A Space-Bounded Anytime Algorithm for the Multiple Longest Common Subsequence Problem

Jiaoyun Yang, Yun Xu, *Member, IEEE*, Yi Shang, *Senior Member, IEEE*, and Guoliang Chen

Abstract—The multiple longest common subsequence (MLCS) problem, related to the identification of sequence similarity, is an important problem in many fields. As an NP-hard problem, its exact algorithms have difficulty in handling large-scale data and time- and space-efficient algorithms are required in real-world applications. To deal with time constraints, anytime algorithms have been proposed to generate good solutions with a reasonable time. However, there exists little work on space-efficient MLCS algorithms. In this paper, we formulate the MLCS problem into a graph search problem and present two space-efficient anytime MLCS algorithms, SA-MLCS and SLA-MLCS. SA-MLCS uses an iterative beam widening search strategy to reduce space usage during the iterative process of finding better solutions. Based on SA-MLCS, SLA-MLCS, a space-bounded algorithm, is developed to avoid space usage from exceeding available memory. SLA-MLCS uses a replacing strategy when SA-MLCS reaches a given space bound. Experimental results show SA-MLCS and SLA-MLCS use an order of magnitude less space and time than the state-of-the-art approximate algorithm MLCS-APP while finding better solutions. Compared to the state-of-the-art anytime algorithm Pro-MLCS, SA-MLCS and SLA-MLCS can solve an order of magnitude larger size instances. Furthermore, SLA-MLCS can find much better solutions than SA-MLCS on large size instances.

Index Terms—Heuristic search, anytime algorithm, space bounded, multiple longest common subsequence (MLCS)

1 Introduction

The multiple longest common subsequence (MLCS) problem is a fundamental problem in many areas such as file comparison [1], syntactic pattern recognition [2], text editing [3], computational biology [4], [5], etc., corresponding to identify the similarity between sequences. This problem has been proved to be NP-hard [6] and extensive works have been done [7], [8], [9], [10]. However existing algorithms have been facing the challenge of large scale data due to the rapid growth of data volumes such as the hundreds gigabytes data per week generated by the next-generation sequencing techniques [11]. When dealing with large-scale data sets, time and space are two essential criterions to evaluate these algorithms. Since waiting a long time for an optimal solution is impractical, anytime algorithms have been developed to quickly produce a suboptimal solution and gradually improve it when given more time, until an optimal one is found. They focus on how to generate a good suboptimal solution within an acceptable time. On the other hand, the memory of any computer is limited, thus algorithms should be designed to limit their space usage not exceeding a given bound. However, time and space efficiency should both be considered in practical algorithms in solving large-size real world problems.

In this paper, we present two new space and time efficient MLCS algorithms, called SA-MLCS and SLA-MLCS. SA-MLCS is an anytime algorithm with a small space increase rate. Different from existing anytime algorithms, SA-MLCS uses an iterative beam widening search strategy to reduce space usage during the iterative search process of finding better solutions and makes use of previous search information to improve efficiency. Although SA-MLCS uses space efficiently, its space usage still grows exponentially and eventually exceeds any given amount of space. To handle this problem, SLA-MLCS, a space-bounded algorithm, is developed based on SA-MLCS. SLA-MLCS uses a replacing strategy when SA-MLCS reaches a given space bound and can effectively handle much larger size problem instances than existing algorithms. When SA-MLCS reaches the space limitation, SLA-MLCS replaces the nodes that have been expanded in order to guarantee the space usage not to exceed the space bound.

The paper is organized as follows. In the next section, the concepts of the MLCS problem, its dominant point method, and the search graph of dominant points are described. Then, previous works of anytime and memory efficient search algorithms are reviewed. In Section 3, a new anytime algorithm, SA-MLCS, is presented. In Section 4, a new space-bounded anytime algorithm, SLA-MLCS, is presented. In Section 5, experimental results comparing the performance of the new algorithms with existing ones are presented. Finally, in Section 6, we summarize the paper.

- *J. Yang, Y. Xu, and G. Chen are with the Key Laboratory on High Performance Computing and the School of Computer Science, University of Science and Technology of China, Heifei, Anhui, China. E-mail: jiaoyun@mail.ustc.edu.cn, xuyun@ustc.edu.cn.*
- *Y. Shang is with the Department of Computer Science, University of Missouri-Columbia, Columbia, MO 65211. E-mail: shangy@missouri.edu.*

Manuscript received 25 July 2013; revised 14 Jan. 2014; accepted 23 Jan. 2014. Date of publication 3 Feb. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by J. Pei.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2304464

2 MLCS Problem Formulation and Related Works

In this section, we first describe the MLCS problem, its dominant point method, and the search graph of dominant points. Then we review related works on anytime search algorithms and memory efficient algorithms.

2.1 Basics of the MLCS Problem

Let Σ represent a finite alphabet for the sequences. Take DNA sequences as an example, $\Sigma = \{A, T, C, G\}$.

Definition 1. *Let $a = s_1 s_2 \dots s_m$ and $b = t_1 t_2 \dots t_l$ be two sequences over Σ. b is a* subsequence *of a if there exists a strictly increasing sequence $< i_1, i_2, \dots, i_l >$ of indices of a such that $t_j = s_{i_j}$ for all j $(1 \le j \le l)$.*

For a set of sequences $\{a_1, a_2, \dots, a_n\}$ over Σ, where $a_k = s_1 s_2 \dots s_{n_k}$, each j $(1 \le j \le n_k)$ is called a *coordinate* of a_k.

Definition 2. *b is a* common subsequence *of $a_1, a_2, \dots, a_n$, if b is a subsequence of all a_j $(1 \le j \le n)$.*

Definition 3. *b is a* longest common subsequence *(LCS) of $a_1, a_2, \dots, a_n$, if b is a common subsequence of $a_1, a_2, \dots, a_n$, and there are no other common subsequences longer than it.*

The *multiple longest common subsequence problem* is to find one or all longest common subsequences of three sequences or more. For two sequences, the problem is commonly called the longest common subsequence problem. Most of the algorithms developed for LCS could not be generalized for MLCS.

The classical exact algorithm for MLCS problem is dynamic programming [5], [12]. It solves this problem by recursively constructing a score matrix T with size $n_1 \times n_2 \times \dots \times n_n$ in which $T[p_1, p_2, \dots, p_n]$ contains the length of the longest common subsequence of prefixes $a_1[1 \dots p_1]$, $a_2[1 \dots p_2], \dots, a_n[1 \dots p_n]$. Then the longest common subsequences can be obtained by tracing back table T. For the two-sequences case, the recursive formula is as follows:

$$T[p_1, p_2] = \begin{cases} 0, & \text{if } p_1 = 0 \text{ or } p_2 = 0, \\ T[p_1 - 1, p_2 - 1] + 1, & \text{if } a_1[p_1] = a_2[p_2], \\ max(T[p_1 - 1, p_2], & \\ T[p_1, p_2 - 1]), & \text{otherwise.} \end{cases} \quad (1)$$

Fig. 1 illustrates an example of dynamic table for DNA sequences GTATGCGAA and AGAGATATG and the cells with the same value are grouped by contours.

The time and space complexities of dynamic programming are both $O(m^n)$ [13]. Many works are proposed to improve dynamic programming algorithm in order to extend the size of data sets that exact algorithms can solve [10], [14], [15], [16], among which the dominant point method is the most efficient.

Before introducing the dominant point method, some notions about dominant point are defined as follows:

Suppose *point*, $P = [p_1, p_2, \dots, p_n]$, denotes a position in the dynamic programming table T for n sequences, where p_i is a coordinate in sequence a_i.

		0	1	2	3	4	5	6	7	8	9
			G	T	A	T	G	C	G	A	A
0		0	0	0	0	0	0	0	0	0	0
1	A	0	0	0	①	1	1	1	1	1	1
2	G	0	①	1	1	1	②	2	2	2	2
3	A	0	1	1	②	2	2	2	2	③	3
4	G	0	1	1	2	2	③	3	3	3	3
5	A	0	1	1	2	2	3	3	3	④	4
6	T	0	1	②	2	③	3	3	3	4	4
7	A	0	1	2	③	3	3	3	3	4	⑤
8	T	0	1	2	3	④	4	4	4	4	5
9	G	0	1	2	3	4	⑤	5	5	5	5

Fig. 1. An example of the dynamic programming table for DNA sequences GTATGCGAA and AGAGATATG, where the cells with circle are dominant points.

Definition 4. *Point $P = [p_1, p_2, \dots, p_n]$ is a* match point *if $a_1[p_1] = a_2[p_2] = \dots = a_n[p_n]$, i.e., identical symbol in all sequences.*

Definition 5. *For points $P = [p_1, p_2, \dots, p_n]$ and $Q = [q_1, q_2, \dots, q_n]$, P* dominates *Q if $\forall l$ $(1 \le l \le n)$, $p_l \le q_l$. P* strongly dominates *Q if $\forall l$ $(1 \le l \le n)$, $p_l < q_l$.*

Definition 6. *Point P is a k-*dominant point *if and only if:*

1. *Point P is a match point.*
2. *$T[P] = k$.*
3. *There is no other match point Q so that $T[Q] = k$ and Q dominates P.*

In Fig. 1, the corner points of contours are dominant points marked with circles, i.e., the skylines of the cells with the same value [17], [18].

The dominant point method is based on the observation that most of the cells in dynamic table are useless and do not need to be computed. Only the dominant points should be expanded and stored. Based on the dominant point method, some parallel algorithms are proposed to improve its efficiency [19], [20], [21]. They differ in how to compute dominant points, which is also known as the skyline problem in multidimensional data analysis.

The search space of the dominant point method can be organized into a graph as follows: A node represents a dominant point in the dynamic table and an edge $< P, Q >$ represents P strongly dominates Q and $T[Q] = T[P] + 1$. A node has at most $|\Sigma|$ neighbors as each character in $|\Sigma|$ corresponds to one neighbor. The graph contains a source node $S = [0, 0, \dots, 0]$ with no incoming edges and a sink node $T = [U, U, \dots, U]$ with no outgoing edges, where U means the infinite value such that T could be a child of any node. A path from the source node to the sink node corresponds to a common subsequence. Thus the MLCS problem becomes finding the longest path from the source to the sink in the graph, a graph search problem where existing graph search algorithms could be applied.

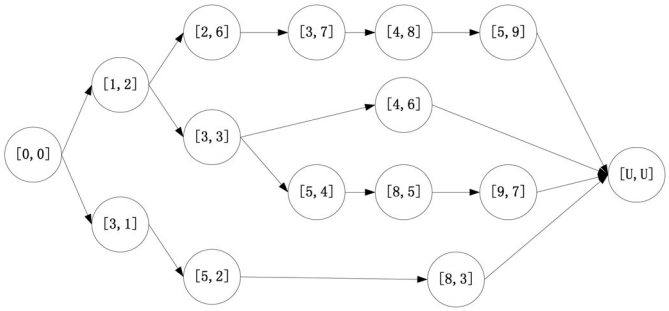


Fig. 2. The search graph corresponding to the dynamic programming table in Fig. 1. Each node $P = [p_1, p_2]$ represents a dominant point in the table, where $[p_1, p_2]$ is the ordinate of the dominant point. Edge $< P, Q >$ represents P strongly dominates Q and $T[Q] = T[P] + 1$. The graph contains a source node $[0, 0]$ and a sink node $[U, U]$.

Fig. 2 illustrates the search graph for the example in Fig. 1.

2.2 Anytime Search Algorithms

Pro-MLCS is a state-of-the-art anytime algorithm for MLCS [22]. It adopts an iterative best first search strategy to progressively output better and better solutions. There are many anytime graph search algorithms, which could be categorized into two groups, A*-based anytime algorithms and beam-based anytime algorithms. They all can be applied to the search graph formulation presented in the previous section.

A*-based anytime algorithms use various strategies to convert general A* algorithm into anytime algorithms. Hansen and Zhou used a weighted A* approach to produce an anytime algorithm named anytime heuristic search (AHS) [23]. Likhachev et al. proposed ARA* to improve AHS by gradually tightening the weights during each iteration [24]. Lately, Likhachev et al. extended ARA* to dynamic graphs [25]. Both AHS and ARA* require parameters, thus an anytime nonparametric A* (ANA*) was designed to avoid parameters input [26]. In order to speed up the time of finding the first solution, an algorithm named Anytime Window A* (AWA*) was proposed [27]. Recently, Vadlamudi et al. combined AWA* and MA* to propose a memory-bounded anytime heuristic-search algorithm MAWA*, which could work under any time or memory restrictions [28]. MAWA* may perform poorly on graphs since the algorithm has no duplication detection. The memory requirement of above algorithms except MAWA* is almost the same as A* algorithm, hereby the high space usage is still the main weakness that limits them to small size problems.

Beam-based anytime algorithms are developed based on beam search algorithms and utilize various approaches to make beam search complete. Zhang developed a complete beam-based anytime search algorithm (CBS) using a technique called iterative weakening [29]. Zhou and Hansen described a different method named beam stack search (BSS) to convert beam search into an anytime algorithm [30]. BSS adopts a beam stack data structure in order to integrate beam search and backtracking. BSS first executes a beam search and then continues to search using backtracking method. ABULB is another anytime algorithm based on beam search proposed by Furcy, which is a local search algorithm by using ITSA* to locally optimize the initial solution [31].

2.3 Memory Efficient Algorithms

There are very few memory efficient exact algorithms for MLCS problem. On the other hand, some approximate algorithms have been developed to use much less space than exact algorithms in order to handle large data sets [32], [33], [34]. For graph search, many memory efficient algorithms are available, which could be grouped into two catalogues: memory efficient algorithms and memory bounded algorithms. The advantage of our MLCS graph search formulation is that the existing memory efficient graph search algorithms can all be applied to solve MLCS problems.

Memory efficient search algorithms only store the nodes in the frontier of the explored region and the solution is recovered by a divide-and-conquer technique. This strategy was first proposed by Korf to improve the memory efficiency of best-first search algorithms [35], [36], [37]. Lately, Zhou and Hansen developed a closely-related approach called sparse-memory graph search [38]. Different from frontier search, sparse-memory graph search deletes the node until all its neighbors have been expanded. Zhou and Hansen also introduced two similar algorithms, including Sweep-A* for the special-case of lattice graphs and breadth-first heuristic search for the special-case of graphs with unit edge costs [39], [40], [41]. Although these algorithms reduce the memory requirement, the minimum memory requirement of these algorithms is at least the width of the graph.

Memory bounded algorithms work under restricted memory and do not let the memory consumption exceed a given bound. Chakrabarti et al. proposed the first memory bounded algorithm MA* based on A* algorithm [42]. When its memory consumption is within the specified memory restriction, MA* performs A* algorithms. Once the available memory runs out, MA* prunes a set of least promising nodes waiting to be expanded and backs up their minimum cost to their parent node. Russell simplified and improved upon MA* to develop SMA* [43]. Kaindl and Khorsand extended SMA* to a graph search algorithm SMAG* [44]. Zhou and Hansen improved the efficiency of SMAG* [45]. Although these memory-bounded algorithms could guarantee the memory bound, they need much longer time to find a good suboptimal solution.

3 SA-MLCS: A New Memory-Efficient Anytime Algorithm with Small Space Increase Rate

In this section, we present a new memory-efficient anytime algorithm for MLCS, SA-MLCS, based on our graph search formulation of dominant points. Before describing its detailed procedure, some data structures are first introduced.

Nodes in the search graph are grouped into different layers based on their distances from the source node. Hence the search graph is divided into layers, one for each depth. For each layer, SA-MLCS maintains three priority queues: New, Open and Closed. The New queue stores the nodes to be expanded, where expanding a node means generating all its child nodes; the Open queue stores the nodes whose

Iteration 1

	Closed	Open	New
Layer 0		[0, 0]	~~[0, 0]~~
Layer 1		[1, 2]	~~[1, 2]~~ ~~[3, 1]~~
Layer 2		[3, 3]	~~[3, 3]~~ ~~[2, 6]~~
Layer 3	[5, 4]		~~[5, 4]~~ ~~[4, 6]~~
Layer 4	[8, 5]		~~[8, 5]~~
Layer 5	[9, 7]		~~[9, 7]~~

Iteration 2

	Closed	Open	New
Layer 0	[0, 0]		
Layer 1	[3, 1]	[1, 2]	~~[3, 1]~~
Layer 2	[3, 3]	[5, 2]	~~[5, 2]~~ ~~[2, 6]~~
Layer 3	[5, 4] [4, 6]		~~[4, 6]~~ ~~[8, 3]~~
Layer 4	[8, 5]		
Layer 5	[9, 7]		

Iteration 3

	Closed	Open	New
Layer 0	[0, 0]		
Layer 1	[3, 1] [1, 2]		
Layer 2	[3, 3] [2, 6]	[5, 2]	~~[2, 6]~~
Layer 3	[5, 4] [4, 6] [3, 7]		~~[3, 7]~~ ~~[8, 3]~~
Layer 4	[8, 5] [4, 8]		~~[4, 8]~~
Layer 5	[9, 7] [5, 9]		~~[5, 9]~~

Iteration 4

	Closed	Open	New
Layer 0	[0, 0]		
Layer 1	[3, 1] [1, 2]		
Layer 2	[3, 3] [5, 2] [2, 6]		
Layer 3	[5, 4] [4, 6] [3, 7] [8, 3]		~~[8, 3]~~
Layer 4	[8, 5] [4, 8]		
Layer 5	[9, 7] [5, 9]		

Fig. 3. The values of the Closed, Open and New queue when applying SA-MLCS on the graph in Fig. 2. There are four iterations. All the points in New are crossed out since they are removed at the end of each iteration.

children have not all been expanded; and the Closed queue stores the nodes whose children have all been expanded. Let New_i, $Open_i$ and $Closed_i$ denote the New, Open and Closed queue of layer i, respectively. In addition, each node contains a Boolean array in which a bit is used to indicate whether a child is expanded. Let $BoolA_P$ denote the Boolean array of node P and $BoolA_P[i]$ for the ith child. If child i is expanded, $BoolA_P[i]$ is marked with 1.

The New queue and Closed queue are ordered based on the value of function h, which is defined as $h(P) = p_1 + p_2 + \cdots + p_n$ for node $P = [p_1, p_2, \ldots, p_n]$. The Open queue is ordered by function Chi_h, which is defined as follows:

$$Chi\_h(P) = \min_{1 \le i \le |\Sigma| \text{ and } BoolA_P[i]=0} \{h(\text{P's } i\text{th child})\}. \quad (2)$$

SA-MLCS adopts an iterative beam widening search strategy to find the longest path in the graph. It gradually widens the search region during successive iterations and remembers some search information of each iteration. The outline of SA-MLCS is as follows:

1) Add source node S into New_0 and initialize $Open_0$ and $Closed_0$ as ϕ.
2) Set two variables $init\_layer$ and cur_layer as 0, where $init\_layer$ and cur_layer denote the initial layer needing to be processed in each iteration and the current layer being processed, respectively.
3) For all the nodes in New_{cur_layer}, generate their children and add them into $New_{cur\_layer+1}$. If $cur\_layer \neq 0$, then
   - Mark their parents' corresponding Boolean array bit as 1.
   - If all the children of a parent node have been expanded, move the parent node to $Closed_{cur\_layer-1}$.
   - Otherwise, resort the parent node in $Open_{cur\_layer-1}$.
4) For the first i nodes in $Open_{cur\_layer}$, where $i = \min(c, |Open_{cur\_layer}|)$ and fixed value c represents the widened beam (the same as in the following), generate their unexpanded children and add them into $New_{cur\_layer+1}$.
5) Move all the nodes in New_{cur_layer} to $Open_{cur\_layer}$.
6) Reserve the first i nodes in $New_{cur\_layer+1}$, where $i = \min(c, |New_{cur\_layer+1}|)$, and delete the other nodes.
7) If layer cur_layer is not the last layer, then
   - If $init\_layer = cur\_layer$ and $|Open_{cur\_layer}| = |New_{cur\_layer}| = 0$, $init\_layer = init\_layer + 1$.
   - $cur\_layer = cur\_layer + 1$, and go to step 3 to continue this iteration.
8) If layer cur_layer is the last layer, then
   - Output the new solution by tracing back nodes in layer cur_layer, if cur_layer is larger than the length of the current approximate solution.
   - If $init\_layer = cur\_layer$ and $|Open_{cur\_layer}| = |New_{cur\_layer}| = 0$, terminate.
   - Otherwise, $cur\_layer = init\_layer$ and go to step 3 to start a new iteration.

In each iteration, SA-MLCS expands i newly generated nodes in each layer. Choosing which nodes to expand affects final solution quality. SA-MLCS chooses the best i nodes among all the unexpanded nodes that could be generated from the nodes in New_{cur_layer} and $Open_{cur\_layer}$. Although in step 4, only i nodes are expanded, SA-MLCS could still choose the best i nodes overall. This is because the Open queue is ordered by $Child\_h$ and the best i unexpanded nodes generated from the nodes in $Open_{cur\_layer}$ must come from the first i nodes.

Fig. 3 illustrates the application of SA-MLCS on the graph in Fig. 2, showing the values of the Closed, Open and New queue for the total four iterations. In this example, the widened beam parameter c is set to 1. In the first iteration, nodes [0,0] [1,2] [3,1] [3,3] [2,6] [5,4] [4,6] [8,5] [9,7] are generated, among which [0,0] [1,2] [3,3] [5,4] [8,5] [9,7] are chosen to be expanded and the others are removed. These nodes constitute one of the optimal solutions. As [0,0] [1,2] [3,3] have unexpanded children, they are moved into the Open queue. [5,4] [8,5] [9,7] have no unexpanded children, thus they are moved into the Closed queue. In the second iteration, nodes [3,1] [5,2] [2,6] [4,6] [8,3] are generated, among which [3,1] [5,2] [4,6] are chosen to be expanded and the others are removed. As [5,2] has unexpanded children, it is moved into the Open queue. [0,0] [3,1] [3,3] [4,6] have no unexpanded children, thus they are moved into the Closed queue. In the third iteration, nodes [2,6] [3,7] [8,3] [4,8] [5,9] are generated, among which [2,6] [3,7] [4,8] [5,9] are chosen to be expanded and [8,3] is removed. [1,2] [2,6] [3,7] [4,8] [5,9] have no unexpanded

children, thus they are moved into the Closed queue. At last, node [8,3] is generated and expanded. [5,2] [8,3] have no unexpanded children, thus they are move into the Closed queue. Since all the Open queue and New queue are empty, the algorithm terminates.

Without loss of generality, we assume all the sequences are of the same length and use m to denote the length of sequences.

Theorem 1. *In each iteration, the number of newly stored nodes is at most cm, where constant c represents the widened beam.*

Proof. In each iteration, the newly generated nodes are stored in the New queue and then moved to the Open queue. Since in step 5, at most c nodes are kept in the New queue and the number of layers is at most m, the number of newly stored nodes is at most cm in each iteration. □

Assume the total number of nodes stored by SA-MLCS is y_i after the ith iteration. Then the following theorem can be obtained.

Theorem 2. *In the ith iteration, SA-MLCS takes $O(cm(|\Sigma|n + \log y_i))$ time.*

Proof. Suppose there are j layers in the ith iteration and each layer l contains x_l nodes. Then $x_1 + x_2 + \cdots + x_j \leq y_i$. The insertion time of priority queue with size x is $O(\log x)$. In step 3, there are at most c parent nodes needing to be moved or resorted. This operation takes $O(c\log x_l)$ time for layer l. The expansion operation takes $O(c|\Sigma|n)$ time as the calculation of heuristic function for each node needs $O(n)$ time. Similarly, in step 4, the expansion operation also takes $O(c|\Sigma|n)$ time. In addition, the insertion operation in steps 3 and 4 takes $O(c\log c)$ time. This is because only the first c nodes should be ordered considering step 6 reserves at most c nodes in the new layer. Therefore steps 3 and 4 take $O(c|\Sigma|n + c\log x_l)$ time for layer l. Step 5 requires $O(c\log x_l)$ time for layer l. Steps 6, 7, and 8 take constant time. Thus, layer l takes $O(c|\Sigma|n + c\log x_l)$ time. The total time for the ith iteration is $O(c|\Sigma|nj + c(\log x_1 + \log x_2 + \cdots + \log x_j)) = O(c|\Sigma|nj + cj(\log(x_1 + x_2 + \cdots + x_j) - \log j)) = O(c|\Sigma|nj + cj\log y_i)$. Since $j \leq m$, SA-MLCS takes $O(cm(|\Sigma|n + \log y_i))$ time for each iteration. □

Theorem 3. *The first approximate solution can be obtained with $O(cnm)$ space and $O(c|\Sigma|nm)$ time.*

Proof. The first approximate solution could be obtained after the first iteration. According to Theorem 1, the first iteration stores cm nodes. As each node needs $O(n)$ space to store the point, the first iteration consumes $O(cnm)$ space. According to Theorem 2, the first iteration takes $O(c|\Sigma|nj + cj\log(cj) - \log j) = O(c|\Sigma|nj + cj\log c) = O(c|\Sigma|nm)$ time as $\log c < |\Sigma|n$. Therefore, SA-MLCS needs $O(cnm)$ space and $O(c|\Sigma|nm)$ time to output the first approximate solution. □

Theorem 4. *Assume the graph contains at most N nodes, then SA-MLCS takes $O(cmN\log N)$ time to finish.*

Proof. Suppose there are j layers and for each iteration at least one new node is stored. Thus the number of iterations is at most N. The total running time is $O(c|\Sigma|nmN + cm(\log y_1 + \log y_2 + \cdots + \log N)) = O(cmN(|\Sigma|n + \log N)) = O(cmN\log N)$ as $|\Sigma|n < \log N$. □

SA-MLCS reduces the memory usage in each iteration by pruning the nodes waiting to be expanded in the New queue. These pruned nodes could be regenerated from the nodes in the Open queue. Because at most c nodes in the Open queue would be expanded for each layer in each iteration, the total extra operation for regenerating nodes takes $O(cmN|\Sigma|)$ time. Compared with the time complexity $O(cmN\log N)$, the extra time is much less since $N >> |\Sigma|$.

Current state-of-the-art algorithms for MLCS problem that could handle large data sets includes MLCS-APP [34] and Pro-MLCS. MLCS-APP is an approximate algorithm and its time and space complexities are both $O(nm^2)$. Pro-MLCS is an anytime algorithm and it needs $O(c|\Sigma|nm)$ time and $O(c|\Sigma|nm)$ space to output the first approximate solution. Similarly, SA-MLCS is also an anytime algorithm. According to Theorem 3, SA-MLCS needs $O(c|\Sigma|nm)$ time and $O(cnm)$ space to obtain the first approximate solution. Therefore, SA-MLCS outperforms MLCS-APP in terms of both time and space complexities, and outperforms Pro-MLCS in terms of space complexity to achieve approximate solutions.

Current state-of-the-art anytime algorithms for graph search could be categorized into two groups, A*-based anytime algorithms and beam-based anytime algorithms. Compared with A*-based anytime algorithms, SA-MLCS only stores part of nodes in each iteration so that it uses less space. In addition, SA-MLCS maintains a priority for each layer and the size of the priority queue for a layer is far smaller than that of the priority queue for A*-based algorithms, which can make the algorithm find solutions faster. Compared with existing beam-based anytime algorithms, SA-MLCS reuses search information from previous iterations, which can improve the time efficiency and guarantee the selection of the best nodes in the frontier.

4 SLA-MLCS: A New Space Bounded Anytime Algorithm

Although SA-MLCS reduces the space usage in each iteration, its space usage will still exceed the available storage when the entire search graph cannot be stored. SLA-MLCS is an extension to SA-MLCS to work under bounded storage. It adopts a replacement method when the space usage of SA-MLCS reaches the memory bound. In each node, SLA-MLCS maintains a variable to indicate whether all its descendants have been expanded completely. Only the nodes whose descendants have all been expanded could be replaced.

The procedure of SLA-MLCS is as follows:

1)-5) are the same as SA-MLCS.
6) Check whether the memory bound is reached.

- If not, keep the first i nodes in $New_{cur\_layer+1}$, where $i = \min(c, |New_{cur\_layer+1}|)$ and delete the other nodes.
- Otherwise, keep the first i nodes in $New_{cur\_layer+1}$, where $i = \min(c, |New_{cur\_layer+1}|, x)$ (x is the

TABLE 1
Comparison of the New Algorithm SA-MLCS with Approximate Algorithm MLCS-APP and Anytime Algorithm Pro-MLCS on Small-Size Test Instances, Where n, m and $Length^*$ Represent the Number of Sequences, the Length of Sequences and the Length of Solutions, Respectively

(a) DNA sequences with $|\Sigma| = 4$

n	m	MLCS-APP			Pro-MLCS			SA-MLCS		
		Time	Space	Length*	Time	Space	Length*	Time	Space	Length*
10	4000	3.34s	207MB	1424	0.1s	95MB	1476	0.13s	13MB	1476
	6000	5.57s	438MB	2124	0.12s	190MB	2191	0.15s	18MB	2191
	8000	8.29s	757MB	2838	0.16s	316MB	2952	0.2s	23MB	2952
15	4000	4.51s	318MB	1321	0.096s	99MB	1348	0.1s	14MB	1348
	6000	7.69s	665MB	1968	0.13s	194MB	2018	0.15s	19MB	2018
	8000	11.36s	1143MB	2609	0.17s	322MB	2699	0.2s	25MB	2699
20	4000	5.47s	395MB	1273	0.1s	101MB	1284	0.11s	16MB	1284
	6000	9.42s	825MB	1905	0.14s	198MB	1921	0.17s	22MB	1921
	8000	14.13s	1430MB	2541	0.19s	328MB	2571	0.23s	28MB	2571

(b) Protein sequences with $|\Sigma| = 20$

n	m	MLCS-APP			Pro-MLCS			SA-MLCS		
		Time	Space	Length*	Time	Space	Length*	Time	Space	Length*
10	4000	4.56s	180MB	405	0.19s	106MB	421	0.18s	12MB	421
	6000	7.43s	383MB	605	0.28s	206MB	635	0.26s	16MB	635
	8000	10.33s	669MB	798	0.38s	338MB	841	0.35s	21MB	841
15	4000	5.72s	276MB	350	0.23s	108MB	355	0.2s	15MB	355
	6000	9.54s	598MB	529	0.35s	210MB	535	0.3s	21MB	535
	8000	14.1s	1048MB	700	0.45s	343MB	711	0.38s	27MB	711
20	4000	6.81s	344MB	324	0.29s	112MB	325	0.27s	18MB	325
	6000	11.24s	746MB	481	0.45s	216MB	492	0.4s	26MB	492
	8000	16.63s	1304MB	647	0.6s	350MB	652	0.53s	34MB	652

number of nodes in $Closed_{cur\_layer+1}$ that could be replaced) and delete the other nodes in $New_{cur\_layer+1}$ and i nodes in $Closed_{cur\_layer+1}$ that could be replaced.

7) If layer cur_layer is not the last layer, then

- If $init\_layer = cur\_layer$ and $|Open_{cur\_layer}| = |New_{cur\_layer}| = 0$, $init\_layer = init\_layer + 1$.
- $cur\_layer = cur\_layer + 1$, and go to step 3 to continue this iteration.

8) If layer cur_layer is the last layer, then

- Output the new solution by tracing back nodes in layer cur_layer, if cur_layer is larger than the length of the current approximate solution.
- If $init\_layer = cur\_layer$ and $|Open_{cur\_layer}| = |New_{cur\_layer}| = 0$, terminate.
- Check whether the memory bound is reached.
  - If not, $cur\_layer = init\_layer$.
  - Otherwise, iteratively check all the nodes in $Closed_{cur\_layer}$ to see whether they could be replaced. If could, mark it and recursively check its parent. After that, find the first layer with nodes that could be replaced and set cur_layer to that layer.
- Go to step 3 to start a new iteration.

SLA-MLCS makes a tradeoff between time and space. If a node can be generated by more than one node and it is deleted from the Closed queue, duplication detection would be ineffective when it is generated by other nodes. This results in more computational time than SA-MLCS. However it can be guaranteed that the space usage of SLA-MLCS will not exceed the memory bound.

Different from existing memory bounded search algorithms, SLA-MLCS replaces the nodes whose descendants have all been expanded. Otherwise, if a node with unexpanded descendants is removed, its parent nodes should be marked signaling that this child needs to be regenerated. This results in the inefficient duplicated regeneration of a node and its descendants from the same parent node.

5 Experimental Results

In this section, we compare the experimental performances of the new algorithms with existing state-of-the-art algorithms on MLCS problems of various sizes.

The experiments were conducted on a Windows server with Core i5-2400 3.1GHz CPU and 4G RAM. DNA sequences and protein sequences were used in the experiments due to their different sizes of Σ, 4 for DNA sequences and 20 for protein sequences.

First, in order to test the performance of SA-MLCS's first solution, SA-MLCS is compared with approximate algorithm MLCS-APP and anytime algorithm Pro-MLCS on small data sets because MLCS-APP could not handle sequences longer than 10,000. Ten data sets were generated for each test. Table 1 shows the running time, space usage and solution quality (length of solution) of these three algorithms on different number of sequences and sequence length. As Pro-MLCS is also an anytime algorithm, we show the results of its first solution. SA-MLCS and Pro-MLCS have similar speed and both are an order of magnitude faster than

TABLE 2
Performance Comparison of Three Anytime Algorithms in Finding Increasingly Better Solutions over Large Size Test Instances of 100 Sequences

(a) DNA sequences with $|\Sigma| = 4$

n=100													
m=10000							m=15000						
Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS		Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS	
	Time	Space	Time	Space	Time	Space		Time	Space	Time	Space	Time	Space
2696	0.78s	631MB	0.8s	92MB	0.8s	92MB	4089	-	-	1.3s	138MB	1.3s	138MB
2728	1.64s	820MB	1.9s	148MB	1.9s	148MB	4095	-	-	4.8s	306MB	4.8s	306MB
2739	-	-	5.4s	316MB	5.4s	316MB	4109	-	-	8.4s	474MB	8.4s	474MB
2753	-	-	20s	935MB	20s	935MB	4115	-	-	18.3s	896MB	18.3s	896MB
2762	-	-	-	-	22s	1000MB	4123	-	-	-	-	25s	1000MB
2767	-	-	-	-	43s	1000MB	4136	-	-	-	-	67s	1000MB

(b) Protein sequences with $|\Sigma| = 20$

n=100													
m=10000							m=15000						
Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS		Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS	
	Time	Space	Time	Space	Time	Space		Time	Space	Time	Space	Time	Space
576	5.1s	685MB	5.2s	173MB	5.2s	173MB	871	-	-	8.28s	258MB	8.28s	258MB
580	16s	919MB	17s	324MB	17s	324MB	876	-	-	17.6s	374MB	17.6s	374MB
585	-	-	35s	551MB	35s	551MB	879	-	-	44.8s	718MB	44.8s	718MB
587	-	-	47s	704MB	47s	704MB	897	-	-	54.2s	838MB	54.2s	838MB
588	-	-	-	-	60s	1000MB	901	-	-	-	-	63.9s	1000MB
589	-	-	-	-	67s	1000MB	902	-	-	-	-	73.6s	1000MB

n and m Represent the number of sequences and the length of sequences, respectively.

MLCS-APP. SA-MLCS and Pro-MLCS have the same solution quality since they use the same heuristic function and find the same solutions. Their solutions are much better than those of MLCS-APP. Among all three algorithms, SA-MLCS consumes the least amount of space, an order of magnitude less than the other two algorithms.

Next, we show the performance of SA-MLCS and SLA-MLCS in dealing with large-size problems, in comparison with Pro-MLCS. MLCS-APP is excluded from this test since it could not run on these data sets. Four data sets were generated for DNA sequences and protein sequences with 100 sequences of varying lengths. Since our aim is to find better solutions within acceptable time and space usage, the time and space limits are set to 120 seconds and $1GB$, respectively. Table 2 shows the results, where "-" denotes an algorithm exceeds the given limits and could not generate a solution. For each data set, the improvement of solutions is shown with the increase of time and space. In obtaining the

TABLE 3
Performance Comparison of Three Anytime Algorithms in Finding Increasingly Better Solutions over Large Size Test Instances with Fixed Sequence Length 10,000

(a) DNA sequences with $|\Sigma| = 4$

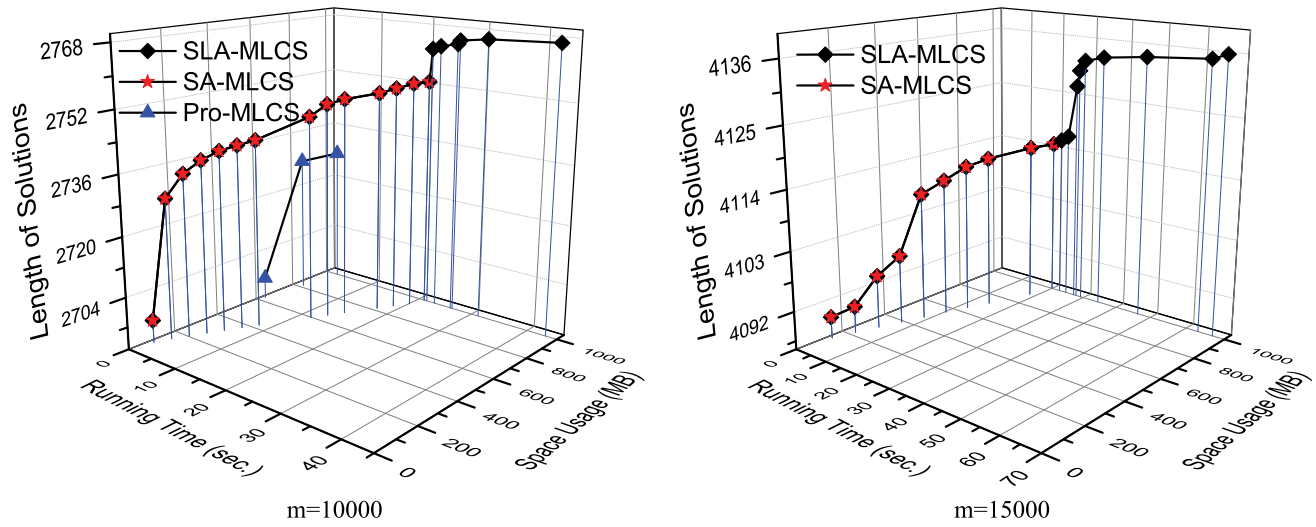
m=10000													
n=150							n=200						
Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS		Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS	
	Time	Space	Time	Space	Time	Space		Time	Space	Time	Space	Time	Space
2645	1.9s	723MB	2s	256MB	2s	256MB	2602	2.6s	811MB	2.7s	165MB	2.7s	165MB
2657	4.2s	983MB	4.4s	355MB	4.4s	355MB	2626	-	-	8.9s	356MB	8.9s	356MB
2680	-	-	26.2s	952MB	26.2s	952MB	2633	-	-	25s	836MB	25s	836MB
2684	-	-	-	-	36.6s	1000MB	2640	-	-	-	-	35s	1000MB
2687	-	-	-	-	40s	1000MB	2644	-	-	-	-	44s	1000MB

(b) Protein sequences with $|\Sigma| = 20$

m=10000													
n=150							n=200						
Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS		Length of Solutions	Pro-MLCS		SA-MLCS		SLA-MLCS	
	Time	Space	Time	Space	Time	Space		Time	Space	Time	Space	Time	Space
539	8.57s	803MB	8.64s	256MB	8.64s	256MB	524	11.3s	922MB	11.4s	340MB	11.4s	340MB
548	-	-	18.2s	355MB	18.2s	355MB	533	-	-	23.8s	433MB	23.8s	433MB
561	-	-	77.4s	952MB	77.4s	952MB	544	-	-	99.8s	910MB	99.8s	910MB
563	-	-	-	-	87s	1000MB	547	-	-	-	-	112s	1000MB
564	-	-	-	-	88s	1000MB	548	-	-	-	-	113s	1000MB

n and m Represent the Number of Sequences and the Length of Sequences, Respectively

(a) DNA sequences with $|\Sigma| = 4$



(b) Protein sequences with $|\Sigma| = 20$

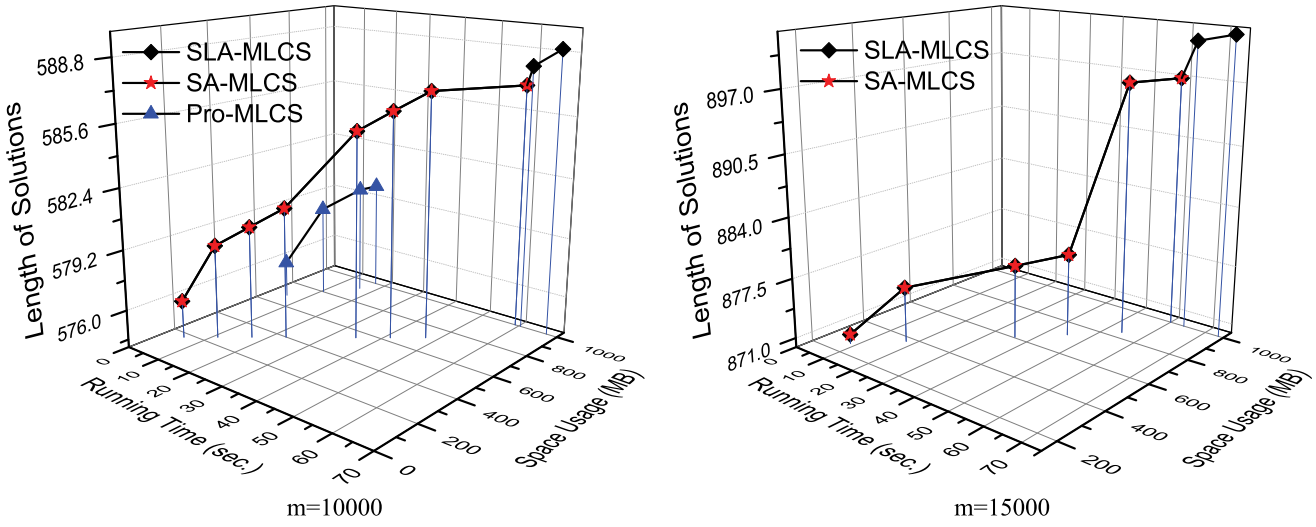


Fig. 4. Improvement of solution quality (length of common sequences) of Pro-MLCS, SA-MLCS and SLA-MLCS on MLCS instances of 100 sequences of different lengths. Since Pro-MLCS could not handle data sets of m=15,000, its result is not shown in the graph for the m=15,000 case.

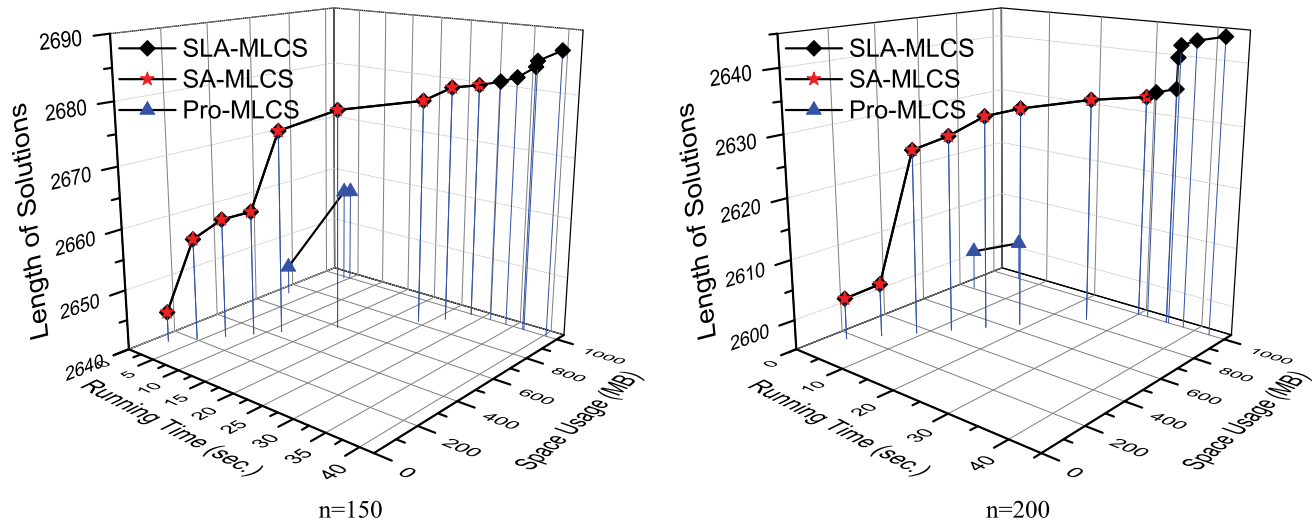
same solution, SA-MLCS utilizes almost the same time as Pro-MLCS, but only 25 percent of the space. Since SA-MLCS reduces space usage, SA-MLCS can find much better solutions than Pro-MLCS with the same space bound. Before reaching the space bound, SLA-MLCS is the same as SA-MLCS. However, SLA-MLCS continues to find better solutions than SA-MLCS after SA-MLCS reaches the space bound. Due to large space usage, Pro-MLCS could not deal with data sets with $\mathrm{m} = 15{,}000$.

Furthermore, SA-MLCS and SLA-MLCS are compared with Pro-MLCS on test cases with varying number of sequences. The length of sequences was fixed at 10,000. The time and space bounds are also set to 120 seconds and $1GB$, respectively. Table 3 shows SLA-MLCS finds better solutions than SA-MLCS and Pro-MLCS. SLA-MLCS and SA-MLCS use much less space than Pro-MLCS.

Since SA-MLCS, SLA-MLCS and Pro-MLCS are anytime algorithms, their solutions would improve with the increase of time and space. Sometimes this improvement is frequent. Tables 2 and 3 only show part of solutions due to the limitation of tables. Thus, Figs. 4 and 5 illustrate the detailed improvement of solution quality over time and space. The results show that SLA-MLCS finds the best solutions with the given time and space bounds and both SA-MLCS and SLA-MLCS use much less space than Pro-MLCS in achieving the same solutions. Since SLA-MLCS is the same as SA-MLCS before SA-MLCS reaching the space bound, their results overlap before SA-MLCS reaches the bound. In addition, Pro-MLCS is not shown for data sets with $n = 100$ and $m = 15{,}000$, because it could not handle this case.

As the data sets for above experiments are generated randomly, in order to test the performance of our algorithms on real biological data, 30 bacteria genomes were obtained from the database of the National Center for Biotechnology Information (NCBI). Table 4 shows the performance of MLCS-APP, Pro-MLCS and SA-MLCS on this data set of

(a) DNA sequences with $|\Sigma| = 4$



(b) Protein sequences with $|\Sigma| = 20$

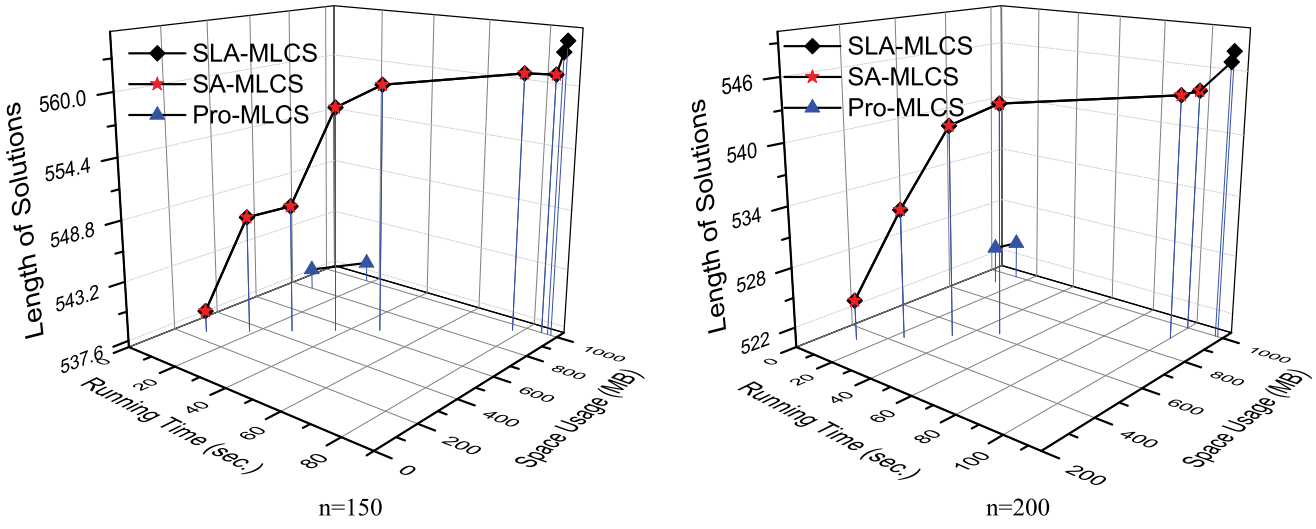


Fig. 5. Improvement of solution quality of Pro-MLCS, SA-MLCS and SLA-MLCS on MLCS instances of different number of sequences with fixed length, 10,000.

varying lengths, where the results of Pro-MLCS and SA-MLCS are obtained from the first iteration. MLCS-APP failed to output a solution when $m = 8{,}000$ as it run out of memory. SA-MLCS consumes the least space and finds better solutions than MLCS-APP. The times of SA-MLCS and Pro-MLCS are very close.

Finally, we push the limit and test the size of data sets that each algorithm can handle. When setting the time and space limits to 120 seconds and $1GB$, respectively, Table 5 shows the output solutions of MLCS-APP, Pro-MLCS, SA-MLCS and SLA-MLCS on 100 bacteria genomes from the database of the NCBI. MLCS-APP

TABLE 4
Comparison of SA-MLCS with MLCS-APP and Pro-MLCS on 30 Bacteria Genomes of Varying Lengths, where m and $Length^*$ Represent the Length of Sequences and the Length of Solutions, Respectively

m	MLCS-APP			Pro-MLCS			SA-MLCS		
	Time	Space	Length*	Time	Space	Length*	Time	Space	Length*
4000	6.75s	562M	1027	0.21s	136M	1051	0.23s	26M	1051
5000	8.92s	857M	1271	0.26s	188M	1298	0.28s	31M	1298
6000	11.8s	1203M	1540	0.31s	251M	1591	0.35s	38M	1591
7000	15.1s	1626M	1813	0.36s	318M	1829	0.4s	43M	1829
8000	-	-	-	0.4s	393M	2051	0.45s	48M	2051

TABLE 5
Scalability Test of MLCS-APP, Pro-MLCS, SA-MLCS and SLA-MLCS on 100 Bacteria Genomes of Varying Lengths, where m Represents the Length of Sequences

m	Length of Solutions			
	MLCS-APP	Pro-MLCS	SA-MLCS	SLA-MLCS
3000	667	668	669	669
4000	-	836	840	850
14000	-	2915	2927	2951
120000	-	-	24884	25985
270000	-	-	-	56327

could only deal with sequences of length up to 3,000, while SLA-MLCS could handle sequences of length up to 270,000, two orders of magnitude larger than MLCS-APP. Besides, as described in Section 2.1, MLCS problem is formulated into a graph search problem, thus graph search algorithms could be applied. Current state-of-art anytime algorithms for graph search include AHS, ARA*, ANA*, beam-stack, AWA*, etc., among which AWA* achieves the best time performance. Therefore, we implemented AWA* to solve MLCS problem, and compared it to MLCS-APP, Pro-MLCS and SA-MLCS on the data sets used in Table 4. Among all algorithms, AWA* finds the worst solution, consumes the most space and costs more time than SA-MLCS, less time than MLCS-APP. For example, AWA* finds a solution with length 1,402 using $3.46s$ time and $1{,}281M$ space for $m = 6{,}000$.

6 Summary

The contribution of this paper is two-fold. One is the formulation of the dominant point method as a graph search problem, which opens the door for the application of existing graph search algorithms in finding longest common subsequence. The other is the development of two space-efficient anytime algorithms, SA-MLCS and SLA-MLCS, for efficiently solve large-size MLCS problems. SA-MLCS adopts an iterative beam widening strategy to reduce space requirement and SLA-MLCS uses a novel replacement technique to keep the search within a given space bound. Both algorithms outperforms existing anytime and approximate algorithms on both randomly generated test cases and real biological cases, using much less space in achieving the same quality solutions or obtaining better solutions within the same space bound. SLA-MLCS is the first space-bounded anytime MLCS algorithm. It can make full use of available storage space to output near-optimal solutions, which makes it very attractive in solving large-size problems, an increasingly important domain in the big data era.

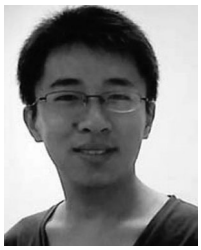
Acknowledgments

Yun Xu is the corresponding author. This work was supported by the National Natural Science Foundation of China (Nos. 61033009 and 61232018) and the Fund for Foreign Scholars in University Research and Teaching Programs (B07033). This work was also supported in part by the United States National Institute of Health (NIH) under Grant R01-GM100701. All findings and results presented in this paper were those of the authors and do not represent those of the funding agencies. Yun Xu is the corresponding author.

References

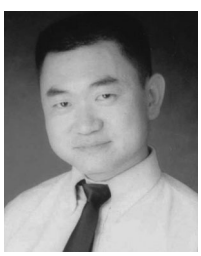
[1] J.W. Hunt and M.D. McIlroy, "An Algorithm for Differential File Comparison," Computing Science Technical Report 41, AT&T Bell Laboratories, 1975.
[2] S.Y. Lu and K.S. Fu, "A Sentence-to-Sentence Clustering Procedure for Pattern Analysis," *IEEE Trans. Systems, Man and Cybernetics*, vol. SMC-8, no. 5, pp. 381-389, May 1978.
[3] D. Sankoff and J.B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley Publication, 1983.
[4] T.K. Attwood and J.B.C. Findlay, "Fingerprinting G-Protein-Coupled Receptors," *Protein Eng.*, vol. 7, no. 2, pp. 195-203, 1994.
[5] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," J. Molecular Biology, vol. 147, pp. 195-197, 1981.
[6] D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *J. ACM*, vol. 25, pp. 322-336, 1978.
[7] I. Simon, "Sequence Comparison: Some Theory and Some Practice," *Electronic Dictionaries and Automata in Computational Linguistics*, vol. 377, pp. 79-92, 1989.
[8] L. Bergroth, H. Hakonen, and T. Raita, "A Survey of Longest Common Subsequence Algorithms," *Proc. Seventh Int'l Symp. String Processing Information Retrieval*, pp. 39-48, 2000.
[9] K. Hakata and H. Imai, "Algorithms for the Longest Common Subsequence Problem for Multiple Strings Based on Geometric Maxima," *Optimization Methods and Software*, vol. 10, pp. 233-260, 1998.
[10] K. Hakata and H. Imai, "Algorithms for the Longest Common Subsequence Problem," *Proc. Genome Informatics Workshop III*, pp. 53-56, 1992.
[11] S.C. Schuster, "Next-Generation Sequencing Transforms Todays Biology," *Nature Methods*, vol. 5, no. 1, pp. 16-18, 2008.
[12] D. Sankoff, "Matching Sequences under Deletion/Insertion Constraints," *Proc. Nat'l Academy Sciences USA*, vol. 69, pp. 4-6, Jan. 1972.
[13] W.J. Hsu and M.W. Du, "Computing a Longest Common Subsequence for a Set of Strings," *BIT Numerical Math.*, vol. 24, no. 1, pp. 45-59, 1984.
[14] D.S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *J. ACM*, vol. 24, pp. 664-675, 1977.
[15] A. Apostolico, S. Browne, and C. Guerra, "Fast Linear-Space Computations of Longest Common Subsequences," *Theoretical Computer Science*, vol. 92, no. 1, pp. 3-17, 1992.
[16] W.J. Masek and M.S. Paterson, "A Faster Algorithm Computing String Edit Distances," *J. Computer System Sciences*, vol. 20, pp. 18-31, 1980.
[17] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. Very Large Data Bases Conf. (VLDB '02)*, pp. 275-286, 2002.
[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive Skyline Computation in Database Systems," *ACM Trans. Database Systems*, vol. 30, pp. 41-82, 2005.
[19] Q. Wang, D. Korkin, and Y. Shang, "A Fast Multiple Longest Common Subsequence (MLCS) Algorithm," *IEEE Trans. Knowledge and Data Eng.*, vol. 23, no. 3, pp. 321-334, Mar. 2011.
[20] Y. Chen, A. Wan, and W. Liu, "A Fast Parallel Algorithm for Finding the Longest Common Sequence of Multiple Biosequences," *BMC Bioinformatics*, vol. 7, no. Suppl. 4, article S4, 2006.
[21] D. Korkin, Q. Wang, and Y. Shang, "An Efficient Parallel Algorithm for the Multiple Longest Common Subsequence (MLCS) Problem," *Proc. 37th Int'l Conf. Parallel Processing (ICPP '08)*, pp. 354-363, 2008.

[22] J. Yang, Y. Xu, G. Sun, and Y. Shang, "A New Progressive Algorithm for Multiple Longest Common Subsequences Problem and Its Efficient Parallelization," *IEEE Trans. Parallel and Distributed Systems*, vol. 24, no. 5, pp. 862-870, May 2013.
[23] E.A. Hansen and R. Zhou, "Anytime Heuristic Search," *J. Artificial Intelligence Research*, vol. 28, pp. 267-297, 2007.
[24] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* with Provable Bounds on Sub-Optimality," *Proc. Advances in Neural Information Processing Systems (NIPS)*, vol. 16, 2003.
[25] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime Search in Dynamic Graphs," *Artificial Intelligence*, vol. 172, no. 14, pp. 1613-1643, 2008.
[26] J.van den Berg, R. Shah, A. Huang, and K. Goldberg, "ANA*: Anytime Nonparametric A," *Proc. AAAI Conf. Artificial Intelligence*, 2011.
[27] S. Aine, P.P. Chakrabarti, and R. Kumar, "AWA*-A Window Constrained Anytime Heuristic Search Algorithm," *Proc. Int'l Joint Conf. Artificial Intelligence*, pp. 2250-2255, 2007.
[28] S.G. Vadlamudi, S. Aine, and P.P. Chakrabarti, "MAWA*-A Memory-Bounded Anytime Heuristic-Search Algorithm," *IEEE Trans. Systems, Man, and Cybernetics. Part B, Cybernetics*, vol. 41, no. 3, pp. 725-735, June 2011.
[29] W. Zhang, "Complete Anytime Beam Search," *Proc. Nat'l Conf. Artificial Intelligence*, pp. 425-430, 1998.
[30] R. Zhou and E.A. Hansen, "Beam-Stack Search: Integrating Backtracking with Beam Search," *Proc. Int'l Conf. Automated Planning and Scheduling (ICAPS)*, pp. 90-98, 2005.
[31] D. Furcy, "ITSA*: Iterative Tunneling Search with A*," *Proc. Nat'l Conf. Artificial Intelligence (AAAI), Workshop Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.
[32] C. Blum, M.J. Blesa, and M. López-Ibáñez, "Beam Search for the Longest Common Subsequence Problem," *Computers Operations Research*, vol. 36, no. 12, pp. 3178-3186, 2009.
[33] C. Blum and M.J. Blesa, "Probabilistic Beam Search for the Longest Common Subsequence Problem," *Proc. Int'l Conf. Eng. Stochastic Local Search Algorithms. Designing, Implementing and Analyzing Effective Heuristics*, vol. 4638, pp. 150-161, 2007.
[34] Q. Wang, M. Pan, Y. Shang, and D. Korkin, "A Fast Heuristic Search Algorithm for Finding the Longest Common Subsequence of Multiple Strings," *Proc. 24th AAAI Conf. Artificial Intelligence*, pp. 1287-1292, 2010.
[35] R.E. Korf, "Divide-and-Conquer Bidirectional Search: First Results," *Proc. 16th Int'l Joint Conf. Artificial Intelligence (IJCAI '99)*, pp. 1184-1189, 1999.
[36] R.E. Korf and W. Zhang, "Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment," *Proc. 17th Nat'l Conf. Artificial Intelligence (AAAI '00)*, pp. 910-916, 2000.
[37] R.E. Korf, W. Zhang, I. Thayer, and H. Hohwald, "Frontier Search," *J. ACM*, vol. 52, no. 5, pp. 715-748, 2005.
[38] R. Zhou and E. Hansen, "Sparse-Memory Graph Search," *Proc. 18th Int'l Joint Conf. Artificial Intelligence (IJCAI '03)*, pp. 1259-1266, 2003.
[39] R. Zhou and E.A. Hansen, "Sweep-A*: Space-Efficient Heuristic Search in Partially Ordered Graphs," *Proc. 15th Int'l Conf. Tools with Artificial Intelligence (ICTAI '03)*, pp. 427-434, 2003.
[40] R. Zhou and E.A. Hansen, "Breadth-First Heuristic Search," *Proc. 14th Int'l Conf. Automated Planning and Scheduling (ICAPS '04)*, pp. 92-100, 2004.
[41] R. Zhou and E.A. Hansen, "Breadth-First Heuristic Search," *Artificial Intelligence*, vol. 170, no. 4, pp. 385-408, 2006.
[42] P.P. Chakrabarti, S. Ghose, A. Acharya, and S.C. de Sarkar, "Heuristic Search in Restricted Memory," *Artificial Intelligence*, vol. 41, no. 2, pp. 197-221, 1989.
[43] S. Russell, "Efficient Memory-Bounded Search Methods," *Proc. 10th European Conf. Artificial Intelligence (ECAI '92)*, pp. 1-2, 1992.
[44] H. Kaindl and A. Khorsand, "Memory-Bounded Bidirectional Search," *Proc. 12th Nat'l Conf. Artificial Intelligence (AAAI)*, vol. 2, pp. 1359-1364, 1994.
[45] R. Zhou and E.A. Hansen, "Memory-Bounded A. Graph Search," *Proc. 15th Int'l Florida Artificial Intelligence Research Society Conf.*, pp. 203-209, 2002.



Jiaoyun Yang received the BS degree in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 2009. He is currently working toward the PhD degree in the Department of Computer Science at USTC. His research interests include parallel algorithms, biological sequence analysis and mining, and high performance computing.

Yun Xu received the PhD degree in computer science from the University of Science and Technology of China (USTC), Hefei, in 2002. He is currently an associate professor with the School of Computer Science of USTC, a member of National High Performance Computing Center at Hefei, and also the supervisor of PhD students under the collaboration scheme between the City University of Hong Kong and USTC. Now, he is leading a group of research students in some high-performance computing and bioinformatics research. His research interests include bioinformatics, biological sequence analysis and mining, parallel algorithms, and parallel programming model, and performance optimization. He has authored or coauthored more than 50 papers. He is a member of the IEEE and the ACM.



Yi Shang received the BS degree from the University of Science and Technology of China, Hefei, in 1988, the MS degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 1991, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1997. He is a professor and the director of Graduate Studies in the Computer Science Department, University of Missouri, Columbia. He has published more than 140 refereed papers in the areas of nonlinear optimization, wireless sensor networks, mobile computing, intelligent systems, and bioinformatics, and received funding from US National Science Foundation (NSF), NIH, Army, US Defense Advanced Research Projects Agency (DARPA), Microsoft, and Raytheon. He is a lifetime member of the ACM and a senior member of the IEEE.

Guoliang Chen received the BSc degree from Xi'an Jiaotong University, China, in 1961. Since 1973, he has been with the University of Science and Technology of China, Hefei, China, a professor with the Department of Computer Science and Technology, and the director of the School of Software Engineering. From 1981 to 1983, he was a visiting scholar at Purdue University, Indiana. He is currently also the director of the National High Performance Computing Center at Hefei. His research interests include parallel algorithm, computer architecture, computer network, and computational intelligence. He has published nine books and more than 200 research papers. He is an Academician of Chinese Academy of Sciences. He was the recipient of the National Excellent Teaching Award of China in 2003.