A Breadth-First Approach to Memory-Efficient Graph Search

Rong Zhou
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
rzhou@parc.com

Eric A. Hansen
Dept. of Computer Science and Eng.
Mississippi State University
Mississippi State, MS 39762
hansen@cse.msstate.edu

Abstract

Recent work shows that the memory requirements of A* and related graph-search algorithms can be reduced substantially by only storing nodes that are on or near the search frontier, using special techniques to prevent node regeneration, and recovering the solution path by a divide-and-conquer technique. When this approach is used to solve graph-search problems with unit edge costs, we have shown that a breadth-first search strategy can be more memory-efficient than a best-first strategy. We provide an overview of our work using this approach, which we call breadth-first heuristic search.

Introduction

It is well-known that the scalability of the A* graph-search algorithm is limited by its memory requirements. A* stores all explored nodes of a search graph in memory, using an Open list to store nodes on the search frontier and a Closed list to store already-expanded nodes. This serves two purposes. First, it allows the optimal solution path to be reconstructed after completion of the search by tracing pointers backwards from the goal node to the start node. Second, it allows nodes that have been reached along one path to be recognized if they are reached along another path, in order to prevent duplicate search effort. It is necessary to store all explored nodes in order to perform both functions, but not to perform just one. This leads to two different strategies for reducing the memory requirements of graph search: one strategy gives up duplicate detection and the other gives up the traceback method of solution reconstruction.

Linear-space variants of A* such as depth-first iterative-deepening A* (DFIDA*) (Korf 1985) and recursive best-first search (RBFS) (Korf 1993) give up duplicate detection. Instead of storing Open and Closed lists, they use a stack to organize the search and rely on depth-first search of the graph to minimize memory use. Since the current best solution path is stored on the stack, solution reconstruction by the traceback method is straightforward. But because they only store nodes on the current path, these algorithms are severely limited in their ability to recognize when newly-generated nodes represent states that have already been reached by a different path. Essentially, linear-space search algorithms convert graph-search problems into tree-search problems. As the depth of the search increases, the number of node regenerations relative to the number of distinct nodes (i.e., the size of the search tree relative to the size of the search graph) can increase exponentially. For complex graph-search problems in which the same state can be reached along many different paths, DFIDA* and RBFS perform very poorly due to excessive node regenerations. Their performance can be improved by using available memory to store as many generated nodes as possible in order to check for duplicates, but this requires as much memory as A* in order to eliminate *all* duplicate search effort.

A second strategy for reducing the memory requirements of A* prevents duplicate search effort, but does not use the traceback method of solution reconstruction. It is based on the development of techniques for duplicate detection that do not require storing all generated nodes, but only nodes that are on or near the search frontier. This strategy was introduced to the artificial intelligence community in the form of an algorithm called *frontier search*, which only stores the Open list and saves memory by not storing the Closed list (Korf *et al.* 2005). A closely-related algorithm called *sparse-memory graph search* stores only a small part of the Closed list (Zhou & Hansen 2003a). Instead of the traceback method, both algorithms use a divide-and-conquer technique to recover the solution path. In this technique, the search algorithm finds an intermediate node along an optimal path and uses it to divide the original problem into two subproblems – the problem of finding an optimal path from the start node to the intermediate node, and the problem of finding an optimal path from the intermediate node to the goal node. The subproblems are solved recursively by the same search algorithm until all nodes along an optimal solution path for the original problem are identified.

We recently showed that when this approach to memory-efficient graph search is adopted for search problems with unit edge costs, a breadth-first search strategy requires less memory than the best-first strategy of A*, among other advantages (Zhou & Hansen 2004; 2006). In the following overview, we summarize the background and key ideas of this approach and discuss its significance.

Divide-and-conquer solution recovery

We begin with a brief review of the divide-and-conquer technique for solution recovery that supports this approach to memory-efficient graph search. The technique was originally developed to reduce the memory requirements of a dynamic-programming algorithm for sequence comparison (Hirschberg 1975). Several years ago, Korf et al. (2000; 2005) introduced it to the AI community and generalized it for graph search. Called *frontier search*, this general approach has been used to reduce the memory requirements of breadth-first search, Dijkstra's single-source shortest-path algorithm, A*, and bidirectional A*.

Frontier search only stores the Open list, and saves memory by not storing the Closed list (Korf *et al.* 2005). It uses special techniques to prevent regeneration of previously closed nodes that are no longer in memory. In addition, since the traditional traceback method of solution recovery requires all closed nodes to be in memory, frontier search uses divide-and-conquer solution recovery. Each node past the middle of the search space stores (via propagation from its parent node) all state information about a node along a best path to it that is about halfway between the start and goal nodes. The halfway point in the search space can be estimated in various ways. For example, it can be estimated by comparing a node's g-cost to its h-cost, or by comparing its depth to an estimate of the optimal solution length. After a goal node is expanded, the midpoint node along a best path is used as a pivot point for divide-and-conquer solution recovery. Two smaller search problems are solved, the problem of finding the best path from the start node to the midpoint node, and the problem of finding the best path from the midpoint node to the goal node. Typically, these problems are *much* easier to solve than the original search problem. Moreover, after one or two levels of recursion, there is usually enough memory to solve the subproblems without frontier search. As a result, solution recovery using this technique is very fast, and a significant amount of memory is saved in exchange for very little time overhead for solution recovery.

Limiting frontier size

Divide-and-conquer solution recovery, although a general technique, is often associated with the multiple sequence alignment problem. For many years, the bioinformatics community has used divide-and-conquer solution recovery to reduce the memory used by dynamic programming algorithms for aligning DNA and protein sequences (Myers & Miller 1988). After this technique was generalized for use by A* and other graph-search algorithms, one of the first problems it was applied to was multiple sequence alignment (Korf & Zhang 2000; Zhou & Hansen 2003a).

It is well-known that the multiple sequence alignment problem can be formalized as a shortest-path problem in a n-dimensional lattice graph, where n is the number of sequences to be aligned. A lattice is a special case of a partially-ordered graph, which means the nodes of the graph are partitioned into layers such that the nodes in one layer cannot have successor nodes in any of the previous layers. Hohwald et al. (2003) compare frontier A* to *bounded dynamic programming* in solving this problem. Bounded dynamic programming is an enhancement of the traditional dynamic programming approach to multiple sequence alignment that uses upper and lower bounds to limit the number of nodes that need to be evaluated to find an optimal alignment. Although bounded dynamic programming uses the same heuristic (i.e., lower-bound function) as A* to prune the search graph, it does not consider nodes in best-first order. Instead, it considers all nodes in one layer of the lattice graph before considering any nodes in the next layer. Unfortunately, bounded dynamic programming requires complex book-keeping and is very difficult to implement, and Hohwald et al. (2003) show that frontier A* outperforms it.

We introduced a search algorithm for multiple sequence alignment, called *Sweep A**, that can be viewed as a hybrid of bounded dynamic programming and A* (Zhou & Hansen 2003b). Like bounded dynamic programming, it considers all nodes in one layer of the lattice graph before considering any nodes in the next layer. But whereas bounded dynamic programming uses a complex indexing scheme to keep track of the nodes to be expanded in each layer, Sweep A* uses an Open list, like A*. The Open list it uses, however, shares the layered structure of the lattice graph, with nodes sorted in increasing order of f-cost in each layer. Nodes are expanded in best-first order within each layer, but the overall search is not best-first since all nodes in one layer are expanded before considering any nodes in the next layer.

Sweep A* is *much* easier to implement than bounded dynamic programming, and dramatically outperforms frontier A*. Not only is it faster, it can use two orders of magnitude less memory. The reason why it uses so much less memory is related to an important insight. When frontier search is used to reduce the memory requirements of graph search, the amount of memory used by the search algorithm depends on the size of the search frontier, and not on the number of nodes expanded. Although expanding nodes in best-first order, as A* does, results in the least number of node expansions, it does not result in the smallest frontier. Instead, the layer-by-layer expansion strategy of Sweep A* has a much smaller search frontier. We note that Thayer (2003), under the direction of Korf, independently developed a similar search algorithm for multiple sequence alignment, called *Bounded Diagonal Search*, that is based on this same insight.

Although Sweep A* can be used to solve other search problems with partially-ordered search graphs, it is not a general search algorithm. Nevertheless its effectiveness for this class of search problems led us to ask whether a similar layer-by-layer search strategy might be more memory-efficient than best-first search for other search problems.

Breadth-first heuristic search

Consider the large class of search problems with unit edge costs. For these problems, breadth-first search can be considered a layer-by-layer search strategy in which each layer of the search graph corresponds to a set of nodes that share the same g-cost. We now show that a breadth-first search strategy has a similar advantage over the best-first strategy of A* for this class of problems.

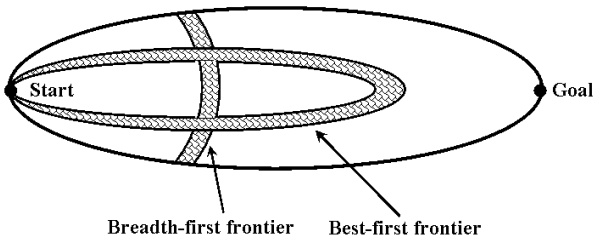

Figure 1: Comparison of best-first and breadth-first search frontiers. The outer ellipse encloses all nodes with f-cost less than or equal to an (optimal) upper bound.

A* uses an admissible node evaluation function (i.e., a lower-bound function) to select the order of node expansions and to prune the search space. Branch-and-bound search algorithms use the same lower-bound function to prune the search space, but expand nodes in either depth-first or breadth-first order. Although depth-first branch-and-bound search is effective for many problems, its inability to detect duplicates makes it ineffective for graph-search problems with many duplicate paths. Like A*, breadth-first branch-and-bound search stores generated nodes in order to detect duplicates. But it is usually considered inferior to A* because it requires an upper bound and must expand (and store) at least as many, and often more, nodes than A*.

When frontier search is used to reduce memory requirements, however, A* no longer has the same advantages over breath-first branch-and-bound search. A* is still optimal in terms of the number of node expansions; breadth-first branch-and-bound search still expands as many, and usually more, nodes. But in frontier search, as we have seen, memory requirements are determined by the size of the frontier, and not by the number of expanded nodes. Figure 1 illustrates, in an intuitive way, how frontier breadth-first branch-and-bound search can have a smaller search frontier than frontier A*. Best-first node expansion "stretches out" the search frontier, which includes nodes with many different g-costs, whereas breadth-first node expansion results in a compact frontier where nodes have only a couple different g-costs, and the width of the frontier is limited by upper and lower bounds. Empirical results for a range of search problems confirm this intuitive picture, and show that a breadth-first frontier can be significantly smaller than a best-first frontier (Zhou & Hansen 2006). Moreover, since a more informative heuristic tends to "stretch out" a best-first frontier further, the relative advantage of a breadth-first over a best-first approach to frontier search improves as the accuracy of the heuristic improves. We call a breadth-first approach to frontier search that uses the same lower-bound function as A* to prune the search space *breadth-first heuristic search*.

Breadth-first iterative-deepening A*

Unlike A*, use of breadth-first branch-and-bound search requires an upper bound as well as a lower-bound function. Often a good upper bound on the cost of an optimal solution can be found by running a fast, approximate search algorithm such as Weighted A* or beam search. When a good upper bound is not easily obtained, however, an iterative-deepening strategy can be adopted. This strategy avoids expanding nodes that have an f-cost greater than a hypothetical upper bound. The algorithm first runs breadth-first heuristic search using the f-cost of the start node as an upper bound. If no solution is found, it increases the upper bound by one (or to the least f-cost of any unexpanded nodes) and repeats the search. Because of the similarity of this algorithm to Korf's (1985) Depth-First Iterative-Deepening A* (DFIDA*), we call it *Breadth-First Iterative-Deepening A** (BFIDA*). The amount of memory it uses is the same as the amount of memory breadth-first branch-and-bound search would use given an optimal upper bound. Although running multiple iterations of breadth-first branch and bound takes extra time, BFIDA* is asymptotically optimal with respect to the number of node expansions in graphs (Zhou & Hansen 2006). By contrast, DFIDA* is only asymptotically optimal with respect to the number of node expansions in trees.

Layered duplicate detection

In addition to improved memory efficiency, the layer-by-layer approach to node expansion used by breadth-first heuristic search has other advantages. One of these is a simpler method of duplicate detection that we call *layered duplicate detection*. It involves keeping one or more previous layer(s) in memory to check for duplicates, rather than all layers. Although this is not always the most efficient approach to duplicate detection, it is simple and can be used when other approaches are not applicable.

Frontier search prevents regeneration of duplicate nodes by recording extra information (called *used-operator bits*) in the data structure of each node; used-operator bits indicate which neighboring nodes have been previously expanded. When applicable, this can be a very efficient approach to blocking node regeneration. But there are some cases in which this approach cannot be easily applied. For example, recent work shows that a symbolic approach to A* in which binary decision diagrams are used to compactly represent set of states can often significantly improve the scalability of A* (Jensen, Bryant, & Veloso 2002). But a symbolic A* algorithm does not represent individual states, and so the data structures used by frontier search to block regeneration of previously-expanded nodes cannot be used. In this case, breadth-first heuristic search with layered duplicate detection provides a simple, alternative approach to memory-efficient symbolic heuristic search (Jensen *et al.* 2006).

Directed graphs present another challenge. Duplicate detection is much easier in undirected graphs because every potential successor of a node is also a potential parent. As a result, in breadth-first search with layered duplicate detection, storing the current layer, the next layer, and the previous layer is sufficient to ensure all duplicates are detected. In undirected graphs, the used-operator bits used in frontier search are also sufficient to block node regeneration. But in directed graphs, these simple techniques are not enough to ensure that all duplicates are detected. In addition to used-operator bits, frontier search does the following. Each time a node is generated, it generates all predecessors of the node in the search graph, even if no path has been found yet to

these nodes. These *dummy nodes* are stored in the Open list with an f-cost of infinity until a path to them is found, at which point they acquire the f-cost of that path. Although this guarantees that no node is generated more than once, these dummy nodes would not be generated by standard A* and there is no bound on the number that could be generated.

In directed graphs, it may not be possible to know how many previous layers need to be stored in memory to detect all duplicates, or there may not be enough memory for all of the layers. But even if layered duplicate detection occasionally allows a node to be regenerated, it bounds the number of times this can happen. We have shown that in the worst case, the number of times a node n can be regenerated in breadth-first search with layered duplicate detection is bounded by

$$\left\lfloor \frac{f^* - g^*(n)}{\text{number of saved layers}} \right\rfloor,$$

where f^* is the length of the shortest solution path to the goal and $g^*(n)$ is the length of the shortest path to node n. This linear bound on the number of node regenerations is in sharp contrast to the potentially exponential number of node regenerations for search algorithms that rely on depth-first search. In practice, we have found that storing a single previous layer in memory is enough to eliminate most node regenerations in a wide range of directed graphs.

Another advantage of layered duplicate detection is that it is much easier to implement in a general-purpose graph-search algorithm. For example, we use breadth-first heuristic search with layered duplicate detection in a domain-independent STRIPS planner, and have shown that this approach leads to state-of-the-art performance.

Beam search

Breadth-first heuristic search with layered duplicate detection can significantly reduce the memory requirements of heuristic search. But its memory requirements are not bounded, and it is still possible to run out of memory if the number of nodes in any layer becomes too large.

If the largest layer (or adjacent layers) does not fit in memory, one way to handle this is to use beam search. Instead of considering all nodes in a layer, a beam search variant of breadth-first branch-and-bound search only considers the most promising nodes when memory is full. A drawback of beam search is that it is not guaranteed to find an optimal solution; in some cases, it may not find any solution, even when one exists. But this can be overcome by allowing it to backtrack to the points at which it could not generate all nodes in a layer, and continue the search from there. Based on this idea, we introduced a complete beam search algorithm called *beam-stack search* that integrates systematic backtracking with beam search (Zhou & Hansen 2005). We also developed a memory-bounded implementation, called *divide-and-conquer beam-stack search*, that stores only a couple layers of the search graph in memory and relies on divide-and-conquer solution recovery. This improves performance by allowing a large beam width, no matter how deep the search. Beam-stack search is an anytime algorithm that finds a good solution quickly and continues to search for improved solutions until convergence to an optimal solution.

Conclusion

Best-first search is traditionally considered more efficient than breadth-first search because it minimizes the number of node expansions. We have summarized some recent results that show that when a reduced-memory approach to graph search is adopted that stores only (or primarily) the search frontier, and relies on a divide-and-conquer method of solution recovery, a breadth-first search strategy can be more memory-efficient than a best-first strategy because the size of its search frontier is smaller. This has led to development of a family of search algorithms that includes a memory-efficient approach to breadth-first branch-and-bound search, a breadth-first iterative-deepening A* algorithm based on it, symbolic versions of these algorithms, a memory-efficient approach to heuristic search for partially-ordered graphs, and a beam-search algorithm that uses divide-and-conquer solution recovery to bound memory requirements.

References

Hirschberg, D. 1975. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* 18(6):341–343.

Hohwald, H.; Thayer, I.; and Korf, R. 2003. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, 1239–1245.

Jensen, R.; Hansen, E.; Richards, S.; and Zhou, R. 2006. Memory-efficient symbolic heuristic search. In *Proc. of the 16th Int. Conf. on Automated Planning and Scheduling (ICAPS-06)*.

Jensen, R.; Bryant, R.; and Veloso, M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *Proc. of the 18th National Conference on Artificial Intelligence (AAAI-02)*, 668–673.

Korf, R., and Zhang, W. 2000. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proc. of the 17th National Conf. on Artificial Intelligence (AAAI-00)*, 910–916.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

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

Korf, R. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Myers, E., and Miller, W. 1988. Optimal alignments in linear space. *Computer Applications in the Biosciences* 4(1):11–17.

Thayer, I. 2003. Methods for optimal multiple sequence alignment. Master's thesis, UCLA. Computer Sci. Dept.

Zhou, R., and Hansen, E. 2003a. Sparse-memory graph search. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1259–1266.

Zhou, R., and Hansen, E. 2003b. Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proc. of the 15th IEEE Int. Conf. on Tools with Artif. Intell. (ICTAI-03)*, 427–434.

Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, 92–100.

Zhou, R., and Hansen, E. 2005. Beam-stack search: Integrating backtracking with beam search. In *Proc. of the 15th Int. Conf. on Automated Planning and Scheduling (ICAPS-05)*, 90–98.

Zhou, R., and Hansen, E. 2006. Breadth-first heuristic search. *Artificial Intelligence* 170:385–408.