TECHINAL NOTE

International Journal of Operations Research

International Journal of Operations Research Vol. 4, No. 2, 98-104 (2007)

A Specialized Branching and Fathoming Technique for The Longest Common Subsequence Problem

Todd Easton* **and Abhilash Singireddy**

237 Durland Hall, School of Industrial and Manufacturing Systems Engineering, Kansas State, University, Manhattan, Kansas, 66506

Received January 2006; *Revised October 2006*; *Accepted December 2006*

Abstract—Given a set $S = \{S^1, ..., S^k\}$ of finite strings, the k-longest common subsequence problem (k-LCSP) seeks a string L of maximum length such that L is a subsequence of each S^i for $i = 1, ..., k$. This paper presents a technique, specialized branching, that solves k-LCSP. Specialized branching combines the benefits of both dynamic programming and branch and bound to reduce the search space. For large k, this method is shown to be computationally superior to dynamic programming.

Keywords—Longest common subsequence, Branch and bound, Dynamic programming

1. INTRODUCTION

Given a finite alphabet set $\sum$ a string, also known as a sequence, is an ordered set of symbols drawn from $\sum$ (repeats are allowed). A subsequence of a string is obtained by deleting 0 or more (not necessarily consecutive) symbols from the string. Given finite strings $S^1, ..., S^k$, the k-longest common subsequence problem (k-LCSP) seeks a string L of maximum length such that L is a subsequence of each S^i for $i = 1, ..., k$. The string L is referred to as the longest common subsequence (LCS) and $|L|$ equals the length of the longest common subsequence (LLCS). This paper presents a new algorithm, called specialized branching (SB), that optimally solves k-LCSP instances.

Finding the longest common subsequence is a combinatorial optimization problem with numerous applications. The majority of the applications use the LCS to compare the similarity between different sets of data. Some of these applications include: the homology of macromolecules such as proteins and nucleic acids (Dayhoff (1969), Smith and Waterman (1981), Sankoff and Kruskal (1983), Bafna et al. (1995) and Jiang et al. (2002)), file comparison (Hunt and McIlroy (1975), Hunt and Szymanski (1977), Aho et al. (1983)), artificial intelligence (Hayes (1989), Jiang and Li (1995)), data compression (Wagner (1973), Storer (1988)), syntactic pattern recognition (Lu and Fu (1978)), text editing (Sankoff and Kruskal (1983)), categorizing visitors based on their interactions on a website (Banerjee and Ghosh (2001)), query optimization in database systems (Sellis (1988)) and the production of smaller circuits in field programmable gate arrays (Brisk et al. (2004)).

The status of k-LCSP is *NP*-complete in general (Maier (1978)), but solvable in polynomial time for any fixed k by dynamic programming (Wagner and Fischer (1974)). The majority of the research involving k-LCSP has been focused on instances where $k = 2$ and 3. However, Gallant et al. (1980), Itoga (1981), Hsu and Du (1984), Irving and Fraser (1992) and Hakata and Imai (1992) have all examined instances where $k \geq 4$. The methodology, specialized branching, presented in this paper is a technique that can computationally solve k-LCSP for large k.

The remainder of the paper is organized as follows. Section 2 explains algorithms that solve k-LCSP. Section 3 provides computational results and the paper concludes in Section 4 with some conclusions and future research.

2. ALGORITHMS THAT SOLVE *k*-LCSP

This section presents two algorithms that solve k-LCSP. Dynamic programming (DP) has been extensively studied and works quickly when $k = 2$ or 3. A new method, specialized branching (SB), is presented here, which is a tree-searching algorithm and works quite well for $k \geq 6$. An integer programming method (Singireddy (2003)) that solves k-LCSP was attempted, but specialized branching always dominated the integer programming technique.

2.1 Dynamic programming

Dynamic programming (DP) was the first method used to solve k-LCSP. Here, we provide a brief description of this method. The interested reader can find a more complete description in Larson (1968). Consider strings S^1, S^2, ..., S^k drawn from $\sum$. For the remainder of the paper, we will assume that $|S^1| = |S^2| = ... = |S^k| = n$. If not, let $n = \max\{|S^i| : i = 1, ..., k\}$, then to each S^i add on $n - |S^i|$ dummy letters. Trivially, such a transformation will not change the LCS.

* Corresponding author's email: teaston@ksu.edu

If the first letter of each S^i is equal, then by definition, the LCS must contain this symbol as the first symbol; if not, then the first letter of the LCS can be the first letter of any one of the sequences or another letter from $\sum$. When any of these two cases occurs, dropping the appropriate symbol(s) from the corresponding string(s) does not change the LCS. This observation leads directly to the recursive nature of dynamic programming.

Let $M_{i_1,\ldots,i_k}$ be the LLCS of the strings $S^1_{i_1} = s^1_1 \ldots s^1_{i_1}, \ldots, S^k_{i_k} = s^k_1 \ldots s^k_{i_k}$. The recursion formula from the k-LCSP is

$$M_{i_1,\ldots,i_k} = \begin{cases} M_{i_1-1,\ldots,i_k-1} + 1 & \text{if } s^1_{i_1} = s^2_{i_2} = \ldots = s^k_{i_k} \\ \max\{M_{i_1-1,i_2,\ldots,i_k}, M_{i_1,i_2-1\ldots,i_k}, \ldots, M_{i_1,i_2\ldots,i_k-1}\} & \text{otherwise} \end{cases}$$

where the boundary conditions are:

$$M_{i_1,\ldots,i_k} = 0 \quad \text{if } i_1 = 0 \text{ or } i_2 = 0 \text{ or } \ldots \text{ or } i_k = 0.$$

The boundary conditions simply specify that the LLCS between any set of strings and the empty string is zero. Since calculating $M_{i_1,\ldots,i_k}$ requires a constant amount of effort, DP's time and space requirements are both $O(n^k)$.

When restricted to 2-LCSP, Hirshberg (1975) observed that calculating $M_{1,2}$ is dependent on its previous row and previous column in matrix M, which resulted in an algorithm with $O(n^2)$ time, but only $O(n)$ space. Irving and Fraser (1992) and Hakata and Imai (1992) have extended this concept to arbitrary k and reduced either the run time or storage requirement from $O(n^k)$ to $O(n^{k-1})$. Even with these improvements, dynamic programming is not suitable to computationally solve k-LCSPs for $k \geq 7$.

Besides the exponential (in the number of sequences) storage and run time, an additional weakness of dynamic programming algorithms is that it does not use the inherent structure of the input sequences. That is, an individual can trivially verify that k sequences are identical; yet, dynamic programming will require the full $O(n^k)$ time and space to obtain the same conclusion.

2.2 A specialized branching algorithm

The specialized branching algorithm (SB) presented here is an enumeration algorithm that combines some aspects of dynamic programming and branch and bound (Land and Doig, (1960)) to limit its search tree. Unlike standard branch and bound, which has two child nodes under each node, when SB branches, $|\sum|$ child nodes are created. Each one of these nodes corresponds to one of the letters in $\sum$. Each node in SB's search tree contains a k-tuple (i_1, ..., i_k), which describes the location of markers for each of the sequences. To determine the k-tuple of a child node, merely scan each sequence, beginning with the letter after the parent node's marker, until the first occurrence of the child node's letter is found. If at least one sequence doesn't have this letter, then the child node is fathomed. Successful creation of a k-tuple indicates that the branches leading to the corresponding node represent a candidate common subsequence for the input sequences. It is important to observe that the depth of a node in this search tree equals the length of the current candidate common subsequence being created. Hence, the longest rooted path in the tree corresponds to the LLCS.

Figure 1 illustrates this type of branching for a 4-LCSP instance created from TGAACGTC, GTAACGTT, TGCACTGA and TTGAGCTA. In Figure 1, the letters along the arcs represent the branched letter. Each node has a bolded number that corresponds to the order in which this node is explored, and a 4-tuple that corresponds to the markers in each of the 4 sequences. An E indicates that the end of at least one of the sequences occurred without finding the letter that should have been branched upon. BF and DF refer to the two new techniques presented here that can be used to fathom a node in the branching tree.

Hsu and Du (1984) first proposed this branching strategy for k-LCSP. Hsu and Du's method only stops this branching process if the end of a sequence is reached or if a k-tuple is identical to the k-tuple of another node in the tree. In such a case, an edge is added between these two nodes. For instance in Figure 1, nodes 24 and 20 would both have edges to node 3. Hsu and Du's algorithm finds all maximal common subsequences by finding every path from the root node to a pendant node and then selects the longest such path for the LCS.

In implementing SB, a depth-first search strategy is used to explore the tree. Theoretically, this doesn't improve the run time, but in practice larger instances can be solved with this strategy than with a breadth-first search strategy due to memory constraints. Unlike Hsu and Du's algorithm, which explores child nodes in a fixed alphabetic order, specialized branching explores child nodes according to the sorted order (smallest to largest) of the maximum i_j with ties broken arbitrarily where i_j is the marker of the jth sequence. With this "greedy" node selection rule and the use of depth first search, SB quickly provides a good solution to k-LCSP, which is stored. If at any point in the branching tree, a better solution is obtained, the better solution replaces the existing solution as the current best solution.

While Hsu and Du's algorithm only fathoms if there are no more letters left in at least one of the sequences or if two nodes have identical k-tuples, specialized branching introduces two new fathoming methods. One fathoming method is based upon branch and bound and is called bounded fathoming. The other method, dominance fathoming, is derived from the principles of dynamic programming.

Let L' represent the current best LCS at some point in specialized branching. A bounded fathom occurs at a node p of depth d if $(n - \max\{i_1, \ldots, i_k\}) + d \leq |L'|$. The length of the current common subsequence contained at node p is equal to its depth d, and at most $n - \max\{i_1, \ldots, i_k\}$ letters can be added to this current common subsequence. Hence, if the sum of these two quantities is no more than $|L'|$, then any common subsequence below node p cannot

improve the current best candidate LCS. Observe that bounded fathoming is a direct modification of the bounding in branch and bound. In Figure 1, node 16 can be bounded fathomed because at most two more letters can be added to the common subsequence contained at this node, which does not improve the current best common subsequence obtained at node 5 ($(8 - 6) + 2 \leq 4 = |L'|$) BF(5).

Dominance fathoming is possible when an equally good or better start to a common subsequence has already been found. Given two common sequences L' and L'' , L' is said to dominate L'' if $L' = L''$ and $i'_{jp} \leq i''_{jp}$ where i'_{jp} and i''_{jp} represent the location of the last letter of L' and L'' in the pth sequence for all $p = 1, ..., k$. This principle is easy to verify and is the heart of DP.

Although dominance fathoming is trivial to verify, care must be taken in implementing this fathoming technique because there can be a large number of nodes that could potentially be checked. The time spent checking for dominance fathoming could easily outweigh the benefit gained by fathoming. To simplify the number of comparisons and minimize storage space, SB tests for the possibility of dominance fathoming once at the creation of each node and compares this node's common subsequence to the current best common subsequence. That is, if a created node is at depth d, then the location of the letter in each sequence at the current node is compared to the corresponding sequence location of the dth letter in the current best candidate solution L'. In Figure 1, a dominance fathom occurs at node 24 from the first letter of the current best solution, which is located at node 2 DF(2).

As mentioned Hsu and Du's algorithm maintains a list of pointers to previously explored nodes so they can match identical k-tuples. Observe that dominance fathoming is stronger (fathoms more nodes) than Hsu and Du's method. In addition, to determine whether a pointer can be created, their algorithm compares the current node to all previously created nodes. This may require a substantial amount of work and may negate any time reduction realized by storing fewer nodes. In contrast, specialized branching only requires a comparison to the current best candidate for the LCS. Thus, the memory associated with a node can be freed as soon as all of its descendant leaf nodes have been explored or fathomed, which improves memory management.

All computational tests throughout this paper include 5 instances of each problem size and were run on a 1.5 GHz. Pentium IV PC with 500 Mb of RAM. The run times are reported in seconds with a limit of 5 hours (18,000 seconds). An N/A in any of the tables indicates that the problem could not be solved by the designated method within the 5-hour time limit.

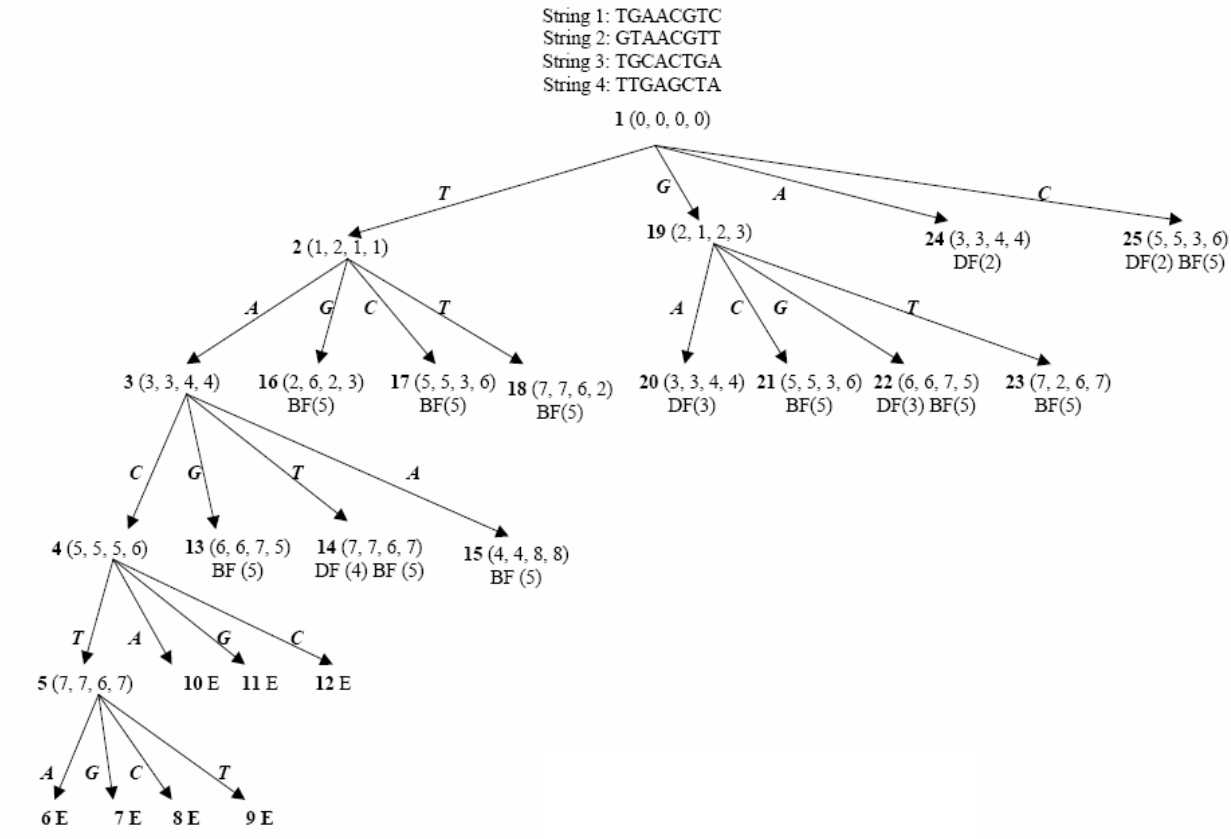

Figure 1. Sample specialized branching tree.

Although the bounded fathoming and dominance fathoming are relatively straightforward methods to prune the branching tree, the computational speed-ups are substantial. Table 2 is included to show the effects of both dominance and bounded fathoming. The 4-LCSP instances generated for Table 2 are randomly created and have $|\sum| = 4$ with a .25 probability of choosing any letter.

The importance of both dominance and bounded fathoming is obvious from the instances with 4 sequences each with 75 letters. With both fathoming techniques, all 5 instances were solved in under an hour. With one fathoming technique, only 40% of the problems were solved in less than 5 hours. With no fathoming techniques, none of these 5 test instances were solved in less than 5 hours. Since Hsu and Du's algorithm is dominated by dominance fathoming, we expect that Hsu and Du's method would have solved at most 40% of the problems in under 5 hours.

The process of storing SB's tree can require $O(k|\sum|^r)$ where r is the LLCS. Thus, SB's run time is $O(k|\sum|^r)$, which is exponential in the LLCS. This time and space complexity can easily be seen since a full $|\sum|$ branching tree of depth r has less than $2|\sum|^r$ nodes for all $|\sum| \geq 2$ and each node stores the current location of the k markers. For large r, the branching tree becomes intractable.

3. COMPUTATIONAL RESULTS FOR DYNAMIC PROGRAMMING AND SPECIALIZED BRANCHING

Table 3 compares SB to the standard dynamic programming approach. As above, the k-LCSP instances are randomly created and have $|\sum| = 4$ with a .25 probability of choosing any letter. The maximum and minimum along with the average values are reported for both the run time and the LLCS. Since the run times of each dynamic program with a given problem size are nearly identical (within 1 second), only the average time is reported.

It can easily be seen that dynamic programming is more effective on the 2 and 4 sequence instances, but for more sequences, SB's performance is vastly superior. The computational results follow the theoretical results as the time requirement for dynamic programming is $O(n^k)$, while SB has a $O(k|\sum|^r)$ run time where r is the LLCS. It is anticipated that implementing a more advanced version (Hakata and Imai (1992), Hirschberg (1977), Irving and Fraser (1992)) of dynamic programming would decrease DP's run times in Table 3. Briefly these advancements reduce either the theoretical run time or storage requirement from $O(n^k)$ to $O(n^{k-1})$. Thus, even these advanced versions of dynamic programming will still be computationally inferior to SB for any $k \geq 7$.

From the results in Table 3, we conclude that the length of the sequences and not the number of sequences limits SB, while the opposite is true for DP. In viewing Table 3, the reader may be surprised to see that the run times for the problems with 10 strings of length 75 require approximately 7 times longer than the problems with 50 strings of length 75. The reason is straightforward and again follows the theoretical run time of SB. As expected, the 50 string problems have substantially smaller LCSs than do the 10 string problems. Therefore, the depths of SB's search trees are smaller in the 50 string sequences than the 10 string instances, which accounts for the faster run time.

Since Table 3's problems were randomly generated, the LLCS's are small. The instances in Table 4 are included to show the performance of SB if the LLCS is close to n. Here instances were created that guaranteed a LLCS of at least 90% or 95% of n. In Table 4, a surprising result is that SB could quickly solve large k-LCSP instances (100 sequences with 500 letters each) when the LLCS is over 90% of the length of the sequences. When this time is compared to the theoretical worst case run times of SB ($O(4^{500})$) or DP ($O(500^{100})$), SB's speed is astounding. SB can solve these problems quickly because a near optimal solution is quickly obtained and the two fathoming techniques almost immediately fathom all other nodes.

Up until now all of the computational results have had $|\sum| = 4$, which is neither a small nor a large alphabet. Table 5 provides the performance of SB on instances for a variety of alphabet sizes. These instances are again randomly generated with each letter equally likely to be at any location in any string. This table helps provide the reader with the size of instances that SB can quickly solve and which instances require too much time.

Table 2. Specialized branching with different types of fathoming

No. of Strings	String Length	Type of Fathoming	Number of Instances Solved	Avg. Time of Solved Instances (Sec)	Max − Min Time
4	50	Neither of the fathoming techniques	5 out of 5	131.4	23 − 398
		Only dominance fathoming	5 out of 5	14.6	2 − 41
		Only bounded fathoming	5 out of 5	8.1	1 − 17
		Both of the fathoming techniques	5 out of 5	1.0	0 − 3
4	75	Neither of the fathoming techniques	0 out of 5	N/A	N/A
		Only dominance fathoming	2 out of 5	5472	3829 − 4905
		Only bounded fathoming	2 out of 5	4367	1253 − 9691
		Both of the fathoming techniques	5 out of 5	931	76 −1862

Table 3. Dynamic programming (DP) vs. specialized branching (SB)

No. of Strings	String Length	Avg. LLCS	Min – Max LLCS	Avg. Time (DP)	Avg. Time (SB)	Min – Max Time (SB)
2	25	14	12 – 15	0	0	0 – 0
	50	29	27 – 31	0	0	0 – 0
	75	45	44 – 47	0	44	2 – 108
	100	62	60 – 63	0	407	25 – 906
	1000	648	643 – 656	0	N/A	N/A
	10000	6526	6515 – 6541	17	N/A	N/A
4	25	9	7 – 10	0	0	0 – 0
	50	21	20 – 23	2	1	0 – 0
	75	33	30 – 35	9	931	76 – 1862
	100	45	44 – 46	45	N/A	N/A
	1000	N/A	N/A	N/A	N/A	N/A
6	25	8	7 – 8	N/A	0	0 – 0
	50	17	16 – 18	N/A	3	1 – 6
	75	29	27 – 29	N/A	2936	1141 – 5793
	100	N/A	N/A	N/A	N/A	N/A
10	25	7	6 – 7	N/A	0	0 – 0
	50	15	14 – 16	N/A	2	1 – 4
	75	24	23 – 26	N/A	3731	1610 – 6982
	100	N/A	N/A	N/A	N/A	N/A
50	25	3	3 – 4	N/A	0	0 – 0
	50	10	10 – 11	N/A	1	0 – 1
	75	17	17 – 18	N/A	505	259 – 939
	100	N/A	N/A	N/A	N/A	N/A
100	25	3	2 – 3	N/A	0	0 – 0
	50	9	8 – 9	N/A	0	0 – 1
	75	15	15 – 16	N/A	320	206 – 460
	100	N/A	N/A	N/A	N/A	N/A
1000	25	1	1 – 1	N/A	0	0 – 0
	50	6	5 – 6	N/A	1	1 – 1
	75	11	11 – 11	N/A	372	310 – 547
	100	N/A	N/A	N/A	N/A	N/A

Table 4. SB on *k*-LCSP instances with similar strings

No. of Strings	String Length	Minimum % of LCS	Avg. LLCS	Avg. Running Time (sec)
10	500	> 90%	459	80
		> 95%	479	2
	1000	> 90%	N/A	N/A
		> 95%	959	62
50	500	> 90%	459	325
		> 95%	479	15
	1000	> 90%	N/A	N/A
		> 95%	959	263
100	500	> 90%	459	1633
		> 95%	479	37
	1000	> 90%	N/A	N/A
		> 95%	959	1104

4. CONCLUSIONS AND FUTURE RESEARCH

This paper introduced both dominance and bounded fathoming and incorporated them into a specialized branching technique (SB) to optimally solve *k*-LCSP problems. Various computational results demonstrated that SB is the best known technique to optimally solve *k*-LCSP for $k \geq 7$.

The substantial computational benefit created by using both dominance and bounded fathoming creates the following important research questions. Is there a way to apply dominance fathoming in a generic branch and bound

routine? That is, by knowing the best solution thus far, can one fathom a node since it will never lead to a better solution because the current best solution has a better "start" than the current node. Such a technique could greatly improve the speed of commercial integer programming codes.

Table 5. SB on *k*-LCSP instances with various size alphabets

Size of Alphabet	No. of Strings	Length of each String	Avg. LLCS	Avg. Running Time Sec.
2	5	25	13.8	0.05
		50	31.2	0.4
		75	46	94.8
		100	N/A	N/A
	50	25	9.2	0.01
		50	21.6	5.6
		75	35.8	1836.4
		100	N/A	N/A
10	5	50	9.8	1
		75	16.4	14.6
		100	22.8	2187
		125	N/A	N/A
	50	100	8.4	6.2
		125	11.6	798.2
		150	16.2	3112
		175	N/A	N/A
25	5	100	10.4	1.6
		125	14.2	28.4
		150	17.8	3486
		175	N/A	N/A
	50	150	4.4	1
		200	7.2	14
		250	10.2	3344
		300	N/A	N/A

ACKNOWLEDGEMENTS

This research was partially supported by the Kansas Technology Enterprise Corporation.

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J. (Eds.) (1983). *Data Structures and Algorithms*, Addison Wiley, MA.
2. Bafna, V., Muthukrishnan, S., and Ravi, R. (1995). Computing similarity between RNA strings. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, Espoo, Finland, pp. 1-16.
3. Banerjee, A. and Ghosh, J. (2001). Clickstream clustering using weighted longest common subsequences. *Proceedings of the Web Mining Workshop at the 1st SIAM Conference on Data Mining*, Chicago, pp. 33-40.
4. Brisk, P., Kaplan, A., and Sarrafzadeh, M. (2004). Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. *Proceedings of the 2004 Design Automation Conference*, San Diego, CA, USA, pp. 395-400.
5. Dayhoff, M.O. (1969). Computer analysis of protein evolution. *Scientific American*, 221(1): 86-95.
6. Dayhoff, M., Schwartz, R., and Orcutt, B. (1978). A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5: 345-352.
7. Gallant, J., Maier, D. and Storer, J.A. (1980). On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1): 50-58.
8. Hakata, K. and Imai, H. (1992). The longest common subsequence problem for small alphabet size between many strings. *Proceedings of the 3rd International Symposium on Algorithms and Computatio*n, Nagoya, Japan, 650: 469-478.
9. Hayes, C.C. (1989). A model of planning for plan efficiency: Taking advantage of operator overlap. *Proceedings of the 11th International Joint Conference of Artificial Intelligence*, Detroit, Michigan, pp. 949-953.
10. Hirschberg, D.S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the Association for Computing Machinery*, 18(6): 341-343.
11. Hirschberg, D.S. (1977). Algorithms for the longest common subsequence problem. *Journal of the Association for Computing Machinery*, 24(4): 664-675.
12. Hsu, W.J. and Du, M.W. (1984). Computing a longest common subsequence for a set of strings. *BIT*, 24: 45-59.
13. Hunt, J.W. and McIlroy, M.D. (1975). An algorithm for differential file comparison. Computing Science Technical Report, 41, AT&T Bell Laboratories, Murray Hill, New Jersey.
14. Hunt, J.W. and Szymanski, T.G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the Association for Computing Machinery*, 20(5): 350-353.
15. Irving, R.W. and Fraser, C.B. (1992). Two algorithms for the longest common subsequence of three (or more) strings. *Lecture Notes In Computer Science*, 644: 214-229.
16. Itoga, S.Y. (1981). The string merging problem. *BIT*, 21: 20-30.
17. Jiang, T. and Li, M. (1995). On the approximation of shortest common and longest common subsequences. *SIAM Journal on Computing*, 24(5): 1122-1139.
18. Jiang, T., Lin, G., Ma, B., and Zhang, K. (2002). A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2): 371-88.
19. Land, A.H. and Doig, A.G. (1960). An automatic method for solving discrete programming problems. *Econometrica*, 28: 497-520.
20. Larson, R. (1968). *State Increment Dynamic Programming*, American Elsevier Publishing Company, Inc., New York, NY.
21. Lu, S.Y. and Fu, K.S. (1978). A sentence-to-sentence clustering procedure for pattern analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 8(5): 381-389.
22. Maier, D. (1978). The complexity of some problems

on subsequences and supersequences. *Journal of the Association for Computing Machinery*, 25: 322-336.
23. Sankoff, D. and Kruskal, J.B. (Eds.) (1983). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA.
24. Sellis, T. (1988). Multiple query optimization. *ACM Transactions on Database Systems*, 13(1): 23-52.
25. Singireddy, A. (2003). *Solving the Longest Common Subsequence Problem for DNA Applications*, Master's Thesis, Industrial and Manufacturing Systems Engineering, Kansas State University, Manhattan, KS.
26. Smith, T.F. and Waterman, M.S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147: 195-197.
27. Storer, J. (1988). *Data Compression: Methods and Theory*, Computer Science Press, MD.
28. Thompson, J.D., Higgins, D.G., and Gibson, T.J. (1994). 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(22): 4673-4680.
29. Wagner, R.A. (1973). Common phrases and minimum-space text storage. *Communications of the Association for Computing Machinery*, 16(3): 148-152.
30. Wagner, R.A. and Fischer, M.J. (1974). The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21: 168-173.