Efficient Implementations of the Savings Method for the Vehicle Routing Problem with Time Windows*

Cor de Jong[1,2] cjong@ortec.nl
Goos Kant[2,3] goos@cs.ruu.nl
André van Vliet[1,2] avliet@ortec.nl

Abstract

One of the most favorite heuristic algorithms for the Vehicle Routing Problem with Time Windows is the Savings Method of Clarke and Wright [1]. In this paper we consider several data structures for this algorithm. This not only yields new theoretical bounds for the implementations, but also efficient practical implementations. Comparisons between the applications are given as well.

1 Introduction

An important problem in distribution systems is the routing of vehicles to deliver products at customers, denoted as the Vehicle Routing Problem (VRP). Since the late fifties, this problem has been studied extensively by operations researchers. In its original setting, vehicles located at a single depot have to deliver goods to a number of customers on different locations, such that the demands of all customers are satisfied and that the total distances travelled by the vehicles are minimal. It is assumed that trucks have a given capacity and that the number of trucks is sufficiently large.

Intrinsically, the VRP is a spatial problem. During the last decades, however, temporal aspects of routing problems have become increasingly important, as there are many practical situations in which time restrictions arise naturally. Often, customers can only be served during certain hours of the day such as office hours or the hours before the opening of a shop. Furthermore, transportation companies try to compete on service differentiation by delivering within prespecified time windows. Therefore, much attention has been given to the Vehicle Routing Problem with Time Windows (VRPTW). The VRPTW is a generalization of the VRP involving the added complexity that every customer should be served within a given time window. Furthermore, the objective evolves to minimizing a combination of both distances travelled and the total duration of the routes.

Due to the added complexity of time windows, computation times of heuristics for solving routing problems are much greater for the VRPTW than for the VRP. However, it is possible to speed up this heuristics significantly by using efficient implementations. In this paper, we

*This research was (partially) supported by ESPRIT Long Term Research Project 20244 (project ALCOM IT: *Algorithms and Complexity in Information Technology*).

[1]Econometric Institute, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, the Netherlands.

[2]ORTEC Consultants, Groningenweg 6-33, 2803 PV Gouda, the Netherlands.

[3]Dept. of Computer Science, Utrecht University, Padualaan 14, 3584 CH Utrecht, the Netherlands.

will discuss how we can speed up the Savings algorithm when we use it to solve the VRPTW. We will only consider problems with one time window per customer. For an extension to problems with multiple time windows we refer to Part 2 of this thesis.

2 Problem formulation

We will define the VRPTW using the notation of Desrocher et al. [3] for the vehicle routing problem with time windows.

Let $G = (V, A)$ be a graph with a set V of vertices and a set A of arcs. We have $V = \{0\} \cup N$, where 0 corresponds to the depot and $N = \{1, \ldots, n\}$ is the set of customers. For the set of arcs, we have $A = (\{0\} \times N) \cup I \cup (N \times \{0\})$, where $I \subseteq N \times N$ is the set of arcs connecting the customers, $\{0\} \times N$ contains the arcs from the depot to the customers, and $N \times \{0\}$ contains the arcs from the customers to the depot. Every customer $i \in N$ has a positive demand q_i, and a time window $[e_i, l_i]$ in which the service should start. For each arc $(i, j) \in A$ we have a cost c_{ij} and a travel time t_{ij}. For $i \in N$, t_{ij} includes the duration of serving customer i. Furthermore, we assume that the vehicles are identical and have capacity Q, and that the cost of one time unit waiting time is equal to C. For $i \in$N, c_{0i} includes the fixed costs of a vehicle.

We have the following variables. For each customer $i \in N$, S_i is the service time (= start of service), W_i is the waiting time of the vehicle at the customer, and y_i is the load of the vehicle when it arrives at the customer. For each arc $(i, j) \in A$, x_{ij} is equal to 1 if arc (i, j) is used by a vehicle and 0 otherwise.

Now the problem is

$$
\begin{array}{lll}
\text{Minimize} & \sum_{(i,j)\in A} c_{ij} x_{ij} + C \sum_{i \in N} W_i & (1) \\
\text{subject to} & \sum_{j \in V} x_{ij} = 1 \qquad \forall\, i \in N & (2) \\
& \sum_{j \in V} x_{ji} = 1 \qquad \forall\, i \in N & (3) \\
& e_i \leq S_i \leq l_i \qquad \forall\, i \in N & (4) \\
& x_{ij} = 1 \Rightarrow S_i + t_{ij} + W_j = S_j \qquad \forall\, (i,j) \in I & (5) \\
& x_{ij} = 1 \Rightarrow y_i - q_i = y_j \qquad \forall\, (i,j) \in I & (6) \\
& q_i \leq y_i \leq Q \qquad \forall\, i \in N & (7) \\
& x_{ij} \in \{0, 1\} \qquad \forall\, (i,j) \in A & (8) \\
& W_i \geq 0 \qquad \forall\, i \in N & (9)
\end{array}
$$

We minimize the total costs, consisting of the total travel costs and the costs of the waiting time. By the equations (2), (3) and (8), we require that every customer is visited exactly once. Equation (4) ensures that the service of each customer starts within his time window. When a vehicle goes from i to j, equations (5) and (9) enforce that there is enough time to do this, while equations (6) and (7) enforce that the loads of the vehicles when arriving at the customers are feasible.

In principle, this problem formulation can be used to solve the VRPTW to optimality with a mixed integer programming solver. For that purpose the equations (5) and (6) can be linearized to

$$S_i + t_{ij} + W_j - S_j \leq M(1 - x_{ij}) \qquad \forall\, (i,j) \in I \tag{5a}$$
$$S_i + t_{ij} + W_j - S_j \geq M(x_{ij} - 1) \qquad \forall\, (i,j) \in I \tag{5b}$$
$$y_i - q_i - y_j \leq M(1 - x_{ij}) \qquad \forall\, (i,j) \in I \tag{6a}$$
$$y_i - q_i - y_j \geq M(x_{ij} - 1) \qquad \forall\, (i,j) \in I \tag{6b}$$

where M is a sufficiently large constant. As we can change the "="-sign of equation (6) without harm into a "$\geq$"-sign, we can even do without (6a). One should note that solving this model cannot be accomplished, unless the problem instance is very small. The merit of this formulation is merely that it describes the problem in a precise way.

3 The Savings method

A popular algorithm to solve the VRP and the VRPTW is the Savings algorithm. This algorithm was developed by Clarke and Wright [1] based on ideas of Dantzig and Ramser [2], and is often referred to as the Clarke-and-Wright algorithm. Initially, in the Savings algorithm every customer is serviced by a separate vehicle. Subsequently we try to reduce the total transportation costs by repeatedly combining the routes of two vehicles to one route. In every iteration, we perform that feasible combination of routes that leads to the maximal cost reduction. The algorithm stops when there is no longer a feasible combination of routes that leads to a cost reduction.

We can describe this algorithm with the following pseudo-code.

```
Procedure Savings
begin
    InitializeRoutes
    repeat
        ComputeBestSaving
        if (best saving > 0)
        begin
            CombineRoutes
        end
    until (best saving ≤ 0)
end
```

First, we initialize the Savings algorithm by defining one route for each customer, which costs $O(n)$, where n is the total number of customers. When searching for the best combination of routes (ComputeBestSaving), we have to evaluate $O(n^2)$ combinations of routes. For every combination of routes we have to test feasibility and, when the combination is feasible, profitability. For the VRP we can test feasibility by checking that the combined loads of the two routes do not exceed the vehicle capacity and that the total duration does not exceed the total available time of the vehicle. When we know the load and the duration of every route, we can determine in constant time the load and the duration of the combination of two routes. So, we can test feasibility for the VRP in constant time.

When we use the Savings method to solve the VRPTW, we have the added complexity of time windows. Feasibility with respect to capacity can still be checked in constant time, but furthermore, we also have to check whether the service of every customer can start within

his time window. We can do this by determining the service times for the combination of the routes i and j, using a forward and a backward pass through the route. When k_i is the number of customers in route i, evaluating the combination of the routes i and j, with a forward and a backward pass has a time complexity of $O(k_i + k_j)$. As the maximal number of customers in a route is equal to n, this time complexity is $O(n)$.

When a combination of the routes i and j is feasible, we have to compute its profitability. The resulting cost reduction or saving is given by

$$sav_{ij} = c_{i0} + c_{0j} - c_{ij} - C\Delta W,$$

where c_{i0} is the cost to go from the last customer of route i to the depot, c_{0j} is the cost to go from the depot to the first customer of route j and c_{ij} is the cost to go from the last customer of route i to the first customer of route j. The increase of waiting time, caused by the combination of the two routes, is denoted by ΔW, and C is the cost of one time unit. This cost reduction can be computed in constant time for the VRP, as waiting times do not exist in this problem. For the VRPTW, however, ΔW can be greater than 0. To compute the profitability of the combination of two routes, we have to determine the minimal increase of waiting time. As we have already determined feasibility of the combination by a forward and a backward pass through the route, we have also determined the minimal duration of the route and thus the minimal increase of waiting time. So, we see that we can test the feasibility and the profitability of the combination of two routes for the VRP in $O(1)$ time and for the VRPTW in $O(n)$ time. The resulting time complexity of ComputeBestSaving is $O(n^2)$ for the VRP and $O(n^3)$ for the VRPTW. To combine two routes to one route, we only need constant time. As there are at most $n-1$ iterations of the Savings algorithm necessary, the total time complexity is $O(n^3)$ for the VRP and $O(n^4)$ for the VRPTW. In the sequel of this article we only discuss upon the VRPTW.

There are a number of possibilities to speed up the Savings algorithm. In the first place it is possible to compute the resulting saving of a combination of two routes in considerably less time than described above. Further, we will show that it is not necessary to evaluate every possible combination of routes in every iteration of the algorithm, when we store the appropriate information.

4 Efficient computation of savings

When we determine the feasibility and profitability of a combination in an efficient manner, we use the convention that we execute every route as early as possible to get a minimal route duration. Note that this convention leads to the earliest possible service time for the last customer of every route. For every customer, we compute the difference between the service time and the end of the time window of the customer. Now we can define the maximal push forward for every route i, denoted by PF_i, as the minimum of these differences over all its customers. This means that the whole route i can be shifted forward in time by at most PF_i time units.

When we want to check whether the combination of the routes i and j (j after i) is feasible, we first determine the earliest possible arrival time at the first customer of route j. This arrival time is the sum of the service time of the last customer of route i and the travel time between this customer and the first customer of route j. Note that the last customer of

route i was serviced as early as possible. If this arrival time is not later than the service time of the first customer of route j plus PF_j, then this combination is feasible.

When the combination of two routes is feasible, we have to compute the resulting cost reduction. When there is no waiting time between the last customer of route i and the first customer of route j, this cost reduction is equal to $c_{i0} + c_{0j} - c_{ij}$. If waiting time occurs, we shift the customers of route i forward by the minimum of PF_i and this waiting time. The remaining waiting time, if any, multiplied by the scaling constant C, should be subtracted from the cost reduction. In both cases the cost reduction can thus be computed in constant time.

To be able to determine feasibility and profitability in this efficient manner, we have to compute the push forward of every route. Initially, we have n routes of one customer each, so we can determine the PF_i's in constant time. This leads to a time complexity of $O(n)$ for the n routes together. In every iteration, when we have performed the best combination of two routes, we have to compute the push forward of one new route. As the maximal number of customers in a route is n, this can be done in constant time per iteration. So, altogether the global variables PF_i can be maintained in $O(n)$.

This way of computing the savings, reduces the time complexity of the Savings Method for the VRPTW from $O(n^4)$ to $O(n^3)$.

5 Selection of the best saving

In the previous section we have shown how we can compute the saving of merging two routes i and j in $O(1)$, by maintaining an appropriate set of global variables. These global variables can be initialized in $O(n)$ time and in every iteration of the Savings method they can be updated in $O(1)$. As the Savings method has at most $n-1$ iterations, these variables can be maintained in $O(n)$ in total.

A straightforward implementation of the Savings method determines the maximal saving in every iteration by just computing the saving for every combination of two routes that are left. As the number of savings that need to be computed is $O(n^2)$ in every iteration, this gives a time complexity of $O(n^3)$. In the following subsections, we discuss some implementations that perform much better than the straightforward implementation. The computational results presented in these subsections, are computed with test problems as described in Section 6.

5.1 Matrix implementation

As in every iteration two routes are deleted and one new route is created, most of the computations of savings are the same in subsequent iterations. One way to circumvent the double work, is to store the savings in a matrix and only update the rows and columns that correspond to the deleted routes and the new route in every iteration. A matrix of dimension $n \times n$ suffices to store these savings, as we give a new route the lowest index of the two routes of which it is composed. Initializing this matrix can be done in $O(n^2)$, and updating the matrix costs only $O(n)$ per iteration and thus $O(n^2)$ in total. Unfortunately, selecting the maximal saving still costs $O(n^2)$ per iteration. Therefore, we still have a time complexity of $O(n^3)$ for the Savings method as a whole.

5.2 Heap implementation

A way to reduce the time in the selection step, is to store the savings in a heap. Using this data structure, the maximum can be found in constant time, and deletions and insertions can be carried out in logarithmic time. Instead of storing just the values of the savings, we now have to store records that contain the value and the two indices of the routes. It suffices to store only the highest saving of the combinations (i, j) and (j, i), and therefore on initialization the heap contains $n(n-1)/2$ records.

If we merge two routes i and j, then in principle all savings of combinations of both i and j with another route should be deleted from the heap. In order to avoid the need of additional variables that indicate where records are located, we propose to leave these records in the heap. When we merge two routes i and j in iteration k, the new route gets the index $n + k$ and we set a flag on the routes i and j that they are not valid anymore. Sooner or later, records with index i or j become the maximal element of the heap and can be deleted without further consideration. Let us now see how large the heap can become. In iteration k, at least one record is deleted and at most $n - k - 1$ records are inserted. Summing over all iterations, this gives that the heap can grow with at most $(n-2)(n-3)/2$ records. So, the size of the heap can never exceed $n^2 - 3n + 3$. The insert and delete operations can be carried out in $O(\log n)$ time each, so the total time spent on them is $O(n^2 \log n)$.

Both the matrix implementation and the heap implementation have the drawback that they use $O(n^2)$ memory. We will now see how we can adjust these implementations so as to reduce the memory requirements, while we still obtain a reasonable time complexity.

5.3 Partial heap implementation

The idea to reduce the memory requirements of the heap implementation is to define a partial heap with a number of cells that is considerably less than n^2. When we create the heap, we place all savings in the heap, as long as there is space available in the heap. When the heap is completely filled, we compare every following saving with the smallest element of the heap. When the new element is greater than the smallest element in the heap, we replace the smallest element of the heap by the new element and restore the heap property. We remember the maximal saving of a combination that could not be placed in the heap. To be able to find the minimal element of the heap in constant time, we place this minimal value in the root instead of the maximal value. Due to the restricted size of the heap, we have to delete all records corresponding to routes that are no longer valid from the heap, in order to avoid that these records completely fill the heap.

When h is the size of the heap, creating the heap can be done in $O(n^2 \log h)$. The maximal element of the heap will be located in the lower half of the heap. So, determining the maximal saving takes $O(h)$ time, which is $O(nh)$ for all iterations together.

When we have combined two routes, we have to update the heap by deleting all combinations consisting of one of these routes. This part of the update costs $O(h)$ time. Next we restore the heap property in $O(h)$ time. Subsequently, we compute the savings of the combinations with the new route, which are added to the heap when the computed saving is greater than or equal to the greatest value that is not placed. This part of the update costs $O(n \log h)$ time. So, the total time we need for an update of the heap is $O(h + n \log h)$ and the time needed for all updates is $O(nh + n^2 \log h)$. Due to the restriction that we only place elements in the heap that are greater than or equal to the greatest value that is not placed in

	size = n		size = $2n$		size = $5n$	
n	cpu	m	cpu	m	cpu	m
100	0.442	5.1	0.417	4.0	0.442	3.0
200	1.914	6.1	1.791	5.0	1.825	4.0
500	14.065	8.0	12.505	6.0	12.325	5.0
1000	64.471	9.4	56.320	7.2	53.654	5.7
1500	160.135	11.0	137.617	8.0	127.000	6.0
2000	304.366	11.9	256.851	9.0	236.372	7.0

Table 1: Sensitivity of the computation time and the number of creations of the heap to the partial heap size (average of 20 runs).

the heap, the heap can become empty. When this situation occurs, we have to create a new heap from scratch. It is clear that the total computing time heavily depends on the number of times that a new heap has to be build. As every combination will not be more than once in the heap, in the worst case we have to create a new heap at most n^2/h times. Therefore, the total time needed to create heaps is $O((n^4/h)\log h)$.

The time complexity for this implementation as a whole is thus $O((n^4/h)\log h + n^2 \log h + nh)$ and the memory requirement is $O(h)$. For example, when the heap size is equal to n, the time complexity is $O(n^3 \log n)$ and the memory requirement is $O(n)$. When the heap size is equal to $n\sqrt{n}$, the time complexity is $O(n^2\sqrt{n}\log n)$ and the memory requirement is $O(n\sqrt{n})$.

On practical problem sets this partial heap implementation performs much better than the time complexity suggests. This is due to the fact that the number of times that the algorithm needs to create a new heap is relatively small in practice. Computational evidence of this phenomenon is provided in Table 1. In the columns with the header 'cpu' we give the computation times in seconds on a 80486 PC. The number of times that a heap is created is denoted by m.

5.4 Grid implementation

The basic idea to reduce the memory requirements of the matrix implementation is to aggregate the information of a block of $b \times b$ cells of the matrix in one record, that contains the maximal saving in this block with the corresponding indices of the routes. The parameter b determines on what level we aggregate the information, and can be chosen to personal taste. We store the aggregated savings information in a matrix of $k \times k$ records, where k satisfies $(k-1)b < n \leq kb$. Clearly, the memory requirement is equal to $O(n^2/b^2)$. The initialization of the savings information can be done in $O(n^2)$. In every iteration, there are $O(n/b)$ blocks that need to be recomputed at a cost of $O(b^2)$ per block. Therefore, updating the savings information can be accomplished in $O(n^2 b)$ for the algorithm as a whole. Selecting the maximal saving costs $O(n^2/b^2)$ per iteration, and thus $O(n^3/b^2)$ in total. All together, this implementation has a time complexity of $O(n^2 b + n^3/b^2)$, which is minimized for $b = O(\sqrt[3]{n})$. Choosing $b < O(\sqrt[3]{n})$ can not be advised, as it increases both the time complexity and the memory requirement. If we choose b such that $O(\sqrt[3]{n}) \leq b \leq n$, then we face a trade-off between the time complexity and the memory requirement. A practical choice is $b = O(\sqrt{n})$, which gives an $O(n)$ memory requirement and a time complexity of $O(n^2\sqrt{n})$. The computation times in

	grid width			
n	$1/2\ \sqrt{n}$	$\sqrt{n}$	$2\sqrt{n}$	$5\sqrt{n}$
100	0.661	0.779	1.105	2.232
200	3.002	3.672	5.305	10.603
500	22.757	30.003	45.223	86.445
1000	106.790	150.520	228.220	436.205
1500	265.022	383.812	588.083	1134.071
2000	506.549	742.868	1154.356	2264.191

Table 2: Sensitivity of the computation time to the grid width (average of 20 runs).

seconds on a 80486 PC, for different values of n and b, are given in Table 2.

Although we have chosen to store both the values of the maximal savings and the corresponding indices of the routes, it suffices to store the values of the savings only. In every iteration, we can simply evaluate $b \times b$ possible combinations after we have selected the block with the maximal saving. All together, these evaluations can be done in $O(nb^2)$, which is much less than the $O(n^2 b)$ time spent on updating the savings information. However, it is necessary to store the indices of the routes, if we want to realize the following speed-up of the algorithm.

When recomputing the maximal saving of a block of $b \times b$ routes, one does not always have to evaluate all b^2 combinations. Suppose that two routes i and j are merged, such that route i becomes the new route and that route j is deleted. For every block that contains route i or j we then need to update the savings information. If the indices of the routes that give the maximal saving of such a block are not equal to i or j, then it suffices to check if a combination with the new route i can improve the current maximum. This can be done in $O(b)$ and is only necessary if the block contains route i. Our experiments indicate that with this approach speed-ups of at least a factor 2 can be obtained.

5.5 Recursive grid implementation

To get a greater reduction in computation time without a reduction in memory requirements, we propose an other modification of the matrix implementation, which we will call the recursive grid method. When using the matrix implementation, the problem is to select the combination of the two routes that gives the maximal saving, which costs $O(n^2)$ time per iteration. The update of the matrix, however, costs only $O(n)$ time per iteration. We have seen that we can reduce the time that we need to select the best combination by aggregating information in blocks of $b \times b$ cells. This leads however to an increase of the time we need to update the matrix. To circumvent this problem, we aggregate information by using different levels of aggregation.

On the first level, we store a complete matrix with the savings of all combinations of routes. On the next level, we aggregate the information of a block of $q \times q$ cells ($q \geq 2$) of the first level in a cell. The matrix on the first level is of size $n \times n$, so the matrix on the second level is of size $\lceil n/q \rceil \times \lceil n/q \rceil$. Each cell of the following level is also an aggregation of $q \times q$ cells of the previous level. We repeat the creation of a new level until we get a level with one cell. The total number of levels is clearly equal to $\lceil {}^q\log n \rceil$. The initialization of the

	array length					
n	3	4	5	6	7	8
100	0.366	0.362	0.372	0.359	0.363	0.377
200	1.521	1.486	1.470	1.466	1.470	1.500
500	10.519	9.983	9.788	9.665	9.601	9.768
1000	44.797	42.625	41.408	40.710	40.227	40.598
1500	104.945	99.144	95.699	93.844	92.398	93.158
2000	191.128	180.355	177.077	170.133	171.149	167.944

Table 3: Sensitivity of the computation time to the array length (average of 20 runs).

first level costs $O(n^2)$ time. The initialization of the m-th level, $m \geq 2$, costs $O((n/q^{m-2})^2)$ time. Summing over m, we see that the creation of al the levels costs $O(n^2)$ time. Working back from the m-th level downwards, we can determine the maximal saving in $O(q^2\ ^q \log n)$. There are at most $n-1$ iterations of the Savings method necessary, so we need $O(nq^2\ ^q \log n)$ to select the best savings. To update the stored information, we have to update the matrices on all levels. The update of the matrix on the first level costs $O(n)$ time. The update of the matrix on the m-th level costs $O(n/q^{m-3})$ time. So, the update of all levels costs $O(n)$ time per iteration and $O(n^2)$ time for the algorithm as a whole. We see that the time complexity of this recursive grid implementation is $O(n^2)$. When we use this recursive method, we have to store the matrices of all levels. The matrix on the first level has n^2 cells, the matrix on the m-th level has $\lceil n/q^{m-1} \rceil^2$ cells. So, the total memory requirement to store the matrices is still $O(n^2)$.

5.6 Array implementation

We have also made an implementation of the Savings method which we will call the array implementation. In this implementation, we have a sorted array for every route r in which we store the k routes, that give the greatest savings when being placed after route r. Creating the arrays costs $O(n^2k)$ time. We can determine the maximal saving in this implementation in $O(n)$ time per iteration, which leads to $O(n^2)$ for the algorithm as a whole. The update of the arrays costs only $O(nk)$ time per iteration and $O(n^2k)$ for the whole algorithm. However, when we update an array we place only new combinations in an array with savings that are greater than or equal to the minimal saving in the array. It can be possible that an array is empty after $k/2$ iterations, so that we have to create a new array. Creating arrays will be done at most $2n/k$ times, so for the whole algorithm this costs $O(n^3)$ time, which leads to a total time complexity of $O(n^3)$ for this implementation. Table 3 shows that in practice, this implementation performs much better. In this table the computation times are given for different values of n and k.

6 Computational results

To test the computation times of several implementations of the Savings method, we have developed a set of test problems. In every test problem the vehicle capacity is equal to 1000, the available time of the vehicles is 12 hours, the location of the depot is at (0,0) and the

	implementation			
n	part. heap	grid	rec. grid	array
100	0.437	0.642	0.413	0.380
200	1.824	2.919	1.687	1.494
500	12.255	22.467	11.369	9.733
1000	53.530	105.298	46.664	40.560
1500	125.972	260.600	105.468	93.563
2000	235.735	499.282	187.966	169.248

Table 4: Computation times for different implementations of the Savings method (average of 20 runs).

loading time of a vehicle is 30 minutes. Further, in every test problem we have the parameters n which denotes the number of customers and $\bar{q}$ which is the average demand of a customer.

We have generated the coordinates of the n customers in the square $[-100, 100]^2$, according to a continuous uniform distribution. The travel times in minutes between the different points are supposed to be equal to the euclidean distances, with a minimum of 5 minutes. The demand of each customer is drawn from a lognormal distribution with parameters $\sigma = 1$ and $\mu = \ln(\bar{q}') - \sigma^2/2$. This type of distribution is preferred over a uniform distribution as it allows for a small number of customers with a high demand and a low average demand. When a demand greater than the vehicle capacity is drawn, this outcome is rejected and a new value is drawn, so $\bar{q}'$ has to be greater than $\bar{q}$ to get an average demand $\bar{q}$. The delivery time at customer i is equal to $5 + 0.05 * q_i$ minutes. The time windows of the customers are uniformly distributed in the time interval during the first and the last possible arrival time of a vehicle at the customer and have an average width of 200 minutes.

With the generated test problems we have done several computations for different implementations of the Savings method. We give the results of computations for the partial heap implementation with a heap size of $5n$, the grid implementation with a grid width of $1/2\sqrt{n}$, the recursive grid implementation with $q = 2$ and the array implementation with an array length of 8. The parameters used for the specific implementations are determined from the results presented in the previous section. For different values of n we give the computation time in seconds on a 80486 PC in Table 4.

From this experiments we see that both the partial heap implementation and the array implementation perform well in practice, although their worst case computation time is not so attractive. The recursive grid implementation combines a nice theoretical behaviour with low computation time in practice. The memory requirements of this implementation, however, can be too demanding for large data sets, as it is $O(n^2)$. The grid implementation provides a nice trade-off between the memory requirements and the theoretical computation time. Its practical computation times, however, are too large compared to the other implementations.

References

[1] G. Clarke and J.W. Wright, Scheduling of Vehicles from a Central Depot to a Number of Delivery Points, *Operations Research 12* (1964), 568-581.

[2] G.B. Dantzig and J.H. Ramser, The Truck Dispatching Problem, *Management Science 6* (1959), 80-91.

[3] M. Desrochers, J.K. Lenstra, M.W.P. Savelsbergh and F. Soumis, Vehicle Routing with Time Windows: Optimization and Approximation, In: Vehicle Routing: Methods and Studies, B.L. Golden, A.A. Assad (eds.), North-Holland, Amsterdam (1988), 65-84.

[4] B.L. Golden, T.L. Magnanti and H.Q. Nguyen, Implementing Vehicle Routing Algorithms, *Networks 7* (1977), 113-148.