From: AAAI-00 Proceedings. Copyright © 2000, AAAI (www.aaai.org). All rights reserved.

A* with Partial Expansion for large branching factor problems

Takayuki Yoshizumi, Teruhisa Miura and **Toru Ishida**

Department of Social Informatics, Kyoto University,
Kyoto 606-8501, Japan
{yosizumi, miura, ishida}@kuis.kyoto-u.ac.jp

Abstract

The multiple sequence alignment problem is one of the important problems in Genome Informatics. The notable feature of this problem is that its state-space forms a lattice. Researchers have applied search algorithms such as A* and memory-bounded search algorithms including SNC to this problem. Unfortunately, previous work could align only seven sequences at most. Korf proposed DCBDS, which exploits the features of a grid, and suggested that DCBDS probably solved this problem, effectively. We found, however, that DCBDS was not effective for aligning many sequences. In this paper, we propose a simple and effective search algorithm, A* with Partial Expansion, for state-spaces with large branching factors. The aim of this algorithm is to store only necessary nodes for finding an optimal solution. In node expansion, A* stores all child nodes, while our algorithm stores only promising child nodes. This mechanism enables us to reduce the memory requirements during a search. We apply our algorithm to the multiple sequence alignment problem. It can align seven sequences with only 4.7% of the stored nodes required by A*.

Introduction

The multiple sequence alignment problem is to align several biological sequences and to extract the common pattern. The alignment is used in various ways for biological sequence analysis in Genome Informatics. We can define the multiple sequence alignment problem as the problem of finding the shortest path in a lattice. The state-space is far different from those of typical search problems such as the sliding-tile puzzle and the maze. There are huge numbers of paths through the same node, because the state-space forms a lattice and the branching factor is $O(2^d)$, when d is the number of sequences to be aligned. The multiple sequence alignment problem has notable features that have not been dealt with in the AI search community.

Ikeda and Imai applied the A* algorithm to the multiple sequence alignment problem (Ikeda & Imai 1994). A* must store all child nodes and because of the large branching factors involved, the memory requirements of A* grow rapidly with search progress. Due to memory constraints, A* cannot align more than seven sequences. On the other hand, linear-space search algorithms such as IDA* (Korf 1985) cannot align more four sequences because of the large number of revisits. We proposed SNC (Miura & Ishida 1998), which can effectively reduce the number of revisits needed by IDA*. SNC, however, cannot align more than seven sequences. Korf was inspired by our research and proposed DCBDS (Korf 1999). This interesting algorithm exploits the features of a grid, stores only the Open list, and performs a series of bi-directional searches. Korf claimed that DCBDS is most effective for a state-space that grows polynomially with problem size, but contains large numbers of short cycles. We applied DCBDS to the multiple sequence alignment problem, which was mentioned as one of the important applications in his paper. We found that not storing the Closed list did not effectively reduce the memory requirements, because the Open list is much larger than the Closed list in the search-space in this case. This weakness is due to the wide distribution of edge costs and a relatively accurate heuristic function. What is worse, DCBDS cannot prevent the search from leaking back into the closed region, when the state-space is a directed graph.

We propose a simple and effective search algorithm, A* with Partial Expansion; it exploits the features of a lattice and effectively reduces the memory requirements. To evaluate the power of our algorithm, we apply it to the multiple sequence alignment problem. We show that it effectively reduces the memory requirements compared to A* and discuss the relation to other search algorithms.

The multiple sequence alignment problem

The alignment of many biological sequences is demanded in various important fields in molecular biology. A biological sequence is composed of alphabetic characters representing its constituents. For example, one protein sequence consists of 20 amino acids. Figure 1 shows a part of the aligned sequences. Hyphens, or gaps, are inserted into the sequences so that the same, or similar, characters occupy the same columns.

In the multiple sequence alignment problem, we want to find the optimal alignment, which is associated with a minimum cost. The cost of the alignment is given by the sum of the costs of pairwise alignments. In a pairwise alignment, the cost of each column is given by the modified PAM-250 matrix in which each sign of score is reversed (Figure 2). The

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

```
Hal   GASQADNAVLVVAA-D---D-GV-QP-QTQEHVFLARTLGIGELIVAVNKMD-L-VDYGESEYKQVEEV-KDLLTQVRFDSENAK
Met   GASQADAAVLVVNVDDA--KSGI-QP-QTREHVFLIRTLGVRQLAVAVNKMD-T-VNFSEADYNEIKKMIGDQLLKMIGFNPEQIN
Tha   GTSQADAAILVISARDG--E-GV-ME-QTREHAFLARTLGVPQMVVAINKMDATSPPYSEKRYNEVAKDA-EKLLRSIGFK-D-IS
Thc   GASQADAAVLVVAV-T---D-GV-MP-QTKEHAFLARTLGINNILVAVNKMD-M-VNYDEKKFKAVEAQV-KKLLMMLGYK-N-FP
Sul   GASQADAAILVVSAKKGEYEAGMSAEGQTREHIILSKTMGINQVIVAINKMDLADTPYDEKRFKEIDVTV-SKFMKSFGFDMNKVK
Ent   GTSQADVAILIVAAGTGEFEAGISKNGQTREHILLSYTLGVKQMIVGVNKMD-A-IQYKQERYEEIKKEI-SAFLKKTGYNPDKIP
Pla   GTSQADVALLVVPADVGGFDGAFSKEGQTKEHVLLAFTLGVKQIVVGVNKMD-T-VKYSEDRYEEIKKEV-KDYLKKVGYQADKVD
Sty   GTSQADAAILIIASGQGEFEAGISKEGQTREHALLAFTMGVKQMIVAVNKMDDKSVNWDQGRFIEIKKEL-SDYLKKIWLQPRQDP
```

Figure 1: A part of the aligned eight sequences

	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W
C	-12																			
S	0	-2																		
T	2	-1	-3																	
P	3	-1	0	-6																
A	2	-1	-1	-1	-2															
G	3	-1	0	1	-1	-5														
N	4	-1	0	1	0	0	-2													
D	5	0	0	1	0	-1	-2	-4												
E	5	0	0	1	0	0	-1	-3	-4											
Q	5	1	1	0	0	1	-1	-2	-2	-4										
H	3	1	1	0	1	2	-2	-1	-1	-3	-6									
R	4	0	1	0	2	3	0	1	1	-1	-2	-6								
K	5	0	0	1	1	2	-1	0	0	-1	0	-3	-5							
M	5	2	1	2	1	3	2	3	2	1	2	0	0	-6						
I	2	1	0	2	1	3	2	2	2	2	2	2	2	-2	-5					
L	6	3	2	3	2	4	3	4	3	2	2	3	3	-4	-2	-6				
V	2	1	0	1	0	1	2	2	2	2	2	2	2	-2	-4	-2	-4			
F	4	3	3	5	4	5	4	6	5	5	2	4	5	0	-1	-2	1	-9		
Y	0	3	3	5	3	5	2	4	4	4	0	4	4	2	1	1	2	-7	-10	
W	8	2	5	6	6	7	4	7	7	5	3	-2	3	4	5	2	6	0	0	-17

Figure 2: The modified PAM-250 matrix

PAM-250 matrix represents the mutation distance between two amino acids or characters (Dayhoff *et al.* 1978).

Formulation as the shortest path problem

The multiple sequence alignment problem can be formulated as the shortest path problem in the d-dimensional lattice (Carrillo & Lipman 1988). Let d be the number of sequences to be aligned and S_k be the k-th sequence.

$L(S_1, .., S_d)$	State-space which is a d-dimensional lattice. i-th axis corresponds to S_i.
s	The start node.
t	The target node.
γ	Path from s to t in $L(S_1, .., S_d)$.
k	The length of γ.
n_i	The i-th node of γ $(0 \leq i \leq k)$.
$m(\gamma)$	Cost of γ.
u, v	Node in $L(S_1, .., S_d)$.
(u, v)	Edge in $L(S_1, .., S_d)$.
$c(u, v)$	Cost of (u, v).

The notations such as γ_{ij} and u_{ij} represent the projections of γ and u onto the plane determined by S_i and S_j, respectively. The path γ also can be represented as a sequence of nodes $\{n_0, n_1, \ldots, n_k\}$, where $n_0 = s$ and $n_k = t$.

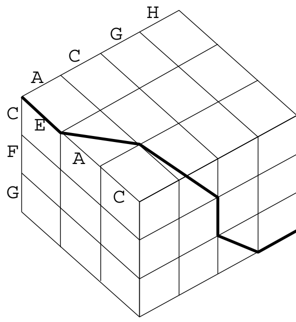
For any given set of d sequences $S_1, \ldots, S_d$, we can define the state-space of the multiple sequence alignment problem. This state-space can be obtained by making the Cartesian product of N sequences (see Figure 3). It forms a d-dimensional lattice. The node that corresponds to the beginning of all the sequences is the start node. The node that corresponds to the end of all the sequences is the target node. The path γ from the start node to the target node in this lattice determines a unique alignment of d sequences. There is a one-to-one relationship between the path and the alignment. The i-th edge of the path corresponds to the i-th column of the alignment. The pairwise alignment of i-th and j-th sequences corresponds to the path γ_{ij}, which is the projection of γ onto the plane determined by the sequences S_i and S_j.

Figure 3 depicts the state-space representation of the multiple sequence alignment problem of three sequences $S_1 =$ ACGH, $S_2 =$ CFG and $S_3 =$ EAC. In this 3-dimensional lattice, the top left-hand corner is the start node and the bottom right-hand corner is the target node. The path drawn with bold line corresponds to the alignment at the lower right of Figure 3. The first edge (n_0, n_1) of this path corresponds to the first column of the alignment, $(-, -, \mathtt{E})$.

In the d-dimensional lattice state-space, the cost of the edge (u, v) is defined as follows.

$$c(u, v) = \sum_{1 \leq i < j \leq d} c(u_{ij}, v_{ij})$$

The edge (u_{ij}, v_{ij}) is the projection of (u, v) onto the S_i-S_j plane (See Figure 4). In the 2-dimensional lattice, each row and column corresponds to each character in the sequences S_i and S_j, respectively. The diagonal edge associates S_i and S_j, while horizontal and vertical edges represent the insertions of gaps into the pairwise alignment. The cost $c(u_{ij}, v_{ij})$ is given by the value of the modified PAM-250 matrix, which corresponds to their characters. Figure 4 depicts the 2-dimensional path which is the projection of the 3-dimensional path of Figure 3. In this figure, path γ_{12} corresponds to the alignment of two sequences, S_1 and S_2. The second edge $(n_{2_{12}}, n_{3_{12}})$ is the projection of the 3-dimensional edge (n_2, n_3) onto the S_1-S_2 plane. This diagonal edge corresponds two characters, C,C. The cost $c(n_{2_{12}}, n_{3_{12}})$ is given by $p(\mathtt{C}, \mathtt{C})$, which represents the value of the PAM-250 matrix corresponding to characters C and C. On the other hand, the first horizontal edge $(n_{1_{12}}, n_{2_{12}})$ is the projection of (n_1, n_2) and corresponds a character and a gap, A,-. The cost of this edge is given by $p(\mathtt{A}, -)$. The $-$ represents the gap and $p(\mathtt{X}, -)$ or $p(-, \mathtt{X})$ gives the gap




```
<before alignment>
A C G H
C F G
E A C

<after alignment>
- A C - G H
- - C F G -
E A C - - -
```

Figure 3: State-space representation

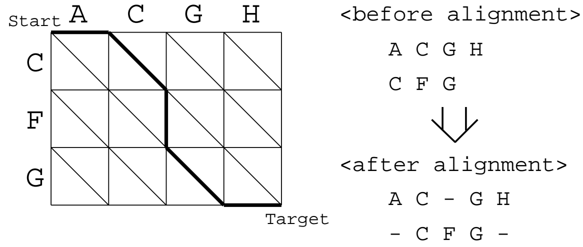


Figure 4: The projection of 3-dimensional path to the S_1-S_2 plane

cost for any character X. There is no corresponding edge for $c(n_0, n_1)$, because the projection of 3-dimensional edge $c(n_0, n_1)$ onto the S_1-S_2 plane is a point. Thus, $p(-,-)$ gives 0 cost, because the pair $(-,-)$ makes no contribution to the pairwise alignment.

Using this definition, we can calculate the cost of the d-dimensional path that corresponds to the alignment of the d sequences. The cost of the path is given by the sum of the edge costs of the path.

$$m(\gamma) = \sum_{i=0}^{k-1} c(n_i, n_{i+1}).$$

Finding the shortest path in d-dimensional lattice is to find the optimal alignment of d-sequences. In Figure 3, $c(n_1, n_2)$ represents the cost of the edge, which corresponds to column $(\mathtt{A}, -, \mathtt{A})$ of the alignment, and equals $p(\mathtt{A}, -) + p(\mathtt{A}, \mathtt{A}) + p(-, \mathtt{A})$.

Using this formulation and the gap cost of 8, Ikeda and Imai (Ikeda & Imai 1994) successfully applied the A* algorithm to the multiple sequence alignment problem. In Ikeda and Imai's experiment, the following heuristic function was used.

$$h(v) = \sum_{1 \le i < j \le d} h^*_{ij}(v_{ij})$$

where $h^*_{ij}(v_{ij})$ represents the shortest path length from v_{ij} to t_{ij} in the 2-dimensional lattice for S_i and S_j. h^*_{ij} is computed by the dynamic programming for each pair of S_i and S_j before the search algorithm is applied. In a high dimensional problem such as aligning seven or eight sequences, the time and space needed for the dynamic programming are negligible compared to those taken for solving the problem and do not increase time and space complexity. This heuristic function is admissible and consistent (Ikeda & Imai 1994).

Problem features

The multiple sequence alignment problem has some remarkable features compared to search problems common in the AI search community, such as the sliding-tile puzzle and the maze. Previous work has applied search algorithms to this problem without taking these features into consideration. The features of the multiple sequence alignment problem are as follows.

State-space The state-space forms a lattice. Therefore, there are a large number of distinct paths to the same node.

Branching factor The branching factor is very large. It becomes $O(2^d)$, where d is the number of sequences to be aligned. When $d = 7$ and $d = 8$, for example, the maximum branching factor becomes 127 and 255, respectively.

Distribution of edge cost In the case of a high dimensional lattice, the edge cost $c(u, v)$ can take on a large number of distinct values.

These features are the reasons why IDA* and A* are not effective against the multiple sequence alignment problem. Linear-space search such as IDA* must generate every distinct path to a given node. In the lattice state-space, there are a large number of distinct paths due to the very large branching factor, and the number of revisits becomes very large. What is worse, each iteration relatively expand few nodes, since most paths have different costs due to the wide distribution of edge cost, and the number of iterations becomes very large. Consequently, there is little or no hope of linear-space search algorithms such as IDA* solving this problem in practical time because of the large number of iterations and revisits. On the other hand, best-first search algorithms such as A* cannot solve high dimensional problems given their large memory requirements. Since the branching factor is very large, many child nodes are generated and stored when a node is expanded. The Open list grows rapidly with search progress and consists of those nodes that might be expanded in the future. Among them, there are some nodes that will never be expanded during a search. It is useless and wasteful to store such nodes. Consequently, A* searches often fail because it stores such nodes.

A* with Partial Expansion

It seems logical not to store unpromising nodes; this reduces the space complexity at the cost of solution quality. Recently, this was mentioned as domain-independent pruning rule for beam search (Zhang 1998). Adopting this idea, we present a new admissible algorithm, A* with Partial Expansion, which reduces the memory requirements of A*. In Partial Expansion, if a node has unpromising child nodes after expansion, then the node is put back into the Open list and its priority is lowered.

The algorithm

In addition to $c(n, n_i)$ and $h(n)$ described in the previous section, we use the following notations. $g(n)$ is the shortest path length from the start node s to the node n found so

far. $f(n)$ is the static value of node n, which is given by $f(n) = g(n) + h(n)$. $F(n)$ is the stored value of node n. $F(n)$ equals the lowest f-value among all unpromising child nodes of n. C is a predefined and nonnegative cutoff value.

We want to store only those nodes that promise to reach the target nodes. For this purpose, we introduce cutoff value parameter C. The child node is regarded as promising and is stored when the f-value of the child node is less than or equal to C plus the F-value of its parent node. Otherwise, the child node is regarded as unpromising, and is not stored. To guarantee optimality, our algorithm uses an additional stored value $F(n)$. If node n has unpromising child nodes after expansion, then n is put back into the Open list with $F(n)$. Initially, the F-value of a node equals its f-value. After expansion of node n, our algorithm sets the $F(n)$ to the lowest f-value among its unpromising child nodes. A* expands nodes in incremental order of f-value, while our algorithm expands nodes in incremental order of F-value. If there are no promising child nodes, then it does not store child nodes at all and only revises the parent's F-value to the lowest f-value among unpromising child nodes. In this case, in other words, it only lowers the priority of its parent node for expansion.

The pseudo-code of our algorithm is as follows. In this code, T represents the set of target nodes, and $succ(n)$ represents the set of child nodes of a node n. The cutoff value C is given in advance.

Algorithm A* with Partial Expansion

```
1  g(s) := 0
2  F(s) := g(s) + h(s)
3  OPEN ← {s}
4  CLOSED ← ∅
5  while OPEN ≠ ∅ do
6     n:=arg min F(n_i), n_i ∈ OPEN
7     OPEN ← OPEN −{n}
8     if n ∈ T then return
9     SUCC_≤C ← {n_j | n_j ∈ succ(n), f(n_j) ≤ F(n)+C}
10    SUCC_>C ← {n_k | n_k ∈ succ(n), f(n_k) > F(n)+C}
11    for each n_l ∈ SUCC_≤C do
12       if n_l ∉ OPEN ∪ CLOSED then
13          g(n_l) := g(n) + c(n, n_l)
14          F(n_l) := g(n_l) + h(n_l)
15          OPEN ← OPEN ∪{n_l}
16       else if n_l ∈ OPEN and
                  g(n) + c(n, n_l) < g(n_l) then
17          g(n_l) := g(n) + c(n, n_l)
18          F(n_l) := g(n_l) + h(n_l)
19       else if n_l ∈ CLOSED and
                  g(n) + c(n, n_l) < g(n_l) then
20          g(n_l) := g(n) + c(n, n_l)
21          F(n_l) := g(n_l) + h(n_l)
22          CLOSED ← CLOSED −{n_l}
23          OPEN ← OPEN ∪ {n_l}
24       end if
25    end for each
26    if SUCC_>C = ∅ then
27       CLOSED ← CLOSED ∪ {n}
28    else
29       F(n) := min f(n_m), n_m ∈ SUCC_>C
30       OPEN ← OPEN ∪ {n}
31    end if
32 end while
```

In this code, there are some additional operations beyond those of A*. In order to selectively store child nodes, we need the operations shown on line 9 and 10 in the pseudo-code. Lines 2, 14, 18, 21 and 29 are needed to manage F-value. The operation on line 30 puts an expanded node back into the Open list.

In extreme cases, when $C = \infty$, our algorithm is identical to A*. On the other hand, when $C = 0$, it stores nodes in best-first order, so those nodes whose f-value exceed the optimal cost will never be stored. This means that we can perform a search with the same size of memory as the Closed list used by A*.

Thus, our algorithm is very effective for those problems wherein the ratio of the Open list to the Closed list is large. Suppose that we apply A* to the problems where the state-space forms a tree and the branching factor is b. Then the ratio of the Open list to the Closed list becomes $b - 1$ to 1. We can reduce the memory requirements by a factor of the branching factor b, since our algorithm only needs the same size of memory as the Closed list by A*. The branching factor of the multiple sequence alignment problem is very large as described in the previous section. In this case, we can effectively reduce the memory requirements by a factor of a few hundred in the best case. Thus, the effect of our algorithm is non trivial for this application.

Search behavior of A* with Partial Expansion

Figure 5 shows the search behavior of our algorithm when $C = 0$. The solid circles represent stored nodes and the dotted circles represent unstored nodes. The digits in the circles represent the F-value, i.e. expansion priority. The digits on the right side of the arrow represent revised F-value after expansion. The digits on the top left of the circles represent the order of expansion.

Figure 5(a) shows the first expansion. The F-value of node *A* is initialized to its f-value, 6. Only one child node *B* is stored at this expansion, because the f-value of node *B* equals the F-value of node *A*. We revise F-value of node A to the lowest f-value, 7, among all unpromising child nodes and put node A back into the Open list. Figure 5(b) shows the second expansion. Node *B* with the lowest F-value is expanded. There are no child nodes with the same f-value as the F-value of *B*, so no child nodes are stored in this expansion. We put node *B* back into the Open list after revising its F-value to the lowest f-value, 8, among all child nodes. Figure 5(c) shows the third expansion. Node *A* with the lowest F-value is expanded again and child node *C* is stored. We set F-value of node *A* to 9 and puts node *A* back into the Open list.

In the same search space, it is necessary for A* to store the nodes represented by the dotted circles in addition to those stored by our algorithm. That is to say, we can find the optimal solution with fewer stored nodes than A*.

Evaluation

Experiment

In our experiments, we used the same conditions as in Ikeda and Imai's experiments, as mentioned before. We use 21 se-

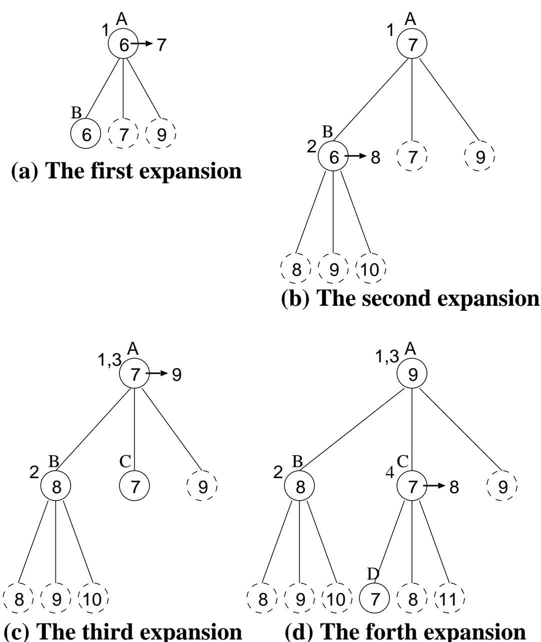


Figure 5: Search behavior of A* with Partial Expansion(C=0)

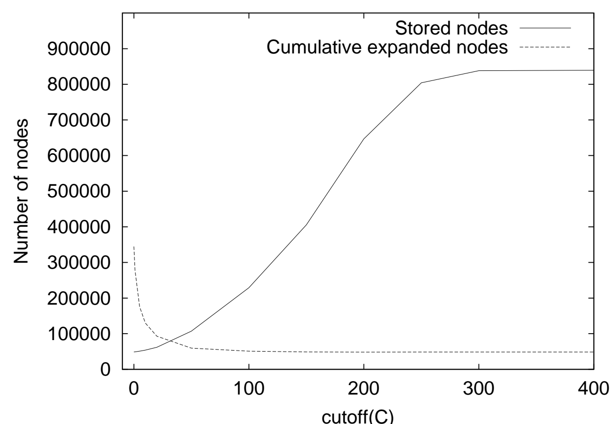


Figure 6: Number of stored and cumulative expanded nodes for each cutoff value(the seven sequence alignment)

quences from various species[1], which code for the elongation factor (EF-TU, EF-1α). The average length of these sequences is 448. The first seven of the 21 sequences are the same as those in their experiments. We applied our algorithm and A* to ten instances of seven and eight sequence alignment problems, respectively. It is natural for the maximum number of sequences, which are aligned by each search algorithm, to depend on sequence length, cost function, heuristic function and so on. We, however, can use their setting as benchmark test, because previous work also uses this setting to evaluate search algorithms. Each instance consists of seven or eight sequences that were randomly selected from 21 sequences. The maximum number of nodes stored by both algorithms is 2,000,000. This corresponds to about 160 megabytes of memory. For each instance, we assess performance from the cumulative number of expanded nodes and the maximum number of stored nodes with the cutoff values (C) of $0, 10, 50$ and ∞. When $C = \infty$, our algorithm is identical to A*. The cumulative number of expanded nodes corresponds to computational complexity and the number of stored nodes corresponds to space complexity.

Table 1 and 2 show the experimental results for the seven and the eight sequence alignment problems. The results are very similar. Due to space limitations, we show only the results of five instances. In the case of A*, the cumulative number of expanded nodes equals the number of nodes included in the Closed list, because the heuristic function we used is consistent. This value is approximately equal to the number of stored nodes in the case of $C = 0$. This shows that our algorithm can perform a search with just the same size of memory as the Closed list used by A*. Averaging the five instances, the cumulative number of nodes expanded by our algorithm is about 5 times larger than that by A*, due to re-expansion of the same node. On the other hand, we can reduce the number of stored nodes to 4.7% of what A* requires. In addition, our algorithm can align the eight sequences, while A* cannot because it demands excessive memory. This result shows the effectiveness of our algorithm against the multiple sequence alignment problem.

Figure 6 shows the cumulative number of expanded nodes and the number of stored nodes for various cutoff values in the case of a first instance of the seven sequence alignment problem. In this instance, the maximum difference in f-values between a node and its child is 396. When $C \geq 396$, our algorithm is virtually identical to A* because it stores all child nodes for each expansion. Figure 6 shows that we can effectively reduce the number of stored nodes, while the cumulative number of expanded nodes increases only a little if the cutoff value is appropriate. In the case of $C = 50$, for example, our algorithm reduces the memory requirements by 87%, while the computational complexity increases by only 20% compared to A*.

In Figure 6, it seems that the number of stored nodes is proportional to the cutoff value, while the cumulative number of expanded nodes is inversely proportional to the cutoff value. The intuitive explanation is as follows. As to the cumulative number of expanded nodes, every node n, whose f-value is lower than the cost of an optimal solution path,

[1]These species are as follows. (0) haloacula marismortui, (1) methanococcus vannielii, (2) thermoplasma acidophilum, (3) thermococcus celer, (4) sulfolobus acidocaldarius, (5) entamoeba histolytica, (6) plasmodium falciparum, (7) stylonychia lemnae, (8) euglena gracilis,(9) dictyostelium dscoideum, (10) lycoperscon esculentum, (11) arabidopsis thaliana, (12) absidia glauca, (13) rhizomucor racemosus, (14) candida albicans, (15) saccharomyces cerevisiae, (16) onchocerca volvulus, (17) artemia salina, (18) drosophila melanogaster, (19) xenopus laevis, (20) homo sapiens. The figure in a parenthesis corresponds to sequence number in this paper. These sequences are available in Genome database on the WWW.

Table 1: Experimental results for the seven sequence alignment problem

instance	sequence number	cutoff	0	10	50	∞(A*)
1	0,1,2,3,	Cumulative Expansion	344,640	129,986	59,592	48,575
	4,5,6	Stored Nodes	48,882	53,948	107,157	839,150
2	0,1,2,8,	Cumulative Expansion	234,582	101,211	53,504	44,405
	10,14,17	Stored nodes	45,144	49,956	87,987	911,218
3	0,2,4,7,	Cumulative Expansion	169,618	82,499	45,859	38,639
	8,14,15	Stored Nodes	38,810	42,851	75,045	859,307
4	5,7,8,9,	Cumulative Expansion	30,702	19,249	12,644	11,650
	10,12,18	Stored Nodes	11,877	13,031	22,801	451,033
5	0,3,6,9,	Cumulative Expansion	463,446	210,544	107,432	85,437
	11,12,19	Stored Nodes	86,611	94,853	164,879	1,576,920

Table 2: Experimental results for the eight sequence alignment problem

instance	sequence number	cutoff	0	10	50	∞(A*)
1	0,1,2,3,	Cumulative Expansion	6,945,069	1,956,430	804,602	unsolvable
	4,5,6,7	Stored Nodes	545,114	596,782	931,689	
2	0,3,4,10,	Cumulative Expansion	6,090,700	2,123,360	935,680	unsolvable
	14,16,17,18	Stored Nodes	648,240	702,099	1,133,568	
3	0,2,5,9,	Cumulative Expansion	1,162,528	525,715	265,425	unsolvable
	11,12,16,19	Stored Nodes	213,999	238,323	461,453	
4	1,4,9,10,	Cumulative Expansion	6,321,726	2,095,884	899,443	unsolvable
	12,15,16,20	Stored Nodes	635,294	697,740	1,165,797	
5	0,1,3,4,	Cumulative Expansion	13,938,989	3,953,400	1,580,348	unsolvable
	6,11,14,17	Stored Nodes	1,016,453	1,098,194	1,639,663	

has to be stored by our algorithm. If the cutoff value is large, the cumulative number of expanded nodes is relatively small, because the expansion of a node stores many child nodes. On the other hand, with a low cutoff value, the cumulative number of expanded nodes is relatively large because the expansion stores few child nodes. As to the number of stored nodes, it is proportional to the cutoff value. This is because the number of stored node, whose f-value is more than the cost of an optimal solution path, increases as the cutoff value increases.

The number of stored nodes for eight sequences is about ten times larger than that for seven sequences in the case of $C = 0$ in our experiments. This implies that the memory requirements for the nine sequence alignment problem may be ten times larger than that for the eight sequence alignment problem. Accordingly, we cannot currently align more than eight sequences by algorithms based on A*, including our algorithm, under the common memory capacity.

Related work

SMA* (Russel 1992) and RBFS (Korf 1993) were proposed to avoid the memory problems of A*. In this section, we compare our algorithm to these algorithms. All explore nodes in best-first order, however, there are some differences between them.

A* Our algorithm is identical to A*, when the cutoff value $C = \infty$. Thus, it includes A* as the special case. Our algorithm stores nodes in best-first order and never stores nodes whose evaluated costs are larger than the cost of an optimal solution path when $C = 0$. It reduces the space complexity at the cost of node re-expansion overhead. As the experimental results show, however, we can effectively reduce the space complexity while only slightly increasing the computational complexity by selecting the appropriate cutoff value.

RBFS (Korf 1993) RBFS is a linear-space best-first search algorithm. For each recursive call, RBFS uses a local cost threshold, which enables it to explore nodes in best-first order. The threshold value equals to the cost of its lowest-cost brother. On the other hand, our algorithm memorizes the cost of its lowest-cost unpromising child for each node and reduces the space complexity without losing admissibility. Unfortunately, RBFS cannot avoid revisits because it stores only nodes along the current search path. Our algorithm has advantages over RBFS when the state-space forms a lattice, because there are no revisits in our algorithm.

SMA* (Russel 1992) SMA* behaves like A* until SMA* stores the maximum number of nodes. When the number of stored nodes reaches the limit, SMA* prunes the node with highest cost in the Open list and continues to search. On the other hand, our algorithm stores only promising nodes and never prunes them. Here is an essential difference between the algorithms. The algorithm of SMA* is much more complicated than that of A*. In addition, we have to use more complicated version of SMA* (Kaindl & Khorsand 1994), when we apply SMA* to problems whose state-space is a graph. Our algorithm is very simple

with little modification of the A* algorithm and is applicable to graph problems without any modification.

Conclusion

We have proposed a simple and effective search algorithm, A* with Partial Expansion, to reduce the memory requirements of A* for problems wherein the branching factor is large. It reduces the space complexity of A*, without losing the merits of A*. It is admissible if a heuristic function is admissible. Consequently, we can solve problems wherein the branching factor is large, while A* cannot due to its excessive memory requirements.

We applied our algorithm and A* to the multiple sequence alignment problem. Experimental results show our algorithm can, on average, align seven sequences with only 4.7% of the amount of memory required by A*. We also applied it to the eight sequence alignment problem, which has not been solved up to now, and successfully aligned eight sequences.

In typical search problems such as the sliding-tile puzzle, the branching factor is much smaller than it is in the multiple sequence alignment problem. For such problems, our algorithm may be less effective in reducing the amount of memory, compared to the case of the multiple sequence alignment problem. However, there are several important applications for which our algorithm will be effective. One such application is the route finding problem in cities with massive and complicated road networks. Usually, the data of a road network occupies a huge volume. It cannot be fitted into main memory and is stored in a geographical database in secondary memory or in distributed databases over the Internet. Suppose that we want to find the shortest route from our house to the nearest bookstore. In a complicated city, there are countless routes because of the many intersections and transportation methods; the branching factor of this problem is very large. Most routes, such as a route to Paris by airplane or to a station by taxi, are useless for finding the shortest route. Thus, it is difficult to directly apply A* to this application. In order to reduce the memory requirements, such useless choices are eliminated when an application is formalized by the search problem. However, it is desirable for the search algorithm to be capable of coping with such useless choices instead of manually eliminating them from the model. The importance of our algorithm lies in its applicability to real world applications.

References

Carrillo, H., and Lipman, D. 1988. The multiple sequence alignment problem in biology. *SIAM Journal Applied Mathematics* 48: 1073-1082.

Dayhoff, M. O.; Schwartz, R. M.; and Orcutt, B. C. 1978. *Atlas of protein sequence and structure*, volume 5, 345-352. National Biomedical Research Foundation.

Ikeda, T., and Imai, T. 1994. Fast A* algorithms for multiple sequence alignment. *Genome Informatics Workshop 94*, 90-99.

Kaindl, H., and Khorsand, A. 1994. Memory-bounded bidirectional search. *AAAI-94*, 1359-1364.

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

Korf, R. E., 1993. Linear-space best-first search, *Artificial Intelligence* 27, 97-109.

Korf, R. E. 1999. Divide-and-conquer bidirectional search: first results. *Proc IJCAI-99*, 1184-1189.

Miura, T., and Ishida, T. 1998. Stochastic node caching for memory-bounded search. *AAAI-98*, 450-456.

Russel, S., 1992. Efficient memory-bounded search methods. *ECAI-92*, 1-5.

Zhang, W., 1998. Complete anytime beam search. *AAAI-98*, 425-430.