Computationally Hard Problems 02249 - Lazy Travelling Salesperson

Niels Thykier s072425
Melvin Winstrøm-Møller s072435
Morten Sørensen s072440

November 9, 2010

Contents

Chapter 1

Abstract

This project was written as part of an assignment in the course 02249 Computationally Hard Problems at DTU, in the fall of 2010. The implementation language for the algorithm was Java.

Chapter 2

Assignment

2.1 a) Problem description

Read and understand the problem. Describe in colloquial terms what the problem is about and explain the main differences to the classical TSP.

2.1.1 Description of the problem

The input consists of 3 positive integers k, m and B, and a graph G. The graph G is described by its size n as well as an edge-weight matrix WM with positive integer weights. The numbers k and m are constrained by the relation:

$$n - k - m \geq 2$$

A problem instance PI gives YES if there is a cycle C in G with a specific form, and has a weight w(C) which is at most B, and NO otherwise. The form of the cycle can be described in two parts.

The first part of the cycle is the first m nodes in WM. The second part tours a number of different nodes $i_1..i_{n-k-m}$. None of $i_2..i_{n-k-m-1}$ is contained in the m first nodes.

The number of edges in C, equal to the number of nodes $|C|$, is described by $\star$. It is seen that the number of edges is equal to m plus $n - k - m - 1$ plus 1, which gives:

$$|C| = n - k$$

A possible modelling for the LazyTSP problem is that a lazy salesperson has to visit a required number of cities $n - k$, starting and ending in city 1. She starts off lazily with simply visiting the first m+1 (starting with city 1) cities on her list of cities, not thinking about how time-effective the route is. Once in the m+1'th city, she discovers that she has a deadline to meet. Now, does there exist at least one route that visits the remaining number of cities and ends in the first city before the deadline ends? Note that the poor salesperson may already have exceeded the deadline during her lazy route.

2.1.2 Comparison with Travelling Salesperson

If m and k could be set to zero, the problem would simply be TSP, since the number of cities that must be visited $n - k$ would become equal to n, which means that the cycle should visit every single node, and not have any duplicate nodes.

Given the similarity with TSP, some of the techniques used for TSP may be useful for LazyTSP. Care should be taken, however, since k is never zero, and thus not all cities have to be visited. This means that some cities must be picked for the solution, while others may not be picked, which is different from TSP, where it is known that all cities must be picked at some point.

2.2 b) NP

Show that LazyTSP is in NP.

To show that LazyTSP is in NP, it must be shown that there exists a polynomial-time algorithm A that takes as input a problem instance of LazyTSP and a random string and determines whether the random string is a solution for LazyTSP. Basically, whether or not a suggested solution can be checked in polynomial time.

Algorithm Below, the algorithm A will be described:

1. Calculate the number $v = (n - p - m - 1) * (log_2(n) + 2$ and check the bitstring. If it is below v bits, skip it.
2. Split the first v bits into (n - p - m - 1) blocks of equal size, and interpret each block as a number, specifically a node index. Check that no number occurs twice.
3. Check that every number is in the range $m + 2..n$.

4. Interpret the (n - p - m -1) as $i_2..i_{n-k-m}$, and calculate the left-hand side from $\star$. If the value is below or equal to B, return YES. Else, return NO.

Running time In this paragraph, the running time will be shown to be polynomial.

Step 1 takes linear time in n to check the number of bits. Step 2 takes linear time in n to split each block of bits and to convert each block. Checking that no number occurs twice takes square time in n. Step 3 takes linear time. Step 4 takes constant time.

Since the number of steps are constant, and each step takes polynomial time in n, the algorithm is polynomial.

Probability of YES and NO In this paragraph, the probability of YES if the problem instance has a solution will be shown to be non-zero, and the probability of NO if the problem instance does not have a solution will be shown to be 1.

YES: It is assumed that there exists some solution. Consider some soluion. This solution can be described as a bitstring encoding the solution in the form of $i_2..i_{n-k-m}$, in which each index is a block of size $log_2(n) + 2$. When run through the algorithm, this solution will pass step 1, 2 and 3. When it comes to step 4, the value will be calculated, and per definition, since it is a solution, it will fullfil the constraint given in $\star$, after which YES will be returned. Since the input string randomly just might happen to be of the solutions form, the probability for YES is positive, and thus non-zero.

NO: Assume that there does not exist any solution. If the bitstring does not pass step 1, 2 and 3, NO will be returned. If the bitstring does pass those steps, it is a valid guess. Now, since there does not exist any solution, it will not fulfill the constraint $\star$, and step 4 will calculate a value which is larger than B, and then return NO. Thus, if there is no solution to the problem instance, NO will always be returned.

2.3 c) NP-complete

Show that LazyTSP is NP-complete. What can you say about the complexity of the problem if you know that $k = n - a$ or $m = n - a$ for some $a \in N$, which does not depend on n?

2.3.1 Proving NP-completeness

In b) it was shown that LazyTSP was in NP. To show that LazyTSP is NP-complete, it only needs to be shown that it is at least as hard as another NP-complete problem. TSP seems to be a suitable candidate for this. If a problem instance of TSP can be converted to a problem instance of LazyTSP in polynomial time, and the YES-NO-property does not change, then it has been shown that LazyTSP is at least as hard as TSP, and is thus NP-complete.

Transformation of problem instance The input to the TSP is a graph G and a value B, similar to LazyTSP. As part of the transformation, m and k is each set to 1, and n is simply the number of nodes in G. Transforming the graph G to the edge-weight matrix WM takes polynomial time.

Now, as part of the transformation, the problem instance is extended. A node r1 is added to WM, at position 2, and r1 have special weights: The weight between the first node fn and r1 is 1, and the weights between r1 and all nodes except fn is the same as the weights from fn to all nodes except r1 and fn. B is furthermore increased by 1. This simulates that the node after fn will have to be visited, since m is equal to one. To handle k, another node r2 is added to the end, and the weight between r2 and all other nodes is infinity. This ensures that r2 will never be added, and handles that k=1 nodes will not be picked for the tour.

Preservance of YES-NO property To prove that LazyTSP is NP-complete, it must be shown that the transformation of TSP problem instance X preserves the YES-NO property when transformed into LazyTSP problem instance X2 = T(X). In order to show this, it will first be shown that if the problem instance X gives YES, then T(X) also gives YES, after which it will be shown that if T(X) gives YES, then X also gives YES.

If X gives YES, then T(X) gives YES

Assume that the problem instance X gives YES, and thus have a solution for TSP. This means that there is a cycle in G that covers all nodes once and sums up to at most B. This cycle will be noted by the indexes,

$$s = fn, i_2, .., i_n$$

, where i_n is the index of the node just before i_1 in the cycle.

The transformation will add two nodes, and the new cycle solution will include the first:

$$s' = fn, r1, (i_2 + 1), (i_3 + 1), .., (i_n + 1) = fn, r1, i'_2, i'_3, .., i'_n$$

. Note that 1 has been added to the indexes to accomodate that r1 was inserted into WM. The number of nodes n' is equal to n+2, and B' is equal to B+1. The edge weight between fn and r1 is 1, and the edge weight between r1 and i'_2 is equal to w(fn, $i\prime_2$).

Now, the first part of the left-hand side of $\star$, the sum of m, becomes equal to 1. The second sum in $\star$ is the sum of the nodes from m+1 to n'-k. This becomes the range 2 to n+1 in s', which fits with

$$w(r1, i'_2) + w(i'_2, i'_3) + .. + w(i'_{n-1}, i'_n)$$

. But since $w(r1, i'_2)$ is equal to w(fn, $i\prime_2$), this sum is equal to the sum of s taken as a path. The third part of $\star$ is simply equal to $w(i_n, fn)$. Taken together, the value thus becomes equal to $w(s') \leq B + 1 = B'$, fulfilling the constraint $\star$. The sequence solution to T(X) is then simply the indexes $r1, i'_2, i'_3, .., i'_n$. It has then been shown how to transform a solution in X to a solution in T(X), and thus preserving that if X gives YES, then T(X) gives YES.

If T(X) gives YES, then X gives YES

Assume that the transformed problem instance T(X) gives YES, and thus have a solution for LazyTSP. This means that there is a cycle in G' that fulfils $\star$. Note this cycle by

$$s' = fn, r1, i'_2, i'_3, .., i'_n$$

. Now, the sum of this is equal to $w(s') \leq B + 1 = B'$. If r1 is removed from the cycle, a link can instead be made between fn and i'_2, giving:

$$s = fn, i_2, .., i_n$$

. This simply gives a decrease in weight of the cycle of 1, since w(fn, r1) is 1, and $w(fn, i'_2) = w(r1, i'_2)$. This results in a weight of the cycle $w(s) \leq B' - 1 = B$. In the original graph, this means that all nodes are covered, and that the cycle has weight less than B, which causes X to give YES. It has then been shown that if T(X) gives YES, then X gives YES.

Conclusion Since it has been shown that there exists a polynomial-time algorithm that transforms problem instances from TSP to LazyTSP, that the YES-NO property of problem instances is preserved in the transformation, and that LazyTSP is in NP, it then follows that LazyTSP is NP-complete.

2.3.2 Considerations regarding complexity

First k will be considered, followed by m.

k

Given the constant a, it follows that for those problem instances for which a does not break the constraints on n, m and k, the difference between k and n will be constant. k can have an impact on the complexity, since k nodes does not have to be picked. If the difference between k and n is constant, then the maximum number of nodes that have to be picked will also be constant. This gives in effect a problem instance where for increasing n, a constant number c of nodes will have to be picked from these, fulfilling $\star$. Ignoring m, this gives a=c, and a number of different combinations of nodes to pick equal to

$$\frac{n!}{(n-a)!} = n*(n-1)*(n-2)*..*(n-a+1) \leq n_1*n_2*n_3*..*n_{a+2} = n^a$$

Note that this formula is similar to the binomial coefficient, but not the same, since the order of the picked nodes matters.

Since a is constant, this means that the time complexity is polynomial.

m

Given the constant a, it follows that for those problem instances for which a does not break the constraints on n, m and k, the difference between m and n will be constant. m has a major impact on the complexity, since the m+1 first nodes does not have to be considered when finding a solution. If the difference between m and n then is constant, the number of nodes that will have to be considered is then also constant. This makes the time complexity polynomial.

2.4 d) Algorithm

Find an algorithm which always gives the correct answer for an input to the LazyTSP, i. e., which always stops and replies YES if it is given a YES-instance and NO otherwise. The algorithm is allowed have exponential worst-case running time but may use "smart" techniques to deal faster with some instances. In case of a YES, your algorithm has to construct a solution $i_1, i_2, ..., i_{n-k-m}$ in accordance with the problem definition. Finally, extend your algorithm in order to solve the optimizing version of the problem, i.e., it should find a solution $i_1, i_2, ..., i_{n-k-m}$ for which B is as small as possible.

Describe in words how the algorithm works.

For the sake of testing and trying things out, two algorihtms have been developed. The first will be explained in detail, while the second will be lightly explained.

As a constraint, only the 2D-euclidean problems are supported.

The two algorithms are somewhat simple. Both of them does not achieve a worst-case running time of O(2^n), but rather O($n!$), which is far from what solutions for TSP can provide (for instance, inclusion-exclusion provides polynomial memory and $O(2^n)$ running time, vastly better than $O(n!)$). However, they are simple and quick to implement, which is fitting given the time constraints. An optimal algorithm should both take advantage of k when it is high, and solve the TSP-like structure when both m and k is low.

2.4.1 Depth-first search with pruning

The first algorithm is based on depth-first search with pruning, in order to decrease the running time. It searches deeply for a solution, and when found, it decreases the global limit for how long any solution may be (since any solution which is longer than that solution is not the best solution). After that, any time a new solution is found, the global limit is updated.

In order to prune away branches in the depth-first search, it finds approximate lower bounds for each branch. These lower bounds are never higher than the best solution in that branch, but may be lower. This ensures that if a lower bound is higher than the global limit, it can safely be cut away, since the best solution in that branch would be longer than the current best solution.

The basic bound is simply calculated as the length of the current sequence added to the distance from the end node in the sequence to start node 1.

For the decision version, instead of setting the initial global upper bound to infinity, set it to B - the weight of the m-part.

The issue with depth-first searching is that branching happens very far out. That means that even when a good upper bound has been found, it may not matter much, since the search may still search branches that could have been cut off at a higher level. And since the search tree increases exponentially, the earlier a branch can be cut, the better. The advantage, however, is that it is guaranteed not to run out of memory, even on large problem instances, since the memory use is minimal.

Pseudo-code description *init*

```
1. Check arguments for validity.
2. Strip the graph of the 2..m nodes, and calculate the weight of their path.
3. return search
```

search

```
1. If (weight(path) + weight(path.end, start) > globalUpperBound) then
   ignore this branch, and back-track.
```

2. If the sequence has full length, test the current path as a possible solution.

3. For each neighbour that is not part of the path and is not the start node:

   (a) Call search with a new path based on that node.

   (b) If the search back-tracked due to no better paths, continue.

   (c) Else, if a new possibly best path was found, select it as the best path if it is better than the current one, and update the global upper bound. If this is decision-version, instead return the solution.

4. Return the best path found so far, or null if decision version.

2.4.2 Branch and bound with greedy path selection

This algorithm is similar to the former algorithm, except for two parts:

The first difference is that the lower bound may be tighther, but it takes more time to compute. For those nodes that constitutes the optimal solution for the current path, the lowest cost possible must be those two edges which are minimal. By taking half of that, a lower bound can be computed for each node. Now, by selecting those nodes that have the lowest half-sum of the lowest edges, which does not occur in the path, a possibly tighther lower bound can be computed.

The second difference relates to the branching. Instead of deep-first searching the possible paths, a greedy search is used instead. This greedy search always picks the path that has the best lower bound. This means that in theory, the greedy search might quickly find a good solution, then later return up in the search tree and cut off a decent amount of branches. This can be much better than the depth-first search. The issue is that the memory use is exponential, which means it scales badly.

2.5 e) Optimizing versions running time

Prove the worst-case running time of your algorithm for the optimizing version.

There are several cases, depending on the values of m and k.

n - m = constant (k ignored) Since it takes O(m) time to remove the 2..m nodes, and the remaining nodes are a constant number of nodes, the worst-case is linear in n.

n - k = constant (m ignored) For init 1 and 2, the time is polynomial.

For search, it is called in the worst case at most n^c times, letting c be a constant. This can be seen by noticing that the depth-first searches all the permutations, and the above formula is derived from the number of permutations calculated in section c.

Search 1 and 2 takes constant time. Search 3 is a loop which is iterated worst-case once per call of search, since search in the worst-case is called once for each iteration of the while. Search 3.a, 3.b and 3.c takes constant time. Search 4 takes constant time.

All in all, O(n^c) time.

No constraints For init 1 and 2, the time is polynomial.

For search, it is called in the worst case at most $n!$ times. This can be seen by noticing that the depth-first searches all the permutations, which for a selection in which the order matters takes $n!$ times.

All in all, O($n!$) time.

2.6 f) Implementation and testing

There is a collection of benchmark inputs to the classical TSP on the web, see http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. These instances are stored in files following the .tsp file format, see the docu- mentation on the homepage.

Implement the algorithm you developed in Part d) and run it on certain .tsp instances, treated as inputs to the optimizing version of LazyTSP. Your software should at least support those .tsp files where TYPE: TSP and EDGE_WEIGHT_TYPE: EUC_2D holds, see the file berlin52.tsp for an example. Solutions should be output along with their costs according to Formula ($\star$). Test how the cost of the solution varies as you increase the parameters k and m.

Instead of developing your software from scratch, you may build upon existing software packages, provided all legal restrictions are obeyed.

The program is implemented in Java. The implementation supports only EUC 2D. Both algorithms previously described has been implemented.

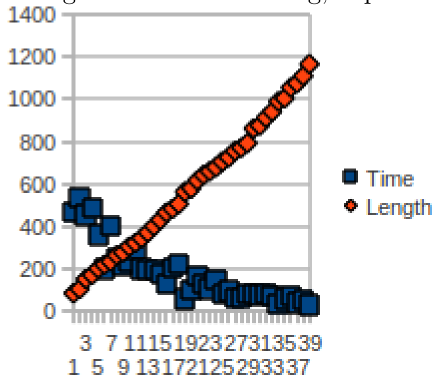
2.6.1 Testing

The testing provides both the solution and visual output. Testing instructions are found in the file "README_TEST".

In the tests performed, it was observed that the depth-search algorithm had trouble handling input over n - k - m ¿ 10. However, it seemed to scale somewhat well, being able to handle pla85900.tsp with m = 1 and k = 85892. In contrast, the greedy algorithm could handle n - m ¡ 20 well, but didn't improve much from increasing k.

Testing for m + k = c Test for eil51.tsp, m+k = 40, depth:

Figure 2.1: m increasing, depth



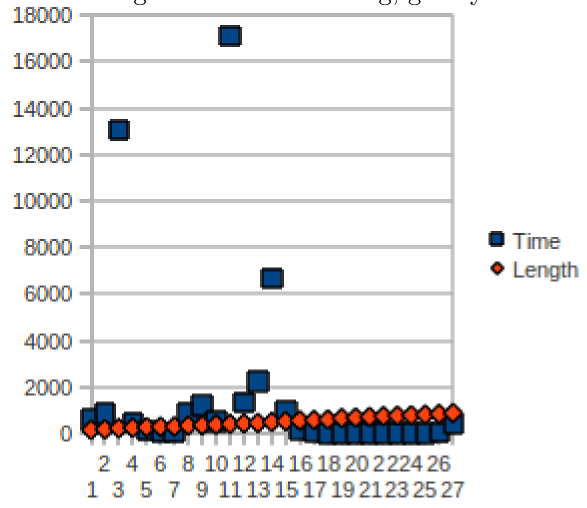
Test for eil51.tsp, m+k = 30, greedy:

The interesting showings from these data sets indicates that the greedy algorithm seem to have a random running time - most of the time it is decent, but it has some problems on some input. In fact, it ran out of memory for m = 29. This seems to show that while the greedy picking of the most promising path can give a better running time, it gives no guarantees.

Conversely, the depth algorithm clearly benefits from a higher m. It never ran out of memory. However, it generally cannot handle n - m - k ¿ 10 very well.

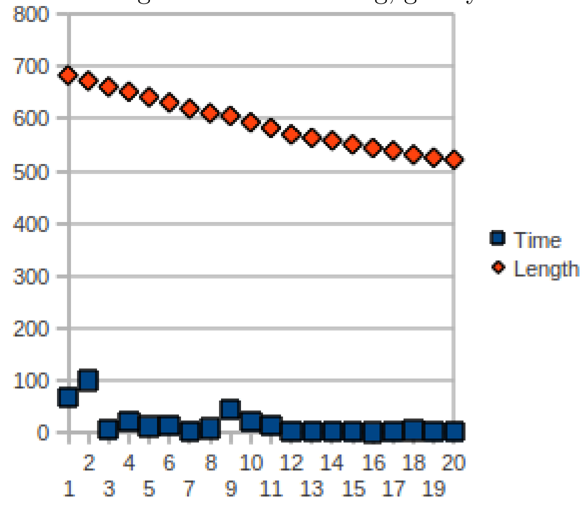
In regards to the cost of the solution, it is clear that the solution becomes higher, the smaller k gets and the higher m gets. This makes intuitively good sense, since the smaller m is, the more options for picking a more effective solution there are. Furthermore, the number of nodes to be picked remains constant.

Figure 2.2: m increasing, greedy



Testing for m = c, k++ Test for eil51.tsp, m = 20, k = 1..20, greedy:

Figure 2.3: k increasing, greedy



As expected, the fewer nodes that has to be included in the cycle, the smaller the distance becomes.

Test for pla85900.tsp, m = 1, k = 85892..85896, depth:

112368.97433105548
110476.24692730559
108638.61430937026
106856.45151641732
56400.0

Two things are of notice here: First, that the distance decreases as expected when fewer nodes have to be selected, but that the distance happens not to decrease linearly; and second, that even though n is very, very large, and m is also very small, the problem can still be handled when the difference between k and n is very small. Another thing to notice is that the depth-search algorithm could handle these inputs without taking much time, while the greedy algorithm happened to fail on lack of memory.