Approximate Multiple Sequence Alignment with A-star

Tristan Cazenave[1]

Abstract. The multiple sequence alignment problem is one of the most important in computational biology. We present algorithms and data structures to improve multiple sequence alignment with A-star. The first improvement is to accelerate the search for the best open node by 15%, using an array of stacks. The second improvement is to detect duplicate nodes efficiently using a transposition table. The third improvement consists in overestimating the admissible heuristic. It works better for aligning long sequences. A typical speedup for sequences of length two hundred fifty is 47 associated to a memory gain of 13 with an error rate of 0.09%. Overestimation can align sequences that are not possible to align with the exact algorithm.

1 Introduction

Multiple sequence alignment is one of the most important problem in computational biology. It is used to align DNA and protein sequences. The problem of aligning more than eight sequences takes too much memory for current exact algorithms such as A-star or dynamic programming. Biologists use programs that give an approximate answer to overcome the difficulty of finding exact alignment.

From a search point of view, the problem has properties that are different from other problems such as the sliding-tile puzzle, or pathfinding on game maps. It has a branching factor in O(2^s), when s is the number of sequences to align. The state space forms a lattice, and there are many paths that go through the same node.

We propose three improvements to basic A-star applied to multiple sequence alignment. The first improvement is general and can be used for other applications of A-star. It consists in using an array of stacks instead of a priority queue for storing the open nodes and finding the open node with the lowest f. The second improvement is to efficiently detect duplicate nodes, i. e. nodes that must not be re-expanded. Traversing the list of open nodes to find if a node is a duplicate is costful, using a transposition table instead speeds up the detection of duplicate nodes. The third improvement is to use overestimation. We have found that it enables large speedups and memory gains at the cost of a low error rate.

The second section presents the multiple sequence alignment problem, the third section details the modifications to the basic A-star algorithm, the fourth section presents experimental results, the last section concludes.

2 Multiple Sequence Alignment

In this section we present the multiple sequence alignment problem, then we show how dynamic programming can be applied to it. We present the approximate algorithms currently used to solve the problem, and we give an overview of the exact algorithms that have been tested on it.

[1] Université Paris 8, France, email: cazenave@ai.univ-paris8.fr

2.1 The problem

The multiple sequence alignment problem can be considered as a shortest path problem in a s-dimensional lattice [2]. Let's first consider the case of dimension two, it consists in aligning two sequences. We can write the letter of the first sequence on the horizontal axis, and the letters of the second sequence on the vertical axis. The path starts at the origin point of the matrix (point (0,0) at the upper left). For each point there are three possible moves: the diagonal, the horizontal, and the vertical moves. A diagonal move is equivalent to aligning two characters of the sequences, an horizontal move is equivalent to aligning a character of the horizontal sequence with a gap in the second sequence, a vertical move aligns a character of the vertical sequence with a gap in the horizontal sequence. All paths stop at the bottom right of the matrix after the last two characters of both sequences have been aligned.

The simple model to evaluate the cost of a move is: 0 for a match (aligning the two same characters), 1 for a mismatch, and 2 for a gap (a gap is represented with a -). The cost of a path is the sum of the costs of its moves.

For example, if the first sequence is ACGTTAGCTA and the second sequence is ACAGTTAGTA the best alignment is:

```
AC-GTTAGCTA
ACAGTTAG-TA
```

and it has a cost of four.

When aligning s sequences, the path goes through a s-dimensional lattice, the branching factor is $2^s - 1$, and the cost of a move is the sum of the costs of the moves for each pair of sequences.

2.2 Dynamic programming

Dynamic programming can be used to efficiently find solution to the problem of sequence alignment. However if the average length of the sequences to align is l, and the number of sequences is s, dynamic programming needs O(l^s) memory and time. A possible improvement trades off time for space [4] but it still requires O(l^{s-1}) memory which is still too much for aligning many sequences.

2.3 Approximate algorithms

The programs currently used by biologist such as CLUSTAL W [12] and DCA [11] find sub-optimal alignments. They consist in series of progressive pairwise alignments.

2.4 Exact algorithms

A* was applied to the optimal alignment of multiple sequences by Ikeda and Imai [6]. The admissible heuristic is computed using the dynamic programming tables for the pairwise alignments. Because of the large branching factor of the problem, and the large number of open nodes, A* cannot align more than seven sequences due to memory limits. To overcome this difficulty and reduce the memory requirements, A* with partial expansion was proposed [13]. It consists in not memorizing in the open list child nodes that have a f value greater than the f value of their parent plus a threshold. Experimental results show that Partial Expansion A* can align seven sequences with less stored nodes than A*, and can align some eight sequences problems. However the gain in memory is acquired at the cost of a greater search time.

Another refinement was proposed to reduce both the memory and the time requirements, using an octree to represent a three-way heuristic [9]. A close approach is the use of external memory pattern databases using strucured duplicate detection [15]. It reduces the memory requirements of the pattern databases by an average factor of 83 times, and makes Sweep-A* [14] run 16% faster than using traditional pattern databases.

Other researchers have compared A-star and dynamic programming [5].

3 A-star

This section presents modifications to the A-star algorithm. In all the paper, the admissible heuristic we have used for A-star is the sum of the pairwise alignments given by the 2-dimensional dynamic programming tables. The first subsection deals with the efficient choice of the best open node. The second subsection is about efficient duplicate node detection. The third subsection explains how we have used overestimation.

3.1 Choosing the best open node

Naive implementation of A-star use a list to store the open and the closed nodes. In this case, when the program has to find the best open node, it has to go through all the list to find the node with the minimum f. The cost of using a list is linear in the size of the list.

A more elaborate, and commonly used, implementation of A-star uses a priority queue to represent the open list. A priority queue uses a heap to maintain the nodes sorted. The insertion of a new node, as well as the finding of the best node require a logarithmic time in the size of the list.

We propose to use an array of stacks to maintain the open list. The index in the array is the value of f for the nodes stored in the corresponding stack. The insertion of an element is performed in constant time, just pushing it on the top of the stack that corresponds to its f value. Finding the best node is also performed in almost constant time. The smallest f value (currentf) over all the nodes is maintained, and updated each time a node is inserted in the open list. When retrieving the best open node, currentf is used to check if the stack at this index has an element. If so the first element is popped and returned as the best node. If the stack is empty, currentf is incremented and the test if performed again for the next index in the array of stacks.

3.2 Duplicate node detection

In the multiple sequence alignment problem, it is very important not to expand again nodes that are already present in the open or closed lists, with a smaller or equal g.

A possible implementation is to go through all the closed and open nodes, to verify if the node to insert in the open list is not already present with a smaller or equal g. When implemented this way, the duplicate node detection takes most of the time of the algorithm.

We propose an implementation of duplicate node detection which uses a transposition table. Transposition tables are often used in game programs so as to memorize the results of search at nodes of the tree [1]. In order to hash position, we have used Zobrist hashing [16]. A position in the state space is defined by its coordinates. There are as many coordinates as there are sequences. For each possible sequence, and each possible coordinate in this sequence, a 64-bit random number has been computed once for all. The Zobrist hashing of a position is the XOR of all the random numbers that correspond to the coordinates of the position. The XOR is used because:

- it is a very fast operation on bits,
- it is incremental: in order to undo the XOR with a number, the only operation needed is to XOR again with this number. When a node is expanded, it is a move to a neighboring position. The program only has to XOR the random numbers of the old coordinates that change, and to XOR also the random numbers of the new coordinates.
- The XOR of random values that have a uniform repartition gives a random value that has a uniform repartition. It is important to lower the collision probability.

Each position in the search space is associated to a 64-bit hashcode. The lowest bits of this hascode are use to index the position in the transposition table. An entry of the transposition table contains a list of structure. Each structure contains a hashcode and a g value. When the algorithm detects duplicate, it goes through the list and verifies if an entry with the same hashcode as the current position, and a less or equal g is present. In that case, the node is cut.

3.3 Overestimation

Overestimation has been used for Sokoban in the program Rolling Stones, adding all the patterns that match instead of only selecting the ones that ensure admissibility [7]. This use of overestimation has helped Rolling Stones solve 52 problems instead of 47 without overestimation.

Pearl has introduced ϵ-admissible search [10] which finds solutions with bounded costs.

A related but different approach is to use likely-admissible heuristics [3]. It consists in relaxing the admissibility requirement in a probabilistic sense. Instead of providing an upper bound to the cost, it guarantees to end-up with optimal solutions with a given probability.

In A-star, the value associated to a node is the sum of the cost of the path from the origin to the node (the g value), and of the admissible heuristic that gives a lower bound on the cost of the remaining path to the goal (the h value). We have for the node n: $f(n) = g(n) + h(n)$.

In order to overestimate the length of the remaining path, we have used the following f: $f(n) = g(n) + w \times h(n)$ where w is a real number greater than one. Overestimation speeds up A-star at the cost of making it inexact.

The main property of this overestimation is that it develops more easily nodes that have a low h value first, i. e. nodes that are closer to the goal than in usual A-star. Nodes that have been discarded early in the search stay discarded longer than in usual A-star. So overestimation privilegiates paths where some search has already been invested, and paths that have a low admissible heuristic at the beginning of the path.

4 Experimental results

Experiments use a Celeron 1.7 GHz with 1 GB of RAM. Given the available memory, we have chosen a limit of 10,000,000 nodes for A-star.

4.1 Generation of test data

In order to test the different algorithms, we have generated random sequences of bases (i.e. strings composed of letters in the {A,C,G,T} alphabet). The tests use sets of strings of length fifty, one hundred, two hundred or two hundred and fifty. This methodology is similar to Korf and Zhang testing methodology [8]. Generating random problems allows to generate a large number of problems, and to easily replicate experiments. For each length, we have generated one hundred problems. Each problem is composed of ten strings.

4.2 Array of stacks

Table 1 gives the time and number of nodes used by A-star with a priority queue, and A-star with an array of stacks. s is the number of sequences to align. Each line describes the result of solving one hundred problems with sequences of length fifty.

Table 1. Comparison of priority queues and array of stacks.

s	$algorithm$	$\Sigma\, time$	$\Sigma\, nodes$
5	array	38.54s	3,154,269
5	queue	45.38s	3,310,618
6	array	744.42s	23,045,612
6	queue	888.92s	23,914,925

For five sequences the speed of the array of stacks is 81,844 nodes per seconds while the speed of the priority queue is 72,953 nodes per second. The array of stacks is 12% faster.

On more complex problems with more nodes, the comparison is even better for the array of stacks: for six sequences it develops 30,957 nodes per seconds versus 26,903 nodes per second for the priority queue. The array of stacks is 15% faster.

4.3 Duplicate node detection

Table 2 gives the time used to solve the one hundred problems with sequences of length fifty, using transposition tables, and using lists. The transposition table uses 65,535 entries. The index of a position is the last 16 bits of its hashcode.

Even for problems with a small number of nodes such as the alignment of four sequences, the transposition table algorithm outperforms clearly the list implementation.

On the more complex problem of aligning five sequences, the list implementation takes 2203 seconds for solving the first six problems, when the transposition table implementation takes 2.05 seconds. Lists become even worse for more than five sequences since the number of nodes grows and the list implementation takes a time proportional to the square of the number of nodes.

Table 2. Comparison of list and transposition table.

s	$algorithm$	$\Sigma\, time$
4	list	124.54s
4	transposition	2.03s

4.4 Straight alignment

In order to find an upper bound to the cost of an alignment and better evaluate overestimation, we have tested the algorithm which consists in aligning all the sequences without introducing gaps. The tests were run for one hundred sets of sequences of length fifty and one hundred sets of sequences of length one hundred. The same sets of sequences are used for testing overestimation. The results are given in table 3 for sequences of length fifty, and in table 4 for sequences of length one hundred.

Table 3. Upper bounds for one hundred sets of sequences of length fifty.

s	$\Sigma\, length$
5	37,341
6	56,041
7	78,561
8	104,711
9	135,545
10	168,401

Table 4. Upper bounds for one hundred sets of sequences of length one hundred.

s	$\Sigma\, length$
4	44,894
5	75,061
6	112,500
7	157,548
8	209,945

4.5 Overestimation

We tested overestimation for different numbers of sequences, different weights, and different lengths of sequences. Results for sequences of length fifty are given in table 5. The first column gives the number of sequences to align, the second column gives the weight used for overestimation (1.00 corresponds to the exact algorithm), the third column gives the cumulated time used to solve one hundred problems, the fourth column gives the sum of the lengths of the shortest paths found for each problem, the fifth column gives the sum of the nodes used for solving each problem.

The results for the exact algorithm are not given for eight or more sequences, since the node limit is reached for these problems before the problem is solved.

We can observe than the 1.05 weight is a safe weight. It reduces significantly the time and the number of nodes, while finding alignments that are better than straight alignments and quite close to optimal alignments. The 1.10 and the 1.20 weights sometimes give worse results than the straight alignment, and should be avoided.

Table 5. Results for one hundred sets of sequences of length fifty.

s	w	$\Sigma\ time$	$\Sigma\ length$	$\Sigma\ nodes$
5	1.00	38.54s	36,654	3,154,269
5	1.05	2.59s	36,747	556,463
5	1.10	0.57s	37,036	212,282
5	1.20	0.30s	37,320	161,983
6	1.00	744.42s	55,362	23,045,612
6	1.05	24.97s	55,477	2,556,103
6	1.10	1.76s	55,929	496,354
6	1.20	0.85s	56,161	328,479
7	1.00	30,844.41s	77,982	168,829,955
7	1.05	268.40s	78,147	12,052,417
7	1.10	6.47s	78,592	1,247,218
7	1.20	2.17s	78,767	653,935
8	1.05	5,639.71s	104,396	58,990,176
8	1.10	26.17s	104,895	3,151,376
8	1.20	4.62s	104,914	1,305,738
9	1.10	94.02s	134,856	8,153,718
9	1.20	16.68s	134,765	2,586,087
10	1.10	572.74s	168,920	21,489,700
10	1.20	44.12s	168,592	5,193,800

Table 6 has been created using table 5. For each number of sequences, and each weight, the speedup, the error and the memory gain are given. The error is calculated dividing the sum of the lengths of the paths found with overestimation by the sum of the lengths of the shortest paths found by the exact algorithm. The memory gain is computed dividing the number of nodes of the exact algorithm by the number of nodes of the approximate algorithm.

We can observe in this table that the memory gains increase exponentially with the number of sequences, and that the speedups increase more than exponentially with the number of sequences.

Table 6. Gains over the exact algorithm for sequences of length fifty.

s	w	$speedup$	$error$	$memory\ gain$
5	1.05	14.88	0.25%	5.67
5	1.10	67.61	1.04%	14.86
5	1.20	128.47	1.82%	19.47
6	1.05	29.81	0.21%	9.02
6	1.10	422.97	1.02%	46.42
6	1.20	875.78	1.44%	70.15
7	1.05	114.92	0.21%	14.01
7	1.10	4,767.30	0.78%	135.36
7	1.20	14,214.00	1.01%	258.17

In order to test if gain are due to a reduction of the branching factor or to a reduction of the depth of the search, we tested the program on sequences of length one hundred. Results are given in table 7. We can observe that the 1.10 weight is always better than the straight alignment, while the 1.20 weight becomes worse for seven and eight sequences. The 1.05 weight gives interesting speedups and memory gains for alignments that are close to optimal.

Table 8 gives the gains and the error calculated with table 7. If we compare table 6 with table 8, we can observe that the gains for four sequences in table 8 are similar to the gain for five sequences in table 6. The branching factor is fifteen for four sequences, and thirty one for five sequences, the average length of the shortest path is four hundred twenty six for four sequences of length one hundred, and three hundred sixty seven for five sequences of length fifty. The gains

Table 7. Results for one hundred sets of sequences of length one hundred.

s	w	$\Sigma\ time$	$\Sigma\ length$	$\Sigma\ nodes$
4	1.00	65.31s	42,605	5,403,857
4	1.05	3.18s	42,696	732,666
4	1.10	0.58s	43,069	243,311
4	1.20	0.36s	43,587	162,110
5	1.00	4547.52s	72,152	88,548,072
5	1.05	109.81s	72,246	8,045,147
5	1.10	2.90s	72,986	788,188
5	1.20	0.84s	74,167	340,647
6	1.05	6250,38s	109,398	93,687,353
6	1.10	17.59s	110,401	2,865,184
6	1.20	1.91s	112,185	700,769
7	1.10	216.68s	155,251	14,090,458
7	1.20	4,47s	157,674	1,415,208
8	1.10	7,288.41s	207,858	89,290,030
8	1.20	10.43s	210,554	2,839,425

Table 8. Gains over the exact algorithm for sequences of length one hundred.

s	w	$speedup$	$error$	$memory\ gain$
4	1.05	20.54	0.21%	7.38
4	1.10	112.60	1.09%	22.21
4	1.20	181.42	2.30%	33.34
5	1.05	41.41	0.13%	11.01
5	1.10	1568.11	1.15%	112.34
5	1.20	5413.71	2.79%	259.94

are slightly greater for four sequences of length one hundred than for five sequences of length fifty, with length of the shortest paths which are also slightly greater.

Concerning five sequences of length one hundred, and seven sequences of length fifty, the average length of the shortest paths are respectively seven hundred twenty two and seven hundred eighty, when the gains in memory are equivalent, and the speedups are three times greater for the seven sequences.

The speedup is more correlated with the length of the shortest path than with the branching factor.

The error rates are more important for five sequences of length one hundred than for seven sequences of length fifty, even if the speedup are lower. It is interesting as it shows that speedup and error rates are not always correlated, and that there are portions of the space of problems (denoted by the length of the sequences and the number of sequences) that are more favorable to overestimation than others.

As the overestimation has a better behavior for sequences of length one hundred than for sequences of length fifty, we have tested the algorithm on sequences of length two hundred (tables 9 and 10), and on sequences of length two hundred and fifty (tables 11 and 12).

For sequences of length two hundred, all weights give much better results than the straight alignment. Moreover, the speedups and the memory gains are also better than for four sequences of length one hundred.

Concerning sequences of length two hundred fifty, the speedups and memory gains are even better, and all the weights give alignments much better than the straight one. For five sequences, the overestimation finds alignments that are much better than the straight one when the exact algorithm exhausts memory. When we have tested the first alignment of five sequences with a weight of 1.05, the node limit

Table 9. Results for one hundred sets of sequences of length two hundred.

s	w	$\Sigma\, time$	$\Sigma\, length$	$\Sigma\, nodes$
4	1.00	3,664.79s	84,102	91,691,701
4	1.05	97.87s	84,197	7,638,509
4	1.10	2.28s	85,190	611,965
4	1.20	0.99s	86,631	336,527
4	straight	0.06s	90,104	0

Table 10. Gains over the exact algorithm for sequences of length two hundred.

s	w	$speedup$	$error$	$memory\ gain$
4	1.05	37.44	0.11%	12.00
4	1.10	1607.36	1.29%	149.83
4	1.20	3701.81	3.00%	272.46

was reached and A-star stopped with no solution after 2,818 seconds. Comparatively, a weight of 1.10 found a path of length 1807 in 0.39 seconds and 57,044 nodes (a straight alignment gave 1865). For this problem the speedup was therefore of much more than 7,225 and the memory gain of much more than 175.

Table 11. Results for one hundred sets of sequences of length two hundred fifty.

s	w	$\Sigma\, time$	$\Sigma\, length$	$\Sigma\, nodes$
4	1.00	12,998.15s	104,565	217,402,086
4	1.05	274.72s	104,660	16,514,033
4	1.10	4.17s	106,002	1,019,825
4	1.20	1.27s	107,831	426,185
4	straight	0.08s	112,060	0
5	1.10	64.88s	179,024	7,012,805
5	1.20	3.44s	183,058	933,319
5	straight	0.15s	187,103	0

In conclusion, overestimation gives better results for long sequences.

5 Conclusion and Future Work

We have shown that using an array of stacks instead of a priority queue makes A-star 15% faster for the multiple sequence alignment problem. Using a transposition table to detect duplicate nodes instead of a list makes it up to 1,000 faster for five sequences. Overestimation of the admissible heuristic gives large speedups and memory gain for small error rates, it works better with long sequences than with short ones.

Future works include combining our improvement with other heuristics used for exact algorithms such as pattern databases [15], partial expansion [13] and dynamic programming [5].

Table 12. Gains over the exact algorithm for sequences of length two hundred fifty.

s	w	$speedup$	$error$	$memory\ gain$
4	1.05	47.31	0.09%	13.16
4	1.10	3,117.06	1.37%	213.18
4	1.20	10,234.76	3.12%	510.11

REFERENCES

[1] D. Breuker, 'Memory versus search in games', Phd thesis, University of Maastricht, (October 1998).

[2] H. Carrillo and D. Lipman, 'The multiple sequence alignment problem in biology', *SIAM Journal Applied Mathematics*, **48**, 1073–1082, (1988).

[3] M. Ernandes and M. Gori, 'Likely-admissible and sub-symbolic heuristics', in *ECAI 2004*, pp. 613–617, Valencia, Spain, (2004). IOS Press.

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

[5] H. Hohwald, I. Thayer, and R. Korf, 'Comparing best-first search and dynamic programming for optimal multiple sequence alignment', in *IJCAI-03*, pp. 1239–1245, (2003).

[6] T. Ikeda and T. Imai, 'Fast A* algorithms for multiple sequence alignment', in *Genome Informatics Workshop 94*, pp. 90–99, (1994).

[7] A. Junghanns and J. Schaeffer, 'Domain-dependent single-agent search enhancements', in *IJCAI-99*, pp. 570–575, (1999).

[8] R. E. Korf and W. Zhang, 'Divide-and-conquer frontier search applied to optimal sequence alignment', in *AAAI-00*, pp. 910–916, (2000).

[9] M. McNaughton, P. Lu, J. Schaeffer, and D. Szafron, 'Memory-efficient A* heuristics for multiple sequence alignment', in *AAAI-02*, pp. 737–743, (2002).

[10] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[11] K. Reinert, J. Stoye, and T. Will, 'An iterative method for faster sum-of-paris multiple sequence alignment', *Bioinformatics*, **16**(9), 808–814, (2000).

[12] J. Thompson, D. Higgins, and T. Gibson, 'CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice', *Nucleic Acids Research*, **22**, 4673–4680, (1994).

[13] T. Yoshizumi, T. Miura, and T. Ishida, 'A* with partial expansion for large branching factor problems', in *AAAI-00*, pp. 923–929, (2000).

[14] R. Zhou and E. Hansen, 'Sweep A*: Space-efficient heuristic search in partially ordered graphs', in *Proceedings of 15th IEEE International Conference on Tools with Artificial Intelligence*, pp. 427–434, (2003).

[15] R. Zhou and E. Hansen, 'External-memory pattern databases using structured duplicate detection', in *AAAI-05*, Pittsburgh, PA, (July 2005).

[16] A. Zobrist, 'A new hashing method with applications for game playing', *ICCA Journal*, **13**(2), 69–73, (1990).