Beam-ACO for the Longest Common Subsequence Problem

Christian Blum

Abstract—The longest common subsequence problem is classical string problem. It has applications, for example, in pattern recognition and bioinformatics. In this work we present a so-called Beam-ACO approach for solving this problem. Beam-ACO algorithms are hybrid techniques that results from a combination of ant colony optimization and beam search, which is an incomplete branch and bound derivative. Our results show that Beam-ACO is able to find new best solutions for 31 out of 60 benchmark instances that we chose for the experimental evaluation of the algorithm.

I. INTRODUCTION

The longest common subsequence (LCS) problem is one of the classical string problems. Given a problem instance (S, Σ), where S = \{s_1, s_2, ..., s_n\} is a set of n strings over a finite alphabet Σ, the problem consists in finding a longest string t^* that is a subsequence of all the strings in S. Such a string t^* is called a longest common subsequence of the strings in S. Note that a string t is called a subsequence of a string s, if t can be produced from s by deleting characters. For example, dga is a subsequence of adagttta. Traditional computer science applications of this problem are in data compression [19], syntactic pattern recognition [13], file comparison [1], text edition [15] and query optimization in databases [16]. More recent applications include computational biology [18], [11] and the production of circuits in field programmable gate arrays [6]. The LCS problem was shown to be NP-hard [14] for an arbitrary number n of input strings.

Due to its classical nature, the LCS problem has attracted quite a lot of research efforts over the past decades. Much work has been dedicated to the development of efficient dynamic programming procedures (see, for example, [2]). The body of work on approximate methods is dominated by constructive one-pass heuristics [9], [10]. Moreover, meta-heuristics have been proposed in [17], [8], [12]. In [3] we recently published the current state-of-the-art algorithm for the LCS problem. The results of this algorithm, which is based on beam search (BS), have shown that none of the earlier algorithms came even close to derive good solutions for difficult problem instances. In other words, BS has shown to be largely superior to any other existing technique for what concerns the LCS problem. In this work we try to improve over the state-of-the-art results of BS by adding a learning component to BS. The resulting algorithm is a hybrid between the metaheuristic ant colony optimization (ACO) [7] and beam search. The interested reader might note that we already published a preliminary version of this Beam-ACO approach in [5]. However, the purpose of [5] was rather didactical in the sense that we aimed at showing the utility of hybridizing ant colony optimization with beam search. The LCS problem exclusively served as an example. In contrast, the algorithm that we present in this work, which is a further development of the one proposed in the above mentioned paper, aims at high performance. As we will show in the section on experimental results our algorithm is able to find new best solutions for 31 of the 60 problem instances that we chose for testing.

The organization of this paper is as follows. In Section II we present the proposed Beam-ACO approach. In Section III we outline the experimental evaluation. Finally, in Section IV we offer conclusions and an outlook to future work.

II. BEAM-ACO

The specific ACO algorithm that we used for adding a learning component to beam search is a standard MAX-MIN Ant System implemented in the hyper-cube framework (HCF), see [4]. The pseudo-code of the algorithm is given in Algorithm 1. As usual, the data structures of the algorithm are: (1) the best-so-far solution T^bs, i.e., the best solution generated since the start of the algorithm; (2) the restart-best solution T^rb, that is, the best solution generated since the last restart of the algorithm; (3) the convergence factor cf, 0 ≤ cf ≤ 1, which is a measure of how far the algorithm is from convergence; and (4) the Boolean variable bs_update, which becomes true when the algorithm reaches convergence.

One of the crucial points of any ACO algorithm is the pheromone model T. In the case of the LCS problem, the definition of the pheromone model is not a trivial task as, for example, in the case of the TSP. After some experimentation we decided to use a pheromone model T that contains for each position j of a string s_i ∈ S a pheromone value 0 ≤ τ_ij ≤ 1, that is, T = \{τ_ij | i = 1, ..., n, j = 1, ..., |s_i|\}. Note that value τ_ij ∈ T indicates the goodness of adding the letter at position j of string s_i to the solution under construction: the greater τ_ij, the greater is the goodness of adding the corresponding letter. This pheromone model implies a specific way of representing solutions, henceforth called ACO-solutions. Note that any common subsequence t of the strings in S—that is, any solution t—can be translated in a well-defined way into a unique ACO-solution T = \{T_ij ∈ [0, 1] | i = 1, ..., n, j = 1, ..., |s_i|\}. This is done as follows: for each string s_i ∈ S we search the positions of the letters of t in s_i such that each letter is in its left-most position possible. Then, these positions j are set to 1 in T, that is, T_ij = 1, and T_ij = 0 otherwise. For example, consider an instance S = \{acbeadbb, cabdaedc, babeddab\} and a
Algorithm 1 Beam-ACO for the LCS problem

1: input: $k_{bs}, \mu \in \mathbb{Z}^+$
2: $T^{bs} := \text{NULL, } T^{rb} := \text{NULL, } cf := 0, bs\_update := \text{FALSE}$
3: $T_{ji} := 0.5, i = 1, \ldots , n, j = 1, \ldots , |s_i|$
4: while CPU time limit not reached do
5: $T^{obs} := \text{ProbabilisticBeamSearch}(k_{bs}, \mu)$ \{see Alg. 2\}
6: if $|T^{obs}| > |T|^{rb}$ then $T^{rb} := T^{obs}$
7: if $|T^{obs}| > |T^{rb}|$ then $T^{obs} := T^{rb}$
8: ApplyPheromoneUpdate($cf, bs\_update, T, T^{obs}, T^{rb}, T^{bs}$)
9: $cf := \text{ComputeConvergenceFactor}(T)$
10: if $cf > 0.99$ then
11: if $bs\_update = \text{TRUE}$ then
12: $T_{ji} := 0.5, i = 1, \ldots , n, j = 1, \ldots , |s_i|$
13: $T^{rb} := \text{NULL}$
14: $bs\_update := \text{FALSE}$
15: else
16: $bs\_update := \text{TRUE}$
17: end if
18: end if
19: end while
20: output: $t^{bs}$ (that is, the string version of $T^{bs}$)

possible solution $t = abed$. This solution translates into the ACO-solution $T = \{101101001, 011001101, 011111000\}$.

The algorithm works as follows. After the initialization of the pheromone values to 0.5, each iteration consists of the following steps. First, beam search (BS) is applied in a probabilistic way, based on pheromone values. The working of BS is outlined further below in Section II-A. This generates a solution $T^{obs}$. Second, the pheromone update is conducted by procedure ApplyPheromoneUpdate($cf, bs\_update, T, T^{obs}, T^{rb}, T^{bs}$). Third, a new value for the convergence factor $cf$ is computed. Depending on this value, as well as on the value of the Boolean variable $bs\_update$, a decision on whether to restart the algorithm or not is made. If the algorithm is restarted, all the pheromone values are reset to their initial value (that is, 0.5). The algorithm is iterated until a maximum computation time limit is reached. Once terminated, the algorithm returns the string version $t^{bs}$ of the best-so-far ACO-solution $T^{bs}$, the best solution found. In the following we describe the two remaining procedures of Algorithm 1 in more detail.

ApplyPheromoneUpdate($cf, bs\_update, T, T^{obs}, T^{rb}, T^{bs}$): As usual in M-MAS algorithms implemented in the HCF, three solutions are used for updating the pheromone values. These are the solution $T^{obs}$ generated by beam search, the restart-best solution $T^{rb}$, and the best-so-far solution $T^{bs}$. The influence of each solution on the pheromone update depends on the state of convergence of the algorithm as measured by the convergence factor $cf$. Each pheromone value $\tau_{ij} \in T$ is updated as follows:

$$\tau_{ij} := \tau_{ij} + \rho \cdot (\xi_{ij} - \tau_{ij}),$$

where $\xi_{ij} := \kappa_{pbs} \cdot T^{pbs}_{ij} + \kappa_{rb} \cdot T^{rb}_{ij} + \kappa_{bs} \cdot T^{bs}_{ij},$ (2)

where $\kappa_{pbs}$ is the weight (that is, the influence) of solution $T^{obs}$, $\kappa_{rb}$ is the weight of solution $T^{rb}$, $\kappa_{bs}$ is the weight of solution $T^{bs}$, and $\kappa_{pbs} + \kappa_{rb} + \kappa_{bs} = 1$. After the pheromone update rule (Equation 1) is applied, pheromone values that exceed $\tau_{\text{max}} = 0.999$ are set back to $\tau_{\text{max}}$ (similarly for $\tau_{\text{min}} = 0.001$). This is done in order to avoid a complete convergence of the algorithm, which is a situation that should be avoided. Equation 2 allows to choose how to weight the relative influence of the three solutions used for updating the pheromone values. For our application we used a standard update schedule as shown in Table I.

ComputeConvergenceFactor($T$): The convergence factor $cf$, which is a function of the current pheromone values, is computed as follows:

$$cf := 2 \left( \frac{\sum_{\tau_{ij} \in T} \max\{\tau_{\text{max}} - \tau_{ij}, \tau_{ij} - \tau_{\text{min}}\}}{|T| \cdot (\tau_{\text{max}} - \tau_{\text{min}})} - 0.5 \right)$$

Note that when the algorithm is initialized (or reset) it holds that $cf = 0$. On the other side, when the algorithm has converged, then $cf = 1$. In all other cases, $cf$ has a value in $(0, 1)$. This completes the description of the learning component of our Beam-ACO approach for the LCS problem.

A. Beam Search

In the following we give a technical description of the (probabilistic) BS algorithm that is used for construction solutions within the ACO framework. The implemented BS algorithm—see Algorithm 2—works roughly as follows. Solutions are constructed from left to right. Partial solutions are extended by appending exactly one letter. The algorithm requires two input parameters: $k_{bs} \in \mathbb{Z}^+$ is the so-called beam width and $\mu \in \mathbb{R}^+ \geq 1$ is a parameter that is used to determine the number of children that can be chosen at each step. For each application the algorithm maintains a set $B$ of subsequences (that is, partial solutions) called the beam. At the start of the algorithm $B$ only contains the empty string denoted by $\emptyset$, that is, $B := \emptyset$. Let $C$ denote the set of all possible extensions of the subsequences in $B$. At each step, $\lfloor \mu k_{bs} \rfloor$ extensions from $C$ are chosen based on a greedy function and the pheromone values. In case a chosen extension represents a complete solution, it is stored in set $B_{\text{con}}$. Otherwise, it is added to set $B$, in case its upper bound value, denoted by $UB()$, is greater than the length of the best-so-far solution $t^{bs}$. At the end of each step, the new beam $B$ is reduced if it contains more than $k_{bs}$ partial solutions. This is done by evaluating the subsequences in $B$ by the above mentioned upper bound function $UB()$, and by selecting the $k_{bs}$ subsequences with the greatest upper bound values. In the following we explain the four different
functions of Algorithm 2 in detail.

Produce_Extensions(B): Given the current beam B as input, function Produce_Extensions(B) produces the set C of non-dominated extensions of all the subsequences in B. This is done as follows. Given a partial solution t to a problem instance \((S, \Sigma)\):

1) Let \(s_i = x_i, y_i\) be the partition of \(s_i\) into substrings \(x_i\) and \(y_i\) such that \(t\) is a subsequence of \(x_i\), and \(y_i\) has maximal length. Given this partition, which is well-defined, we keep track of position pointers \(p_i := |x_i|\) for \(i = 1, \ldots, n\). The partition into substrings as well as the corresponding position pointers are shown by means of an example in Figure 1.

2) The position of the first appearance of a letter \(a \in \Sigma\) in a string \(s_i \in S\) after the position pointer \(p_i\) is well-defined and denoted by \(p_i^a\). In case letter \(a \in \Sigma\) does not appear in \(y_i\), \(p_i^a\) is set to \(\infty\). Again, see Figure 1 for an example.

3) Letter \(a \in \Sigma\) is called dominated, if there exists at least one letter \(b \in \Sigma\), \(a \neq b\), such that \(p_i^b < p_i^a\) for \(i = 1, \ldots, n\). For an example consider the partial solution \(t = b\) for the problem instance considered in Figure 1. As letter \(a\) always appears before letter \(d\) in \(y_i\) (\(i \in \{1, 2, 3\}\)), we say that \(a\) dominates \(d\).

4) \(\Sigma_d \subseteq \Sigma\) denotes the set of non-dominated letters of the alphabet \(\Sigma\) with respect to \(t\). Obviously it is required that a letter \(a \in \Sigma_d\) appears in each string \(s_i\) at least once after the position pointer \(p_i\).

More specifically, \(C\) is the set of subsequences \(ta\), where \(t \in B\) and \(a \in \Sigma_d\).

Filter_Extensions(C): This second function extends the non-domination relation—as defined above—from the extensions of one specific subsequence to the extensions of different subsequences of the same length. Formally, given two extensions \(ta, zb \in C\), where \(t \neq z\) but not necessarily \(a \neq b\), \(ta\) dominates \(zb\) if and only if the position pointers concerning \(a\) appear before the position pointers concerning \(b\) in the corresponding remaining parts of the \(n\) strings. All dominated partial solutions are removed from \(C\).

Choose_Extension(C): The choice of an extension from \(C\) is done as follows. First, based on the values of a greedy function and the phenomone values a probability for each extensions \(ta \in C\) is generated. More specifically, the greedy function of an extension \(ta \in C\) is the following one:

\[
\eta(ta) := \left( \sum_{i=1}^{n} \frac{p_i^a - p_i^b}{|y_i|} \right)^{-1}
\]  \(3\)

However, instead of using directly these greedy values, we decided to use the corresponding ranks instead. For evaluating a child \(v = ta \in C\) we use the sum of the ranks of the greedy weights that correspond to the construction steps performed to construct string \(v\). Let us assume that \(v\) is on the \(i\)-th level of the search tree, and let us denote the sequence of characters that forms string \(v\) by \(v_1 \ldots v_i\), that is, \(v = v_1 \ldots v_i\). Then,

\[
\nu(v) := \sum_{j=1}^{i} r(v_1 \ldots v_j)
\]

\(4\)

where \(v_1 \ldots v_j\) denotes the substring of \(v\) from position 1 to position \(j\), and \(r(v_1 \ldots v_j)\) denotes the rank of the corre-

---

Algorithm 2 Procedure ProbabilisticBeamSearch\((h_{\text{bs}}, \mu)\)

of Algorithm 1

1: input: \(h_{\text{bs}}\) and \(\mu\)
2: \(B_{\text{curr}} := \emptyset, B := \{\emptyset\}, t_{\text{ref}} := \emptyset\)
3: while \(B \neq \emptyset\) do
4: \(C := \text{Produce}\_\text{Extensions}(B)\)
5: \(C := \text{Filter}\_\text{Extensions}(C)\)
6: \(B := \emptyset\)
7: for \(k = 1, \ldots, \min\{ |\mu|_{\text{bs}}, |C| \}\) do
8: \(za := \text{Choose}\_\text{Extension}(C)\)
9: \(t := za\)
10: if \(|UB(t)| = |t|\) then
11: \(B_{\text{curr}} := B_{\text{curr}} \cup \{t\}\)
12: if \(|t| > |t_{\text{ref}}|\) then \(t_{\text{ref}} := t\) end if
13: else
14: if \(|UB(t)| \geq |t_{\text{ref}}|\) then \(B := B \cup \{t\}\) end if
15: end if
16: \(C := C \setminus \{t\}\)
17: end for
18: \(B := \text{Reduce}(B, h_{\text{bs}})\)
19: end while
20: output: The ACO-version \(T_{\text{bs}}\) of \(
\arg\max \{|t| \mid t \in B_{\text{curr}}\}
\)
Fig. 1. In this graphic we consider instance \( I^{fa} = (S = \{s_1, s_2, s_3\}, \Sigma = \{a, b, c, d\}) \), where \( s_1 = beadedc, s_2 = caabadd, \) and \( s_3 = baceddd \). Moreover, the current partial solution is \( t = ba \). Figures (a), (b), and (c) show the corresponding division of \( s_i \) into \( x_i \) and \( y_i \), as well as the setting of the position pointers \( p_i \) and the next positions of the 4 letters in \( y_1 \). In case a letter does not appear in \( y_i \), the corresponding pointer is set to \( \infty \). This is the case for letters \( a \) and \( b \) in \( y_1 \): \( p^b_1 := \infty \) and \( p^a_1 := \infty \).

The probability for a deterministic choice, also called uniform probability. In case of a deterministic choice, we probabilistically, or deterministically. This is decided with the position of letter \( a \). Remember in this context, that \( n \) is the number of occurrences of letter \( v \). Figure \( 2(a) \) shows the corresponding division of \( s_i \) into \( x_i \) and \( y_i \), as well as the setting of the position pointers \( p_i \) and the next positions of the 4 letters in \( y_1 \). In case a letter does not appear in \( y_i \), the corresponding pointer is set to \( \infty \). This is the case for letters \( a \) and \( b \) in \( y_1 \): \( p^b_1 := \infty \) and \( p^a_1 := \infty \).

### III. Experimental Evaluation

We implemented Beam-ACO in ANSI C++ using GCC 3.2.2 for compiling the software. The experimental results that we outline in the following were obtained on a PC with an AMD64X2 4400 processor and 4 Gigabyte of memory.

The set of benchmark instances that we chose for testing was introduced by Shyu and Tsai in [17]. Their instances are biologically inspired, and thus, they consider alphabets of size \(|\Sigma| = 4\), corresponding to DNA sequences, and of size \(|\Sigma| = 20\), corresponding to protein sequences. Shyu and Tsai provided three different types of instances. One is randomly generated and we denote this set henceforth as Random. The other two sets consist of real DNA and protein sequences of rats and viruses, Rat respectively Virus. Each of these three sets consists of 20 instances, 10 with an alphabet size of 4 and another 10 with an alphabet size of 20. The string length is always 600, while the number of strings per instance varies from 10 to 200, that is \( n \in \{10, 15, 20, 25, 40, 60, 80, 100, 150, 200\} \).

First we conducted tuning experiments. We considered the following three parameters for tuning: (1) The beam width \( k_{sus} \in \{20, 30, 40\} \), (2) parameter \( \mu \in \{1.5, 2.0, 3.0\} \), and (3) the determinism rate \( q \in \{0, 0.2, 0.4, 0.6, 0.7, 0.8, 0.9\} \). For tuning we used the 20 instances from the Virus set. We applied Beam-ACO for each combination of \( k_{sus}, \mu \) and \( q \) 10 times with a computation time limit of 200 seconds to each problem instance. The tuning results can be summarized as follows. The beam width has no significant influence on both the quality and the running time of the algorithm. The setting the with \( k_{sus} = 40 \) seemed to work slightly better than the other settings, we decided to use this setting for the rest of the experiments. In contrast, both the determinism rate and the setting of parameter \( \mu \) have a significant effect on the quality of the achieved results. Both is shown by means of graphics in Figure 2. Figures 2(a) and 2(b) show for each setting of the determinism rate \( q \) the ranks of the average results achieved by the algorithm over a set of problem instances in form of box plots. For these experiments we adopted a setting of \( \mu = 2.0 \). Figure 2(a) shows the box-plots for the 10 instances with \(|\Sigma| = 4\), while Figure 2(b) shows the same information for the 10 instances with \(|\Sigma| = 20\). Note that the lower a box the better the corresponding algorithm setting. From the results it is clear that a determinism rate of \( q = 0.0 \) does not work well at all. This means that at least some degree of determinism is needed. While the results for the instances with \(|\Sigma| = 4\) are not really conclusive for what concerns the choice of \( q \) from \{0.2, 0.4, 0.6, 0.7, 0.8, 0.9\}, the results for the instances with \(|\Sigma| = 20\) clearly show that the
algorithm’s performance keeps increasing with increasing
determinism rate until $q = 0.8$. Therefore, we decided to
adopt this setting for all further experiments. The graphics in
Figures 2(c) and 2(d) show the results of Beam-ACO for the
different settings of $\mu$. Note that for these experiments $k_{ow}$ was fixed to 40 and $q$ to 0.8. The curves show the
evolution of the algorithm rank that was computed with
respect to the average algorithm performance over 10 trials
for the 10 instances with $|\Sigma| = 4$, respectively $|\Sigma| = 20$. This
time the results for the instances with $|\Sigma| = 4$ are more
conclusive. While for small instances—with respect to the
number of strings—a rather small setting of $\mu$ seems
required, larger instances seem to require a larger $\mu$. In
particular, for $n \in \{10, 15\}$ the setting $\mu = 1.5$ is best,
for $n \in \{20, 25, 40, 60\}$ the setting $\mu = 2.0$ is best, while
for $n \in \{80, 100, 150, 200\}$ the setting of $\mu = 3.0$ is
best. In fact, these are the settings that we have adopted
for the final experiments for what concerns instances
with $|\Sigma| = 4$ from sets Rat and Random. Concerning
the instances with $|\Sigma| = 20$, the tuning results are less
conclusive (see Figure 2(d)). Therefore, we decided for a
reasonable compromise achieved by the setting $\mu = 2.0$ for
all remaining experiments concerning instances with
$|\Sigma| = 20$.

With the settings as described above we applied Beam-
ACO for 10 times with a computation time limit of 200
seconds to each of the 60 problem instances. The final
results—separated for the three instance sets—are presented
in Tables II, III and IV. All three tables provide information
about the results of three different algorithms. Apart from
the results of Beam-ACO we also present the results of the
standard ACO algorithm proposed by Shyu and Tsai in [17]
and the results of beam search [3], which is the current state-
of-the-art algorithm. For ACO we provide for each instance
the information that was given in [17], that is, the average
results and the average computation times over 10 runs, as
equivalently as the corresponding standard deviations. For beam
search, which is a deterministic heuristic, we provide the
result in addition to the computation time. For Beam-ACO
we provide the same information as for ACO. In addition, the
first column concerning Beam-ACO contains the best result
achieved for each instance over 10 runs. In case of a gray
background, Beam-ACO has achieved a new best solution for
the corresponding instance. On the other side, when Beam-
ACO was not able to reach the performance of beam search,
the corresponding result of beam search is printed in italic
style.

The following observations can be made. Altogether,
Beam-ACO is able to produce new best solutions in 31
out of 60 cases. Only in 7 cases, Beam-ACO produces a
result that is inferior to the one of beam search. Most of
the new best solutions are obtained for instances with $|\Sigma| = 4$.
On the other side, all cases in which Beam-ACO does not
reach the performance of beam search concern instances
with $|\Sigma| = 20$. In our opinion, this indicates that beam
search is working better for instances with larger alphabets,
while for instances with smaller alphabets there was still
room for improvement. Concerning the computation times,
the processor that we used for evaluating Beam-ACO is
presumably more than twice as fast as the processors used
for the evaluation of ACO and beam search. This means
that Beam-ACO is significantly slower than ACO and beam
search. However, as the LCS problem is not a time critical
problem, it is much more important to improve in terms
of solution quality, even with the disadvantage of increased
running times. A critical reader might remark at this point
that the performance of beam search may be increased by
choosing a larger beam width. Therefore, we applied beam
search (as published in [3]) to all 60 problem instances with
an increased beam width of 500. Note that the original setting
in [3] was 100. The results that are presented in Table V
show indeed that the performance of beam search increases.
However, this increase in quality comes at the cost of a
considerably elevated computation time. New best solutions
are found in 17 cases (as marked by a gray background).
However, in 16 cases (especially for instances with $|\Sigma| = 4$
and a rather small number of strings) beam search with the
setting of $k_{ow} = 500$ is inferior to Beam-ACO. These cases
are marked by an asterisk. This demonstrates clearly the
utility of adding a learning component to beam search as
done in Beam-ACO.

IV. Conclusions

In this work we have proposed an enhanced Beam-ACO
approach for the longest common subsequence problem. The
computational results, obtained on 60 problem instances from
the literature, have shown that Beam-ACO is able to find new
best solutions in 31 cases. In general, the results indicate the
usefulness of adding a learning component in the style of
Beam-ACO to beam search.

Future work will be concerned with the development of an
effective local search algorithm for enhancing Beam-ACO.
This is a challenging task, because at first sight the longest
common subsequence problem does not seem suitable for the
development of local search algorithms.

Acknowledgements

This work was supported by grant TIN2007-66523 (FOR-
MALISM) of the Spanish government. Moreover, Christian
Blum acknowledges support from the Ramón y Cajal pro-
gram of the Spanish Ministry of Science and Innovation.

References

Addison-Wesley, Reading, MA, 1983.
subsequence algorithms. In Proceedings of SPIRE 2000 – 7th Inter-
national Symposium on String Processing and Information Retrieval,
longest common subsequence problem. Computers & Operations
optimization. IEEE Transactions on Man, Systems and Cybernetics –
Fig. 2. Tuning results presented in graphical form. Figures (a) and (b) concern the tuning of the determinism rate, whereas Figures (c) and (d) present results concerning the tuning of parameter $\mu$. More information is given in the text.


<table>
<thead>
<tr>
<th>Instance</th>
<th>ACO</th>
<th>Beam search</th>
<th>Beam-ACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>197.2 (2.0)</td>
<td>211 (9.8)</td>
<td>216 (215.70 (0.48)</td>
</tr>
<tr>
<td>15</td>
<td>185.2 (1.3)</td>
<td>194 (13.2)</td>
<td>197 (196.60 (0.52)</td>
</tr>
<tr>
<td>20</td>
<td>176.2 (1.3)</td>
<td>184 (14.9)</td>
<td>186 (185.60 (0.52)</td>
</tr>
<tr>
<td>25</td>
<td>172.2 (0.7)</td>
<td>179 (15.8)</td>
<td>181 (179.90 (0.57)</td>
</tr>
<tr>
<td>40</td>
<td>161.4 (1.3)</td>
<td>167 (21.0)</td>
<td>170 (169.30 (0.67)</td>
</tr>
<tr>
<td>60</td>
<td>155.4 (1.3)</td>
<td>161 (27.6)</td>
<td>162 (161.30 (0.53)</td>
</tr>
<tr>
<td>80</td>
<td>151.6 (0.8)</td>
<td>156 (33.5)</td>
<td>156 (155.90 (0.32)</td>
</tr>
<tr>
<td>100</td>
<td>148.8 (1.3)</td>
<td>154 (40.3)</td>
<td>154 (153.20 (0.48)</td>
</tr>
<tr>
<td>150</td>
<td>143.4 (0.8)</td>
<td>148 (56.4)</td>
<td>149 (148.20 (0.42)</td>
</tr>
<tr>
<td>200</td>
<td>141.0 (0.6)</td>
<td>146 (74.3)</td>
<td>146 (145.30 (0.48)</td>
</tr>
<tr>
<td>20</td>
<td>54.0 (1.1)</td>
<td>61 (33.3)</td>
<td>60 (60.00 (0.00)</td>
</tr>
<tr>
<td>15</td>
<td>46.2 (1.6)</td>
<td>51 (37.6)</td>
<td>51 (50.70 (0.48)</td>
</tr>
<tr>
<td>20</td>
<td>42.4 (1.3)</td>
<td>47 (39.5)</td>
<td>45 (45.00 (0.00)</td>
</tr>
<tr>
<td>25</td>
<td>40.0 (1.1)</td>
<td>43 (39.5)</td>
<td>43 (43.00 (0.00)</td>
</tr>
<tr>
<td>40</td>
<td>34.2 (0.7)</td>
<td>37 (43.2)</td>
<td>38 (38.00 (0.00)</td>
</tr>
<tr>
<td>60</td>
<td>30.6 (0.8)</td>
<td>34 (46.5)</td>
<td>34 (34.00 (0.00)</td>
</tr>
<tr>
<td>80</td>
<td>29.0 (1.1)</td>
<td>32 (53.2)</td>
<td>32 (32.00 (0.00)</td>
</tr>
<tr>
<td>100</td>
<td>28.4 (0.8)</td>
<td>31 (59.2)</td>
<td>31 (31.00 (0.00)</td>
</tr>
<tr>
<td>150</td>
<td>26.0 (0.4)</td>
<td>29 (75.6)</td>
<td>29 (29.00 (0.00)</td>
</tr>
<tr>
<td>200</td>
<td>25.0 (0.2)</td>
<td>27 (98.0)</td>
<td>27 (27.00 (0.00)</td>
</tr>
</tbody>
</table>

### TABLE III

Results for the Rat instances.

<table>
<thead>
<tr>
<th>Instance</th>
<th>ACO</th>
<th>Beam search</th>
<th>Beam-ACO</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>182.0 (2.4)</td>
<td>191 (9.7)</td>
<td>197 (195.00 (1.33)</td>
</tr>
<tr>
<td>15</td>
<td>166.6 (1.3)</td>
<td>173 (12.3)</td>
<td>177 (176.80 (0.42)</td>
</tr>
<tr>
<td>20</td>
<td>160.0 (1.3)</td>
<td>163 (12.6)</td>
<td>168 (167.40 (0.52)</td>
</tr>
<tr>
<td>25</td>
<td>155.8 (1.3)</td>
<td>162 (15.8)</td>
<td>164 (163.20 (0.79)</td>
</tr>
<tr>
<td>40</td>
<td>143.4 (0.8)</td>
<td>146 (19.4)</td>
<td>152 (150.70 (1.06)</td>
</tr>
<tr>
<td>60</td>
<td>142.4 (1.7)</td>
<td>144 (26.7)</td>
<td>147 (146.60 (0.52)</td>
</tr>
<tr>
<td>80</td>
<td>128.8 (0.7)</td>
<td>135 (31.8)</td>
<td>135 (133.90 (0.88)</td>
</tr>
<tr>
<td>100</td>
<td>124.6 (2.0)</td>
<td>132 (38.5)</td>
<td>134 (133.10 (0.57)</td>
</tr>
<tr>
<td>150</td>
<td>115.6 (1.3)</td>
<td>121 (51.1)</td>
<td>123 (122.10 (0.32)</td>
</tr>
<tr>
<td>200</td>
<td>114.6 (2.3)</td>
<td>121 (69.1)</td>
<td>121 (119.80 (0.63)</td>
</tr>
<tr>
<td>20</td>
<td>63.4 (1.3)</td>
<td>69 (27.4)</td>
<td>70 (69.80 (0.42)</td>
</tr>
<tr>
<td>15</td>
<td>56.6 (0.8)</td>
<td>60 (36.7)</td>
<td>61 (60.20 (0.42)</td>
</tr>
<tr>
<td>20</td>
<td>47.8 (0.7)</td>
<td>51 (34.4)</td>
<td>53 (52.40 (0.52)</td>
</tr>
<tr>
<td>25</td>
<td>46.2 (1.3)</td>
<td>51 (39.0)</td>
<td>51 (50.50 (0.53)</td>
</tr>
<tr>
<td>40</td>
<td>44.2 (1.3)</td>
<td>49 (47.4)</td>
<td>48 (48.00 (0.00)</td>
</tr>
<tr>
<td>60</td>
<td>43.0 (0.4)</td>
<td>46 (60.3)</td>
<td>45 (45.00 (0.00)</td>
</tr>
<tr>
<td>80</td>
<td>39.6 (0.8)</td>
<td>43 (64.4)</td>
<td>42 (42.00 (0.00)</td>
</tr>
<tr>
<td>100</td>
<td>37.0 (1.1)</td>
<td>38 (64.8)</td>
<td>37 (37.00 (0.00)</td>
</tr>
<tr>
<td>150</td>
<td>34.0 (1.1)</td>
<td>36 (77.8)</td>
<td>36 (35.10 (0.32)</td>
</tr>
<tr>
<td>200</td>
<td>32.4 (1.3)</td>
<td>33 (101.0)</td>
<td>33 (32.10 (0.32)</td>
</tr>
</tbody>
</table>
### TABLE IV
Results for the Virus instances.

| Instance | $|\Sigma|$ | $n$ | ACO avg time (s) std avg time (s) std | Beam-ACO result avg time (s) std | best avg time (s) std |
|----------|--------|-----|-------------------------------------|---------------------------------|-----------------------|
| 4        | 4      | 10  | 197.6 (1.3) 3.7 (0.7)              | 212 11.6                        | 217 216.00 (0.47) 76.67 (51.96) |
| 15       | 15     | 15  | 183.6 (1.3) 7.9 (2.0)              | 193 15.4                        | 200 199.10 (0.32) 64.56 (50.53) |
| 20       | 20     | 20  | 173.8 (2.5) 20.4 (6.5)             | 181 17.2                        | 184 183.00 (0.67) 49.84 (40.51) |
| 25       | 25     | 25  | 179.0 (1.8) 18.3 (5.3)             | 185 17.9                        | 189 187.90 (0.74) 98.31 (67.51) |
| 40       | 40     | 40  | 155.0 (2.1) 20.5 (3.4)             | 162 21.9                        | 166 162.80 (1.62) 93.51 (56.30) |
| 60       | 60     | 60  | 150.6 (1.3) 30.8 (9.3)             | 158 29.1                        | 160 158.60 (0.70) 73.50 (47.35) |
| 80       | 80     | 80  | 145.8 (1.3) 45.5 (6.9)             | 153 36.0                        | 154 153.30 (0.67) 84.78 (59.96) |
| 100      | 100    | 100 | 143.4 (2.7) 23.8 (10.3)            | 150 43.9                        | 151 150.00 (0.82) 89.55 (61.29) |
| 150      | 150    | 150 | 141.6 (0.8) 50.0 (21.3)            | 148 64.5                        | 149 147.70 (0.95) 91.72 (51.28) |
| 200      | 200    | 200 | 140.6 (1.3) 65.6 (15.6)            | 145 84.5                        | 148 146.50 (1.08) 116.19 (61.79) |

### TABLE V
Results for beam search (published in [3]) with an increased beam width of 500, that is, $k_{beam} = 500$.

| Instance | $|\Sigma|$ | $n$ | Random result avg time (s) std avg time (s) std | Rat result avg time (s) std avg time (s) | Virus result avg time (s) std avg time (s) |
|----------|--------|-----|-------------------------------------|---------------------------------|-----------------------|
| 10       | 10     | 10  | *213 147                            | 61 601                          | *196 166              | 70 297                        |
| 15       | 15     | 15  | 200 219                             | 52 929                          | 176 224              | *60 419                        |
| 20       | 20     | 20  | 188 249                             | 47 1120                         | 165 236              | 53 444                        |
| 25       | 25     | 25  | 181 249                             | 43 1060                         | 164 252              | 51 523                        |
| 40       | 40     | 40  | *169 294                            | 38 955                          | *148 284              | 48 677                        |
| 60       | 60     | 60  | 162 295                             | 34 858                          | 147 350              | 46 771                        |
| 80       | 80     | 80  | 157 281                             | 33 861                          | 137 288              | 44 856                        |
| 100      | 100    | 100 | 155 314                             | 31 849                          | *133 323              | 39 676                        |
| 150      | 150    | 150 | *148 336                            | 29 833                          | 126 342              | 37 499                        |
| 200      | 200    | 200 | 148 377                             | 27 920                          | 121 360              | 34 636                        |