

Sweep A*: Space-Efficient Heuristic Search in Partially Ordered Graphs

Rong Zhou and Eric A. Hansen
Department of Computer Science and Engineering
Mississippi State University, Mississippi State, MS 39762
{rzhou,hansen}@cse.msstate.edu

Abstract

We describe a novel heuristic search algorithm, called Sweep A, that exploits the regular structure of partially ordered graphs to substantially reduce the memory requirements of search. We show that it outperforms previous search algorithms in optimally aligning multiple protein or DNA sequences, an important problem in bioinformatics. Sweep A* also promises to be effective for other search problems with similar structure.*

1. Introduction

Alignment of multiple protein or DNA sequences is a fundamental operation in bioinformatics. It is used to identify shared patterns among sequences that reflect common evolutionary history, structure or function. Although dynamic programming is the traditional method of aligning sequences, recent work has shown that the A* search algorithm outperforms dynamic programming in finding optimal alignments by using a lower-bound function (i.e., an admissible heuristic) to direct the search and limit how much of the state space must be explored [4, 10, 14, 9, 11, 3, 16].

It is well known that the limiting factor in scaling up A* is its memory requirement. A* keeps track of the explored part of the search space using an Open list to store nodes on the search frontier, and a Closed list to store already expanded nodes. The space needed to store all explored nodes can quickly fill available memory. Although limited-memory variants of A* such as IDA* [5] and RBFS [7] conserve memory by not storing Open and Closed lists, they are ineffective for complex graph-search problems with many duplicate paths, such as the multiple sequence alignment problem, due to excessive node regenerations. In such graphs, an ability to detect and avoid duplicate search effort is critical. Thus, recent work on improving the memory efficiency of A* for multiple sequence alignment has followed the approach of retaining the Open and Closed lists

in order to prevent duplicate search effort, but using various techniques to limit their size.

For the multiple sequence alignment problem, the size of the Open list is a particular concern. The relative size of the Open list compared to the Closed list grows with the branching factor of a search problem, and the branching factor for multiple sequence alignment is $2^n - 1$ where n is the number of sequences being aligned. One technique for reducing the size of the Open list uses partial node expansion to delay or avoid insertion of nodes into the Open list [14]. Another technique makes use of an upper bound on the optimal solution cost. If the lower-bound estimate (or f -cost) of any node is greater than the upper bound, the node is not inserted in the Open list because it cannot lead to an optimal solution and will never be expanded by A* [4, 15]. Both techniques have proved effective in improving the scalability of A* for multiple sequence alignment.

Other researchers have developed techniques for reducing the size of the Closed list. The Closed list performs two important functions in graph search. First, it allows an 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 already been reached along one path to be recognized if they are reached along another path, in order to prevent duplicate search effort. Inspired by Hirschberg's divide-and-conquer technique for reducing the memory requirements of dynamic programming for multiple sequence alignment [2, 12], Korf and Zhang [9] introduced a variant of A* called *Divide-and-Conquer Frontier A** (DCFA*) that prevents duplicate search effort without storing a Closed list; only an Open list is used. DCFA* has been shown to outperform a very efficient implementation of dynamic programming on a range of multiple sequence alignment problems [3]. Zhou and Hansen [16] describe a related search algorithm that stores a limited amount of nodes in the Closed list in exchange for some performance advantage. Because these algorithms do not store all expanded nodes in a Closed list, they cannot extract an optimal solution path in the conventional way of tracing back pointers. Instead, they employ

a divide-and-conquer strategy for reconstructing an optimal solution path after the search terminates. This requires storing nodes that are potentially in the middle of an optimal path, and, when the search ends, using a node in the middle of an optimal path to divide the original problem into two sub-problems. The sub-problems are solved by the same search algorithm to find nodes in the middle of their optimal paths, and the process continues recursively until all nodes along an optimal path have been identified. This method of solution reconstruction is very fast relative to the time required to solve the original search problem.

In this paper, we introduce a new variant of A* that limits the memory needed to store the Open and Closed lists by a combination of the above strategies, plus a novel strategy that has not been used before. We use the fact that the multiple sequence alignment problem has a great deal of structure that is not exploited by a general-purpose graph search algorithm such as A* or DCFA*. In particular, the search graph for the multiple sequence alignment problem is a lattice, which is a special case of a partially ordered graph. We introduce a variant of A* that is able to exploit this structure to further reduce memory requirements by adopting a novel strategy for node expansion that we call “sweeping” through the search space. We show that Sweep A* significantly outperforms previous space-efficient search algorithms for optimal multiple sequence alignment.

The paper is organized as follows. We begin with a discussion of partially ordered graphs and some examples of search problems with this structure. Then we describe the Sweep A* algorithm and report computational results that demonstrate its improved performance.

2. Partially ordered graphs

It is well-known that the multiple sequence alignment problem can be formulated as a shortest-path problem in a graph that corresponds to an n -dimensional lattice, where n is the number of sequences to be aligned [1]. Figure 2 shows an example of a two-dimensional lattice for a pairwise alignment problem, and Figure 1 shows an example of a three-dimensional lattice corresponding to a problem of aligning three sequences.

A lattice is a special case of a partially ordered graph. In this section, we define the concept of a partially ordered graph and consider several examples. In the rest of the paper, we show how this structure can be leveraged to improve search efficiency.

2.1. Definitions

Before defining a partially ordered graph, we begin with some preliminary definitions. Let \mathbf{G} be a directed graph (digraph) with a set of nodes \mathbf{N} and a set of edges \mathbf{E} .

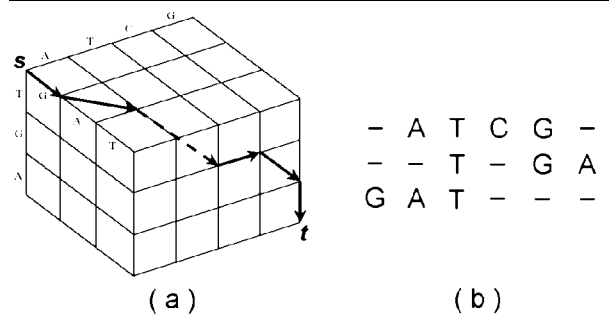


Figure 1. (a) The lattice for alignment of the three sequences ATCG, TGA and GAT, and (b) the alignment that corresponds to the path marked by a chain of arrows.

Let $Successors(n)$ be the set of successor nodes of node $n \in \mathbf{N}$, that is, the set of nodes that can be reached directly from n in the graph. Let $\phi(n)$ be the 1-step transitive closure of a node n defined as follows,

$$\phi(n) = n \cup Successors(n).$$

Similarly, the 2-step transitive closure $\phi^2(n)$ is defined as

$$\phi^2(n) = \bigcup_{n^1 \in \phi(n)} \phi(n^1),$$

and the k -step transitive closure $\phi^k(n)$ is defined as

$$\phi^k(n) = \bigcup_{n^{k-1} \in \phi^{k-1}(n)} \phi(n^{k-1}).$$

As $k \rightarrow \infty$, the k -step transitive closure becomes the *transitive closure*, which we denote as $\phi^*(n)$.

We generalize the above notation to sets of nodes as follows. The 1-step transitive closure of a set of nodes N is defined as

$$\Phi(N) = \bigcup_{n \in N} \phi(n),$$

and the transitive closure of N is defined as

$$\Phi^*(N) = \bigcup_{n \in N} \phi^*(n).$$

We say that a set of nodes X *precedes* another set of nodes Y if and only if X and the transitive closure of Y are disjoint but the transitive closure of X and Y are not disjoint, or mathematically,

$$X \prec Y \iff X \cap \Phi^*(Y) = \emptyset \wedge \Phi^*(X) \cap Y \neq \emptyset.$$

It can be shown that the above *precedence relation* (denoted by \prec) is irreflexive, asymmetric, and transitive. Note that $X \prec Y$ implies $X \cap Y = \emptyset$.

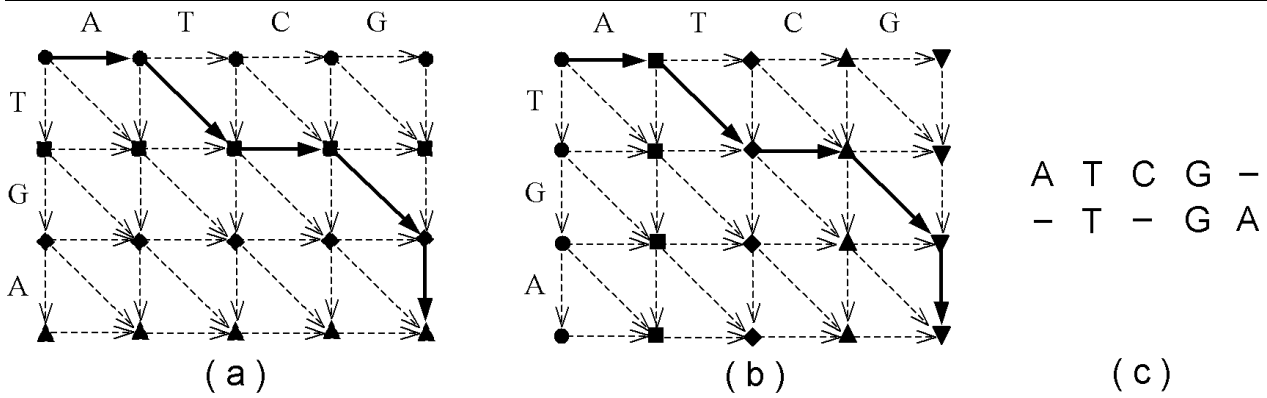


Figure 2. Two possible partially ordered state-space graphs for aligning two sequences TGA and ATCG. A path corresponding to the alignment shown in panel (c) is denoted by a chain of solid arrows. Panel (a) shows one possible definition of layers corresponding to sequence TGA, and panel (b) shows a different definition of layers corresponding to sequence ATCG. Using the longest sequence to define layers results in smaller layers.

We define a *partially ordered graph* as a digraph G with a set of nodes N and a set of edges E such that

$$N = \bigcup_{\ell=0}^T N_{\ell}, \forall 0 \leq \ell < T, N_{\ell} \prec N_{\ell+1}.$$

According to this definition, the set of nodes of a partially ordered graph can be partitioned into $T + 1$ disjoint subsets based on the precedence relation. Each of these subsets is called a *layer* of the partially ordered graph and is indexed by a number ℓ which we refer to as the *layer number*. Naturally, the layer number ℓ of a node n is defined as the layer number of N_{ℓ} that includes the node as an element. Abusing notation slightly, we use the function $\ell(n)$ to get the layer number of a node n . For convenience, we assume that the start node of a search problem is always in layer N_0 and a goal node is always in layer N_T . If this is not the case, it is obviously possible to remove all layers that precede the layer containing the start node or that follow the layer containing the goal node, in order to create a partially-ordered search graph that satisfies this assumption.

We define the *interleaving constant* Δ of a partially ordered graph as the minimum positive integer such that

$$\forall n, 0 \leq \ell < T - \Delta, n > \Delta \Rightarrow \Phi(N_{\ell}) \cap N_{\ell+n} = \emptyset.$$

The interleaving constant Δ describes the locality of a partially ordered graph by limiting the 1-step transitive closure of a layer within a certain range of subsequent layers that follow the current layer. A partially ordered graph is said to have no locality if and only if $\Phi(N_0) \cap N_T \neq \emptyset$, i.e., the 1-step transitive closure of N_0 intersects with the terminal layer N_T . In this case, we have $\Delta = T$. Note that the interleaving constant Δ of a partially ordered graph may vary depending on how its layers are defined.

2.2. Examples

To make our discussion of partially ordered graphs concrete, consider the following three examples. In the figures that accompany our examples, we use basic shapes to distinguish among different layers of a partially ordered graph. Only nodes with the same shape belong to the same layer.

2.2.1. Multiple sequence alignment Figure 2 shows two different partially ordered state-space graphs for the same pairwise sequence alignment problem. Using basic shapes to represent the layers of the graph (e.g. \bullet represents N_0 , \blacksquare represents N_1 , and so on), we see that the precedence relation defined for Figure 2(a) is: $\bullet \prec \blacksquare \prec \blacklozenge \prec \blacktriangle$. Figure 2(b) shows a different partial order for the same graph, which illustrates that there may be more than one way to define the layers of a partially ordered graph. The graph is divided into 4 or 5 layers depending on which sequence is used to determine the precedence relation that forms the layers.

2.2.2. Most probable state path Finding the most probable sequence of states in a hidden Markov model, given a sequence of observations, has applications in many areas that include speech recognition and bioinformatics. The state-space graph of this problem is also a partially ordered graph, as shown in Figure 3(a). The start node is located in layer N_0 denoted by \bullet and the goal node is located in layer $N_{T=4}$ denoted by \blacktriangledown . Note that the nodes in layer N_{ℓ} have immediate successors only in layer $N_{\ell+1}$ for all $0 \leq \ell \leq 3$, and so the interleaving constant Δ is 1.

2.2.3. Job sequencing The problem of job sequencing is to schedule N jobs sequentially on a machine such that a penalty function on job completion time is minimized. Sen et al. [13] show that A^* is effective in solving this problem. Figure 3(b) shows the partially ordered state-space graph

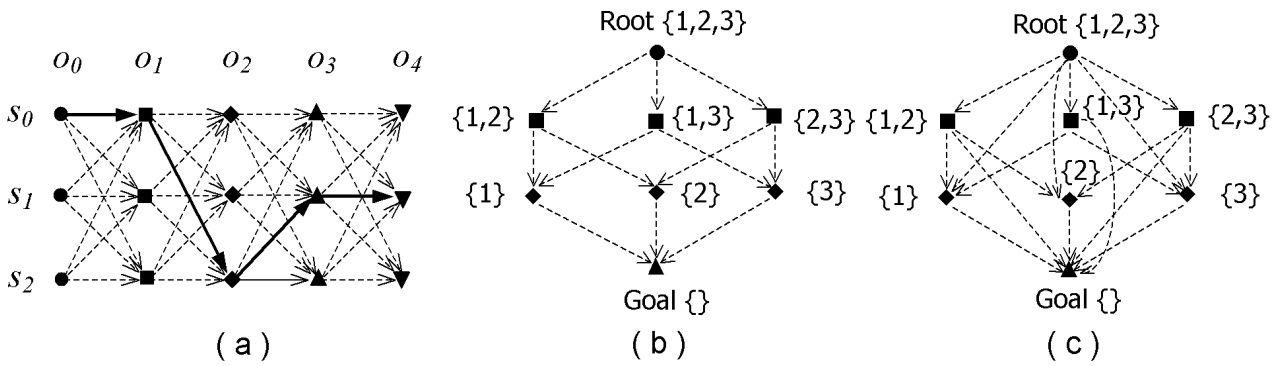


Figure 3. (a) shows a partially ordered state-space graph for determining the most probable state path in a hidden Markov model. The rows correspond to the hidden states and the columns to the sequence of observations of the Markov process. The other panels show partially ordered state-space graphs for a 3-job sequencing problem on a machine that can process (b) one job at a time, or (c) one or two jobs at a time. The numbers in braces represent jobs waiting to be scheduled.

for a simple 3-job sequencing problem in which only one job is processed on a machine at a time. When two jobs can be processed simultaneously on a single machine, the corresponding state-space graph shown in Figure 3(c) is more interesting because the interleaving constant Δ is greater than 1. In fact, Δ equals the maximum number of jobs that can be executed simultaneously on a single machine.

3. Sweep A* algorithm

We now introduce a variant of A*, called Sweep A*, that exploits the structure of partially ordered graphs to substantially reduce the memory requirements of search. Like other variants of A* that do not keep all closed nodes in memory [8, 9, 16], Sweep A* has two stages: first it searches forward from the start node to the goal node in order to find the optimal cost of a solution, as well as one or more intermediate nodes along an optimal path. Then it uses a divide-and-conquer approach to recursively reconstruct the optimal path. We consider these stages in turn.

3.1. Cost-only pass

The key idea of Sweep A* is to search a partially ordered graph on a layer-by-layer basis. That is, Sweep A* expands nodes in layer N_0 , then nodes in layer N_1 , and so on. We call this layer-by-layer search a “sweep” of the state space. When Sweep A* is expanding nodes in one layer, nodes in subsequent layers are not eligible for expansion, even if they have lower f -costs, until all eligible nodes in the current layer have been expanded. Thus, node expansion in Sweep A* does not follow a strictly best-first order. Best-first node expansion is honored only for nodes expanded in the same layer.

In keeping with this layer-by-layer strategy of node expansion, Sweep A* maintains multiple Open lists, each corresponding to a layer of the partially ordered graph. (We assume that Sweep A* always know which layer of the graph a node belongs to, so that it can insert each newly generated node in the proper list.) Let $Open_\ell$ denote the Open list for the currently expanding layer N_ℓ and let $Open_{\ell+i}$ denote the Open list for layer $N_{\ell+i}$. Recall that it is not possible for a node in N_ℓ to have immediate successors in layers that follow $N_{\ell+\Delta}$, where Δ is the interleaving constant. Thus, Sweep A* only needs to maintain Open lists for layers as far ahead as $N_{\ell+\Delta}$. (For the multiple sequence alignment problem and the definition of layers in Figure 2, the interleaving constant is 1 and Sweep A* only needs to maintain two Open lists.) Sweep A* also uses an upper bound U on the cost of an optimal solution to limit the size of the Open lists. No node with an f -cost greater than the upper bound is inserted into any Open list, since such a node cannot lead to an optimal solution. This results in significant space savings.

Since only nodes in $Open_\ell$ are selected for expansion, eventually $Open_\ell$ becomes empty, which tells Sweep A* to switch to the next layer for node expansion. The most important consequence of the layer-by-layer strategy for node expansion is that after Sweep A* starts expanding nodes in a layer, it never again considers nodes in any previous layer. Because of the precedence relation between layer N_ℓ and layer $N_{\ell+1}$, expanding any node in layer $N_{\ell+1}$ cannot create a path back to layer N_ℓ . This means that Sweep A* does not need to store any nodes in previous layers in order to prevent duplicate paths in the continued search. Closed nodes in previous layers can be removed from memory, and only nodes in the current layer are kept in the Closed list. In this way, Sweep A* achieves substantial space savings.

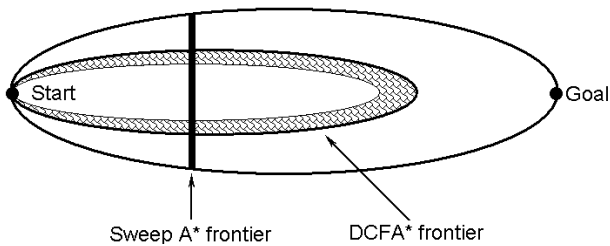


Figure 4. Comparison of frontiers of DCFA* and Sweep A*. The outer ellipse encloses all nodes with f -cost less than or equal to an upper bound.

3.2. Solution extraction

Since previous layers of the search are removed from memory, the problem is how to extract a solution path after the search ends. The way in which Sweep A* reconstructs a solution path is very similar to that of divide-and-conquer frontier A* (DCFA*) [9], which in turn is based on a technique developed by Hirschberg [2].

Let ℓ_m be the layer number of a “middle” layer that roughly divides the state space in half. Each node whose corresponding layer number is greater than ℓ_m keeps track of the state information of its ancestral node that is located in the middle layer. After a goal node is expanded and the search terminates, the stored ancestral state information is used as a divide-and-conquer point to break the original problem into two smaller subproblems, which are solved recursively. One subproblem has the original start node and takes the midpoint as the goal node. The other subproblem takes the midpoint as the start node and aims for the original goal node. The recursion ends when a subproblem is trivial, i.e., the least-cost path is a single edge.

Unlike DCFA*, each node of Sweep A* also stores the cost of the shortest path from the start node to its ancestral node in the middle layer. Note that the cost of the path from the ancestral node to the goal node can be computed by subtracting the previously-mentioned cost from the cost of the complete solution path, which is known when the goal node is expanded. Both costs are very useful because they provide *optimal* upper bounds for solving the subproblems. An optimal upper bound minimizes the number of node expansions required to find optimal solutions to the subproblems, improving the efficiency of solution reconstruction.

3.3. Complexity analysis

The space complexity of Sweep A* depends on the structure of the search graph for a particular problem. The amount of memory needed by Sweep A* is equal to the size of the largest layer in the graph, or more precisely, it is equal to the largest number of nodes that have an f -cost

less than the f -cost of the upper bound in any layer. As a rule, the more layers in a partially ordered graph, the greater the memory efficiency of Sweep A*. If the layers are all of equal size, the factor by which Sweep A* reduces memory use is approximately equal to the number of layers. It is interesting to note that Sweep A* functions correctly in any graph, since any graph can be thought of as a partially ordered graph with a single layer N_0 and an interleaving constant of $\Delta = 0$. But if the search graph does not have more than one layer, Sweep A* acts exactly like A* with upper-bound pruning and provides no extra memory savings.

We can give a more precise analysis of the space complexity of Sweep A* for the multiple sequence alignment problem. The partially ordered state-space graph has $O(l^n)$ nodes divided into l layers, where n is the number of sequences being aligned and l is the length of the longest sequence. The interleaving constant Δ of the partially ordered state-space graph is 1 (assuming the definition of layers in Figure 2), which means that the maximum number of layers of the partially ordered graph that need to be stored in memory by Sweep A* is $\Delta + 1 = 2$. Since this is a constant, it can be ignored in the asymptotic complexity analysis, and the space complexity of Sweep A* for multiple sequence alignment is the space complexity of a single layer of the partially ordered graph, which is $O(l^{n-1})$. This is the same as the space complexity of DCFA* [9]. But as we will see, Sweep A* has the further advantage that by expanding nodes on a layer-by-layer basis, it reduces memory use further by reducing the size of the search frontier.

3.4. Relation to best-first search

Unlike a traditional best-first search algorithm like DCFA*, Sweep A* does not explore the search space in a strictly best-first order determined by the node evaluation function f . Strictly best-first node expansion prevents DCFA* from exploiting structural regularities in the search graph. Instead, Sweep A* explores the nodes of the search graph in an order that reflects the precedence relation among layers of a partially ordered graph. Figure 4 shows a picture that gives some intuition about how this alternative strategy can reduce memory use.

Both DCFA* and Sweep A* store nodes that are on the frontier of the search. When nodes are expanded in order of the node evaluation function f , as by DCFA*, the frontier tends to have the stretched-out shape shown in Figure 4. But when nodes are expanded in an order that respects the layered structure of a partially ordered graph, the frontier has the more regular shape indicated by a vertical line in Figure 4. As this illustration suggests, the more regular-shaped frontier is in some sense “smaller” and can be represented by a smaller set of nodes. This intuition is born out in experimental results.

Length	DCFA*			Sweep A*			Sweep A* (Opt.)		
	Exp(K)	Stored(K)	Secs	Exp(K)	Stored(K)	Secs	Exp(K)	Stored(K)	Secs
4,000	27,293	1,953	1,435	57,350	51	1,543	27,728	25	1,043
5,000	56,150	3,159	2,437	113,446	81	2,958	56,842	41	2,002
6,000	102,632	4,628	6,329	206,616	122	5,230	103,655	61	3,551
7,000	162,984	6,252	10,546	321,823	155	6,936	164,367	83	4,963
8,000	259,170	8,461	18,319	487,501	205	10,146	261,001	113	7,362

Table 1. Comparison of DCFA*, Sweep A* using a sub-optimal upper bound found by modified RTA*, and Sweep A* using an optimal upper bound. Results are for aligning 3 random sequences of length 4000 through 8000. The comparison shows the number of nodes expanded during the cost-only pass, in thousands (Exp); the maximum number of nodes stored at any time, in thousands (Stored); and the running time in CPU seconds (Secs).

3.5. Upper bound computation

We have seen that Sweep A* can use an upper bound on the cost of an optimal solution to prune the search space. An upper bound is obtained by finding an approximate solution to the search problem. The closer the solution is to optimal, the more effective the bound will be in pruning the search space. But since the time and space needed to find an approximate solution must be added to the overall search cost, we must consider the tradeoff between the quality of the upper bound and the resources needed to find it.

There are many possible ways of computing an approximate alignment to obtain an upper bound. The approach we describe can be used for any search problem, and not only sequence alignment. We propose a variation of real-time A* (RTA*), a search algorithm that creates a solution path in stages [6]. We modify the algorithm so that it uses all available memory, but never exceeds it. Our version of RTA* searches forward from the current state until a memory limit is reached. Then it selects the node with the lowest f -cost in the Open list as a new start state, adds the best path from the old start state to the new start state to the overall solution, deletes the Open and Closed lists to recover memory, and continues the search from the new start state. This process repeats until a goal state is reached. Although we have found that this is an effective way to find a close-to-optimal solution, any method for computing an upper bound can be used with Sweep A*.

3.6. Iterative-deepening bounds

It is possible to define a version of Sweep A* that does not need a previously-computed upper bound. Instead, it uses an iterative-deepening strategy to avoid expanding nodes that have an f -cost greater than a hypothetical upper bound. The algorithm first runs Sweep A* using the f -cost of the start node as an upper bound. If no solution is found, it increases the upper bound by a variable called *step-size* and repeats the Sweep A* search. This process con-

tinues until Sweep A* finds a solution. This variation of Sweep A*, which we call *Iterative-Deepening Sweep A**, can minimize memory use. That is, the amount of memory it uses is the same as the amount of memory Sweep A* would use given an optimal upper bound (assuming *step-size* is chosen appropriately). However, Iterative-Deepening Sweep A* may run more slowly than Sweep A* with a previously-computed upper bound, because it takes extra time to run multiple iterations of Sweep A*.

4. Computational results

Sweep A* can solve any search problem that has a partially ordered search graph. In this section, we consider its performance in solving the multiple sequence alignment problem. We consider two sets of test problems used previously in the literature. The experiments we report were performed on a 300Mhz Sun UltraSparc II workstation with 2 gigabytes of RAM.

4.1. Three random sequences

First, we consider the identical test domain used by Korf and Zhang [9]: alignment of three random sequences of lengths 4000 through 8000 using their simple cost function, with results averaged over 100 trials. For this test domain, the number of closed nodes is many times larger than the number of open nodes, due to the dissimilarity of the sequences and the relatively low branching factor. As a result, no general-purpose search algorithm has been shown to be as space-efficient as DCFA*. Table 1 compares the performance of DCFA* to Sweep A*. Results are shown for Sweep A* using both a sub-optimal upper bound (computed by our modified RTA* algorithm) and using the cost of an optimal solution as an upper bound. This comparison shows how the quality of the upper bound affects the performance of Sweep A*.

When Sweep A* does not have an optimal upper bound (and it will rarely have one in practice), it expands more

# of seqs.	PEA*		IDSweep A*	
	Stored (K)	Secs	Stored (K)	Secs
6	18	11	2	85
7	67	141	9	177
8	280	9,375	37	3,156

Table 2. Comparison of A* with Partial Expansion (PEA*) and Iterative-Deepening Sweep A* (IDSweep A*) in aligning protein sequences of length about 450 from [14].

nodes than DCFA* or A*. The reason for this is that Sweep A* does not expand nodes in a strictly best-first order. But although expanding nodes on a layer-by-layer basis may conflict with a strictly best-first order, it dramatically reduces the amount of memory used, as clearly shown in Table 1. Sweep A* uses roughly 2% of the memory used by DCFA*. Moreover, even when Sweep A* expands twice as many nodes as DCFA*, it often runs as fast or faster. There are a couple of reasons for this.

First, duplicate detection is more efficient in Sweep A* than in DCFA* because Sweep A* only searches a small Open list and Closed list for each node generated, whereas DCFA* searches a much larger, monolithic Open list for duplicates. Second, Sweep A* can have better CPU cache performance because it expands nodes in the current layer before it goes to the next layer. Nodes in the same layer are often stored close to each other, increasing the chance of CPU cache hits.

4.2. Six to eight similar protein sequences

The second test domain we consider is alignment of six to eight sequences from a set of similar protein sequences of length about 450 used by Yoshizumi *et al.* [14]. The cost function is the PAM-250 matrix with a linear gap penalty of 8. In this test domain, the number of open nodes is many times larger than the number of closed nodes, due to the close similarity among the sequences and the high branching factor. As a result, no search algorithm has been shown to be more space-efficient than A* with Partial Expansion (PEA*) [14] which partially expands nodes in order to reduce the size of the Open list. Table 2 compares its performance to the performance of Iterative-Deepening Sweep A* using a fixed *step-size* of 50. We set the *cutoff value* of A* with Partial Expansion to zero, since this minimizes memory use. Results are averaged over 100 trials. Table 2 shows that Iterative-Deepening Sweep A* stores only 11% to 14% of the nodes stored by PEA*.

It is also interesting to compare the running time of the two algorithms. Because Iterative-Deepening Sweep A*

needs multiple iterations to find a solution, PEA* runs 6.7 times faster in aligning six sequences. But it runs only 26% faster in aligning seven sequences, and approximately three times slower than Sweep A* in aligning eight sequences. Because PEA* allows partially expanded nodes, the same node may be expanded multiple times before it is closed, and this slows performance. The average number of expansions for a node is influenced by (a) the cutoff value and (b) the branching factor. As the number of sequences being aligned increases from six to eight, the branching factor grows from $2^6 - 1$ to $2^8 - 1$. With a cutoff value of zero, the extra overhead of node expansion in PEA* is likely to increase exponentially in the number of sequences. On the other hand, Iterative-Deepening Sweep A* does not have this extra overhead, which effectively compensates for the multiple iterations it requires to find a solution. This gives Iterative-deepening Sweep A* an overall advantage over PEA* as the number of sequences being aligned increases.

Sweep A* also has the potential to perform more efficiently if it has a good upper bound. For example, if Sweep A* has an optimal upper bound, it stores only 7.5% of the nodes needed by PEA* and runs 20% faster in aligning 7 sequences.

5. Conclusion

We have described a novel heuristic search algorithm called Sweep A* that substantially reduces the space complexity of search in partially ordered graphs by expanding nodes in an order that reflects the layered structure of the search graph. We have shown that Sweep A* outperforms previous space-efficient algorithms for multiple sequence alignment by an order of magnitude or more in terms of memory requirements.

The algorithm presented in this paper may be improved in a couple of ways. We used A* to search inside each layer. Memory requirements could be reduced further by using DCFA* [9] or Sparse-memory A* [16] to search inside each layer, allowing the Closed list of a layer to be pruned before all nodes in the layer have been expanded. Preliminary results indicate that this can reduce memory use by about half compared to the results reported in this paper. In solving the multiple sequence alignment problem, it may also be possible to improve performance by defining diagonal layers in the search graph, instead of the horizontal and vertical layers illustrated in Figure 2, since an optimal path follows a diagonal direction from the upper left-hand corner to the lower right-hand corner of the lattice.

Finally, a search strategy similar to that of Sweep A* could be used to leverage problem structure in more general graphs. This extension will be described in future work.

Acknowledgments This work was supported in part by NSF CAREER grant IIS-9984952.

References

- [1] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988.
- [2] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
- [3] H. Hohwald, I. Thayer, and R. Korf. 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)*, pages 1239–1245, 2003.
- [4] T. Ikeda and H. Imai. Enhanced A* algorithms for multiple alignments: Optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science*, 210(2):341–374, 1999.
- [5] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
- [6] R. Korf. Real-time heuristic search. *Artificial Intelligence*, 42:197–221, 1990.
- [7] R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62:41–78, 1993.
- [8] R. Korf. Divide-and-conquer bidirectional search: First results. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1184–1189, Stockholm, Sweden, 1999.
- [9] R. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 910–916, 2000.
- [10] M. Lermen and K. Reinert. The practical use of the A* algorithm for exact multiple sequence alignment. *Journal of Computational Biology*, 7(5):655–671, 2000.
- [11] M. McNaughton, P. Lu, J. Schaeffer, and D. Szafron. Memory-efficient A* heuristics for multiple sequence alignment. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 737–743, 2002.
- [12] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.
- [13] A. Sen, A. Bagchi, and R. Ramaswamy. Searching graphs with A*: Applications to job sequencing. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 26:168–173, 1996.
- [14] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 923–929, 2000.
- [15] R. Zhou and E. Hansen. Multiple sequence alignment using Anytime A*. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 975–976, 2002.
- [16] R. Zhou and E. Hansen. Sparse-memory graph search. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 1259–1266, 2003.

Appendix

Notation

$s(n)$	State information of node n
$\ell(n)$	Layer number of node n
$m(n)$	Node n 's ancestral state information in middle layer
$u(n)$	Cost of the least-cost path from start node to $m(n)$
$c(u, v)$	Cost of edge from node u to v
$g(n)$	Cost from the start node to node n
$h(n)$	Estimated cost from node n to goal node
$f(n)$	Estimated cost from start to goal node via node n
U	Upper bound
ℓ_m	Layer number for the middle layer

Pseudocode for algorithm Sweep A*

```

Algorithm SweepA* (State start, State goal, Real  $U$ , Integer  $\ell_m$ )
1   $s(\text{root}) \leftarrow \text{start}$ ,  $g(\text{root}) \leftarrow 0$ ,  $\ell(\text{root}) \leftarrow 0$ 
2   $f(\text{root}) \leftarrow g(\text{root}) + h(\text{root})$ 
3   $\text{Open}_0 \leftarrow \{\text{root}\}$ ,  $\text{Open}_1 \leftarrow \dots \leftarrow \text{Open}_\Delta \leftarrow \emptyset$ 
4   $\text{Closed} \leftarrow \emptyset$ ,  $\ell \leftarrow 0$ 
5  while  $\exists 0 \leq i \leq \Delta \wedge \text{Open}_{\ell+i} \neq \emptyset$  do
6    while  $\text{Open}_\ell \neq \emptyset$  do
7       $n \leftarrow \arg \min_n \{f(n) \mid n \in \text{Open}_\ell\}$ 
8       $\text{Open}_\ell \leftarrow \text{Open}_\ell \setminus \{n\}$ ,  $\text{Closed} \leftarrow \text{Closed} \cup \{n\}$ 
9      if  $s(n) = \text{goal}$  then /* Extract a solution path */
10     if  $c(\text{start}, m(n)) = u(n)$  then /* trivial problem */
11        $\text{sol}_0 \leftarrow \{\text{start}, m(n)\}$ 
12     else
13        $\text{sol}_0 \leftarrow \text{SweepA}^*(\text{start}, m(n), u(n), \lfloor \ell(m(n))/2 \rfloor)$ 
14     if  $c(m(n), \text{goal}) = g(n) - u(n)$  then /* trivial problem */
15        $\text{sol}_1 \leftarrow \{\text{goal}\}$ 
16     else
17        $\text{sol}_1 \leftarrow \text{SweepA}^*(m(n), \text{goal}, g(n) - u(n),$ 
18          $\lfloor (\ell(n) - \ell(m(n)))/2 \rfloor)$ 
19     return  $\text{sol}_0 \cup \text{sol}_1$ 
20   if  $\ell(n) = \ell_m$  then  $u(n) \leftarrow g(n)$ 
21   for  $i = 0$  to  $\Delta$  do
22     for each  $x \in \text{Successors}(n) \wedge \ell(x) = \ell + i$  do
23       if  $g(n) + c(n, x) + h(x) > U$  continue /* skip */
24       if  $x \in \text{Open}_{\ell+i}$  then
25         if  $g(n) + c(n, x) < g(x)$  then
26            $g(x) \leftarrow g(n) + c(n, x)$ 
27           if  $\ell(x) > \ell_m$  then
28              $m(x) \leftarrow m(n)$ ,  $u(x) \leftarrow u(n)$ 
29         else if  $i \neq 0$  or  $x \notin \text{Closed}$  then /* new node */
30            $g(x) \leftarrow g(n) + c(n, x)$ ,  $\ell(x) \leftarrow \ell$ 
31           if  $\ell = \ell_m$  then
32              $m(x) \leftarrow s(x)$ 
33           else if  $\ell > \ell_m$  then
34              $m(x) \leftarrow m(n)$ ,  $u(x) \leftarrow u(n)$ 
35            $\text{Open}_{\ell+i} \leftarrow \text{Open}_{\ell+i} \cup \{x\}$ 
36   for each  $n \in \text{Closed}$  do
37     delete  $n$ 
38    $\text{Closed} \leftarrow \emptyset$ ,  $\ell \leftarrow \ell + 1$ 
39    $\text{Open}_{\ell+\Delta} \leftarrow \emptyset$ 
40 return  $\emptyset$ 

```