Searching for Shortest Common Supersequences by Means of a Heuristic-Based Genetic Algorithm

Jürgen Branke, Martin Middendorf
University of Karlsruhe
Institut für Angewandte Informatik
und Formale Beschreibungsverfahren
D-76128 Karlsruhe, Germany
Email: {branke, middendorf}@aifb.uni-karlsruhe.de

July 1996

Published in Proceedings of the 2nd Nordic Workshop on Genetic Algorithms and Their Applications, Vaasa, Finland, Finish Artificial Intelligence Society, 1996, pp. 105-114.

Abstract

In this paper we describe a genetic algorithm (GA) for the Shortest Common Supersequence (SCS) problem which is a classical problem from stringology. The SCS problem has applications in artificial intelligence (specifically planning), mechanical engineering and data compression. It is NP-complete even under severe restrictions concerning the alphabet size, the length of the given strings, or their structure. Using a Genetic Algorithm to solve SCS is not easy, e.g. because the search space contains only a few valid solutions of reasonable length and a natural representation would lead to varying string lengths. To circumvent these difficulties, we base the Genetic Algorithm on a slightly modified Majority Merge heuristic. The resulting GA/heuristic hybrid yields significantly better results than Majority Merge alone and other well-known heuristics, though its running time is much higher.

Keywords: evolutionary algorithm, genetic algorithm, shortest common supersequence, heuristic

1 Introduction

The Shortest Common Supersequence (SCS) problem is a classical problem from stringology which has applications e.g. in artificial intelligence (specifically planning), mechanical engineering and data compression [2, 11]. The problem is as follows: Given a finite set L of strings over an alphabet Σ, find a string of minimal length that is a supersequence of each string in L. A string S is called a *supersequence* of a string T if S can be obtained from T by inserting zero or more symbols.

For example, given the alphabet $\Sigma = \{a, b, c\}$ and the set of strings $L = \{cbbc, abc, cba\}$, a shortest common supersequence of L is the string $cbabc$. Another, longer supersequence would be for example $cabbac$.

The SCS problem is NP-complete even under various restrictions concerning the alphabet size, the length of the given strings, or their structure [7, 8, 9]. To find optimal solutions, dynamic programming algorithms as well as Branch-and-Bound algorithms have been investigated by Fraser [4]. However, the dynamic programming algorithms are successful only for a very small number of given strings, because, otherwise, their space requirements are too large. And Branch-and-Bound algorithms need too much time to be practical, except for very small alphabets. Several deterministic heuristics have been proposed to find approximate solutions for the problem [2, 3, 4, 6].

In this paper, we describe how to tackle the Shortest Common Supersequence problem by means of a heuristic-based genetic algorithm.

In the next section we mention the special difficulties when applying a GA to the SCS problem, how we work around it, and how the actual algorithm looks like. Section 3 contains empirical results and comparisons to other heuristics. Conclusions and future work are given in Section 4.

2 The Genetic Algorithm

The Shortest Common Supersequence problem has some properties that make it hard to apply a GA:

- a natural representation of candidate solutions, the direct representation as a string, would lead to genotypes of varying length,
- changing a good solution just slightly will very likely yield an invalid string, i.e. a string which is not a supersequence,
- most of all possible strings of reasonable length are invalid,
- there exists a trivial solution of size at most $|\Sigma|$ times the optimal solution length (if l is the maximal length of a string in L and S is a concatenation of the characters in Σ then S^l is a solution). Thus the GA has to get very close to the optimum to justify the effort.

To still be able to apply a GA successfully, we decided to base the GA on one of the best known heuristics, Majority Merge. The result is a powerful GA/Majority Merge hybrid that we henceforth call GA/M for short.

The basic idea is to change the heuristic such that it can be influenced by many parameters, and then to use the genetic algorithm to tune the parameters. Compared to encoding the problem directly, we thereby gain an easier fitness landscape for the GA. This idea has already been applied successfully in [1, 5].

The Majority Merge heuristic builds a supersequence starting from the empty string as follows: It looks at the first symbol of every string in L (i.e. the front), appends the most frequent symbol, say a, to the supersequence and then removes a from the front of the strings. This process is repeated until all strings in L are exhausted.

For random strings Majority Merge is the best known heuristic when the number of strings is large compared to the alphabet size. If the number of strings is small and the alphabet is large then heuristics perform better that iteratively take two strings out of

the set L and replace them by their optimal shortest common supersequence until a single string is left [4].

Clearly, choosing always the most frequent symbol from the front of the strings as the next symbol in the supersequence is unlikely to be a globally optimal strategy. We therefore extend Majority Merge

i) by assigning a weight to each symbol of every string, and

ii) by having the heuristic choose that symbol of the front of the strings whose sum of weights is maximal.

We thus obtain a mapping from the weight assignments to the possible supersequences (only in the rare case of ties several supersequences may be obtained from one weight assignment). The mapping has the following properties:

- each weight assignment encodes a supersequence of L, i.e. a valid solution, and
- there exists a weight assignment that produces a shortest common supersequence.

The task of the genetic algorithm then remains to assign appropriate weights to the symbols of the strings in L. This problem can be encoded by storing a real-valued weight for each symbol of each string on the chromosome. Thus the length of a chromosome is given by the sum of the lengths of all strings in L. To keep the weights of the different chromosomes in a comparable range of values, we normalize the weights such that the sum of weights equals 1. The quality of each chromosome is just the length of the supersequence that is produced by the adapted Majority Merge heuristic with the given set of weights.

As it turned out, the ordering of the weights on the chromosome also plays an important role, presumably for the following reason: to decide which symbol is removed next, the heuristic only looks at the weights of the symbols at the front, and one might regard this group of weights as building blocks. Since, in general, the indices (i.e. their position in the string) of the symbols that appear at the front will not differ too much, it is favorable to encode symbols with the same index from different strings closely together on the chromosome.

The GA/M was based on the PGA program [10] and uses an island model approach with 10 subpopulations of size 150 each. For most parameters we used the default values of PGA: mutation probability is 0.02 for each gene [1], crossover probability is 1.0, migration between the populations is done every 10 generations such that one of the populations selects one individual by ranking selection and sends it to all other populations. The algorithm is of steady state type, i.e. in each generation, it selects two parents, according to ranking selection, performs two-point crossover to obtain one child, perhaps mutates it and installs the result back into the population. The worst individual of the population is deleted. The run stops when all individuals have the same fitness (or after 1,000,000 evaluations, but even for the most difficult test problems GA/M needed only about 250,000 evaluations on the average to converge).

In addition to the just described version, we also implemented some variants that are more strongly biased towards the underlying heuristic by adding a basic value to each

[1] If a gene is selected for mutation, its value is changed randomly by at most 5 percent of the domain range.

weight before evaluation. In the extreme case, when the basic value is high enough, the GA/M can merely break ties, i.e. decide only when several symbols occur equally often. Depending on the basic value we tested the following variants of our genetic algorithm:

- G_0: without a basic value
- $G_{1/||L||}$: with basic value $\frac{1}{||L||}$ where $||L||$ is the sum of the lengths of the strings in L
- G_1: with basic value 1.

As it turned out, there is no basic value which performs best for all test instances. This led us to include the basic value in the genotype and have it optimized by the GA/M. This version of our genetic Algorithm is called G_v. G_v allows the basic value to vary between 0 and $\frac{1}{||L||}$.

3 Results

The GA/M was tested on a number of problem instances with different numbers of strings, different alphabet sizes and different string lengths. Some of the problem instances were generated randomly, some of them with similar strings, and some were special sets of strings for which the optimal solution is obvious, but Majority Merge performs badly.

All results are averaged over 4 runs with different random seeds. The results are compared to the average of 150 runs of Majority Merge with ties broken randomly.

We also ran Majority Merge as often as there were evaluations in a GA/M, with ties broken randomly and keeping track of the best solution found so far. The improvements that could be obtained by this brute force approach were always much smaller than by any of the GA/M-variants. This confirmed that our good results were actually due to the GA/M-design and not just a result of massive computational power.

3.1 Results on Random Strings

Here, the strings to be merged were generated randomly and independently of each other. We tested instances with string-lengths 40, 80, and 160, alphabet sizes of 2, 4, and 16 and numbers of strings in L of 4, 8, and 16. For each combination of parameters, we constructed 5 random instances of sets of strings.

Figure 1 shows the relative improvements of G_0, $G_{1/||L||}$, G_1, and G_v on Majority Merge.

In general, the relative improvement for a fixed string length increases with a shrinking number of strings and a growing size of the alphabet. E.g. with $l = 40$, the least relative improvement for G_0 ($G_{1/||L||}$, G_1, G_v) was obtained for $|\Sigma| = 2$ and $k = 16$ with 4.12% (respectively 4.36%, 2.58%, 4.28%), whereas for $|\Sigma| = 16$ and $k = 4$ the relative improvement was 23.43% (respectively 23.26%, 22.42%, 23.54%).

The very high improvement for a small number of strings and large alphabet shows that the GA/M was able to adapt the heuristic so that the results were good even in cases where the underlying heuristic is relatively weak. The most interesting cases are the improvements for large numbers of strings ($k \geq 8$) and small alphabet ($a \leq 4$). In these cases Majority Merge was at least as good as all other heuristics investigated by Fraser [4] (for string length 100), but still the GA/M was able to improve on the results significantly.

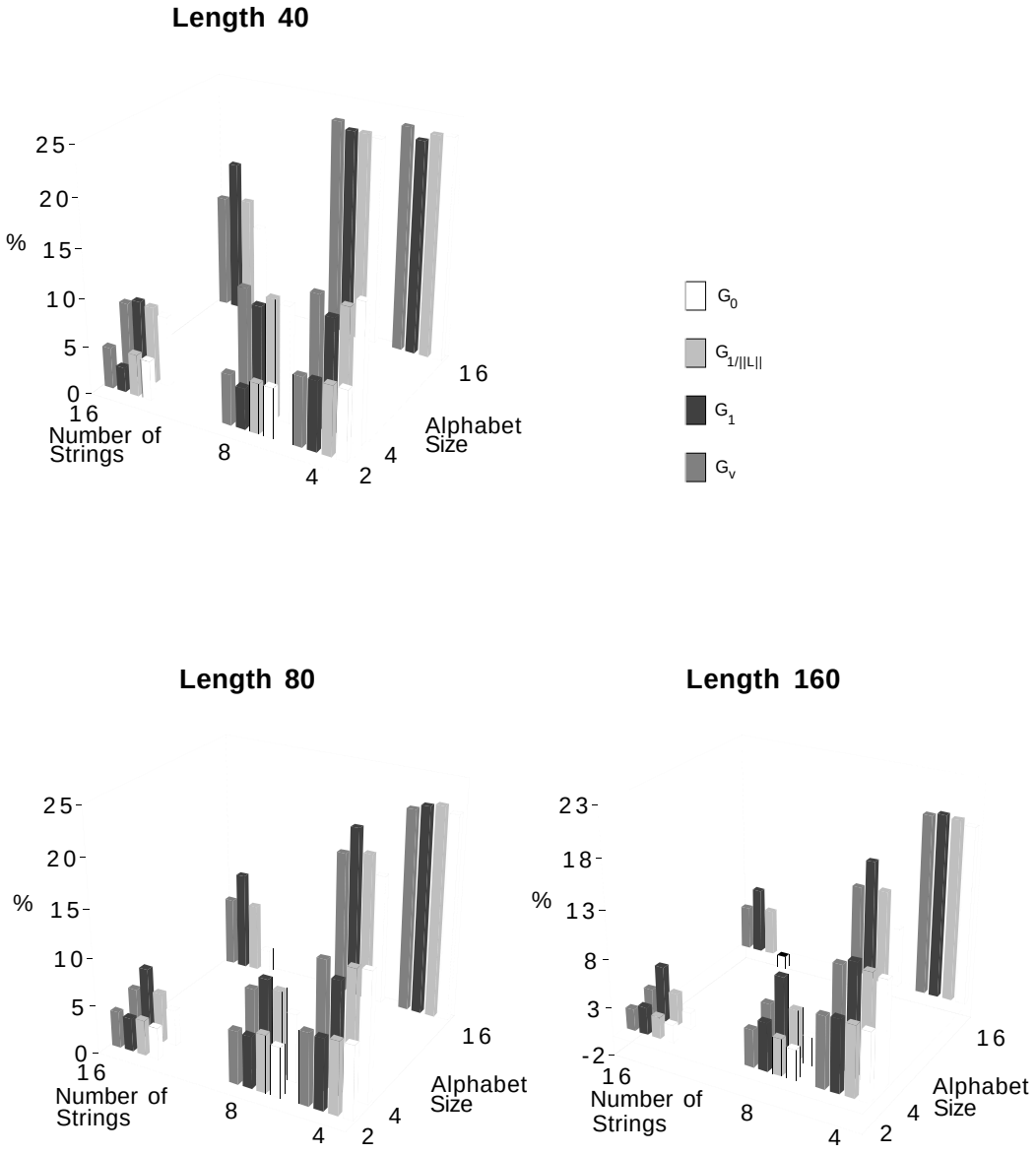

Figure 1: Relative improvement of the 4 GA/M-variants compared to simple Majority Merge on random strings of length 40 (upper left), length 80 (lower left) and length 160 (lower right). Each bar represents the average over 20 runs (5 random instances, 4 random seeds).

$\|\Sigma\| = 4$	Optimum	G_0	$G_{1/\|\|L\|\|}$	G_1	G_v	Maj. Merge
$n = 90, k = 8$	≤ 100	100.0	100.0	105.4	100.0	115.1
$n = 90, k = 16$	≤ 100	100.0	100.0	103.8	100.0	106.8
$n = 80, k = 8$	≤ 100	100.0	100.0	122.7	100.0	168.1
$n = 80, k = 16$	≤ 100	100.0	100.0	155.3	100.0	179.5

Table 1: Best string length found: Results on similar strings, averaged over 20 runs (5 random instances, 4 random seeds)

For a fixed number of strings and fixed alphabet, the relative improvement of the GA/M shrinks with a growing length of strings. This is probably because the chromosomes become very long when the size of the problem instance grows (e.g. for 16 words of length 160 the chromosome consists of 2560 weight values) which makes it harder for the GA to find good solutions in a limited time.

G_1 with basic value 1 is the best version of the GA/M for long strings or a large number of strings. G_0 is better than G_1 for $l = 80$, $k = 4$, $|\Sigma| \leq 4$ as well as $l = 40$, $k = 4$, or $k = 8$, $|\Sigma| \leq 4$ or $k = 16$, $|\Sigma| = 2$. In nearly all cases $G_{1/||L||}$ shows a performance between that of G_0 and G_1. With some exceptions, G_v performs very similar to $G_{1/||L||}$ on these random problems.

3.2 Results on Similar Strings

In the real world, the strings to be merged are usually interdependent and quite similar, like e.g. the sequences of operations necessary to construct several variants of the same product. Therefore we also tested the GA/M with sets of strings that were obtained as randomly chosen subsequences from a random string of length 100 over an alphabet of size 4. Since these strings all originate from the same supersequence, they are relatively similar. Also, it is very likely that the original supersequence is an optimal supersequence. Tests were conducted with sets of strings containing 8 or 16 supersequences of length 80 or 90.

The results obtained by G_1 and the Majority Merge heuristic are shown in Table 1. In all cases G_0, $G_{1/||L||}$, and G_v obtained a supersequence of length 100 which might be optimal. G_1 performed better than Majority Merge but both are much worse than G_0, $G_{1/||L||}$, and G_v.

Fraser [4] tested several deterministic heuristics on 8 and 16 randomly chosen subsequences of a random string of length 100 over an alphabet of size 4. None of the heuristics found a supersequence of length 100 (averaged over 4 test instances), although the heuristics that rely on computing optimal supersequences for 2 strings came close to 100.

3.3 Results on Special Sets of Strings

We also tested our GA/M on several instances on which Majority Merge behaves badly and for which the shortest common supersequence is known. Our test sets were the following:

- L_1: 9 strings a^{40}, 4 strings ba^{39}, 2 strings bba^{38}, and 1 string $bbba^{37}$.
  The optimal solution of length 43 is to first remove the b's, and then the a's of

	Optimum	G_0	$G_{1/\|\|L\|\|}$	G_1	G_v	Maj. Merge
L_1	43	43.0	43.0	157.0	43.0	157.0
L_2	79	79.5	94.0	121.0	80.0	121.0
L_3	60	60.0	60.0	60.0	60.0	73.2

Table 2: Best string length found: Results on special strings

all strings together. However, Majority Merge would inevitably first remove the 9 complete a-strings, then one b, then the newly obtained a-strings, one b, etc.

- L_2: 9 strings a^{40}, 4 strings $b^{13}a^{27}$, 2 strings $b^{26}a^{14}$, and 1 string $b^{39}a$.
  Similar to L_1, it is optimal to first remove all b's and then all a's. Again, Majority Merge can not find the optimum. For the GA/M, the number of necessary "decisions" against the majority-rule to reach the optimum is higher than for L_1 because the sequence of b's is longer.

- L_3: 8 strings $a^{20}b^{20}$, 8 strings $b^{20}c^{20}$.
  Here it is optimal to first remove all a's, then the b's, and finally the c's. Since there are as many strings with prefix a^{20} than there are strings with prefix b^{20} Majority Merge can find the optimal solution only if it breaks all ties correctly which is quite unlikely.

Table 2 shows the results for GA/M on L_1, L_2, and L_3. G_0 and G_v outperform $G_{1/||L||}$ and G_1. The optimum was found by G_0, $G_{1/||L||}$, and G_v for L_1 and L_3. G_0 and G_v also came quite close to the optimum for L_2. Since G_1 can only break ties it could find the optimum for L_3 only, which it did, but otherwise was as bad as Majority Merge.

4 Conclusion and Future Work

Compared to the Majority Merge heuristic alone, the GA/M improved the results significantly. On random strings, it was able to overcome the weaknesses of Majority Merge where it is not as good as other known heuristics, and also improved on the results when Majority Merge is best compared to other heuristics.

On more regular strings, which are closer to the real world, GA/M found the assumed optimum in all runs and thus performed much better than any other common heuristic.

The results on special instances showed that Majority Merge can be completely misled and that it may therefore be dangerous to bias GA/M too much towards Majority Merge, i.e. to choose a high basic value. GA/M variants with small or variable basic values always found the optimal solution or came very close to it.

Overall, given the nature of the problem, the improvements obtained by GA/M compared to Majority Merge are remarkable. Naturally, a simple heuristic runs much faster than it takes GA/M to converge, so there is a tradeoff between accuracy of the solution and computation time.

Future research in this area will include

- testing some other GA/M variants that might better determinine an optimal basic value; e.g. initializing all weights to the same value, or initializing different subpopulations with different basic values.
- applying similar approaches to related problem domains
- adapting other deterministic heuristics so that the GA can use them as an underlying strategies.

Acknowledgement

We thank Marcus Engels-Lindemann for interesting discussions and for implementing most of the code.

References

[1] J. Branke, U. Kohlmorgen, H. Schmeck, and H. Veith. Steuerung einer Heuristik zur Losgrössenplanung unter Kapazitätsrestriktionen mit Hilfe eine parallelen genetischen Algorithmus. In J. Kuhl and V. Nissen, editors, *Tagungsband zum Workshop Evolutionäre Algorithmen in Management Anwendungen*, pages 21–31. University of Göttingen, Institut für Wirtschaftsinformatik, 1995.

[2] D.E. Foulser, Q. Yang, and M. Li. Theory and algorithms for plan merging. *Artificial Intelligence*, 57:143–181, 1992.

[3] C. B. Fraser and R. Irving. Approximation algorithms for the shortest common supersequence. *Nordic Journal on Computing*, 2:303–325, 1995.

[4] C.B. Fraser. *Subsequences and supersequences*. PhD thesis, University of Glasgow, Departement of Computer Science, 1995.

[5] K. Haase and U. Kohlmorgen. Parallel genetic algorithm for the capacitated lot-sizing problem. In Kleinschmidt et al., editor, *Operations Research Proceedings*, pages 370–375. Springer-Verlag, 1996.

[6] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM J. Comput.*, 24:1122–1139, 1995.

[7] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, 1978.

[8] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoret. Comput. Sci.*, 125:205–228, 1994.

[9] K.-J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoret. Comput. Sci.*, 16:187–198, 1981.

[10] P. Ross. *About PGA v2.7*. University of Edinburgh, 1994.

[11] V. G. Timkovsky. Complexity of common subsequence and supersequence problems and related problems. *Cybernetics*, 25:565–580, 1990.