Comput. & Ops. Res. Vol. 12, No. 3, pp. 273–283, 1985
Printed in the U.S.A.

0305–0548/85 $3.00 + .00
Pergamon Press Ltd.

IMPLEMENTATION TECHNIQUES FOR THE VEHICLE ROUTING PROBLEM

MARVIN D. NELSON,† KENDALL E. NYGARD,‡ JOHN H. GRIFFIN§ and WARREN E. SHREVE‖
Division of Mathematical Sciences, North Dakota State University, Fargo, ND 58105, U.S.A.

(*Received October* 1982; *revised April* 1984)

Scope and purpose—The vehicle routing problem (VRP) arises in designing routes for school buses, garbage collection, package delivery and many other pickup and delivery systems. The VRP allows the available vehicles to have limited capacities. The Clarke-Wright heuristic algorithm and related variations are very widely used to produce solutions to the VRP. This paper presents and compares alternative data structures and computer science techniques which can be used to implement the Clarke-Wright algorithm.

Abstract—Six methods for implementing the widely used Clarke-Wright algorithm for the vehicle routing problem (VRP) are presented and compared. Fifty-five large test problems are used to compare the methods. The methods involve alternative ways to access adjacency information in both low and high density problems. The results clearly establish methods of choice for VRP problems with given characteristics.

INTRODUCTION

Alternative data structures and computer science techniques can be used to efficiently implement widely used heuristic algorithms for the vehicle routing problem (VRP). In the VRP, there is a given set of stops (customers) each demanding some known quantity of service (pickup or delivery) and a set of vehicles with fixed capacities available to serve these customers. The objective is to assign stops to vehicles and specify a routing order for each vehicle which minimizes total operating cost for the fleet. The VRP and closely related variations arise frequently in designing routings for garbage collection, school buses, package delivery, milk delivery and many other urban and rural delivery systems. If there are m vehicles, each with unlimited capacity, the VRP is known as the m-travelling salesman problem (m-TSP). The m-TSP can be reformulated as a single travelling salesman problem (1-TSP or simply TSP). Thus, only the presence of vehicle capacities separates the VRP from the TSP.

This paper begins with a survey of algorithms which have been used for the VRP. Six methods of implementing the widely used Clarke-Wright algorithm are described in detail. Fifty-five test problems of various sizes and characteristics are used to compare CPU times and memory requirements of the methods. CPU times required by the methods show that considerable improvement over straightforward implementations can be obtained. The results clearly establish rankings among the methods. Special problem characteristics which influence performance of the alternative methods are identified.

†Marvin D. Nelson is a graduate research intern in the computer science Master's degree program at North Dakota State University. He previously served on the support staff of the NDSU Computer Systems Institute. His research interests are in the areas of computer-based operations research methods, combinatorial optimization and computer graphics software.

‡Kendall E. Nygard is assistant professor of Operations Research in the Mathematical Sciences Department at North Dakota State University. He earned a Ph.D. in Industrial Engineering and Operations Research at Virginia Tech University. Software developed under Dr. Nygards' direction has been used to deyelop efficient school bus routes for *13* districts in North Dakota, Minnesota and South Dakota. In addition, he has major involvement with a large-scale network flow model used in North Dakota grain marketing analyses.

§John H. Griffin is assistant professor of Computer Science in the Mathematical Sciences Department at North Dakota State University. He has also served on the teaching faculty of the NDSU Computer Systems Institute. He received his Ph.D. from Washington State University. Dr. Griffin is interested in algorithm analyses, design and complexity. He is also involved with operating system design, including adaptations of the Bell Labs UNIX operating systems to microprocessor-based systems.

‖Warren E. Shreve is associate professor of Mathematics at North Dakota State University. He earned his Ph.D. in mathematics at the University of Nebraska. His research program in matrix differential equation currently involves the calculation of inverses, eigenvalues and eigenvectors of large sparse matrices. Dr. Shreve is also interested in the methods of dynamic programming and their application to operations research problems.

ALGORITHMS FOR THE VEHICLE ROUTING PROBLEM

A substantial number of vehicle routing algorithms have appeared in the literature. Detailed surveys have been written by Christofides *et al.* [4], Watson-Gandy and Foulds [29] and Bodin *et al.* [1]. This section briefly summarizes alternative approaches to the vehicle routing problem (VRP).

The VRP can be formulated as an integer programming problem in several ways [9, 10, 22, 25]. In these formulations, the close connection between the VRP and the TSP is apparent, and even relatively small problems require large numbers of variables and constraints. Although some exact algorithms have been developed, the computational complexity of the problem renders methods for obtaining a true optimal routing practical for only small problems, roughly on the order of thirty customers [25]. The TSP and VRP are both members of the set of NP-complete problems [19]. Thus, all currently known exact algorithms for these problems require a number of computational steps that grows as an exponential function of the number of stops which must be visited.

In practice, most research has been concerned with heuristics. In some cases, the structure of integer programming formulations has provided insight into problem structure which can be used to advantage in developing heuristic algorithms. Classification ideas of Fisher and Jaikumar [9], and Watson-Gandy and Foulds [29] can be utilized to categorize existing heuristics as follows: (1) tour-building methods; (2) two-phase methods; and (3) optimization-based methods.

Tour-building methods have the richest history, with much of the work involving ideas which originated in the 1964 work of Clarke and Wright [5]. The Clarke–Wright method begins with each customer on a tour by itself, served by a single vehicle. Thus, the mileage incurred to service a given customer i is $d(0, i) + d(i, 0)$, where 0 is the index of the depot. The algorithm uses a largest mileage savings criteria to determine the order in which tours are combined. For example, a tour which combines node i and node j would proceed from the depot to node i, then directly to node j followed by a return to the depot. The "savings" associated with combining these routes is given by

$$s(i, j) = d(0, i) + d(0, j) - d(i, j).$$

As the expression indicates, combining the nodes into a single tour saves retracing of the two arcs which connect nodes i and j to the depot, but incurs the distance from i to j in so doing. Combined tours can be further merged if mileage savings result. The original Clarke–Wright algorithm processes the savings in order, beginning with the largest. In considering any given savings, tour feasibility is evaluated before a change is made. The Clarke–Wright method is the most widely used algorithm in practice, and many variations have appeared in the literature [23, 28]. For example, Gaskell [11] and Yellow [30] modify the savings calculation $s(i, j)$ by placing weights on the terms. Gaskell also suggests including average distances from nodes to the depot. Several authors have developed "look ahead" features which attempt to evaluate the consequences of choosing a particular savings before actually accepting it [15, 17, 26, 27]. Golden *et al.* [14] initiated the use of special data structures in implementing the Clarke-Wright method. These ideas are expanded upon in this paper.

Two-phase methods, also known as cluster-first methods, operate in phase 1 by assigning customers to vehicles before any actual routing is done. Phase 2 involves the use of a TSP algorithm to route the customers in each tour once clustering is completed. Widely differing phase 1 techniques have been suggested in the literature [9]. For example, Tyagi [28] uses a nearest neighbor rule to sequentially assign customers to vehicles. Christofides *et al.* [4], use a similar clustering rule, but invoke a tour improvement algorithm as the customers are assigned to vehicles. The Gillett-Miller SWEEP algorithm [13] utilizes a simple phase 1 technique in which customers are ordered by angle and a radial arm sweep assigns them until vehicle capacity is met.

Optimization-based methods for the VRP have roots which are associated with the evolution of combinatorial optimization through the 1950s and 1960s [22]. Recently developed optimization-based heuristics for the VRP have produced excellent solutions in reasonable CPU time for problems of up to 200 nodes [9, 25]. Some of these methods could

be properly classified as two phase methods, but are a separate category because of their strong connection with integer programming formulations of the problem. Heuristic VRP algorithms based on Lagrangian relaxation of these formulations have been developed by Stewart and Golden [25], Christofides *et al.* [4] and Gavish and Graves [12]. The ideas are well presented in a survey article by Magnanti [22]. Research in this area suggests that this approach to the VRP might be pursued with success in the near future.

The highly successful method of Fisher and Jaikumar [7, 9] begins with an integer programming formulation and works with generalized assignment and TSP subproblems. Their heuristic method is an outgrowth of a Benders decomposition approach described in their earlier paper [8]. Finally, Cullen *et al.* [6] utilize a set-partitioning formulation of the VRP as the basis for a heuristic which alternates between assignment and location subproblems as long as further improvement is possible.

After a VRP solution has been created, the individual TSP tours can often be improved through the use of a tour improvement routine. The k-opt branch exchange concept associated with Lin [20] and Lin and Kenighan [21] is the most widely used. The k-opt method exchanges k branches from the tour for k branches not in the tour if improvement results. For n tour stops, consideration of all possible exchanges involves computational work of order n^k, a fact that has often limited k to the values two or three in practice. A promising variation known as Or-opt [23] examines only branch exchanges which insert short strings of nodes into new tour locations. Stewart [25] reports that Or-opt performs nearly as well as 3-opt with only about five percent of the comparisons.

METHODOLOGY

When solving large vehicle routing problems (200 or more nodes) computational time and solution quality are both important considerations. The Clarke–Wright algorithm and closely related variations yield reasonably good solutions without excessive computational time and for this reason are widely used in commercial software packages. In this section, six implementations of the Clarke-Wright are presented. Empirical work, presented in the section on computational results, establishes that good data structures and efficient use of available information results in very fast "savings" algorithms.

(a) *Basic characteristics of the six alternative methods*

Salient features of the six methods for implementing the Clarke-Wright algorithm are presented below. Detailed descriptions and diagrams of the data structures for the methods are in the next sub-section.

In the Clarke–Wright algorithm, there is a savings

$$s(i, j) = d(0, i) + d(0, j) - d(i, j)$$

for each possible pair of nodes i and j. The savings are processed in descending order of magnitude and an arc (i, j) is accepted (added to a route) if:

(1) Customers i and j do not belong to the same route.

(2) A vehicle capacity constraint would not be violated if the two routes containing customers i and j are joined.

(3) Neither i nor j is interior to a route (an interior customer is not adjacent to the depot).

Savings are processed until all arcs have been considered. During the early stages of the algorithm, total mileage is rapidly reduced and the number of vehicles is reduced by one each time an arc is accepted. The final stages of the algorithm are characterized by little reduction in mileage while each arc added to an emerging route reduces the number of vehicles required by one.

In the Clark–Wright algorithm, the number of savings to consider may be quite large. A good implementation of the algorithm must provide an efficient method for processing the

savings in descending order and also allow a simple way of "looking ahead" to eliminating those arcs associated with customers which become interior to routes. Although there are several sorting algorithms which perform well (e.g. quicksort, heapsort, shellsort and mergesort), only the heap (data structure used by the heapsort) appears to offer a good way of using interior node information to eliminate arcs from the savings array when they become inadmissable as new links are added to the routes. This is because the heap maintains only a partial ordering among the savings, and is easy to adjust when a savings is eliminated.

A heap is a complete binary tree in which the key value (this will always be a savings in the algorithms to be described) of any node is greater than or equal to the key of either of its children (if they exist). This implies that the root node is the largest element in the heap. A sequential representation of a heap using an array is usually handled by identifying the elements at positions $2i$ and $2i + 1$ as the children of node i. Figure 1 illustrates a heap representation of the savings in a six arc VRP network. Array positions are indicated by the numbers to the left of the nodes. The -1 savings value represents an artificial node used to complete the binary tree. In addition, the heapsort is the only sort procedure that requires almost no additional storage and has a worst case and an expected case order of $O(N \log_2 N)$ (proved to be the fastest order achievable for any sorting algorithm [16]).

Method 0 is a standard Clarke–Wright implementation which uses a shellsort to totally order the savings list prior to any arcs being considered for acceptance. Method 1 is similar to method 0 but a heap is used to partially order the savings array. The root node is removed, considered for inclusion in a route and the heap reformed. This process is repeated until the heap is empty. Since the savings are processed from the heap, a totally ordered list is never actually created and two array references for each arc are saved. In addition, the efficiency of the heapsort ensures that method 1 will perform at least as efficiently as any method which uses a sorting algorithm to produce a total ordering of the savings list before considering them for route inclusion.

The partial ordering present in a heap structure suggests that adjacency information [14] can be used to remove values from the savings list as it is determined that they are associated with nodes which have become interior and can no longer be considered. The arcs removed by adjacency information would not be explicitly considered for inclusion in a route, thereby reducing computational effort. Methods 2 and 3 are alternative ways of explicitly storing pointers needed to eliminate these arcs. Early elimination of these arcs reduces the size of the heap which must be reformed each time a savings is extracted. When methods 2 and 3 are compared with method 1 it is apparent that the use of adjacency information reduces CPU time, but increases memory requirements.

The success of methods 2 and 3 suggests that it is worthwhile to consider ways to efficiently store and reference adjacency information. Method 4 uses a very compact and efficient means of handling adjacency information in complete graph (fully dense) problems. All the methods must store the origin, terminal and savings due to the sorting process, and all must explicitly reference each arc. Therefore, when adjacency information is not used, a complete graph offers no advantage over a noncomplete graph. However, complete graphs have special properties that allow the use of hashing functions and mathematical relation-

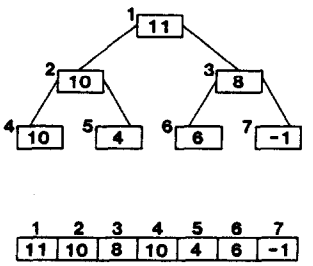

Figure 1. Heap data structure.
(a) Heap interpretation.
(b) Array representation.

ships to quickly locate adjacency information implicitly present in a compact storage scheme.

Method 4 assumes a complete graph and is able to store and solve the same problems as methods 2 and 3 with three-sevenths the storage and less computational time. This offers substantial benefits in solving problems that are complete or can be augmented to be complete.

Finally, method 5 involves using a number of sequentially created heaps to store the savings. In this method, savings which exceed a threshold value are stored in a heap and processed to completion using the data structures of method 3. Subsequent heaps are created for only those nodes which have not become interior to emerging routes. The smaller heaps allow for faster processing and the total number of entries in all the smaller heaps is less than the number of entries in the single heap of method 3. This multiple heap process reduces the sort time at the expense of some additional savings calculation time. Empirical work with method 5, using a 2-heap process, produced the best CPU time for all test problems while using the least storage for 45 of the 55 test problems in this study.

(b) *Detailed descriptions of the Clarke–Wright implementations*

This section describes the details of the six implementations of the Clarke–Wright algorithm. The following notation is used throughout this section:

m = number of arcs

n = the number of nodes.

All six methods require two lists of length n. The first node-length array is used to record the route each node is presently on and mark nodes that become interior to a route. The second array stores the demand for each vehicle, and initially contains the demand of the individual nodes. In the figures, a simple VRP network with four non-depot nodes and six arcs is used to illustrate the methods.

Method 0. This method is a straightforward implementation of the Clarke–Wright algorithm which was used as a benchmark to measure improvements in CPU time and memory requirements. Three additional lists of length m are required. These arrays contain the origin, the terminal and the savings associated with each arc. A shellsort with optimal sorting intervals (16) is used to totally order the savings which are then selected from the list in descending order. The lists used by this technique require $3m + 2n$ storage units plus additional sorting storage of a length equal to three times the highest sorting interval. Figure 2 illustrates this implementation.

Method 1. This method is similar to method 0 except that a heap is used to store the three lists. The current root node of the heap is removed, chosen if the selection criteria are satisfied and the heap is reformed. This process is repeated until all savings have been considered. This technique requires only one memory location for sorting, thereby reducing the required storage to $3m + 2n$.

Method 2. This method utilizes a heap and additional lists which store adjacency information used to eliminate all arcs associated with a node which becomes interior to an emerging route (Fig. 3). This technique has the effect of constantly reducing the size of the heap and results in considerable savings in CPU time. All adjacency information is explicitly stored; the method can therefore be applied to complete or noncomplete graphs.

1	2	2	1	3	1
2	3	4	4	4	3
11	10	10	8	6	4

m long

Figure 2. Data structure for method 0 and 1.

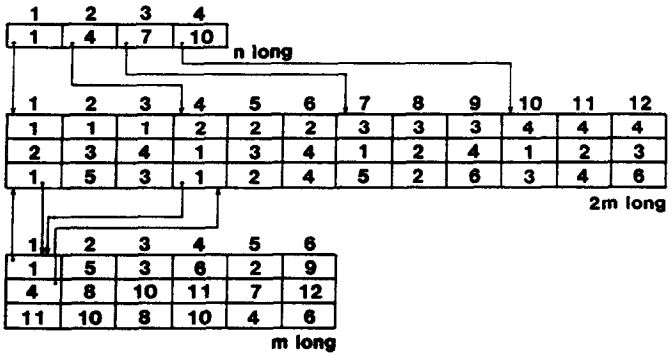

Figure 3. Data structure for method 2.

Storage areas necessary for this implementation include:

Network topology

Bookkeeping. Three parallel arrays of length $2m$ to store origin and terminal nodes for each arc and a pointer from this arc to the associated savings in the heap structure. Arcs are stored in ascending origin node order. Every arc is stored both forwards and backwards, e.g., arc (1–2) also appears as arc (2–1). This allows direct access to sublists which contain every arc involving an individual node.

Starting location pointer. An array of length n in which the starting position of the adjacency information sublist in the "bookkeeping" arrays is stored.

Heap

Savings and pointers out of the heap. Three parallel arrays of length m, maintained as a heap and used to store the savings and two pointers from the heap structure to the associated information in the "bookkeeping" arrays.

When a node becomes interior to a route, savings of all associated arcs are removed from the heap. The appropriate sublist is found via the "starting location pointer" list. Each arc in this sublist is removed form the heap by reheaping the binary tree from the location indicated by the "heap pointer." The procedure described in Horowitz and Sahni [16] for reheaping is quite simple and efficient. The method requires $9m + 3n$ storage locations.

Method 3. In method 2, the representation allows arcs to be eliminated from the heap during route formation. For a large problem, however, the storage requirements can become excessive. Method 3 is a technique which reduces the storage requirements to $7m + 3n$. This is done by eliminating the redundant information stored in the "bookkeeping" lists of method 2.

The structure is easily understood by considering an example. Referring to Fig. 4,

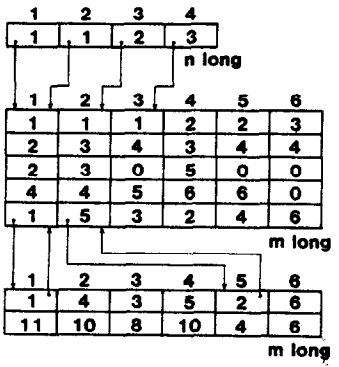

Figure 4. Data structures for method 3.

suppose that all arcs associated with node 2 are to be eliminated. The first occurrence of an arc involving node 2 is found by following the "starting location pointer" at position 2 to position 1 of the "bookkeeping" lists. The arc (1–2) would then be eliminated by setting the savings at position 3 of the "heap" to a negative value and reforming the heap. Subsequent arcs would be eliminated by the following rules:

(a) If node 2 was in the "node A" list then follow the "node A pointer"; otherwise follow the "node B pointer" to the next occurrence of an arc involving node 2. If the appropriate pointer is equal to zero, stop.

(b) Use the pointer in the "bookkeeping" array to obtain the position in the heap of the corresponding savings.

(c) Set the savings to a negative value and readjust the heap.

(d) Return to step (a).

Method 4. In complete graph problems, further savings in memory and CPU time can be realized over method 3 by taking advantage of implied adjacency information. As in method 3, the savings are stored in a heap along with a pointer to the appropriate arc information in the "bookkeeping" arrays (Fig. 5). The arcs are stored in the "bookkeeping" array in the following order:

	A_1	A_2	····	$A_{(n-1)}$
origin	1 1 1 · · · 1	2 2 · · · 2	····	$(n-1)$
terminal	2 3 4 · · · n	3 4 · · · n	····	n

Using this arrangement, it is not necessary to explicitly store the arc origin and terminal nodes because the array position implies this information. For example, arc (1–2) always appears implicitly in position 1 of the array. The adjacency information is also implied. In particular, all arcs which originate from node i are in positions

$$A_i + j \qquad j = 0, 1, \ldots, (n - i - 1)$$

and all arcs which terminate with node i are in positions

$$A_{(i-j)} + (j - 1) \qquad j = 1, 2, \ldots, (i - 1).$$

The savings are processed in a manner similar to method 3. A heap is formed and the root node considered for inclusion in an emerging route. A hash function is used to determine the origin and terminal node for each arc that is considered. The heap is reformed and the process repeated until the heap is empty. When a node becomes interior, a hash function is used to determine the locations of all arcs which have that node as an endpoint. The savings associated with these arcs are removed from the heap as in method 3. The storage requirements for this method are $3m + 3n$. Methods 2 or 3 will clearly place much greater demands on memory unless the problem being solved is of relatively low density.

Method 5. In methods 2–4, heap creation and adjustments are a significant part of the computational overhead. Method 5 is a technique in which several smaller heaps are used. The algorithm is carried out in several phases and different heaps are generated during each

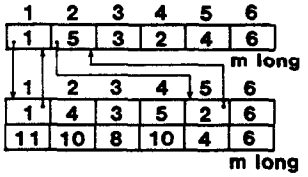

Figure 5. Data structures for method 4.

phase. During a given phase, only the savings of arcs between non-interior nodes which exceed a threshold value are included in the heap being constructed. Once the heap is generated, all arcs in the heap are processed as before. New phases with successively lower threshold values are initiated until no unprocessed arcs remain. This occurs when the threshold value equals zero. The heaps are formed and processed using the data structures of method 3.

The method was tested by choosing a threshold value experimentally and running the algorithm in two phases. Experiments using ten uniformly distributed test problems showed that the algorithm performed well with threshold values between 20 and 40% of the maximum possible savings value. A 30% threshold value was used for the test problem runs. The method was also effective for the non-uniform problems which were tested. The CPU times were typically 30% shorter than the best times of all the other methods.

COMPUTATIONAL RESULTS

Computational work with the six methods described in this paper leads to four conclusions. First, when compared with other techniques for storing and processing the entire savings list in order, methods which use the heap data structure perform the best. Second, using adjacency information in conjunction with the partial ordering among savings stored in the heap structure results in a substantial reduction in CPU time. Third, using special characteristics of complete graphs to store and access adjacency information with minimum memory requirements result in reduced CPU time when compared with conventional methods of storing adjacency information. Finally, when used with a carefully chosen threshold value, sequentially creating and processing heaps with only part of the eligible savings in each step results in the smallest CPU times and storage requirements among all the methods tested. Empirical work which leads to these conclusions is presented in this section.

(1) Test problems

CPU times for all the methods are shown in Tables 1 and 2. Problems 1.1–1.5 are from Christofides *et al.* [3], and have been used as test problems by several authors [4, 9, 10, 22, 25]. The remaining problems are fully dense (complete graph) problems and range in size from 200 to 1000 nodes. These test problems were created with a random test problem generator. Using parameters specified by the user, the generator uniformly distributes nodes within a rectangular geographical region and within round areas of high node density called clusters. The test problems have from zero to three clusters and the fraction of nodes within the clusters and their distribution among the clusters are varied. Locations of the clusters center and the depot are also varied. All the methods were run and compared on 30 problems with 200, 400 and 600 nodes. The results are shown in Table 1. Because of their superior performance on these smaller problems, only methods 4 and 5 were run on 800 and 1000 node problems. These results are shown in Table 2.

(2) Comparison of methods

The Pascal language was chosen for the study because of its readability and ease of implementing data structures. Much effort was expended to take advantage of every possible computational and storage savings within the framework of Pascal coding and good programming style. The codes were run on a Pascal 8000 compiler [2] with the run time error checking turned off. We avoided special machine and language dependent capabilities like bit level manipulation and alternative storage classes (such as the two byte integers available in FORTRAN and assembly languages). We note, however, that manipulations performed on a heap lend themselves very well to assembly language coding. Experiments with assembly language heap manipulation resulted in savings of an additional 8.4–14.2% over the CPU times reported in this section. In order to maintain consistancy for comparison purposes, the programs reported in this paper were written entirely in Pascal.

The CPU times including loading the problem, generating the savings, sorting and processing the savings and outputting the arcs as they are accepted. Compilation time is not

Table 1.

Problem Number	Number of Nodes	Method 0	Method 1	Method 2	Method 3	Method 4	Method 5
1.1	50	1.73	1.39	1.27	1.15	1.09	1.07
1.2	75	3.51	2.60	2.25	1.93	1.83	1.83
1.3	100	6.28	4.39	3.43	2.92	2.74	2.53
1.4	150	13.52	9.65	7.17	5.98	5.56	4.98
1.5	199	24.73	17.19	12.22	10.21	9.59	8.43
2.1	200	24.93	17.15	12.33	10.13	9.44	7.57
2.2	200	25.67	17.26	12.27	10.11	9.36	7.63
2.3	200	27.63	17.22	12.65	10.43	9.83	9.63
2.4	200	27.05	17.14	12.39	10.42	9.57	8.67
2.5	200	25.75	17.06	12.13	10.30	9.39	7.78
2.6	200	24.69	17.08	12.42	10.41	9.61	8.13
2.7	200	26.41	17.27	12.24	10.32	9.43	7.93
2.8	200	26.99	17.17	12.19	10.29	9.39	7.66
2.9	200	25.80	17.11	12.25	10.30	9.46	7.61
2.10	200	25.98	17.13	12.17	10.24	9.47	7.94
3.1	400	105.30	72.59	47.30	38.91	36.56	28.73
3.2	400	104.16	72.93	47.28	38.85	36.45	28.46
3.3	400	126.46	73.35	49.19	39.92	38.39	36.02
3.4	400	128.76	73.18	48.10	39.31	37.32	32.10
3.5	400	117.60	73.51	47.46	38.98	37.05	29.45
3.6	400	125.94	73.64	47.88	39.20	37.55	30.49
3.7	400	124.17	73.73	47.32	38.89	37.08	29.79
3.8	400	110.82	72.97	47.21	38.79	36.67	28.89
3.9	400	129.11	72.53	47.29	38.91	36.92	28.59
3.10	400	122.23	72.46	47.18	38.82	36.74	29.06
4.1	600	251.50	170.25	106.53	87.50	83.55	63.95
4.2	600	245.05	170.23	107.63	87.65	83.57	64.49
4.3	600	312.14	169.50	112.83	90.06	88.83	81.05
4.4	600	310.28	169.44	108.94	88.81	86.03	74.18
4.5	600	278.19	169.60	106.72	87.52	84.55	65.66
4.6	600	275.54	170.04	107.56	87.99	87.31	68.19
4.7	600	288.35	170.30	106.94	87.70	86.82	65.58
4.8	600	254.68	170.06	106.12	87.18	84.70	63.38
4.9	600	290.32	170.89	107.29	87.51	86.07	63.37
4.10	600	292.18	171.13	106.55	87.38	86.42	65.16

Table 2.

Problem Number	Number of Nodes	Method 4	Method 5
5.1	800	155.06	113.19
5.2	800	156.14	113.11
5.3	800	167.86	148.73
5.4	800	159.93	130.38
5.5	800	155.33	116.39
5.6	800	158.22	122.66
5.7	800	156.10	116.23
5.8	800	153.58	112.96
5.9	800	158.35	112.24
5.10	800	156.72	115.30
6.1	1000	245.60	176.18
6.2	1000	239.87	176.93
6.3	1000	260.64	**
6.4	1000	257.79	207.60
6.5	1000	249.77	185.79
6.6	1000	255.58	193.94
6.7	1000	250.78	186.33
6.8	1000	250.46	181.18
6.9	1000	253.78	180.55
6.10	1000	251.52	184.54

** Not Executed Because of Memory Limitations

included. A preprocessor was used to generate the problems and a post processor to assemble the routes and output demand and solution quality. All problems were run on an IBM 4341 computer.

The computational data supports four main conclusions, each of which is described below.

First, use of the heap structure reduces the computational time. When compared with the bench-mark method 0, the simplest heap-based method (method 1), reduced CPU time by an average of 38.45 percent. Methods 2–5 also use heaps and exhibit additional improvements in CPU times. Use of the quicksort-based method in a way similar to the method 1 procedure may produce comparable CPU time results, since it has an expected order of $n\log n$. A quicksort method would, however, require substantial additional memory and has a worst case order of n.

Second, use of pointers to explicitly store adjacency information in combination with a heap (methods 2 and 3) cut execution time when compared to method 1. Method 3 consistently outperformed method 2 and reduced execution time by an average of 46.56% compared to method 1. Most of the success of adjacency information lies in reducing the number of arcs that are considered for acceptance, and in removing elements before incurring the cost of sorting them.

Third, a substantial reduction in storage can be realized without loss of computational efficiency if a complete graph is used. In terms of storage, one would have to eliminate four-sevenths of the possible arcs before a noncomplete graph would be competitive with a complete graph (assuming that adjacency information is being used). Thus, in actual routing problems where only a few arcs can be removed in practice, it is clearly advantageous to use a complete graph and associate a prohibitively large distance with the invalid arcs. These large distances would force the arcs savings to lie near the bottom of the heap where they will be rejected if a feasible routing exists. On the other hand, it is important to realize that as arcs are eliminated, the CPU time of method 3 will ultimately become better than that of method 4.

Finally, experiments with method 5 resulted in the shortest run times of all the methods for each test problem, and the smallest storage requirements in nearly all cases (25–40% of the storage required to run the same problem as method 3 if a reasonably good threshold value is chosen). However, method 5 should be applied somewhat cautiously in practice. The rule used to determine a threshold value for placing savings in the first heap was arrived at by experimenting with test problems which had nodes uniformly distributed in a square region with the depot at the center. Deviations from a uniform density of nodes affects the distribution of the savings values for the arcs and their acceptance into emerging routes. However, the results shown in Tables 1 and 2 are encouraging in general. Forty-five of the 55 test problems are not uniform and represent a wide range of induced clustering. Problems with many nodes in a relatively small single cluster require substantial memory in method 5, but only problem 6.3 required more storage then it was possible to request on the computer system used. This problem is a large fully dense problem, with 1000 nodes and 499500 arcs. In this problem, 75% of the nodes are located in a small cluster far from the depot. Problems 2.3, 3.3 4.3 and 5.3 have the same characteristics with fewer nodes, but ran successfully on the system.

Our experiments clearly show that careful coding and judicious data structure choices can substantially reduce CPU times and make it possible to easily handle problems up to 1000 nodes or more.

REFERENCES

1. L. Bodin, B. Golden, A. Assad and M. Ball. The state of the art in the routing and scheduling of vehicles and crews. *Comput. Ops Res.* **9**, 63–212 (1983).
2. G. Cox and J. Tobias, *Pascal 8000 IBM 360/370 Version for OS and VS Environment.* Reference Manual, Version 1.2, Feb. (1982).
3. N. Christofides, S. Eilon and C. D. T. Watson-Gandy, *Distribution Management: Mathematical Modelling and Practical Analysis.* Griffin, London (1971).
4. N. Christofides, A. Mingozzi and P. Toth, The vehicle routing problem. *Combinatorial Optimization.* Wiley, New York (1979).

5. G. Clarke and J. Wright, Scheduling of vehicles from a central depot to a number of delivery points. *Ops Res.* **12,** 568–581 (1964).
6. F. Cullen, J. Javis and H. D. Ratliff, Set partitioning based heuristics for interactive routing. *Networks* **11,** 125–143 (1981).
7. M. L. Fisher, Lagrangian relaxation method for integer programming problems. *Management Sci.* **27,** 1–18 (1981).
8. M. L. Fisher and R. Jaikumar, *A Decomposition algorithm for large-scale vehicle routing,* Decision Science Working Paper 78-11-05, University of Pennsylvania, July (1978).
9. M. L. Fisher and R. Jaikumar, A generalized assignment heuristic for vehicle routing. *Networks* **11,** 109–124 (1981).
10. B. A. Foster and D. M. Ryan, An integer programming approach to the vehicle scheduling problem. *Ops Res. Quart.* **27,** 367–384 (1976).
11. T. J. Gaskel, Bases for vehicle fleet scheduling. *Ops Res. Quart.* **18,** 281–295 (1967).
12. B. Gavish and S. C. Graves, *The traveling salesman problem and related problems.* Working Paper OR 078-78, Operations Research Center, MIT, July (1978).
13. B. E. Gillett and L. R. Miller, A heuristic algorithm for the vehicle-dispatch problem. *Ops Res.* **22,** 340–349 (1974).
14. B. Golden, T. Magnanti and H. Nguyen, Implementing vehicle routing algorithms. *Networks* **7,** 113–148 (1977).
15. R. A. Holmes and R. G. Parker, A vehicle scheduling procedure based upon savings and a solution perturbation scheme. *Ops Res. Quart.* **27,** 83–91 (1976).
16. E. Horowitz and S. Sahni, *Fundamentals of Data Structures.* Computer Science Press, Maryland (1976).
17. K. Knowles, The use of a heuristic tree-search algorithm for vehicle routing and scheduling. *Operational Research Conf.,* Exeter, England (1967).
18. D. E. Knuth, *The Art of Computer Programming.* Addison-Wesley, Philippines (1973).
19. J. L. Lenstra and A. H. G. Rinnooy Kan, Complexity of vehicle routing and scheduling problems. *Networks* **11,** 221–227 (1981).
20. S. Lin, Computer Solutions of the T.S.P. *Bell System Tech. J.* **44,** 2245–2270 (1965).
21. S. Lin and B. W. Kernighan, An effective heuristic algorithm for the T.S.P. *Ops Res.* **21,** 498–516 (1973).
22. T. L. Magnanti, Combinatorial optimization and vehicle fleet planning: perspectives and prospects. *Networks* **11,** 179–213 (1981).
23. I. Or, Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking. Ph.D. Thesis, Dept. of Industrial Engineering and Management Sciences, Northwestern University (1976).
24. J. A. Robbins and W. C. Turner, *Clarke and Wright—Lin interaction program for vehicle routing problems.* Working Paper, Department of Industrial Engineering, Oklahoma State University (1976).
25. W. Stewart and B. Golden, New algorithms for deterministic and stochastic vehicle routing problems D.B.A. thesis, University of Maryland, (1981).
26. F. Tillman and H. Cochran, A heuristic approach for solving the delivery problem. *J. Ind. Engng.* **19,** 354–358 (1968).
27. F. Tillman and R. D. Hering, A study of a look-ahead procedure for solving the multiple delivery problem. *Transportation Res.* **5,** 225–229 (1971).
28. M. S. Tyagi, A practical method for truck dispatching problem. *J. Ops Res. Soc. Japan* **10,** 76–92 (1968).
29. C. D. T. Watson-Gandy and L. R. Foulds, The vehicle scheduling problem: a survey. *New Zealand Op. Res.* **9,** 73–91 (1981).
30. P. C. Yellow, A computational modification of the savings method of vehicle scheduling. *Op. Res. Quart.* **21,** 281 (1970).