search. The algorithm continues until expanding the nodes in the input file doesn't generate any more nodes in the output file.

3.2 Sorting the Disk Files

Algorithms for sorting disk files are well-known. See [Garcia-Molina, Ullman, & Widom, 2000], pp. 42-48. The basic algorithm is to read as much of the unsorted file as will fit into memory, sort it in memory using quicksort, for example, and write the sorted portion of the file to a new subfile. Continue until the entire original file has been read and written into a set of sorted subfiles. Then, all the sorted subfiles are merged in one pass, storing the head of each file in memory, and writing the lowest record to a final sorted output file.

3.3 DDD in Memory

Surprisingly, delayed duplicate detection is useful even when all nodes fit in memory, resulting in reduced running time due to improved cache performance. In the standard implementation of breadth-first search in memory, the Open list is stored in a hash table. As each new node is generated, it is looked up in the hash table, which often results in a cache miss, since the hash function is designed to randomly scatter the nodes. A DDD implementation doesn't use a hash table, but a single FIFO queue in memory, reading nodes off the head of the queue, and appending them to the tail. Once a level of the search is completed, the queue is sorted in memory using an algorithm such as quicksort, and the sorted queue is scanned, merging duplicate nodes. The advantage of this approach is that the queue is only accessed at the head and tail, or at two points in between during quicksort, and hence most memory references will reside in cache, reducing the running time.

4 Experiments

We implemented a breadth-first search on sliding-tile puzzles, and the 4-Peg Towers of Hanoi problem.

4.1 Sliding-Tile Puzzles

[Schofield, 1967] published a complete breadth-first search of the 3 x 3 Eight puzzle. We completed a BFS for all sliding-tile puzzles up to the 2 x 7 Thirteen Puzzle. Table 1 below shows the results. The first column gives the *x* and *y* dimensions of the puzzle, and the second column gives the number of moves needed to reach all solvable states, starting with the blank in a corner position. This is also the worst-case optimal solution length, for a goal with the blank in a corner. The third column gives the number of solvable states, which is *(xy)V2,* and the fourth column gives the width of the problem space, which is the maximum number of nodes at any depth.

Size	Moves	Total States	Max Width
2 x 2	6	12"	2
2 x 3	21	360	44
2 x 4	37	20,160	1,999
3 x 3	31	181,440	24,047
2 x 5	55	1,814,400	133,107
2 x 6	80	239,500,800	13,002,649
3 x 4	53	239,500,800	21,841,159
2 x 7	108	43,589,145,600	1,862,320,864

Table 1: Sliding-Tile Puzzle Results

The 3 x 4 and 2 x 6 Eleven Puzzles were the largest that we could solve in memory. We implemented both a standard BFS algorithm, using one bit of memory per state, and also breadth-first frontier search with delayed duplicate detection. The standard BFS required 17.5 minutes, while the frontier search with DDD required only 9.6 minutes, on a 440 Megahertz Sun Ultra 10 workstation. This demonstrates that DDD frontier search is useful even for problems that fit in memory.

The 2 x 7 Fourteen Puzzle was the largest we could search exhaustively with our 120 gigabyte disk. At 8 bytes per state, this problem required about 15 gigabytes of disk storage, and ran for 51 hours, 13 minutes. The ratio of the problem size to the problem width is about 23.4, illustrating the advantage of frontier search.

4.2 Towers of Hanoi

For the standard 3-peg Towers of Hanoi problem, there is a simple algorithm that guarantees the shortest path between any pair of states. The 4-peg Towers of Hanoi puzzle, known as Reve's puzzle, is much more interesting. There exists a algorithm for finding a solution, and a conjecture that it generates optimal solutions, but the conjecture remains unproven. [van de Liefvoort, 1992] contains a good bibliography for this problem. [Szegedy, 1999] gives bounds for the A:-peg version, but they include an unspecified constant in the exponent.

For the sliding-tile puzzles, all paths between any pair of states have the same even-odd parity, and the algorithm described in Section 3.1 works correctly. For the Towers of Hanoi, however, two states can have both even and odd length paths between them. In that case, frontier search with delayed duplicate detection can fail by

leaking into the interior of the search. One solution to this problem is to store two complete levels of the search at a time. See our full paper for the details [Korf, 2003].

Using this algorithm, we were able to exhaustively search all problems through 18 disks, starting with all disks on one peg. Table 2 shows the results. The first column gives the number of disks, and the second column shows the minimum number of moves required to move all the disks from one peg to another. The third column gives the total number of states of the problem, which is 4^n, where n is the number of disks. The fourth column shows the maximum width of the graph, starting from a state with all disks on one peg. In all cases, the optimal number of moves equals the conjectured minimal number of moves. The 18-disk problem took almost 6 days to run, and at 8 bytes per state required about 30 gigabytes of disk space. Note that the ratio of the total number of states to the problem width for the 18-disk problem is almost 34.5.

Disks	Moves	Total States	Max Width
1	1	4	3
2	3	16	6
3	5	64	30
4	9	256	72
5	13	1,024	282
6	17	4,096	918
7	25	16,284	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	129	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

Table 2: 4-Peg Towers of Hanoi Results

5 Conclusions and Further Work

We showed that delaying the detection of duplicate nodes in a breadth-first search can effectively make use of very large capacity disk-storage systems. We also showed how to combine this idea with frontier search. In addition, we demonstrated that these ideas can also speed up a search that occurs entirely in memory. Using these algorithms, we completed breadth-first searches of sliding-tile puzzles with up to 43 billion nodes, and 4-peg Towers of Hanoi problems with up to 68 billion nodes. For the 4-peg Towers of Hanoi, we verified a conjecture regarding optimal solution lengths to 18 disks.

Current work is focussed on implementing these techniques for more complex best-first search algorithms such as Dijkstra's algorithm [Dijkstra, 1959] and A* [Hart, Nilsson, & Raphael, 1968].

6 Acknowledgments

Larry Taylor brought the 4-Peg Towers of Hanoi problem to my attention, and also used disk storage in a breadth-first search to find duplicate operator strings [Taylor & Korf, 1993]. Thanks to Jianming He for verifying the Towers of Hanoi results. 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.

References

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

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

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

[Korf & Zhang, 2000] 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, 1985] Korf, R. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97-109.

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

[Korf, 2003] Korf, R. 2003. Delayed duplicate detection: initial results. In *Proceedings of the IJCAI-03 Workshop on Model Checking and Artificial Intelligence.*

[Schofield, 1967] 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.

[Szegedy, 1999] Szegedy, M. 1999. In how many steps the k peg version of the towers of hanoi game can be solved? In *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS '99), LNCS 1563.* Trier, Germany: Springer-Vcrlag.

[Taylor & Korf, 1993] Taylor, L., and Korf, R. 1993. Pruning duplicate nodes in depth-first search. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-93),* 756-761.

[van de Liefvoort, 1992] van de Liefvoort, A. 1992. An iterative algorithm for the reve's puzzle. *The Computer Journal* 35(I):91-92.