Sparse-Memory Graph Search

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 framework for reducing the space complexity of graph search algorithms such as A* that use Open and Closed lists to keep track of the frontier and interior nodes of the search space. We propose a sparse representation of the Closed list in which only a fraction of already expanded nodes need to be stored to perform the two functions of the Closed List - preventing duplicate search effort and allowing solution extraction. Our proposal is related to earlier work on search algorithms that do not use a Closed list at all [Korf and Zhang, 2000]. However, the approach we describe has several advantages that make it effective for a wider variety of problems.

1 Introduction

Graph search algorithms such as Dijkstra's algorithm and A* use an Open list to store nodes on the search frontier, and a Closed list to keep track of already expanded nodes. The Closed list performs two important fiinctions.

1. 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.
2. In graph search problems, 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.

Although both functions are important, storing all expanded nodes in a Closed list can quickly use all available memory.

If one is willing to give up the ability to recognize when the same node is reached along different paths, IDA* and RBFS can solve search problems in linear space [Korf, 1985; 1993]. Related search algorithms provide a limited ability to prevent duplicate search effort by storing a limited number of expanded nodes [Reinefeld and Marsland, 1994; Miura and Ishida, 1998]. In complex graph search problems with many duplicate paths, however, an inability to prevent *all* duplicate search effort often leads to poor performance.

Fortunately, it is not necessary to store all expanded nodes in a Closed list in order to eliminate all duplicate search effort. It is only necessary to store enough nodes to form a "boundary" in the search graph that prevents previously closed nodes from being revisited. This is the strategy adopted by a pair of recent search algorithms [Korf, 1999; Korif and Zhang, 2000], which are related to earlier work on reducing the memory requirements of dynamic programming algorithms for sequence comparison [Hirschberg, 1975; Myers and Miller, 1988]. If not all expanded nodes are stored in a Closed list, an alternative method of solution extraction is needed - since closed nodes along an optimal solution path may no longer be in memory when the search ends. So, these algorithms preserve information about nodes that are in the middle of a solution path. After the search ends, they use this information to divide the original problem into sub-problems - employing a divide-and-conquer strategy to recursively reconstruct an optimal solution path. This approach to reducing memory requirements results in an algorithm with two distinct phases; the first phase searches from the start to the goal node, and the second phase uses a divide-and-conquer method to reconstruct an optimal solution path.

In this paper, we describe some improvements of this search strategy. Earlier search algorithms that use this strategy do not store a Closed list at all [Korf, 1999; Korf and Zhang, 2000]. Instead they include extra information in open nodes, and even insert additional nodes into the Open list, to ensure that it forms a "boundary" that prevents "leaks" back into the closed region. Instead of adding information to the Open list, our algorithm preserves some already expanded nodes in a Closed list - but only enough to create a boundary and allow efficient solution reconstruction. Much of the closed region of the search can be removed from memory. Thus, we call this a sparse representation of the Closed list, and a sparse-memory approach to graph search. We show that it results in a more flexible approach to reducing memory requirements that offers several advantages.

2 Background

The most closely related algorithms to the search algorithm we introduce in this paper are Divide-and-Conquer Bidirectional Search (DCBS) [Korf, 1999] and Divide-and-Conquer Frontier Search (DCFS) [Korf and Zhang, 2000]. Although both are general graph-search algorithms, they were developed as an approach to performing multiple sequence alignment, and the memory-saving technique they use was first used by related dynamic programming algorithms for sequence comparison [Hirschberg, 1975; Myers and Miller, 1988]. To provide some perspective, we begin with a brief description of the multiple sequence alignment problem. For a detailed description of this problem, we refer to our references.

2.1 Multiple sequence alignment

Alignment of multiple DNA or protein sequences is an important problem in computational biology. It provides a way to measure the similarity between sequences and detect related segments. Alignment involves inserting gaps into sequences in order to maximize the number of matching "characters." It is well-known that this problem can be formalized as a shortest-path problem in a n-dimensional lattice, where *n* is the number of sequences to be aligned. The number of nodes in the lattice is l^n, where / is the average length of sequences, and thus grows polynomially with the length of sequences, and exponentially with the number of sequences. Although dynamic programming is the traditional method for solving this problem, A* has been shown to outperform it by using an admissible heuristic to limit the number of nodes in the lattice that need to be examined to find an optimal alignment [Ikeda and Imai, 1999; Lermen and Reinert, 2000]. But given the large number of nodes that usually must be examined, memory is a bottleneck, and techniques for solving this problem in reduced memory can be very helpful.

2.2 Divide-and-conquer frontier search

Both Divide-and-Conquer Bidirectional Search (DCBS) [Korf, 1999] and Divide-and-Conquer Frontier Search (DCFS) [Korf and Zhang, 2000] use the same strategy for reducing memory requirements. The difference is that DCBS uses bidirectional search and DCFS uses unidirectional search. Korf and Zhang [2000] find that unidirectional search results in better performance, and so we refer to DCFS in the rest of this paper.

DCFS reduces memory requirements by storing only the frontier nodes of the search, not the interior nodes - that is, by using an Open list but not a Closed list. Because it does not use a Closed list, DCFS must use some other method to avoid duplicate node expansions and to extract an optimal solution path at the end of the search. The problem of avoiding duplicate node expansions is particularly important in the case of multiple sequence alignment because there are combinatorially many paths from the start node to any other node. For this reason, linear-space search algorithms that do not test for duplicates, such as IDA*, have "pitiful results" (in the words of Korf [1999]), when used for multiple sequence alignment. Even bounded-memory search algorithms that detect some but not all duplicates have been reported to perform poorly on this problem [Yoshizumi *et ai,* 2000].

To avoid duplicate node expansions, DCFS uses the following techniques to prevent the search from "leaking" back into the closed region, that is, to prevent closed nodes from being re-generated. First, it stores in each node a list *of forbidden operators.* This list includes one operator for each neighbor (predecessor or successor) of this node that has already been generated. When the node is later selected for expansion, only operators that are not among the forbidden operators are used to generate its successor nodes. This prevents the search from re-generating nodes that may have been already closed and removed from memory. However, a complication arises in the case of directed graphs - in particular, directed graphs in which a node can have predecessors that are not also potential successors. If a node is expanded and removed from memory before all of its predecessors have been generated and inserted into the Open list, it can be regenerated if one of these predecessor nodes is later expanded. To prevent this, DCFS modifies the Open list in a second way. When a node is expanded, it generates not only its successor nodes, but all of its predecessor nodes - even if the search has not yet found a path to these predecessor nodes. If there is not yet a legal path to these predecessor nodes, they are assigned an infinite /-cost to prevent them from being expanded until a legal path is found.

To make it possible to reconstruct the solution path at the end of the search, DCFS modifies nodes in a further way. In each node past the midpoint of the search, it stores all information about a node on its path that is about halfway between the start and goal node. Then, when the search is completed, DCFS knows a middle node on the optimal solution path, and can reconstruct the solution path by a divide-and-conquer approach. Using the same search algorithm, it recursively solves the subproblems of finding an optimal path from the start node to the middle node, and then from the middle node to the goal node. This recursive method of solution reconstruction is essentially the divide-and-conqucr idea first proposed by Hirschberg [1975].

2.3 Limitations

Korf and Zhang [2000] show that DCFS can be very effective in "problem spaces that grow polynomially with problem size, but contain large numbers of short cycles," including path-planning in two-dimensional grids and multiple sequence alignment, assuming a small number of sequences. They acknowledge that their search algorithm is not as effective in an "exponential problem space with a branching factor of two or more." In that case, they explain, the Open list is much larger than the Closed list and "not storing the Closed list doesn't save much." In fact, the multiple sequence alignment problem is an exponential problem space (and NP-complete [Just, 2001]), when the number of sequences to be aligned is not bounded. Its branching factor is 2^n — 1 (where *n.* is the number of sequences), and the size of the Open list can dwarf the size of the Closed list when aligning as few as five or six sequences. This problem is well-known and has motivated development of techniques for reducing the size of the Open list when using A* to align multiple sequences. These include use of an upper bound to prune open nodes that cannot lead to an optimal solution [Ikeda and Imai, 1999], and use of partial node expansions [Yoshizumi *et al.,* 2000].

As it turns out, it is difficult to combine DCFS with techniques for reducing the size of the Open list without creating inefficiencies, or even "leaks" back into the closed region. It is possible to combine DCFS with Partial Expansion A*, a technique for reducing the size of the Open list by allowing partially expanded nodes [Yoshizumi *et al.,* 2000]. This can be beneficial in some cases. But allowing nodes to be partially expanded has the effect of reducing the number of nodes that are closed. In turn, this reduces the memory-saving effect of DCFS by reducing the number of nodes eligible to be removed from memory, as our experimental results will show.

Another approach to reducing the size of the Open list is to prune open nodes when their /-cost is greater than an upper bound on the optimal /-cost, as in Enhanced A* [Ikeda and Imai, 1999]. Because this technique does not reduce the number of closed nodes, the idea of combining it with DCFS appears more promising. However, it introduces other difficulties. Recall that DCFS generates all predecessors of a node when it is expanded, even if it has not yet found a path to these predecessors. If nodes are pruned from the Open list, no path may ever be found to these extra nodes, and the Open list may become cluttered with useless nodes that can never be removed. In directed graphs with the special property that the set of successors of each node is disjoint from the set of predecessors, such as the search graph of the multiple sequence alignment problem, this inefficiency is the only negative effect of pruning the Open list. But in directed graphs that do not share this property, and in all undirected graphs, pruning nodes from the Open list can also result in "leaks" back into the closed region, as follows. The first time a node is generated, there is no guarantee that the best path to it has been found. So, if it is pruned using an upper bound, it may later be re-generated if a better path to it is found. But since the forbidden operators associated with the node when it was first generated were lost when it was pruned, a node that was previously closed could be re-generated.

The difficulty of combining DCFS with techniques for reducing the size of the Open list is a significant limitation. Another potential limitation worth mentioning is the overhead for storing a list of forbidden operators in each node. For problems with a small branching-factor, this overhead is slight. But for the multiple sequence alignment problem, this overhead grows as the number of sequences grows. Recall that the number of operators (i.e., the branching factor) of multiple sequence alignment is $2^n - 1$, and DCFS stores incoming as well as outgoing edges in the list of forbidden operators. Thus, the total number of edges incident to a node is $(2^n - 1) \times 2 = 2^{n+1} - 2$. Although Korf and Zhang [2000] claim that the space complexity of DCFS for multiple sequence alignment is $O(l^{n-1})$, compared to the $O(l^n)$ space complexity of A*, this claim rests on the assumption that a node takes constant storage. As *n* increases, the storage required for the list of forbidden operators increases exponentially. In fact, the space complexity of DCFS for mulitple sequence alignment is $O(2^n l^{n-1})$, and its advantage over the $O(l^n)$ space complexity of A* disappears when $n > log_2 l$.

3 Sparse-Memory Search Algorithm

The algorithm we describe in the rest of this paper adopts a strategy for reduced-memory search that is similar to the strategy used by DCFS. However, it implements this strategy in a simpler way that is more compatible with techniques for reducing the size of the Open list, and does not require storing lists of forbidden operators in nodes.

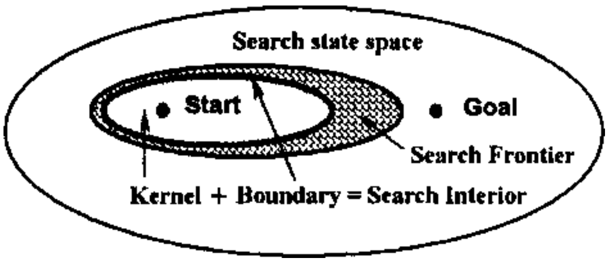

Figure 1: An illustration of the relationships among the kernel and the boundary of the search interior, the search frontier, and the entire state space.

A key difference from DCFS is that our algorithm does not entirely eliminate the Closed list. Instead, we propose a sparse representation of the Closed list that allows many (but not all) closed nodes to be removed from memory. To explain our approach, we note that the search interior (the set of closed nodes) can be partitioned into two disjoint subsets that we call the set of *boundary nodes* and the *kernel* of the search interior.

Definition 1 *Let I be the set of search interior nodes, i.e., nodes whose lowest-cost paths have been found. The* kernel *of I, denoted K (I), is defined as follows:*

$$\mathcal{K}(I) = \{k \mid k \in I, \forall p \in Pred(k) \Rightarrow p \in I\},$$

where Pred(k) denotes the set of predecessor nodes ofk, that is, the set of nodes that can make a transition into node k in the underlying graph, which may be directed or undirected.

Basically, the kernel is the set of nodes whose predecessor nodes are all interior (i.e., closed) nodes.

Definition 2 *The set of* boundary nodes *of search interior I, denoted B(I), is defined as the non-kernel nodes of I, that is:*

$$\mathcal{B}(I) = I \setminus \mathcal{K}(I).$$

Another way of describing a boundary node is to say that at least one of its predecessor nodes is not an interior node, or mathematically,

$$\mathcal{B}(I) = \{b \mid b \in I, \exists p \in Pred(b), p \notin I\}.$$

Figure 1 illustrates the relationship between the kernel and boundary of the search interior. Note that nodes in the boundary can enter the kernel, but once a node is in the kernel it remains there. The nodes in the kernel are eligible for removal from memory because they are not needed to prevent duplicate search effort.

The intuitive meaning of "boundary" can be explained as follows. Because every closed node is reachable from the start node, the set of closed nodes can be considered a "volume" in the underlying graph that encompasses the starting node. Nodes outside this volume cannot reach nodes inside the volume without passing through some node in the boundary. Thus, storing only the boundary nodes in the Closed list is as effective as storing the entire "volume," with respect to preventing the search from "leaking" back into the closed region. (From this perspective, one could say that DCFS adds nodes to the Open list - i.e., the search frontier - in order to ensure that the frontier forms a boundary, and it treats the Closed list as the kernel that can be removed from memory.)

In addition to keeping boundary nodes in the Closed list, our sparse-memory approach keeps some other, non-boundary nodes that allow solution reconstruction. The method of solution reconstruction is different from the one used by DCFS. Recall that DCFS stores in each node all information about a "middle" node on the path through this node to the goal. In our approach, each node in the search graph maintains a pointer to its predecessor node along the best path to this node, *or to some earlier, ancestral node along this path}* In the latter case, we have a *sparse solution path.*

Definition 3 *A* sparse solution path *is an ordered list of nodes* $P = \{P[0], P[1], \cdots, P[d]\}$,*such that* $d \geq 1$, $P[0]$ *is a start node,* $P[d]$ *is a goal node, and* $\sum_{i=0}^{d-1} \delta(P[i], P[i+1]) = \delta(start, goal)$, *where* $\delta(u, v)$ *denotes the cost of the lowest-cost path between two nodes* u *and* v.

If a node in a sparse solution path has a pointer to an ancestral node instead of a pointer to its predecessor, we call the ancestral node a *relay node* to indicate that it skips over some nodes in an original, "dense" solution path. We also consider the start node (which has no backward pointer) a relay node. How relay nodes are created and used in solution reconstruction is explained below. Here, we simply note that relay nodes are never removed from the Closed list because they are needed for solution reconstruction. Thus, a sparse representation of the Closed list includes all boundary nodes and all relay nodes. Nodes that do not fall in either category may be removed from memory.

It is well-known that A* can be viewed as Dijkstra's algorithm applied to a transformed graph with the same set of nodes and edges, but modified edge costs given by the equation $\hat{c}(u,v) = c(u,v) - h(u) + h(v)$, where $c(u,v)$ $(\text{or } \hat{c}(u,v))$ is the cost of edge (u,v) in the untransformed (or transformed) graph and $h(u)$ $(\text{or } h(v))$ is the cost estimate of the lowest-cost path from node *u* (or *v)* to the goal node. Therefore, we present our sparse-memory algorithm as a modification of Dijkstra's algorithm, and note that our description applies to A* also. Our sparse-memory approach requires two assumptions. First, the transformed (or untransformed) graph cannot contain any negative-cost edges. This means the heuristic used by A* must be consistent. Without this assumption, it is impossible for our algorithm (or any heuristic search algorithm) to accurately identify the interior of the search, and DCFS makes the same assumption. Second, we assume that our algorithm knows the in-degree (in the underlying graph) of every node it visits. For comparison, DCFS assumes that it knows every predecessor (in the underlying graph) of a node it visits. Thus, our second assumption is weaker than that made by DCFS.

```
Procedure ExpandNode (Node $u$)
for each $v \in Successors(u)$ do
    if $v \in Open$ then
        $\rho(v) \leftarrow \rho(v) - 1$
        if $g(u) + c(u,v) < g(v)$ then
            $g(v) \leftarrow g(u) + c(u,v)$
            $ancestor(v) \leftarrow u$
    else if $v \in Closed$ then
        $\rho(v) \leftarrow \rho(v) - 1$
    else /* generate new node */
        $\rho(v) \leftarrow in\text{-}degree(v) - 1$
        $g(v) \leftarrow g(u) + c(u,v)$
        $ancestor(v) \leftarrow u$
        $Open \leftarrow Open \cup \{v\}$
        if memory is full then PruneClosedList ()

Procedure PruneClosedList ()
for each $v \in Open \cup Closed$ do
    if $ancestor(v) \in Pred(v)$ then
        $\alpha \leftarrow ancestor(v)$
        while $\alpha \in Closed$ and $\rho(\alpha) = 0$ do
            $\alpha \leftarrow ancestor(\alpha)$
        if $\alpha \neq ancestor(v)$ then
            $ancestor(v) \leftarrow \alpha$
            $\rho(\alpha) \leftarrow \infty$ /* never prune relay node */
for each $v \in Closed$ do
    if $\rho(v) = 0$ then
        $Closed \leftarrow Closed \setminus \{v\}$
        Delete $v$

Algorithm SMGS (Node $start$, Node $goal$)
$g(start) \leftarrow 0$
$\rho(start) \leftarrow \infty$ /* never prune start node */
$ancestor(start) \leftarrow null$
$Open \leftarrow \{start\}$
$Closed \leftarrow \emptyset$
while $Open \neq \emptyset$ do
    $u \leftarrow \arg\min_u \{g(u) \mid u \in Open\}$
    $Open \leftarrow Open \setminus \{u\}$
    $Closed \leftarrow Closed \cup \{u\}$
    if $u$ is $goal$ then
        $SSP \leftarrow ExtractSparseSolutionPath(u)$
        $DSP \leftarrow \emptyset$
        for $i = 1$ to $length(SSP)$ do
            if $SSP[i-1] \in Pred(SSP[i])$ then
                $DSP.addTail(SSP[i])$
            else
                $DSP.addTail(SMGS(SSP[i-1], SSP[i]))$
        return $DSP$
    else
        ExpandNode ($u$)
```

Figure 2: Pseudocode for *SMGS* (Sparse-Memory Graph Search), in which procedure *ExpandNode* assumes a directed graph. See Figure 3 for the pseudocode of *ExpandNode* in undirected graphs.

Because pointers require less memory than state information about a midpoint node, the memory required to store all pointers plus relay nodes could be less than the memory required to copy state information about the same midpoint node into many other nodes.

3.1 Pruning the Closed list

Our search algorithm must be able to efficiently distinguish between the kernel and boundary of the search interior, in order to prune nodes from the kernel. Recall that a node is in the kernel if all of its predecessors are closed. To identify such nodes, we introduce a technique for keeping track of the number of unexpanded predecessors for each generated-and-stored node. We call this number the ρ-value of a node. It is initially set to the number of predecessors (the in-degree) of the node in the underlying graph minus one to account for the predecessor that generated it. The **ρ-value** is updated during node expansion. As each successor of a node is considered (some of which may already be in the Open or Closed lists), its p- value is decremented. (See lines 3, 8 and 10 of procedure ExpandNode in Figure 2.) This requires negligible time and space overhead, and, given **ρ-values,** kernel-membership for a node can be determined in constant time by checking whether it is a closed node with a **ρ-value** of zero.

An advantage of the sparse-memory approach is that it does not immediately remove closed nodes from memory, unlike DCFS. A sparse-memory version of Dijkstra's algorithm (or A*) acts exactly like Dijkstra's algorithm (or A*) until it senses that memory is about to be exhausted. Only then does it invoke procedure *PruneClosedList* in Figure 2 to recover memory. This procedure prunes nodes from the Closed list in two steps. First it updates the ancestral pointer of any boundary node whose predecessor is about to be pruned (lines 1-8). This is necessary to allow solution reconstruction and requires finding the *relay node* that is the closest boundary node along its solution path (lines 3-5), and updating its ancestral pointer accordingly (line 7). This makes this boundary node a relay node, and to prevent it from being pruned in the future, its **ρ-value** is set t(∞(line 8). After this step, kernel nodes are pruned unless they are a start node or relay node created in a previous pruning step (lines 9-12). Updating the ancestral pointers of nodes (followed by pruning) creates a sparse solution path, from which a complete or "dense" solution path can be reconstructed after the search terminates.

3.2 Solution Path Reconstruction

The fact that the Closed list is not pruned unless memory is close to being exhausted means that the overhead of solution reconstruction can be avoided if memory resources are adequate. In that case, the sparse-memory algorithm acts exactly like Dijkstra's algorithm (or A*) and an optimal solution path is extracted in the conventional way.

If the Closed list has been pruned, an optimal solution path is reconstructed by invoking the Sparse-Memory Graph Search algorithm (SMGS) in Figure 2 recursively. First the sparse solution path (SSP) is extracted (line 11) in the conventional way by tracing pointers backward from the goal. Then the corresponding *dense solution path* (DSP) is reconstructed as follows. For each pair of consecutive nodes $(SSP[i-1], SSP[i])$ in the *SSP* (starting from the start node), the algorithm checks to see if $SSP[i-1]$ is a predecessor of $SSP[i]$ (line 14). If so, n o$SSP[i]$ s added to the tail of the *DSP* (line 15); otherwise, the search algorithm calls itself recursively with $SSP[i-1]$ and $SSP[i]$ as the new start and goal nodes, in order to get a dense solution path between the two which is added to the tail of *DSP* (line 17).

```
Procedure ExpandNode (Node $u$)
1  $\rho(u) \leftarrow degree(u)$
2  for each $v \in Successors(u)$ do
3      if $v \in Open$ then
4          if $g(u) + c(u,v) < g(v)$ then
5              $g(v) \leftarrow g(u) + c(u,v)$
6              $ancestor(v) \leftarrow u$
7      else if $v \in Closed$ then
8          $\rho(v) \leftarrow \rho(v) - 1$
9          $\rho(u) \leftarrow \rho(u) - 1$
10     else /* generate new node */
11         $g(v) \leftarrow g(u) + c(u,v)$
12         $ancestor(v) \leftarrow u$
13         $Open \leftarrow Open \cup \{v\}$
14         if memory is full then PruneClosedList ()
```

Figure 3: Pseudocode for *ExpandNode* in undirected graphs.

Another difference from DCFS that is worth noting is that DCFS divides a problem into two subproblems at each level of the recursion. The sparse-memory approach can divide a problem into two *or more* subproblems. The extra flexibility is possible because the sparse-memory approach uses relay nodes with ancestral pointers, whereas DCFS stores all information about a middle node in each node. When a problem is divided into more than two subproblems, the subproblems are smaller and easier to solve and solution reconstruction can be faster. In fact, relay nodes can be spaced at half intervals, one-third intervals, or any other interval, and allow a tradeoff between the sparseness of the search interior and the speed of solution reconstruction.

3.3 Pruning the Open list

We now consider how easily the sparse-memory approach to pruning the Closed list can be combined with techniques for pruning the Open list. Pruning nodes on the Open list that have an /-cost greater than an upper bound on the optimal /-cost creates difficulties for DCFS, as discussed earlier. An advantage of the sparse-memory approach is that it does not create difficulties, at least in solving problems for which the set of predecessors of a node is disjoint from the set of successors, such as the multiple sequence alignment problem. This advantage will be illustrated in our experimental results.

For other directed graphs, that is, for directed graphs in which the same node can be both predecessor and successor of another node, DCFS allows leaks back into the closed region when the Open list is pruned. The sparse-memory approach does not, and this is another advantage. Unfortunately, the sparse-memory approach has a problem in such graphs. If the best path to a node has not been found when it is first generated, it could be pruned and re-generated later, when a better path is found. Since **ρ-values** are erased when the node is pruned, the **ρ-value** of such nodes will not be decremented to zero. As a result, the sparse-memory algorithm may not recover as much memory from the Closed list. However, our experimental results suggest that this inefficiency is minor.

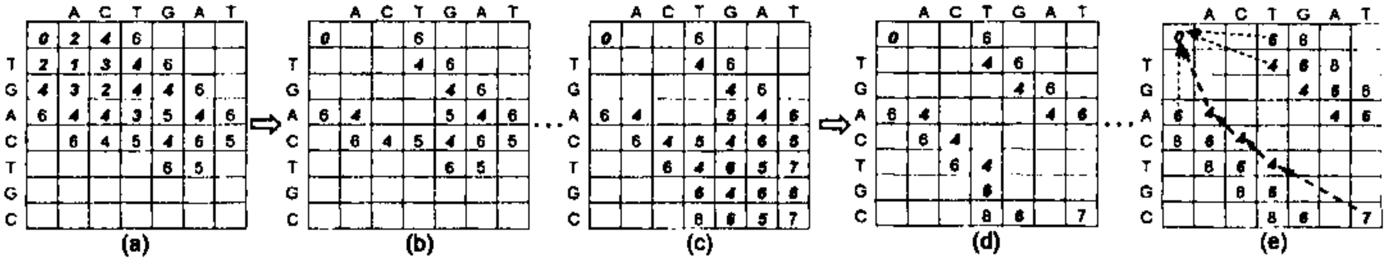

Figure 4: Example of Sparse-memory Dijkstra searching for an optimal alignment of two sequences. Panels (a) and (b) show the search space just before and after the first pruning of the Closed list. Panels (c) and (d) show the search space just before and after the second pruning. Panel (e) shows the sparse solution path at the end of the search.

In undirected graphs, we noted earlier that DCFS also allows "leaks" back into the closed region when the Open list is pruned. The sparse-memory algorithm does not, if we make a minor modification, as explained in the following.

3.4 Undirected graphs

The pseudocode in Figure 2 shows the *ExpandNode* procedure for directed graphs. Figure 3 gives the pseudocode of the *ExpandNode* procedure in undirected graphs. The difference is that in undirected graphs, ρ-**values** are only set and decremented for closed nodes, not open nodes. Thus, pruning open nodes cannot cause any problems by distorting ρ-values. In fact, the pseudocode in Figure 3 could be used for directed graphs also. But in directed graphs, it would require considering all predecessors of a node as well as successors during node expansion, in order to set ρ-**values** correct. The overhead for this would be significant. This is not a problem in undirected graphs because the set of potential successors of a node is equal to the set of potential predecessors.

3.5 Hybrid algorithms

The sparse-memory algorithm we have described is based on more than one idea. It uses nodes on the Closed list to create a "boundary" that prevents re-generation of closed nodes, instead of using the Open list as a boundary. In addition, it uses relay nodes for divide-and-conquer solution reconstruction, instead of storing in each node state information about a middle node along the search path.

Because these ideas can be considered separately, it is possible to create search algorithms that are hybrids of DCFS and the sparse-memory approach. For example, the DCFS technique of using the Open list as a boundary could be combined with relay nodes. This would create a version of DCFS that is able to divide a solution path into more than two pieces, allowing flexible (and potentially faster) solution reconstruction; use of relay nodes would also allow DCFS to delay pruning of the Closed list until memory is full. Similarly, some of the techniques used by DCFS can be considered independently. For example, forbidden operators can be used in combination with the sparse-memory approach, instead of ρ-**values.** One advantage of using forbidden operators is that they sometimes speed up search, since it is faster not to generate a node than to generate it and check for a duplicate on the Open or Closed list. We consider the performance of these hybrid algorithms in the computational results that follow.

4 Computational Results

We first illustrate how a sparse-memory version of Dijkstra's algorithm works by showing how it solves a small pairwise sequence alignment problem. Then we consider the performance of sparse-memory A* on more challenging problems.

4.1 Example

Consider the problem of aligning two sequences, ACTGAT and TGACTGC, using a very simple cost function: zero for a match, one unit for a substitution, and two units for a gap. The state space of the problem can be represented by a two-dimensional grid in which the columns correspond to one sequence and the rows to the other. The problem of finding an optimal alignment of the two sequences corresponds to the problem of finding an optimal path from the start node in the upper-left corner to the goal node in the lower-right corner, where horizontal and vertical moves correspond to gap insertions in one or the other sequence, and diagonal moves correspond to a substitution or match of characters.

Figure 4 shows the behavior of sparse-memory Dijkstra\s algorithm at the critical points when memory becomes full and the Closed list is pruned. It assumes that memory capacity is 30 nodes. Figure 4(a) shows the explored state space when memory is full for the first time. The number in each cell is the g-value of the corresponding state (or node). For closed nodes, the g-value is highlighted in bold italics. Among the 17 closed nodes, II are identified as kernel nodes and pruned. Figure 4(b) shows the result of pruning. When memory is full the second time, as shown in Figure 4(c), the Closed list is pruned again and 12 kernel nodes are removed from memory, as shown in Figure 4(d), freeing enough memory for continued search. Figure 4(e) shows the state space in memory when the goal node (the cell in the lower-right corner) is expanded. The SSP solution is shown as a chain of thick dashed arrows. The arrows drawn in thin dashed lines represent ancestor pointers that are created during pruning.

In this example, it is also possible to see how the Open list can be pruned using the sparse-memory approach. Suppose we know that 8 is an upper bound on the cost of an optimal alignment. Then all nodes with g-values greater than or equal to 8 can be pruned from the Open list, and the search progresses in the same way except for not storing the seven nodes whose g-values equal 8. Note that pruning nodes in the Open list will *not* change the boundary of the search interior.

4.2 Multiple sequence alignment

We tested Sparse-memory A* (Sparse-A*) on a series of challenging multiple sequence alignment problems, in order to compare its performance to DCFA* (the DCFS version of A*). We used a 300Mhz Sun UltraSparc II workstation with two gigabytes of RAM. Results are displayed in Table 1.

First, we considered the identical test domain used by Korf and Zhang [2000]: alignment of three random sequences of length 4000 using their simple cost function, with results averaged over 100 trials. Both DCFA* and Sparse-A* are effective in this domain, whereas A* could not solve any problem instance due to memory limitations. Versions of A* that use special techniques to recover memory by pruning the Open list [Ikeda and Imai, 1999; Yoshizumi *et al.,* 2000] were also unable to solve any of these problem instances. This is because the Closed list, not the Open list, fills most of memory in this case. One reason there are so many closed nodes is that the accuracy of the pairwise heuristic used for multiple sequence alignment depends on the similarity of the sequences. Because random sequences have only random similarities, the heuristic is weak, resulting in many node expansions and closed nodes.

For this problem set, DCFA* is more memory-efficient than Sparse-A*. However, Sparse-A* runs faster. It runs faster for long sequences like these because the solution path is correspondingly long, and solution reconstruction can be faster using relay nodes. At each recursion level, relay nodes make it possible to divide a solution path into several smaller, easier-to-solve subproblems, instead of always dividing it in half. Faster solution reconstruction comes at the expense of a small increase in memory for extra relay nodes, as can be seen by comparing the performance of DCFA* using relay nodes (a hybrid algorithm) to standard DCFA*.

Why can DCFA* be more memory-efficient than Sparse-A*? One factor is that using relay nodes to divide a solution path into more than two pieces requires extra memory for the extra relay nodes - a classic space-time tradeoff. Another factor is that use of forbidden operators allows a closed node to be pruned as soon as all its predecessors are *generated.* By contrast, use of p-values requires waiting until all predecessors are *closed,* before pruning the node. As a result, a hybrid algorithm that combines the sparse-memory approach with forbidden operators outperforms the sparse-memory approach alone, for this problem set. These two factors may not account for all of the difference in memory efficiency, but we currently have no other explanation.

We next compared Sparse-A* and DCFA* on real protein sequences using the PAM250 cost matrix, which is widely-used by biologists. In aligning five sequences randomly selected from a pool of low-similarity protein sequences of length 300 used in previous experiments [McNaughton *et ai,* 2002], Sparse-A* and DCFA* were again effective in solving all instances. By contrast, A* ran out of memory on most instances. For this problem set, the Enhanced A* algorithm [Ikeda and Imai, 1999], which uses an upper bound to prune nodes from the Open list, performs very well. This is because the number of open nodes is very large for this set of problems. Unlike DCFA*, the sparse-memory approach can be safely combined with this technique of pruning the Open list. This gives it an overall advantage, and Enhanced Sparse-A* is the best-performing algorithm for this problem set. (Although Enhanced Sparse-A* stores slightly more nodes than DCFA*, it uses less memory because forbidden operators makes the DCFA* nodes larger. We also emphasize the following point: although the table shows that the running time of Enhanced A* is less than the running time of Enhanced Sparse-A*, this does not mean that it is a faster algorithm. If Enhanced A* by itself can solve a problem in available memory, the sparse-memory version of Enhanced A* does not need to prune the Closed list at all, and has identical performance. But for these experiments, we adjusted the sparse-memory algorithm to minimize its memory use - since memory is the key factor we are evaluating here.)

Algorithm	Statistics	3 seqs. (4,000)	5 seqs. (300)	7 seqs. (450)
DCFA*	[Sees.	1,435	2,930	1721
	Nodes(K)	1,953	9,997	1,200
	1 Mbytes	284	551	112
DCFA* + relay nodes	Sees.	712	2,863	97
	Nodes(K)	2,202	10,910	1,215
	Mbytes	293	561	114
Sparse-A*	Sees.	908	3,073	65
	Nodes(K)	4,510	17,626	1,093
	Mbytes	389	766	72
Sparse-A* + forbidden ops.	Sees.	871	3,218	101
	Nodes(K)	3,388	13,100	1,071
	Mbytes	355	673	104
Enhanced A*	Sees.		1,940	24
	Nodes(K)	can't	16,844	76
	Mbytes	solve	720	28
Enhanced Sparse-A*	Sees.	917	2,785	24
	Nodes(K)	4,510	10,600	76
	Mbytes	389	465	28

Table 1: Performance comparison of DCFA*, sparse-memory A*, Enhanced A*, and hybrid algorithms. The node count is the number of stored nodes, measured in thousands, and memory includes storage of the pairwise heuristic.

A drawback of DCFA* is that its node size increases as the number of sequences being aligned increases. This becomes very apparent when DCFA* is used to align seven protein sequences of length around 450, randomly selected from a set of similar protein sequences used in previous experiments [Yoshizumi *et al,* 2000]. The high similarity of these sequences gives rise to a very accurate heuristic, and makes these sequences much easier to align. Even A* can solve these problems. Although A*, Sparse-A* and DCFA* always expand the same number of nodes, DCFA* generates and stores an average of 10% more nodes in this domain because it inserts extra nodes in the Open list. A more serious problem is that the nodes created by DCFA* are two times bigger than the nodes created by the other algorithms, because they include lists of forbidden operators. As a result, DCFA* runs slower and uses more memory than any other algorithm, including A*! (In aligning ten sequences, the nodes created by DCFA* are seven times larger than the nodes created by A*, and their relative size almost doubles with each additional sequence thereafter.) The table shows Sparse-A* is not as effective as Enhanced A*, because pruning the Open list is much more important than pruning the Closed list, for this problem set. Nevertheless, Enhanced Sparse-A* performs no worse than Enhanced A*, and this point is important. The sparse-memory approach improves performance when possible, and complements other techniques for reducing memory.

For perspective, we discuss what happens if we combine DCFA* with Partial Expansion A* [Yoshizumi *et al,* 2000]. (In our experiments, we set the cutoff value for Partial Expansion A* to zero, to minimize memory use.) For the set of seven similar protein sequences, Partial Expansion DCFA* required an average of 29 Mbytes of memory, and 437 CPU seconds. The much greater running time is due to the overhead of partial expansions. The slightly higher memory requirement is because nodes are partially expanded, and nodes that are not closed cannot be pruned by DCFA*. For the set of three random 4000-length sequences, Partial Expansion DCFA* stored 40% more nodes and ran 42% slower than DCFA*. Again, the reason is that partially expanded nodes are not pruned. Finally, we note that when DCFA* is combined with Enhanced A*, it consistently requires more memory (and CPU time) than DCFA* alone, due to the complications discussed earlier.

4.3 15-Puzzle

The 15-puzzle is not a problem for which duplicate detection is a crucial issue. Nevertheless, it is interesting to consider as both a benchmark and an example of an undirected graph. We used 91 of the 100 problem instances in Korf (1985) as a test set. Sparse-A* by itself used an average of 79% of the memory used by A*; DCFA* used an average of 44% of the memory used by A*; and Enhanced A* used an average of 57% of the memory used by A*. A sparse-memory version of Enhanced A* performed best. It used 32% of the memory used by A*. We also tested DCFA* combined with Enhanced A*. It is unable to prevent leaks back into the closed region, for reasons discussed earlier, and we noticed many node re-expansions. Interestingly, it used only 10% of the memory used by A*, due to aggressive pruning, and ran faster than the other algorithms. But this is only because the 15-puzzle is a domain in which node re-generation has less overhead than managing the Open and Closed lists, since IDA* outperforms all of these algorithms on this problem. In undirected graphs with exponentially many duplicate paths, a sparse-memory version of Enhanced A* is likely to perform best.

5 Conclusion

We have proposed a sparse-memory approach to graph search that builds on ideas in earlier work, but implements them in a way that is often more effective. A key advantage of this approach to reducing the size of the Closed list is that it can be combined with a technique for reducing the size of the Open list by upper-bound pruning. This is especially useful for large branching-factor problems where the size of the Open list would otherwise dramatically exceed the size of the Closed list, such as the multiple sequence alignment problem when there are more than four or five sequences.

The sparse-memory approach has other advantages. It allows more flexible, and potentially faster, solution reconstruction. And last but not least, it can behave exactly like A* (or Enhanced A*, or Dijkstra's algorithm) until it reaches a memory limit, and only then removes nodes from the Closed list. Thus, there is (virtually) no overhead for this technique unless a search problem cannot be solved within a given memory bound. Then the overhead for solution reconstruction is compensated for by the reduced memory requirements.

Acknowledgments We thank the anonymous reviewers for helpful comments. This work was supported in part by NSF CAREER grant I1S-9984952 and NASA grant NAG-2-1463.

References

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

[Ikeda and Imai, 1999] T. Ikcda 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.

[Just, 2001] W. Just. Computational complexity of multiple sequence alignment with SP-score. *Journal of Computational Biology,* 8:615-623, 2001.

[Korf and Zhang, 2000] 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.

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

[Korf, 1993] R. Korf. Linear-space best-first search. *Artificial Intelligence,* 62:41-78,1993.

[Korf, 1999] R. Korf. Divide-and-conquer bidirectional search: First results. In *Proc. of the 16th InternationalJoint Conference on Artificial Intelligence (IJCAI-99),* pages 1184-1189, 1999.

[Lermen and Reinert, 2000] 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.

[McNaughton et al.,2002] M. McNaughton, P. Lu, J. Schaefler, 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.

[Miura and Ishida, 1998] T. Miura and T. Ishida. Stochastic node caching for memory-bounded search. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98),* pages 450-456, 1998.

[Myers and Miller, 1988] E. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences,* 4:11-17,1988.

[Reinefeld and Marsland, 1994] A. Reincfeld and T. Marsland. Enhanced iterative-deepening search. *IEEE Trans, on Pattern Analysis and Machine Intelligence,* 16:701 -710, 1994.

[Yoshizumi *et al,* 2000] 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.