An Algorithm for Special Case Shortest Path Problems with Resource Constraints and Time-Based Costs

Ritish P.W. Oemraw
321878
June 2011
Bachelor Thesis
Erasmus University

Supervisors:
Adriana Gabor
Remy Spliet

Table of contents

1 Introduction:

The Shortest path problem (SPP) has been around for quite a while. Over the years it has seen many variations and extensions. Solutions to any of these problems have proven to be of importance for many practical issues. Perhaps one of the most well-known examples is that of a car's navigational system determining what route to take.

Besides the practical implementations the problem can also be used for theoretical purposes. In both the practical and theoretical implementations the time it takes to solve the problem has been essential. In certain instances, it has been proven that in order to solve the problem (to an optimum) an excessive amount of time is required. For such instances we have come to rely on algorithms that sacrifice the quality of the solution in favour of reducing the solving time.

These algorithms can provide a trade-off between solving time and solution quality that matches the desires of those needing solutions quickly. Imagine having a simple problem that might take just ten seconds to solve. In many cases people are willing to wait ten seconds, but if, like the authors of [1], you have to solve a problem 10.000 times, then waiting ten seconds each time can become an agonizing ordeal.

A fast algorithm does not necessarily have to be a good algorithm. If for instance an algorithm was used that would provide solutions to the problem in two seconds, that would seem good at first. However, the quality of those answers should never be overlooked. It could be that the solutions it provides are so bad in terms of costs or length, that it would simply be unusable. Therefore a good algorithm is one, where the quality of the solutions does not deteriorate too heavily for the time it saves.

Problem description

In this thesis we will be studying a specific extension of the SPP, one with capacity constraints, time-windows and linear time costs (SPPCCTWTC). The goal is to find the shortest path from a given starting node, the source, to another given node, the sink.

Each node has a specific time window in which they can be serviced, a demand that must be met when visiting it and the costs associated with that node are linearly dependant on the start time of the service. Costs are incurred when crossing an arc. The travelling-time of

that arc will also have passed. The load from which to supply the demand is set at the beginning. All costs can be either positive or negative. The network on which this path is to be found does not contain any cycles and for each node, other than the source and sink, there exists an arc that leads directly to the sink.

Research question

For the SPPCCTWTC an algorithm to solve it exactly has already been found by the authors of [2]. For the larger instances however, it will take too long to solve, as described in [2]. Therefore the research question is:

"Can a method be developed that provides a near-optimal solution for the SPPCCTWTC in a short amount of time?"

We will attempt to find a heuristic that can provide a solution faster, while still providing a near-optimal path. To determine the quality of our solutions, we will compare our solution's costs with the costs of the exact algorithm's solution. Due to the use of different programming languages a comparison of the solving time will not be done.

Literature review

Because the SPPCCTWTC is an extended form of the SPP it is a good start to look at the solution methods of the SPP. The SPP can be described as finding a path along the arcs of a weighted network that starts at a given node, ends at another given node and minimizes the sum of the weights on the arcs it traverses. This problem has two prominent solution methods:

- Bellman-Ford's algorithm (as described in [3])
- Dijkstra's algorithm (as described in [4])

The algorithm in [3] can be applied to networks with negative weights, as long as there are no negative weight cycles. The computational bound of that algorithm is $O(V^3)$, where V is the amount of nodes in the network. The algorithm of [4] has a computational bound of $O(V^2)$ and

that makes it faster than that of [3], but it only works in networks that have positive weights on each arc.

Over the years there have been many research papers on various extensions of the SPP. [2, 5, 6] are related to the SPPCCTWTC, while [7] is related to SPP in general but can be useful for the development of the heuristic.

[2] describes an exact algorithm for the SPPCCTWTC . Understanding how the exact method works is essential when attempting to find a fast heuristic. The method used in this article make use of a modified labelling method and a node cost function to determine the optimal service time at nodes to minimize the costs. The authors developed a definition for the node cost function after thorough analysis of the cost structure at the nodes, while taking into account the effects of the time windows of the node's predecessors.

In [5] the authors describe an adaptation of the algorithm of [4] to make it work in networks with negatively weighted arcs. It is still faster than the algorithm in [3], but it is more restrictive; it only works in networks where the negatively weighted arcs are not part of a cycle, not even if that cycle has a positive weight. Seeing as how the SPPCCTWTC can contain negative arc weights but does not contain cycles, this can help to create a faster algorithm than when using the approach of [3].

[6] is about SPP extended with resource constraints. The authors present an example containing capacity constraints and time windows. It describes how time windows can be seen as a resource constraint on the resource time. The presented solution makes use of a labelling method and the elimination of dominated paths. [6] however, does not take into account the time costs that the SPPCCTWTC has.

[7] describes and evaluates several existing heuristics that are being used in practice. These heuristics were developed for situations in which the computation time needs to be very short. Think for example of the car navigational system that has to quickly recalculate the route whenever the driver takes a wrong turn or otherwise deviates from the initial route. These heuristics have proven to reduce the computation time significantly while still resulting in near-optimal solutions. Hopefully a feature that will be maintained when applied to SPPCCTWTC.

Overview

An in-depth analysis of the problem will be provided in chapter 2. This will provide an easy understanding that will be necessary for understanding the algorithm presented in this thesis. The algorithm itself will be explained in chapter 3. The explanation will be done in two parts. The first will explain in detail how the algorithm works. The second part will provide the pseudo-code of the algorithm for a more specific view on the actions taken at each step. In chapter 4 we will take a look at the numerical results of the algorithm. For this purpose we have generated a series of test cases. The creation of these data sets will be detailed before we move on to the results. For each of these cases the solution has been determined exactly in order to compare them with our results. Thereafter, in chapter 5, we will present our conclusion followed by a discussion on possible future research in chapter 6.

2 Problem analysis

The SPPCCTWTC we will examine will be defined on a specific type of directed networks. These networks do not contain any cycles. Furthermore, there exists an arc at each node that leads directly to the sink, except for the source node. The network has a finite amount of nodes, which, together with the no-cycle limitation, implies that there exists a maximum on the amounts of steps there can be in a path.

Associated with each node are their respective time window, costs, time costs and demand. Time costs are linearly tied to the time at the moment of service, that is, the time cost is a constant that is multiplied with the starting time of the service of that node. For each arc we have a cost and travel time. Every type of costs is a real number; they can be negative. This complicates the problem, because a longer path could have lower total costs.

The problem starts with a given load and at each node, that is serviced, the load is reduced by the demand of the serviced node. In our case the load cannot be replenished. Time can pass in two ways. The first is when traversing an arc; the travel time of that arc will have passed when reaching the end of that arc. The second way is when we decide to wait before starting service at a node. This way service can be delayed allowing us to have some influence in the time costs incurred at the serviced nodes.

We will be using the following notation to describe the different aspects of the network, $G = (N, A)$:

$N = \{o, 1, 2, \ldots, n, s\}$: the set of nodes, where o is the source node and s is the sink node;
nc_i : the node costs of node $i \in N$, a real number;
tc_i : the time costs of node $i \in N$, a real number;
$[a_i, b_i]$: the time window of node $i \in N$, with $a_i \leq b_i$ and both being nonnegative real numbers;
d_i : the demand of node $i \in N$, a nonnegative integer;
tt_{ij} : the travel time of the arc between node i and node j, $i, j \in N$, a nonnegative number;
ac_{ij} : the arc costs of the arc between node i and node j, $i, j \in N$, a real number;
T_i : the time at which node $i \in N$ is serviced;
L_i : the remaining load after servicing node $i \in N$;

For a path to be regarded as feasible it must meet the following criteria:

- The path starts at the source, o ;
- The path ends in the sink, s ;
- Each node, i, along the path is serviced within their time window, $a_i, \leq T_i \leq b_i$;
- At the end of the path the remaining load is nonnegative, $L_s \geq 0$;

3 Method description

Description

Initial feasible path

For our heuristic we will start by quickly searching for a low-cost path for the SPPCCTWTC. The path should satisfy all the feasibility criteria. (Lines 1-26 of Algorithm 1) Then we will attempt to improve upon this initial path through the use of a local search method. (Lines 27-68)

To find our initial path in a manner that does not require excessive solving time, we will ignore the time costs for this part of the heuristic. Note that this only affects the costs of a path and that any feasible path that we find for this modified problem is still a feasible path in the SPPCCTWTC, since no constraints were changed.

By ignoring the time costs we are left with the arc costs, the node costs, the capacity constraints and the time-window constraints. The authors of [5] have shown that the time-window constraint can be described as a resource constraint; thereby this modified problem, has just two resource constraints. This new problem can be solved exactly through the use of a labelling method. However, since we are not necessarily interested in the optimal path for this problem, we will be using a non-exact but faster method to construct our initial path.

This path will be constructed iteratively, adding one node to the path at each step. At the beginning the path exists of just the source node. (Lines 1-2 in Algorithm 1) To determine what node to add at each step, all nodes that are incident to an arc leaving the previously added node are examined. (Lines 4-9) This examination is to determine whether the nodes are a feasible option, that is, whether adding it to the path still returns a path that satisfies the requirements of a feasible path, minus the requirement of ending in the sink. If a node is deemed infeasible it is removed from further analysis.

After the feasible nodes have been identified, the next step is to rate them on their likeliness to be part of a near-optimal path in our graph. (Lines 16-19) For this purpose we define the function $R^*(K,i)$ to give an indication of this likeliness. $R^*(K,i)$ is defined as follows:

$$R^*(K,i) = ac_{ji} + nc_i + MinO_i - S(PLT_{K,i} + PLL_{K,i})$$

Where:

- *K is the path that we want to add node i to. The last node of K is referred to as node j.*
- *$MinO_i$ is the minimum arc cost of all arcs leaving node i.*
- *S is a predetermined variable to scale the effects of $PLT_{K,i}$ and $PLL_{K,i}$.*
- *$PLT_{K,i}$ is the percentage of time left after adding node i to path K.*
- *$PLL_{K,i}$ is the percentage of load left after adding node i to path K.*

The derivation of *R*(K,i)* can be explained as follows. High costs are a sign that this node is most likely not part of a near-optimal path, therefore ac_{ji} and nc_i increase the value of *R*(K,i)*. A high *R*(K,i)* value indicates that node *i* is not likely to be part of a near-optimal path.

We choose to define $MinO_i$ as the minimum arc costs of those outbound arcs. We did this because, it could negatively affect the *R*(K,l)* of an otherwise favourable node, *l*, if we let the high outbound arc costs influence *R*(K,l)*. It could in fact have a low-cost outbound arc that would be used to move to the next node if node *l* was added to the path. If we had let the high outbound arc costs influence *R*(K,l)*, we might not have chosen node *l*.

Incidentally, in the same case, if that one low-cost arc would not be used it might have been better to not have chosen node *l* in favour for a node, *m*, which has a higher $MinO_m$ but, with an average outbound arc cost that is lower than that of node *l*. As it stands it is not certain what definition of $MinO_i$ yields the best results overall, but in this thesis we have gotten the best results when using the minimum.

$PLT_{K,i}$ specifies how much time is left as a percentage of the upper limit of the time window of the sink. As the path gets longer, $PLT_{K,i}$ will decrease. This is in accordance with the idea that possible future benefits will decrease as the leftover time decreases. Lower leftover time means fewer options to choose from due to time becoming a limiting factor. If the upper limit of the time window of the sink is trivial, another node could be used as a reference point. With a trivial upper limit of a time window we refer to an upper limit of a node whose time costs are zero and whose upper time window limit will only be crossed when actively choosing to do so. Perhaps the highest non-trivial upper limit of the time windows would be a decent alternative.

$PLL_{K,i}$ specifies the amount of load left as a percentage of the starting load. Like $PLT_{K,i}$, $PLL_{K,i}$ also decreases as the path gets longer as there will be more demand to satisfy.

Again, this will lead to lower possible future benefits as the path gets longer, due to supplies becoming a limiting factor.

S, like $MinO_{i,}$ is also a parameter that is open to discussion. It is used to amplify or dampen the effects of $PLT_{K,i}$ and $PLL_{K,i}$. This is necessary because between different networks the magnitude of the costs can differ greatly, so we need to scale the effects of PLT^{K}_{j} and PLL^{K}_{j} in order to hold their own against the effects of the directly incurred costs. We have

$$S = |\ (min_i(\ nc_i\ ) + min_{ij}(\ ac_{ij}\ )\ )|$$

We use the absolute value so that it never increases the value of *R*(K,i)*. Making *S* dependent on the minimal nc_i and ac_{ij} allows it to be used in different networks whereas a static *S* would not work when all the costs are increased or decreased greatly. The use of the minimization is again linked to the fact that it provided the best results when compared to the alternative options; average and maximization.

The combination of *S*, $PLT_{K,i}$ and $PLL_{K,i}$ is meant to roughly indicate the possible future benefits of adding node *i* to path *K*. In a network where longer paths do not necessarily have equal or higher costs, it is important for the algorithm to attempt to look ahead. If the algorithm only picks nodes that provide a direct benefit, it could be that one cost-increasing node stands between the path and a slew of cost-decreasing nodes. Therefore when picking a node, it is worth looking ahead at the possibilities a node might unlock for future node selections. In our case, we try to estimate the possible future benefits of a node.

When we have determined *R*(K,i)* for all the feasible nodes, *i*. We add the node with the lowest *R*(K,i)* value to the path. It is possible that at one point there are not enough supplies or time left to move on to a new node, thereby making the whole path infeasible. To prevent that this happens, we will use a specific property of the network that we are working with. From each node, except the source, there exists an arc leading directly to the sink. At each step, except the first, the sink is also examined. Adding a check to see if the sink is still a feasible option, can help us prevent that we end with an infeasible path. Now, whenever the sink becomes infeasible we know that in the previous iteration it was still feasible, so if we now remove the previously added node and in its place we put the sink, we are left with a feasible path. (Lines 10-15)

Local Search

Now that we have a feasible path we will attempt to improve upon it by use of a local search method. This method will make use of the time costs of nodes to find possible replacements; therefore we will no longer ignore these costs. In fact, even if this method did not rely on the time costs, the cost comparison of the newly created paths would be meaningless if the time costs would still be ignored.

The first thing we need to do is calculate the costs of our feasible path. As there are most likely multiple time windows that are not binding, all leading to different feasible solutions with different costs, we need to determine at what time we service each node. Non-binding time windows indicate that the service at those nodes can be moved earlier/later without violating the requirements of a feasible path. Servicing a node late may prevent us from servicing subsequent nodes early, so it is important to look at it as a whole, rather than per node.

If the time costs at a node are high it is beneficial to service it earlier, as the time costs are linearly tied to the service time. With negative time costs however, it is better to arrive later. While it is possible to create a heuristic to determine the arrival times quickly, it is but a small sacrifice of solving time to calculate the optimal values for a given path. It would be a great loss if we choose to forsake optimally distributed service times that result in the lowest possible cost for the given path, in favour for a marginal decrease in solving time after carefully creating our path.

The method used to determine the optimal values is described in [2]. The authors of [2] have shown that for a given (partial-)path the lowest costs can be determined by looking at the node-cost function of the last node in that path. The node-cost function provides the time cost component at each point in the time window. It is constructed iteratively, so the node-cost function of the last node is constructed using that of its predecessor. The basic idea of the node-cost function is explained as follows. You start at the successor of the source node, node *i*, for this node the node-cost function is a straight line, depending only on the start time of service at node *i*. For the next node, node *j*, the node-cost function of node *i* is expanded. If the function was increasing we flatten it to a horizontal line, starting and ending at the lowest costs of the line before the flattening. This is done because it is never favourable to start service at any point other than at its lowest cost point. Therefore if we want to leave node *i* at a time later than when the cost function is at its lowest point, we still service node *i* at the time where it results in the lowest costs and then just wait till the time arrives to leave node *i*. The next step is performed by adding the arc and node costs of node *j* to the modified node-cost

function of *i*. Next we extend the domain of the node-cost function to the last moment at which we can leave node *i* and still be able to service node *j* within its respective time window. The costs associated with the extended segment are the same as the cost at the start of the segment. Next the node-cost function is shifted to the right as much as the travel time of the arc from *i* to *j*. For this new domain we add the time-based portion of the costs of servicing node *j* to the node-cost function. Any increasing segment is again replaced with a horizontal segment for the same reasons as stated earlier. The last step is to limit the domain to the time window of node *j*. Now we know for each moment of the time window of *j* what the lowest achievable costs are for servicing node *j* at that time. Repeating this process for all the nodes in the path eventually results in the node-cost function of the sink. In [2] they have proven that the node-cost function is a non-increasing, piecewise linear function. Therefore the minimum of the function can always be found at a breakpoint of the function. Finding the minimum, of the node-cost function of the sink, results in finding the lowest possible cost for the entire path and the service time of the sink that goes with it.

After the total costs have been calculated and the service time of the sink is determined it is possible to return to the preceding nodes to determine the service times of those nodes. This is done by subtracting the travel time of the arc used to reach the sink from the service time at the sink. This new moment is the last possible time at which you can service the predecessor of the sink. Limiting the node-cost function of the sinks predecessor to that new moment, results in a new modified node-cost function. Finding the time, that corresponds to the minimum of this new function, results in finding the time to start service at the predecessor of the sink. This process can be repeated for all the preceding nodes. (Lines 25 -26)

After deriving the costs for our initial path we can begin to apply local search to try to improve on it. We will attempt to replace one node at first and then try to replace a second node and see if the costs of these new paths are lower than our previous lowest cost. If so then that new path will become our best path and its cost will become the new lowest cost.

Using two loops we pick every combination of two nodes to be replaced. (Lines 27 & 45) The first loop runs from the successor of the starting node, till the second predecessor of the sink. This way the first replacement does not try to replace the source or sink node and it always leaves one node option, the predecessor of the sink, available for the second replacement. The second loop runs from the successor of the node of where the first loop is at, till the predecessor of the sink.

The node that will replace the first picked node is chosen following a string of selections. (Lines 31-33) First we need to determine which nodes could replace the picked node. This is done by looking at the predecessor of the picked node and determining which other arcs lead to a node that has an arc leading to the successor of the picked node.

After those nodes have been selected, we need to check whether the constraints are not violated. For each node we check whether there exists at least one set of service times such that each node is serviced within their respective time-windows. This is easily done by choosing the service times as early as possible. That is, never wait with servicing a node so that you have the maximum amount of time available when moving on for subsequent nodes. This way, future possibilities for service times are never limited by earlier service times.

Next we will look at whether the capacity constraint is still met. This is done by adding the demand of the picked node to the load that was left after crossing the path in full and then subtracting the demand of the node that might replace the picked node. If this new value is negative it violates the capacity constraint and thus the node is not a replacement candidate for the picked node.

After these checks we are left with a set of nodes that can replace the picked node. Since we ignored the time costs before, it would be smart to look only at the nodes that don't have higher time costs, than the picked node. So we eliminate nodes with time costs that are higher than that of the picked node. However, time costs, are not the only property to look at, so from the list of leftover nodes, we choose the node that has the lowest node cost to replace the picked node. (Lines 34-35) If by chance no node has proven to be a feasible replacement candidate, then we start over with the next node in line for replacement.

If we have found a node that was a feasible replacement we go on to determine the optimal service time distribution and the associated costs. If the costs associated with this new path are lower than our previously lowest costs, then this new path will become the new best path and its cost, becomes the new lowest costs. This way if the combination of the two replacements does not prove to be better, we can still have a better path than we started with thanks to the single node replacements, which resulted in a better path. (Lines 36-44) Note that after each iteration, in which we update the best path, we start the local search all over again with the new best path.

Next we will attempt to make a second change. The same process is applied to choose the node that will replace the second picked node. However, we do make the first replacement in the path, before deciding on the second replacement. The reason for this is that the first replacement may open up or close some possibilities for the second replacement. It might be

that, when making the first replacement the amount of left over supplies has increased and as a result, a node, that would otherwise be inaccessible, becomes a feasible option. (Lines 45-50)

After both nodes have been replaced, the new costs are calculated and if they are lower than the previous lowest costs, we set the new path as the new best path and start at the beginning of the local search. If the costs are not lower we move on to the next iteration. (Lines 51-62) In the end we will have our best feasible low-cost path with corresponding costs, service time distribution and remaining load.

Algorithm 1

Next we will give a brief overview of our algorithm through its pseudo-code:

Details:

P is the path

ln stands for last added node

candidates can refer to possible nodes to add to the path, or to replace a node from the path

pred(i) indicates the predecessor of node *i*.

suc(i) indicates the successor of node *i*.

breaks cause the current for-loop to be ended pre-maturely

Initialization:

1 $P = [o]$,

2 $ln = o$,

Main execution:

3 **while:** $s \subset N^1$ is false

4 $candidates = \emptyset$

5 **for every** node $n \subset out(ln)$ **do:**

6 **if** $T_{ln} + tt_{ln,n} \leq b_n$ **&** $L_{ln} - d_n \geq 0$

7 $candidates = [candidates, n]$

8 **end**

9 **end**

10 **if** s $\not\subset$ *candidates* **&** $ln \neq o$

11 $P = [P \backslash ln, s]$

12 $L_s = L_{ln} + d_{ln} - d_s$

13 $T_s = T_{ln} - tt_{pred(ln),ln} + tt_{pred(ln),s}$

15 **end**

16 $R = \emptyset$

17 **for every** node $n \subset candidates$ **do:**

18 $R = [R, R^*(n)]$

19 **end**

20 $ln = n^* \mid R^*(n^*) = min(R)$

21 $P = [P, ln]$

22 $T_{ln} = T_{pred(ln)} + tt_{pred(ln),ln}$

23 $L_{ln} = L_{pred(ln)} - d_{ln}$

24 **end**

25 Calculate optimal distribution of T_n, $n \subset P$, in regard to the costs.

26 $cost = C(P, T_o, T_{suc(o)}, \ldots, T_{pred(s)}, T_s)$

27 startover = 1

28 **while** *startover = 1*

29 *startover = 0*

30 **for every** node $n \subset P \backslash [o, pred(s), s]$ **do:**

31 Define *candidates* to the collection of nodes that can replace *n* in path *P* while keeping *P* feasible.

32 Define *candidates* as the nodes, $cn \subset candidates$, for which $tc_{cn} \leq tc_n$

33 Define *NC* as the collection of nc_{cn}, with $cn \subset candidates$

34 $n^* = cn \mid nc_{cn} = min(NC)$

35 $P^* = P$, *with* $n = n^*$

36 Calculate optimal distribution of T^*_i, $i \subset P^*$, in regard to the costs.

37 $cost^* = C(P^*, T^*_o, T^*_{suc(o)}, \ldots, T^*_{pred(s)}, T^*_s)$

38 **if** $cost^* < cost$ **do:**

39 $P = P^*$

40 $T = T^*$

41 $cost = cost^*$

42 $L_s = L_s + d_n - d_{n^*}$

43 *startover = 1*

44 **end**

45 **for every** node *n2* ⊂ *P*\[o, suc(o), s]* **do:**

46 Define *candidates2* to the collection of nodes that can replace *n2* in path *P** while keeping *P** feasible.

47 Define *candidates2* as the nodes c*n2*, for which $tc_{cn2} \leq tc_{n2}$, with *cn2* ⊂ *candidates2*

48 Define *NC2* as the collection of nc_{cn2}, with *cn2* ⊂ *candidates2*

49 *n2* = cn2 | nc_{cn2} = min(NC2)*

50 *P* = P, with n2 = n2**

51 Calculate optimal distribution of T^*_i, *i* ⊂ *P**, in regard to the costs.

52 *cost* = C(P*, T^*_o, $T^*_{suc(o)}$, …, $T^*_{pred(s)}$, T^*_s)*

53 **if** *cost** < *cost* **do:**

54 *P = P**

55 *T = T**

56 *cost = cost**

57 $L_s = L_s + d_n + d_{n2} - d_{n*} - d_{n2*}$

58 **if** *n* ≠ *pred(pred(s))* **&** *n2* ≠ *pred(s)*

59 *startover = 1*

60 **BREAK**

61 **end**

62 **end**

63 **end**

64 **if** *startover = 1*

65 **BREAK**

66 **end**

67 **end**

68 **end**

69 **return** [*P, T, cost,* L_s]

4 Numerical results

Construction of the data sets

In order to measure the algorithm's performance we need a series of data sets. We received 20 random datasets from Remy Spliet. These consist of ten small sets and ten large sets. All datasets represent a network that is the result of column generation. This means that there is a set of nodes, N^{first}, that will be copied and added as many times as there are nodes in the set. The set does not include the source and sink nodes. The arcs of the nodes that start and end within the set are modified to start in the set, but to end in the next copy of the set. The nodes in the last copy will only have arcs that lead to the sink. Table I shows the characteristics of these data sets.

Table I. Characteristics of the random test instances

Amount of nodes	nc-lower limit	nc-upper limit	tc-lower limit	tc-upper limit	demand-lower limit	demand-upper limit	ac-lower limit	ac-upper limit	tt-lower limit	tt-upper limit
227	-5.2359	0	0	0	1	9	0.166767	4.81455	0.166767	4.81455
227	-5.03124	2.69639	-0.2162	0.169007	1	9	0.166767	4.81455	0.166767	4.81455
227	-3.9979	0	-0.00572	0	1	9	0.166767	4.81455	0.166767	4.81455
227	-3.83406	0	-3.65E-05	8.74E-05	4	8	0.166767	4.81455	0.166767	4.81455
227	-4.54184	1.67921	-0.2283	0.296447	4	8	0.166767	4.81455	0.166767	4.81455
227	-4.6093	0	0	0	1	7	0.299435	3.80277	0.299435	3.80277
227	-3.5022	3.60442	-0.33973	0.146784	1	7	0.299435	3.80277	0.299435	3.80277
227	-2.2552	0	-0.00428	0.0106	1	7	0.299435	3.80277	0.299435	3.80277
227	-2.1002	0	-6.86E-17	3.82E-17	1	7	0.299435	3.80277	0.299435	3.80277
227	-2.84315	0	-0.08496	0.075902	4	8	0.299435	3.80277	0.299435	3.80277
2502	-6.13782	0	0	0	2	9	0.067587	5.86956	0.067587	5.86956
2502	-5.724	7.59662	-0.61536	0.267738	2	9	0.067587	5.86956	0.067587	5.86956
2502	-5.86218	0.597143	-0.16316	0.346709	2	9	0.067587	5.86956	0.067587	5.86956
2502	-3.12825	0.64168	-0.12762	0.116518	2	9	0.067587	5.86956	0.067587	5.86956
2502	-4.93843	1.16779	-0.20789	0.384952	2	10	0.067587	5.86956	0.067587	5.86956
2502	-6.28702	0	0	0	2	9	0.050918	5.84449	0.050918	5.84449
2502	-6.43841	2.2458	-0.38315	0.428071	2	9	0.050918	5.84449	0.050918	5.84449
2502	-3.59207	5.24855	-0.51796	0.170427	2	9	0.050918	5.84449	0.050918	5.84449
2502	-2.70108	1.82481	-0.25572	0.067905	2	9	0.050918	5.84449	0.050918	5.84449
2502	-7.49692	3.68216	-0.5022	0.410013	0	8	0.050918	5.84449	0.050918	5.84449

In addition, we have created our own data sets. Our datasets can be separated into five scenarios. For each of these scenarios we created three small and three large data sets. This adds up to a total of 30 self-created datasets, with a grand total of 50 data sets.

The first scenario is the basic scenario; the others are deviations of this first scenario. In this scenario the lower and upper time window limits of the source and the sink nodes are set to 5 and 25 respectively. The demand, node costs and time costs of the source and sink are set to zero.

The lower time window limits of the first set of nodes, N^{first}, are generated by drawing a random number of the uniform distribution, between 5 and 10 for each node and then rounding it. For the upper window limit the range is changed to 25 to 30. The limits do not have to be integers, but this way it becomes easier to check whether the algorithm is working correctly. In the random test instances, these parameters are not integers.

$$a_i = round(\ unifrnd(\ 5,\ 10)\ ),\ \ i \in N^{first}$$
$$b_i = round(\ unifrnd(\ 25,\ 30)\ ),\ \ i \in N^{first}$$

The node costs are uniformly distributed between -5 and 5.

$$nc_i = unifrnd(\ -5,\ 5),\ \ i \in N^{first}$$

The time costs are calculated in a similar manner as the node costs. Instead of -5 and 5, we now let it range from -1/3 to 1/3. The 1/3 is chosen so that the average service time, *15 = mean (5, 25),* multiplied with a 1/3 results in 5 and thereby have an average impact that is roughly the same as the impact of the node cost.

$$tc_i = unifrnd(\ -1/3,\ 1/3),\ \ i \in N^{first}$$

The demand of a node is gotten by rounding up a uniform random number between 0 and 10.

$$d_i = ceil(\ unifrnd(\ 0,\ 10)\ ),\ \ i \in N^{first}$$

Copying and adding this set of node as many times as there are nodes in the set gives us a new set. Adding the source and sink to this new set results in the final set of nodes.

For the arcs, we start with the arcs that start in the source node. These arcs end in the nodes of the initial node set, N^{first}. For the costs of these arcs we draw a random number from the uniform distribution between -3 and 3.

$$ac_{s,i} = unifrnd(-3, 3), \ s \in [o, d], \ i \in N^{first}$$

The travel time of these arcs is calculated rounding up a uniform random number between 0 and 3. Again, it is not required that these values are integers, but it makes it easier to check the outcome of the algorithm. In the random test instances, these parameters are not integers.

$$tt_s = ceil(unifrnd(0, 3)), \ s \in [o, d]$$

The other arc costs and travel times are calculated in the same way. Those arcs run from a node to every node of the next copy of N^{first}, except the copy of themselves. The arcs are copied to the node's copies; that is that an arc leaving node *k* and ending in node *l* will be copied and will start in the copy of *k* and run to the copy of *l*. This holds for every set of copies, except the last one. For the last set there exist only the arcs leading to the sink.

Using these steps we have generated 3small and 3 large data sets. The small ones have a total of 227 nodes; 25 initial nodes that result in a total of 225 when added to the copies and the last 2 are the source and sink. The large sets have 2502 nodes, 50 initial nodes, 2500 with copies, and 2 for the source and sink nodes. Every instance has a starting load of 30.

The second scenario is identical to the first scenario with the exception that the arc and node costs are nonnegative. The upper limits on these parameters are still the same, but the lower limits have been set to zero. This will allow us to see if our algorithm can effectively keep the costs low, when only the time costs can reduce the costs once they have been incurred.

The third scenario deviates from the first scenario in that it only allows nonnegative time costs. This time it is the other way around. The upper limit of the time costs has not changed. We expect that the service is always started upon arrival in a node, as there are no benefits to be reaped in postponing it.

In the fourth scenario, the time costs are set to zero for all nodes. This way the local search will not exclude nodes based on higher time costs. This will show whether it was a good choice to exclude nodes with higher time costs or not.

In the fifth scenario the arc and node costs are zero. In this scenario

$$R^*(K,i) = 0 + 0 + 0 - 0(PLT_{K,i} + PLL_{K,i})$$

So the initial path will just chose the first node it can find until it can't go on anymore and is forced to go to the sink. The path that is generated only depends on the way the node set has

been arranged. This will show how much the local search will improve when only time costs matter.

The data sets were created in Matlab, while using seed-values to allow for a reproduction of our datasets. The seed-value started at zero for the first data set of a scenario and increased by one for every data set of that scenario created after the first one.

Results

To evaluate our algorithm, we solved all 50 test instances with our algorithm. For each of these test instances we have the optimal costs. The results of the random test cases can be found in Table II and the results of the generated test cases are in Table IIII.

The first column of the tables specifies what test case was used. The second indicates what scenario it was from. The third shows the amount of nodes in the test case. The fourth column shows the amount of nodes in the optimal path, while the fifth column shows the amount of nodes in the path the algorithm found. Column six contains the optimal costs and columns seven and eight indicate the costs of our algorithm before and after applying local search. The last two columns show the gap the costs have with the optimal costs, *gap*, and the improvement of the local search, *impLS*, in percentages. Their definitions are as follows:

$$gap = | Costs - Opt.Costs | / | Opt.Costs |$$
$$impLS = | Costs\ before\ LS - Costs | / | Costs |$$

Table II. Results of the random test instances

Test Instance	Scenario	No. nodes	Opt.Path Length	Alg. Path length	Opt. Costs	Costs before Local Search	Costs after Local Search	Gap with Opt. Cost	Improv. Of Local Search
1	Random	227	8	8	-18,4452	-18,4452	-18,4452	0,00%	0,00%
2	Random	227	6	6	-6,75655	-3,5803	-3,5803	47,01%	0,00%
3	Random	227	6	3	-0,184233	0,8839	0,8839	579,78%	0,00%
4	Random	227	7	3	-0,000903669	1,1052	1,1052	122395,97%	0,00%
5	Random	227	6	6	-8,80618	-0,0079	-0,0079	99,91%	0,00%
6	Random	227	15	11	-22,5068	-20,6595	-20,6595	8,21%	0,00%
7	Random	227	8	8	-7,9424	-2,3656	-2,3656	70,22%	0,00%
8	Random	227	11	3	-0,214256	0,0292	0,0292	113,63%	0,00%
9	Random	227	8	3	-0,010142	0,0313	0,0313	408,90%	0,00%
10	Random	227	8	7	-1,02066	-0,1445	-0,1445	85,84%	0,00%
11	Random	2502	10	9	-33,4104	-31,1715	-31,1715	6,70%	0,00%
12	Random	2502	11	8	-28,456	-18,4341	-18,4341	35,22%	0,00%
13	Random	2502	7	7	-7,71646	-7,4624	-7,4624	3,29%	0,00%
14	Random	2502	9	3	-3,59939	0,3230	0,3230	108,97%	0,00%
15	Random	2502	8	8	-6,46699	-5,2055	-5,2055	19,51%	0,00%
16	Random	2502	8	8	-30,9816	-30,9816	-30,9816	0,00%	0,00%
17	Random	2502	7	7	-17,9964	-7,3444	-11,5555	35,79%	36,44%
18	Random	2502	8	10	-6,83081	-5,5209	-5,5209	19,18%	0,00%
19	Random	2502	10	7	-2,55573	-1,1265	-1,1265	55,92%	0,00%
20	Random	2502	8	8	-11,0712	-2,6467	-6,0182	45,64%	56,02%

In Table II we see that 15 cases resulted in negative cost values for our algorithm, whereas all 20 of them had a negative optimal cost value. Four of those positive values were in cases where the optimal value itself was relatively close to zero; instance 14 is the case where this does not hold. Incidentally, this is also the second-most deviating case, when comparing path length, indicating that the algorithm failed to estimate the future benefits of the nodes correctly in this case.

In two cases the algorithm found the optimal path and in three others it came very close. The cases in which it did find the optimal value, 1 and 16, were not limited to the shorter paths or the smaller cases. However, it did find the optimal values before the local search. Both scenario 1 and 16 have non-positive node costs and time costs of zero for all nodes. The arc costs range and demand are also similar. The same characteristics apply to instance 6 and 11. For those instances we see that the gap is below 10%. What is surprising however is that the gap of instance 13 is below 5% even, but this instance is not that comparable as the others, to instances 1 and 16. Test instance 13 can actually be compared with instance 14 and instance 14 has a gap of almost 109%. This could again be caused by the path length; whereas instance 13 has a path of the same length as the optimal path.

The average gap of these instances is 6206.98%, but this is mostly due to instance 4 with its gap of 122395.97%. Ignoring this gap gives an average gap of 91.78%. This is still greatly

impacted by test instance 3 and 9 with their gaps of 579.78% and 408.90% respectively. When these are ignored the average reaches the much improved value of 44.41%. All these instances that featured such extreme deviations in their gap have one thing in common; their optimal costs are very close to zero. In the calculation of the gap, when dividing by the optimal costs, this will produce very high values.

Examining the effects of the local search, shows that there are only two cases in which the local search improved the answer; case 17 and 20. Both of them are in the large data sets. The improvements themselves provide a 36.44% and56.02% improvement. Further examination of the effects of the local search will be done after analysing Table IIII.

Table IIII. Results of the generated test instances

Test Instance	Scenario	No. nodes	Opt.Path Length	Alg. Path length	Opt. Costs	Costs before Local Search	Costs after Local Search	Gap with Opt. Cost	Improv. Of Local Search
21	1	227	13	9	-51,8763	-32,3477	-32,8917	36,60%	1,65%
22	1	227	11	6	-41,9892	-13,7763	-17,0882	59,30%	19,38%
23	1	227	11	7	-36,0471	-19,5891	-19,5891	45,66%	0,00%
24	1	2502	19	10	-86,7974	-44,5470	-51,1593	41,06%	12,92%
25	1	2502	16	8	-57,1974	-30,7166	-30,7166	46,30%	0,00%
26	1	2502	21	12	-84,6935	-59,8524	-59,8524	29,33%	0,00%
27	2	227	4	4	0,172138	2,4265	2,4265	1309,63%	0,00%
28	2	227	4	3	1,41338	1,7698	1,7698	25,22%	0,00%
29	2	227	4	4	0,771179	4,8022	4,8022	522,71%	0,00%
30	2	2502	11	8	-7,4342	6,0363	4,8407	165,11%	24,70%
31	2	2502	9	3	-5,22301	1,8027	1,8027	134,51%	0,00%
32	2	2502	7	5	-4,76133	2,9480	2,9480	161,92%	0,00%
33	3	227	10	9	-33,0701	-27,8516	-27,8516	15,78%	0,00%
34	3	227	11	9	-20,6988	-17,5334	-17,5334	15,29%	0,00%
35	3	227	7	7	-20,4705	-17,2302	-17,2302	15,83%	0,00%
36	3	2502	17	13	-61,1194	-49,2449	-52,3627	14,33%	5,95%
37	3	2502	18	11	-64,4645	-50,2386	-50,2386	22,07%	0,00%
38	3	2502	16	12	-81,9946	-60,0580	-60,3009	26,46%	0,40%
39	4	227	12	9	-41,3048	-27,0927	-27,0927	34,41%	0,00%
40	4	227	13	9	-44,049	-28,7908	-28,7908	34,64%	0,00%
41	4	227	16	16	-62,1982	-62,1982	-62,1982	0,00%	0,00%
42	4	2502	16	11	-73,7487	-52,7560	-52,7560	28,47%	0,00%
43	4	2502	13	11	-63,2048	-53,1766	-53,1766	15,87%	0,00%
44	4	2502	18	10	-81,238	-52,3492	-52,3492	35,56%	0,00%
45	5	227	10	9	-15,8235	-1,4521	-7,7334	51,13%	81,22%
46	5	227	12	7	-15,3293	1,3501	-3,3694	78,02%	140,07%
47	5	227	15	9	-17,9877	-8,1323	-8,1960	54,44%	0,78%
48	5	2502	19	10	-35,1076	-4,6745	-14,9542	57,40%	68,74%
49	5	2502	18	9	-30,7597	-0,0307	-8,2209	73,27%	99,63%
50	5	2502	21	7	-40,4793	6,2039	-5,4402	86,56%	214,04%

In the generated test cases, there were 24 negative costs values where there were 27 cases with negative optimal costs. All the positive values can be traced to scenario 2. Scenario 2 is

the scenario where the arc and node costs are nonnegative. It is the only scenario that features a gap of over 100% and it even has a gap of over 1000%. The average gap of Scenario 2 is 386.52% and is thereby the scenario with the worst performance. The average gap is influenced mostly by instance 27 and 29; these instances also featured optimal costs relatively close to zero.

Scenario 1 performs has an average gap of 43.04%. This was the basic scenario where all costs could be positive or negative. Note that this percentage is comparable to the earlier found average gap of the random test cases when ignoring the three most deviating cases. That average resulted in 44.41%, setting them only 1.37 percentage points apart.

Scenario 3 has the smallest gaps, with an average gap of 18.29%. Scenario 3 featured nonnegative time costs. This would indicate that our algorithm does best when the time costs are nonnegative and the arc and node costs can be negative.

Scenario 4 is not far behind with its average gap of only 24.82%. In scenario 4 the time costs were set to zero. This scenario is the only scenario that features a test instance where the optimal path was found, that is test instance 41.

Scenario 5 does not do so well as 1, 3 or 4 with its average gap of 66.80%. Scenario 5 featured arc and node costs of only zeros. This is in accordance with our earlier indication that the algorithm does better when the arc and node costs can be negative.

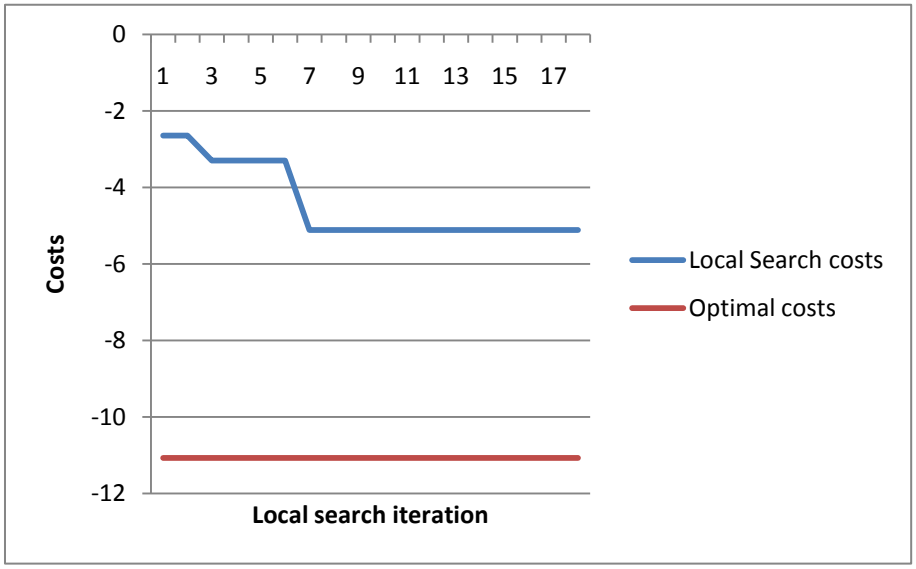
This time our algorithm only found the optimal value once; in instance 41. This is in a small data set and again we see that it was found before the local search. This only reinforces our earlier suspicion, that it might have been easier than usual to find the optimal path. However, we know that our local search does not change the amount of nodes in our path. If it were to find the optimal value, then the amount of nodes in the path before local search should already be of the same length as the optimal path. In Table II, eight paths shared their length with the optimal path. In Table IIII, we find four of those paths. Two out of those eight were optimal, 25%, and one of the four was optimal, 25%. It would be presumptuous of us to say that in 25% of the cases where the length of the path matches that of the optimal path, the optimal path is found. However, it would be interesting to generate an initial path of the same length as the optimal path and then look at how the local search performs.

In Table IIII we have 10 instances where the local search improves on the initial path. This is significantly more than we found in the random test cases. These cases are 21, 22, 24, 30, 34, 45, 46, 48, 49 and 50. The first three of these cases are in scenario 1 and the last four are in scenario 5. There are none in scenario 4 as could be expected, since we focused our

local search to exclude nodes with higher time costs, but in scenario 4 all the time costs are the same, zero. In scenario 5 time costs were the only costs as the rest was set to zero, therefore it is as expected that scenario 5 shows the most improvements through local search.

Now we will take a look at how the local search works when improving a path in the different scenarios.

Figure I. Test Instance 20; Random scenario; large data set



In Figure I we see that only iteration 3 and 7 improve the path and that all iterations after that have no effect. We can clearly see that this implementation of local search only keeps a changed path if the costs are lower, whereas some implementations sometimes choose to disrupt their solution in the hopes of opening new and better opportunities. The improvement in the costs is, as stated earlier, 56.02%. All the improving happens in the first half of the local search.

Figure II. Test Instance 24; Scenario 1; large data set

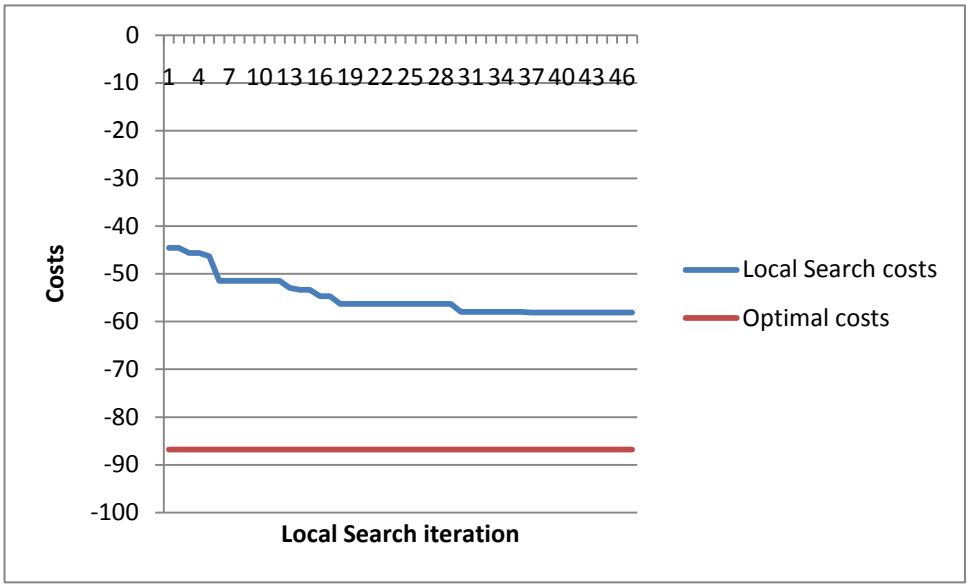
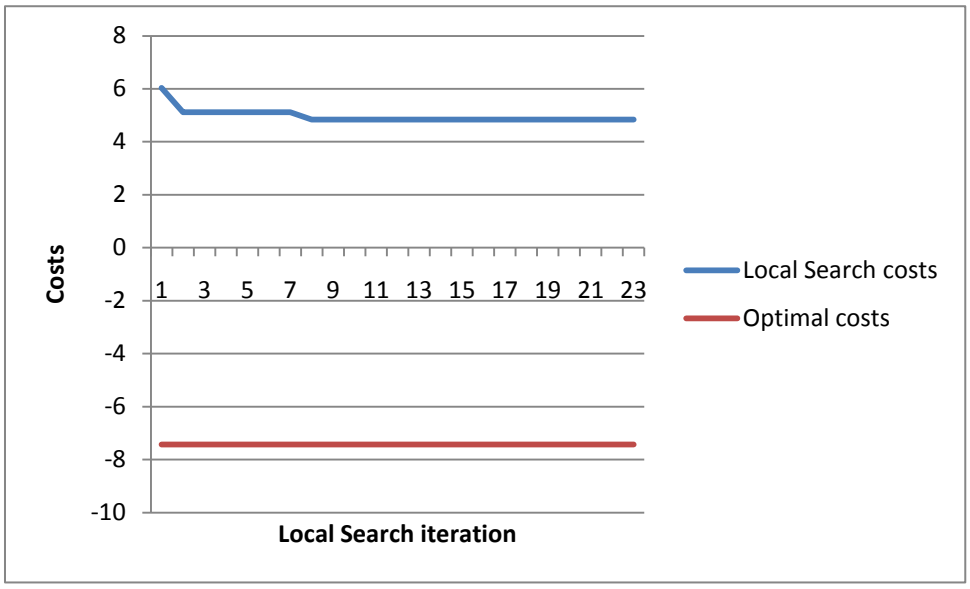


Figure II shows a local search process where there are a lot more updates to the path. This test instance is from scenario 1, where all the costs could be both negative and positive. The path is only 10 nodes long, whereas the path of figure I was 8 nodes long. Still we can see that there are a lot more iterations going on here. The amount of iterations our local search goes through does not only depend on the amount of nodes in the path. There are quite a few causes for the local search to not find a feasible replacement candidate. We only count the iterations in which such a candidate has been found. That is why the amount of iterations is not solely dependent on the length of the initial path. This process caused a 12.92% improvement in the costs. Like the last figure, the majority of the improvement is achieved in the first half.

Figure III Test Instance 30; Scenario 2; large data set



In Figure III we see the local search improve a path from scenario 2, where the arc and node costs are nonnegative. The improvement in costs is 24.70%. This is also the only time in scenario 2 that the local search found a better path, therefore it will come as no surprise that the amount of cost updates are small.

Figure IV Test instance 36; Scenario 3; large data set

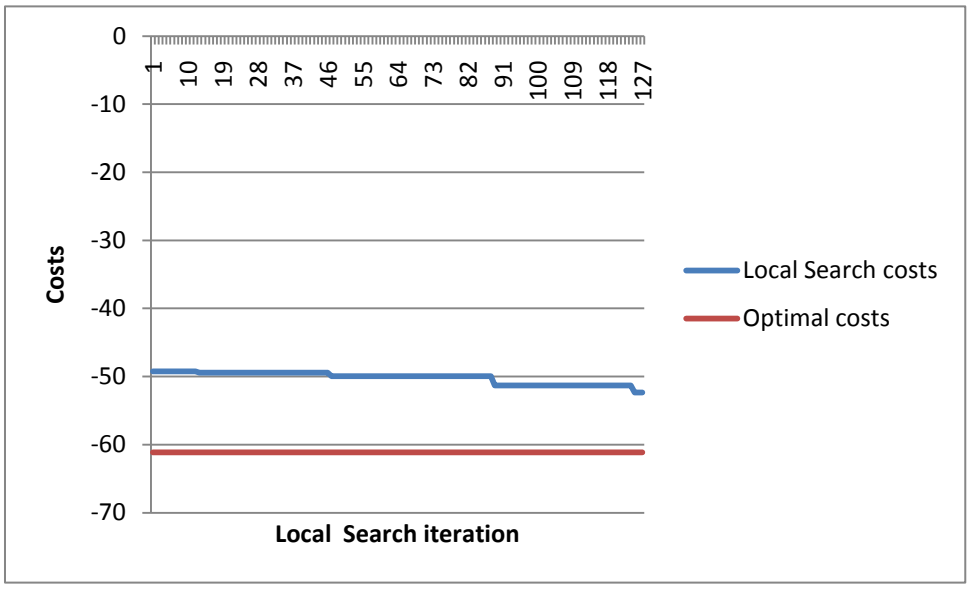
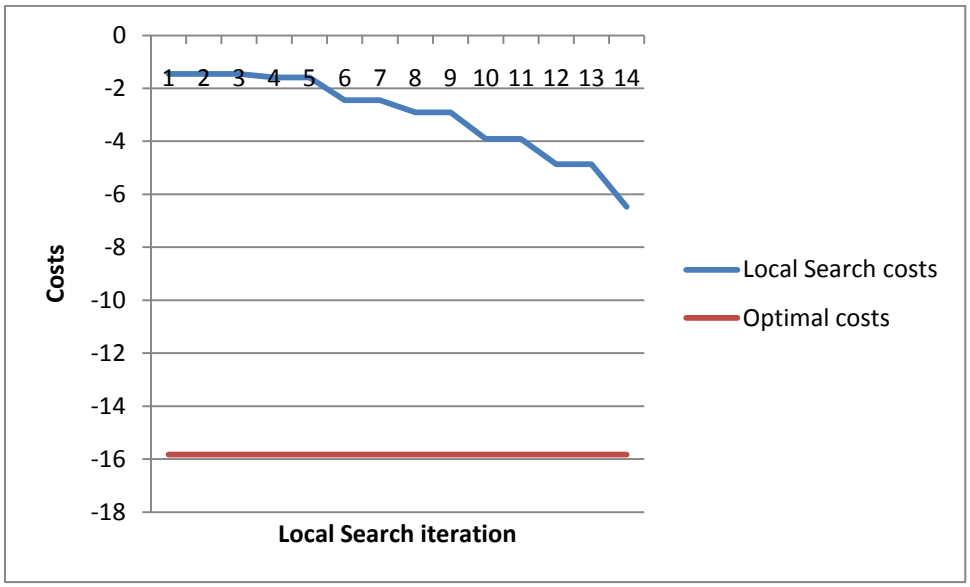


Figure IV shows the improvement of a path of scenario 3. In this scenario all time costs are nonnegative. The improvement is 5.95% and this is achieved in 127 iterations. A clear sign that more iterations, does not mean bigger improvements. Scenario 3 has only 2 instances

where local search finds a better path. The other improves the cost by 0.40%, so again it is no surprise that this one does not update as much as we would like it to.

Figure V. Test Instance 45; Scenario 5; small data set



In Figure V we see the local search process of a test instance from scenario 5; where the time costs are the only costs. Here, we can clearly see that the local search proves to be a valuable asset in finding a better path. It has improved the costs by 81.22%.

This demonstrates that in networks where time costs are a significant portion of the total costs, the local search is very important. Networks where time costs are not so significant, might not improve at all, or could try a modified local search method, which puts less focus on the time costs when selecting nodes as replacements.

Contrary to some of the earlier figures, the majority of the improvement in this figure is achieved in the second half.

Next we will show the average improvement per iteration for the scenarios where the local search improved the costs of at least one of the test instances.

Figure VI. Average relative improvement of scenario 1

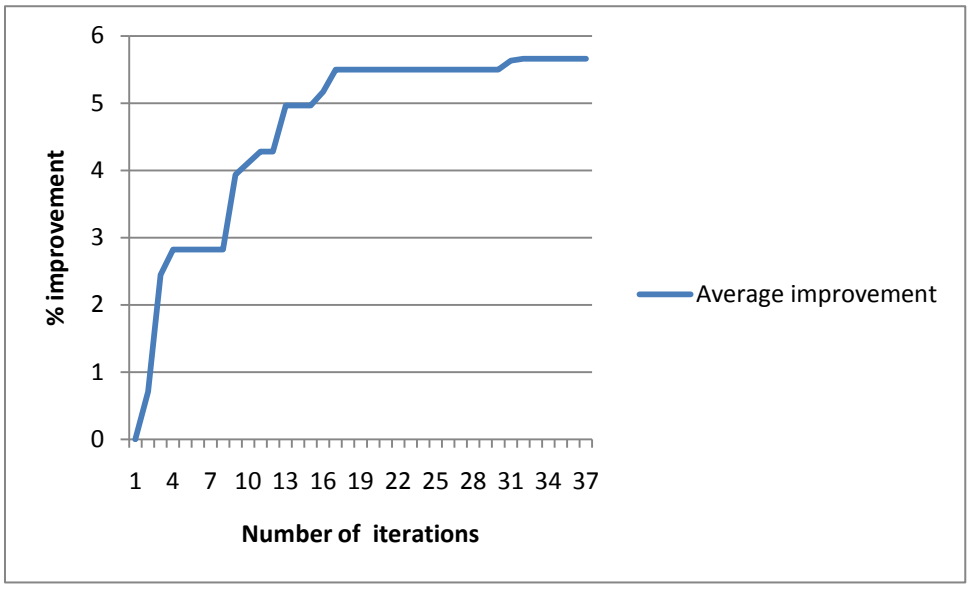
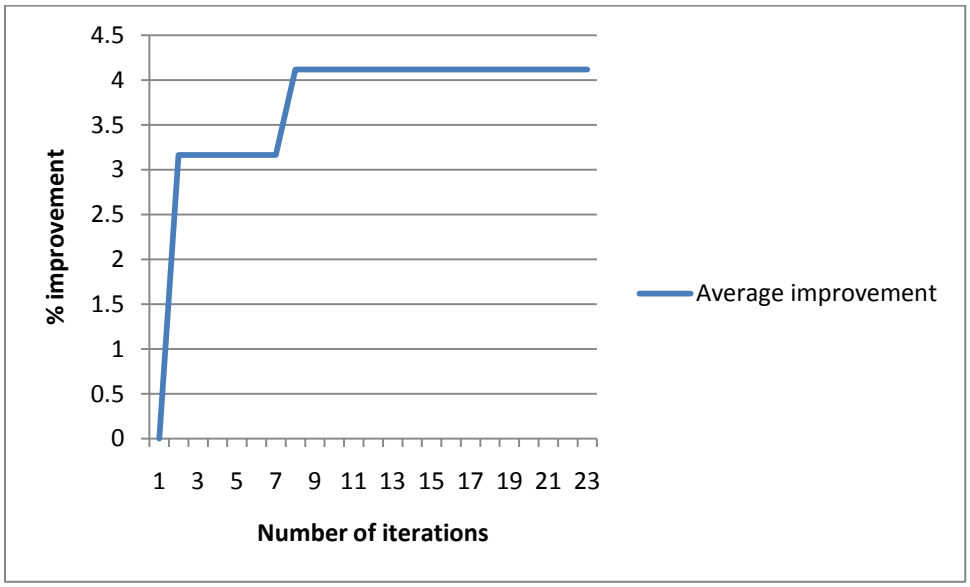


Figure VI indicates that the majority of the improvement in scenario 1 is reached in the first 16 iterations. After 16 iterations the average improvement in percentage is 5.167%.

If one is to work with networks where all the costs can range from negative values to positive values, then it might be wise to limit the amount of iterations performed to around 20.

Figure VII. Average relative improvement of scenario 2



In figure VII we that more than three quarters of the improvement is reached after 2 iterations, while the last jump in improvement is after 8 iterations. The average improvement reached 3.16% in 2 iterations and 4.12% after 8 iterations.

When working in networks with nonnegative arc and node costs one could choose to limit the amount of iterations to around 10, or if solving time needs to be kept to an absolute minimum, then one might consider limiting it to just around 3 iterations.

Figure VIII. Average relative improvement of scenario 3

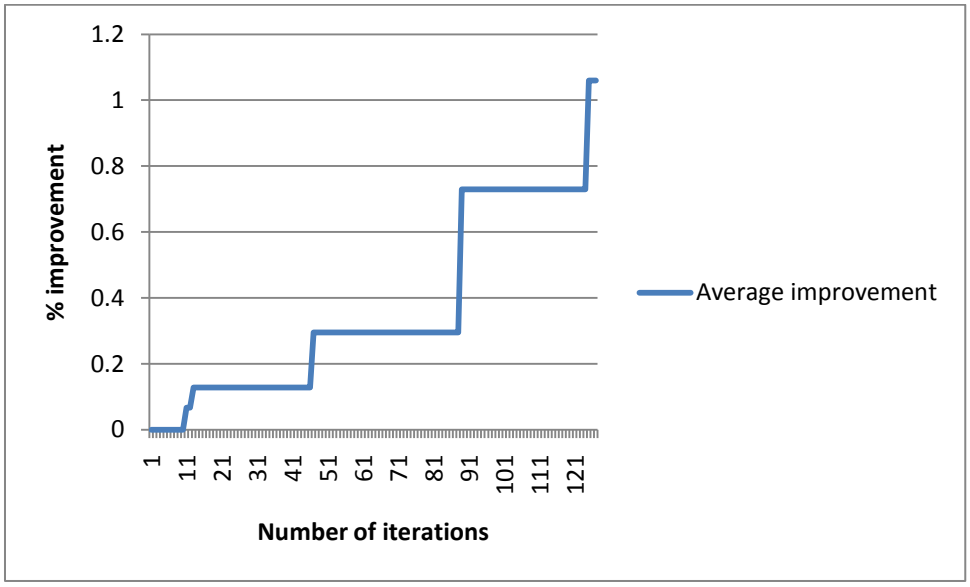
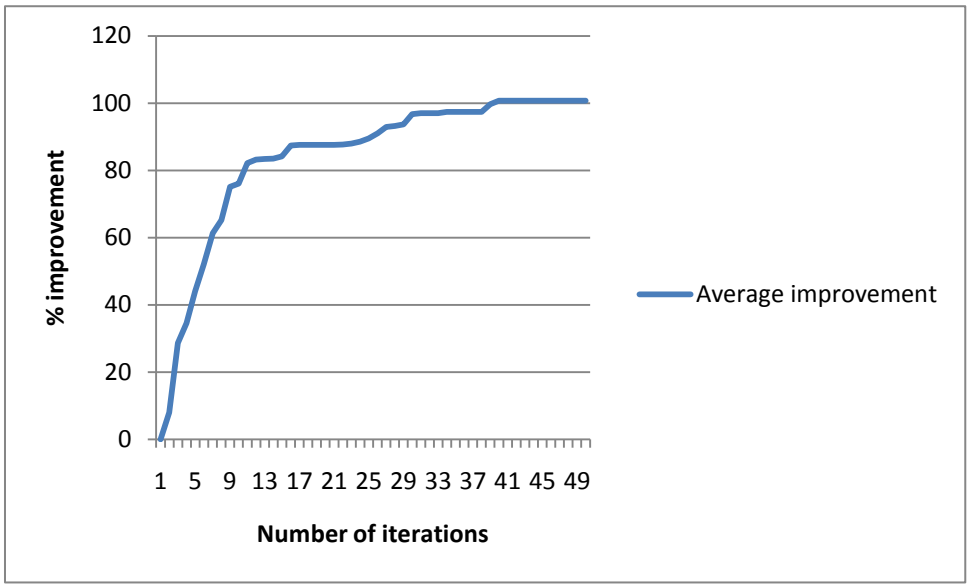


Figure VIII has its average improvement divided more equally over the iterations. However the average improvement does only reach 1.06% in the end. This was in scenario 3, where the time costs were nonnegative.

If that 1.06% is important then completing around 120 iterations is what is needed. However if that 1.06% is too small of an improvement, then one might choose to skip the local search altogether.

Figure IX. Average relative improvement of scenario 5



For scenario 5 we see in figure IX that the majority of the average improvement is reserved to the first 17 iterations. After just 17 iterations it has reached an average improvement of 87.59%. The next 31 iterations cause the average improvement to reach it peak at 100.75% after which it does not improve anymore for the last 2 iterations.

Scenario 5 describes the situation where the arc and node costs are zero. When working with this type of network it proves favourable to let the local search complete at least the first 17 iterations and depending on the severity of the solving time limitations one might choose to perform more iterations of not.

5 Conclusion

In this thesis we set out to find out if a method can be developed that provides near-optimal solutions for the shortest path problem, with capacity constraints, time windows and time costs, in a short amount of time. The algorithm we proposed has shown that this is indeed possible.

Although our algorithm does not perform as well as we desired in every scenario, it has shown where improvements can be made to increase its performance in those scenarios. In scenario 1 we have seen an average gap of 43.04%. The performance of the local search has improved the answers somewhat, but there are still three instances where it did not change anything. The local search is definitely in need of reworking. The criteria on which it selects it replacement nodes could be improved in order to let it better estimate the effects the replacements will have on the costs.

Scenario 2 featured an average gap of 386.52%. This is the scenario where our algorithm performed the worst. Since the arc and node costs are nonnegative in this scenario, it would seem that our algorithm needs to be improved when selecting nodes for its initial path. Especially the estimation of the future benefits a node might bring with, needs to be reconsidered for this kind of situation.

The scenario where it did best was scenario 3. The nonnegative time costs cause the service to be initiated as soon as possible. The average gap was only 18.29%.

Scenario 4, with a gap of 24.82%, was the second best scenario for our algorithm. Here the time costs were set to zero. It seems that our algorithm performs best when the time costs are never negative; they can be positive or zero, but not negative. This would seem weird at first, because negative time costs would only decrease the total costs, but looking closer shows that while that is true, our algorithm fails to reap the benefits thereof and thereby does relatively better when the time costs cannot be negative.

The final scenario, scenario5, had an average gap of 66.80%, making it the second worst scenario, although not nearly as bad as scenario 2. This did come as a bit of a surprise, since both scenarios, 2 and 5, changed the range of the arc and node costs. It appears that while arc and node costs equal to zero are not benefitting the algorithm, having arc and node costs that are nonnegative is even worse.

The local search needs improvement too. Focussing on nodes with a lower time cost seemed a good idea at first and it worked in certain scenarios, but overall we think it is better to have it be more selective of what nodes it chooses. Incorporating the time costs when creating the initial path and then having the local search look at it from a general perspective could also allow the algorithms performance to be more evenly distributed over the different scenarios.

In most cases the biggest improvements to the path were made in the first half of the total amount of iterations. The second half could be removed in order to reach lower solving times. If the total amount of iterations cannot be estimated one could just apply the local search to the first half of the initial path or restrict the amount of iterations to around 20, since that seemed to be the section where most of the improvements were found.

6 Further research

While working on this thesis we have come across a few aspects that would be interesting to investigate further. The first is about the definition of *R*(K,i)*. We mentioned earlier that some variables in its definition were chosen based on the results they produced, but it would be interesting to see how these results are actually produced. The variables in question are $MinO_i$ and *S*. Investigating what definition of *R*(K,i)* is the best may open up new possibilities that result in better initial paths or increase the algorithm's robustness.

The local search section of our algorithm can definitely be improved. We looked at the time costs, but perhaps there are other parameters or combinations of parameters that yield better results. Analysing how each parameter influences the performance of the local search could not only prove valuable for SPPCCTWTC, but for networks in general.

Another field that might be interesting to explore is the relationship between the initial path length and the improvement through local search. Longer paths generally open up more possibilities, but are not necessarily better. Finding the right path length for the initial path could assist the local search in increasing its performance.

An eccentricity worth taking a look at is the large difference in the average gap between when the arc and node costs are zero and when they are nonnegative. It seems odd that the difference in the average gap would be so big. However this could be just a fluke in the results, so perhaps it is wise to create more test instances of these scenarios and see if the discrepancies continue to hold.

We used networks that represented problems to which column generation was applied, but the only two aspects of the network that we actively used was the arc that connected the sink with every other node, with the exception of the source and that there were no cycles. It would be good to know if our algorithm performs differently in random networks of which the only common features are that there are no cycles and that there exists an arc for every node, except the source, that leads directly to the sink.

The last thing I would recommend investigating is how an algorithm would perform if it would create multiple initial paths. Creating an initial path can be done quickly. Having multiple paths, of which each was created using different methods, could create a more robust algorithm. Where one path fails to deal with the crux of the problem, another path might still find a near-optimal solution.

7 References

[1] Spliet, R., Gabor, A., 2011, The Time Window Assignment for Vehicle Routing Problems (working paper)

[2] Ioachim, I., Gelinas, S., Soumis, F., Desrosiers, J., 1997, A Dynamic Programming Algorithm for the Shortest Path Problem with Time Windows and Linear Node Costs, *Networks Vol: 31 Issue: 3*, pp. 193-204

[3] Wolsey, L.A., 1998, Integer Programming, Wiley

[4] Dijkstra, E.W., 1959, A note on two problems in connexion with graphs, *Numerische Mathematik*, pp. 269-271

[5] Kundu, S., 1980, A Dijkstra-like shortest path algorithm for certain cases of negative arc lengths, *BIT 20*, pp. 522-524

[6] Gueguen, C., Dejax, P., Gendreau, M., Feillet, D., 2004, An Exact Algorithm for the Elementary Shortest Path Problem with Resource Constraints: Application to some Vehicle Routing Problems, *Networks Vol: 44 Issue: 3*, pp. 216-229

[7] Fu, L., Sun, D., Rilett, L. R., 2006, Heuristic shortest path algorithms for transportation applications: State of the art, *Computer and Operation Research Vol: 33*, pp. 3324-3343