Linear Time Dynamic-Programming Algorithms for New Classes of Restricted TSPs: A Computational Study

Egon Balas • Neil Simonetti

Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA

Bryn Athyn College of the New Church, 2895 College Drive, Box 717, Bryn Athyn, PA 19009, USA

eb17@andrew.cmu.edu • nosimone@newchurch.edu

Consider the following restricted (symmetric or asymmetric) traveling-salesman problem (TSP): given an initial ordering of the n cities and an integer $k > 0$, find a minimum-cost feasible tour, where a feasible tour is one in which city i precedes city j whenever $j \geq i + k$ in the initial ordering. Balas (1996) has proposed a dynamic-programming algorithm that solves this problem in time linear in n, though exponential in k. Some important real-world problems are amenable to this model or some of its close relatives.

The algorithm of Balas (1996) constructs a layered network with a layer of nodes for each position in the tour, such that source-sink paths in this network are in one-to-one correspondence with tours that satisfy the postulated precedence constraints. In this paper we discuss an implementation of the dynamic-programming algorithm for the general case when the integer k is replaced with city-specific integers $k(j)$, $j = 1, \ldots, n$.

We discuss applications to, and computational experience with, TSPs with time windows, a model frequently used in vehicle routing as well as in scheduling with setup, release and delivery times. We also introduce a new model, the TSP with target times, applicable to Just-in-Time scheduling problems. Finally for TSPs that have no precedence restrictions, we use the algorithm as a heuristic that finds in linear time a local optimum over an exponential-size neighborhood. For this case, we implement an iterated version of our procedure, based on contracting some arcs of the tour produced by a first application of the algorithm, then reapplying the algorithm to the shrunken graph with the same k.

(*Dynamic Programming, Algorithms, Complexity, Heuristics, Vehicle Routing, Traveling Salesman Problem*)

1. Introduction

Given a set N of points called *cities* and a cost c_{ij} of moving from i to j for all $i, j \in N$, the traveling salesman problem (TSP) seeks a minimum-cost permutation, or tour, of the cities. If $c_{ij} = c_{ji}$ the TSP is symmetric, otherwise it is asymmetric. Stated on a directed or undirected graph $G = (N, A)$ with arc lengths c_{ij} for all $(i, j) \in A$, the TSP is the problem of finding a shortest Hamiltonian cycle in G. It is well known that the TSP is NP-complete, but some special cases of it are polynomially solvable. Most of these special cases owe their polynomial-time solvability to some attribute of the cost matrix. Other cases arise when the problem is restricted to graphs with some particular (sparse) structure. For surveys of the literature on this topic see Burkard et al. (1997), Gilmore et al. (1985), and Ven der Veen (1992).

Balas (1996) introduced a class of polynomially solvable TSPs with precedence constraints: Given an integer $k > 0$ and an ordering $\sigma := (1, \ldots, n)$ of N, find a minimum cost tour, i.e. a permutation π of σ, satisfying $\pi(1) = 1$ and

$$\pi(i) < \pi(j) \text{ for all } i, j \in \sigma \text{ such that } i + k \leq j. \tag{1}$$

Problems in this class can be solved by dynamic programming in time linear in n, though exponential in k:

Theorem 1.1 *(Balas 1996) Any TSP with condition (1) can be solved in time $O(k^2 2^{k-2} n)$.*□

So for fixed k, we have a linear-time algorithm for solving TSPs — whether symmetric or asymmetric — with this type of precedence constraint. Furthermore, the conditions can be generalized by replacing the fixed integer k with city-specific integers $k(i)$, $1 \leq k(i) \leq n-i+1$, $i \in N$. In this case condition (1) becomes

$$\pi(i) < \pi(j) \text{ for all } i, j \in \sigma \text{ such that } i + k(i) \leq j, \tag{1a}$$

and a result similar to Theorem 1.1 applies (Balas 1996). Frequently occurring practical problems, like TSPs with time windows, which play a prominent role in vehicle routing (Solomon 1987), can be modeled using (1a).

Another direction in which the model can be generalized is to require a tour of $m < n$ cities subject to precedence constraints of the above type, where the choice of cities to be included into the tour is part of the optimization process. This type of problem, known as the *Prize Collecting TSP* (Balas 1989), serves as a model for scheduling steel rolling mills.

Whenever the problem to be solved can be expressed in terms of the above type of precedence constraints, the dynamic-programming procedure finds an optimal solution. On the other hand, for problems that cannot be expressed in this way, the dynamic-programming procedure finds a local optimum over a neighborhood defined by the precedence constraints. Note that this neighborhood is exponential in n, i.e. in such cases the algorithm can be used as a linear-time heuristic to search an exponential size neighborhood exhaustively.

In this paper we discuss several variants of our implementation of the dynamic-programming algorithm and our computational experience with these variants. The next section (2) briefly restates the dynamic programming procedure proposed in Balas (1996). This associates with any given TSP a network such that the tours satisfying condition (1) are in a one-to-one correspondence with the source-sink paths of the network. Section 3 describes our implementation of this procedure for the more general case of a TSP with condition (1a). We construct an auxiliary structure whose main building blocks are problem-independent and can therefore be generated and stored in advance of attacking any particular instance. Section 4 discusses the use of our procedure for solving TSPs with time windows. Section 5 addresses the related problem of TSPs with target times, introduced here for the first time. Section 6 introduces an iterative version of our procedure for the case when it is used as a heuristic for solving standard TSPs without precedence constraints. Sections 4-6 contain the results of extensive computational testing and comparisons with the best available alternative methods. Finally, Section 7 draws some conclusions.

2. The Dynamic-Programming Algorithm

The idea of solving TSPs by dynamic programming is not new. For any TSP, the cost of an optimal tour segment visiting the cities of a subset $M \subset N$ in positions 1 through $i-1$ and visiting city j in position i, can be calculated recursively as

$$C(M, i, j) = \min_{\ell \in M}\{C(M \setminus \{\ell\}, i-1, \ell) + c_{\ell j}\}.$$

The difficulty with this is that there are $\binom{n}{i-1}$ different subsets M of N of size $i-1$, and as i grows, the number of such subsets grows exponentially. It is the precedence constraints involving k that make it possible to eliminate this exponential growth in the case of TSPs satisfying condition (1), and to express each such M by a pair of subsets whose joint size is bounded by k. To be specific, for any permutation π and any position i in the tour defined by π, the set M of cities visited in positions 1 through $i-1$ can be expressed as

$$M := (\{1,\ldots,i-1\} \setminus S^+(\pi,i)) \cup S^-(\pi,i),$$

where $\{1,\ldots,i-1\}$ is the set of the first $i-1$ cities in the initial ordering σ, while $S^-(\pi,i)$ is the set of cities numbered i or higher but visited in a position lower than i, and $S^+(\pi,i)$ is the set of cities numbered below i but not visited in a position higher than i; i.e.,

$$S^-(\pi,i) := \{h \in N : h \geq i,\ \pi(h) < i\},$$
$$S^+(\pi,i) := \{h \in N : h < i,\ \pi(h) \geq i\}.$$

The reason that this representation is economical, is that for all π satisfying condition (1),

$$|S^-(\pi,i)| = |S^+(\pi,i)| \leq \lfloor k/2 \rfloor$$

i.e. the set of the first $i-1$ cities in a tour, for any i, is represented by at most k entries.

It is this characteristic of the problem at hand that allows us to overcome the "curse of dimensionality" that dynamic programming usually entails.

It is well known that dynamic-programming recursions can be expressed as shortest-path problems in a layered network whose nodes correspond to the states of the dynamic program. Accordingly, the method proposed in Balas (1996) associates with a TSP satisfying (1), a network $G^* := (V^*, A^*)$ with $n+1$ layers of nodes, one layer for each position in the tour, with the home city (city 1) appearing at both the beginning and the end of the tour, hence both as source node s (the only node in layer 1) and sink node t (the only node in layer $n+1$) of the network. The structure of G^*, to be outlined below, is such as to create a one-to-one correspondence between tours in G satisfying condition (1) (to be termed *feasible*) and $s-t$ paths in G^*. Furthermore, optimal tours in G correspond to shortest $s-t$ paths in G^*.

Every node in layer i of G^* corresponds to a state specifying which city is in position i, and which cities are visited in positions 1 through $i-1$. What makes G^* an efficient tool is that the state corresponding to any node can be expressed economically by the following four entities: i (the position in the tour), j (the city assigned to position i, i.e. having $\pi(j) = i$), $S^-(\pi,i)$ and $S^+(\pi,i)$ (the two sets defined above). The nodes of G^* can be referenced by the notation (i, j, S^-, S^+), where S stands for $S(\pi,i)$.

A layer i will be referred to as *typical* if $k+1 \leq i \leq n-k+1$. It was shown in Balas (1996) that a typical layer contains $(k+1)2^{k-2}$ nodes (the first k and last k layers contain fewer nodes). More importantly, the structure of each layer is the same, in the sense that the nodes of any layer i can be expressed in terms of i and k only. Consider, for instance, the TSP with condition (1) and $k=4$, and examine the nodes corresponding to the state where city 10 is assigned to position 12, and the cities visited in positions 1 through 11 are $\{1, \ldots 9, 12, 13\}$. This is the node $(12, 10, S^-, S^+)$, with $S^- = \{12, 13\}$, $S^+ = \{10, 11\}$. It is part of layer 12. However, every typical layer has an analogous node, which can be expressed as $(i, i-2, \{i, i+1\}, \{i-2, i-1\})$. For $i = 12$, this is the above node; and for $i = 125$, say, the corresponding node is $(125, 123, \{125, 126\}, \{123, 124\})$. This node expresses the state where city 123 is in position 125, and the cities visited in positions 1 through 124 are $\{1, \ldots, 122, 125, 126\}$.

All the arcs of G^* join nodes in consecutive layers; namely, a node in layer i, say (i, j, S^-, S^+), is joined by an arc of G^* to a node in layer $i+1$, say $(i+1, \ell, T^-, T^+)$, if the states corresponding to these two nodes can be part of the same tour. When this occurs, the two nodes are said to be *compatible*. Compatibility can be recognized efficiently due to the fact that the symmetric difference between S^- and T^-, and between S^+ and T^+, is 0, 1 or 2, depending on the relative position of i, j and S^- (see Balas 1996 for details). The cost assigned to the arc of G^* joining the two nodes is then $c_{j\ell}$, the cost of the arc (j, ℓ) in the original TSP instance. It can be shown that no node of G^* has in-degree greater than k, which bounds the number of arcs joining two consecutive layers by $k(k+1)2^{k-2}$, and the total number of arcs of G^* by $k(k+1)2^{k-2}n$. Since the complexity of finding a shortest s-t path in G^* is $O(A^*)$, the complexity of our dynamic programming stated in Theorem 1.1 follows.

Although linear in the number n of cities, our dynamic-programming algorithm, when applied to an arbitrary TSP as an improvement heuristic, finds a local optimum over a neighborhood whose size is exponential in n. Indeed, we have

Proposition 2.1 *The number of tours satisfying condition (1) is* $\Omega((\frac{k-1}{e})^{n-1})$.

Proof. Partition $\sigma \setminus \{1\}$ into consecutive subsequences of size $k-1$ (with the last one of size $\leq k-1$). Then any reordering of σ that changes positions only within the subsequences yields a tour satisfying (1), and the total number of such reorderings (i.e. feasible tours) is $((k-1)!)^{\frac{n-1}{k-1}} \geq \left(\frac{k-1}{e}\right)^{n-1}$.$\square$

When condition (1) is replaced by (1a), one can build an appropriate network $G^{*(a)}$ with properties similar to those of G^*, but with layers $V_i^{*(a)}$ that depend on the numbers $k(j)$ for the cities j such that $i - k(j) + 1 \leq j \leq i$.

3. Implementation

The computational effort involved in solving an instance of our problem breaks up into (a) constructing G^*, and (b) finding a shortest source-sink path in G^*. Constructing G^* requires (a 1) identifying the nodes, (a 2) identifying the arcs, and (a 3) assigning costs to the arcs, of G^*. Here the bulk of the effort goes into (a 1) and (a 2), since the arc costs of G^* are obtained trivially from those of the original graph G.

But (a 1) and (a 2) can be executed without any other information about the problem instance than the value of k and n. Further, as explained above, the structure and size of each typical layer, i.e. node set V_i^*, $k+1 \leq i \leq n-k+1$, is the same; and the structure of each arc set $A^* \cap (V_i^*, V_{i+1}^*)$, $k+1 \leq i \leq n-k$, (apart from costs) is also the same. Hence it suffices to generate and store one such node set and the associated set of outgoing arcs, and use it for all i to which it applies; i.e., n need not be known in advance.

Finally, note that if such a node set (layer) is generated and stored for a value K of k as large as computing time and storage space allows, then any problem instance with $k \leq K$ can be solved by repeatedly using this node set – call it W_K^* – and the associated arc set, provided that one finds a way to retrieve the smaller node set V_i^* from the larger one, W_K^*, constructed beforehand.

We will show below how this can be done efficiently. Furthermore, the technique that allows this retrieval can be generalized to the case of city-dependent constants $k(i)$, i.e. to TSPs satisfying condition (1a) and the corresponding network $G^{*(a)}$. The upshot of this is that problems of the type discussed here can be solved by constructing an auxiliary structure in advance, without any knowledge about specific problem instances, and using it to solve arbitrary problem instances whose constants k (or $k(i)$) do not exceed the size of the structure.

We will now describe the auxiliary structure and its properties that allow the retrieval of the node sets of particular problem instances. Since the problem with a single k for all cities is a special case of the problem with city-specific $k(i)$, we discuss only the latter one. Let G_K^{**} denote the network whose layers, other than the first and last, i.e. the source and sink, are identical copies of the node set W_K^*, and whose arcs, other than those incident from the source or to the sink, are those determined from W_K^* by the compatibility conditions mentioned in Section 2. For any problem instance whose constants $k(i)$ satisfy $\max_i k(i) \leq K$, any $s-t$ path in $G^{*(a)}$ corresponds to an $s-t$ path in a graph G_K^{**} with the same number of layers as $G^{*(a)}$. The reverse of course is not true, i.e. G_K^{**} contains $s-t$ paths not in $G^{*(a)}$.

One of the more delicate aspects of our implementation is the method we use to identify and remove those $s-t$ paths of G_K^{**} that do not belong to $G^{*(a)}$. To explain this method, we will illustrate it on an example.

Table 1: Nodes of W_4^* and their Compatible Pairs

		Compatible	
No.	Node Label (i,j,S^-,S^+)	Predecessors	Successors
1:	$(i,i,\emptyset,\emptyset)$	1,3,8,20	1,2,4,9
2:	$(i,i+1,\emptyset,\emptyset)$	1,3,8,20	3,5,10
3:	$(i,i-1,\{i\},\{i-1\})$	2,6,16	1,2,4,9
4:	$(i,i+2,\emptyset,\emptyset)$	1,3,8,20	6,7,11
5:	$(i,i+1,\{i\},\{i-1\})$	2,6,16	8,15
6:	$(i,i-1,\{i+1\},\{i-1\})$	4,12	3,5,10
7:	$(i,i,\{i+1\},\{i-1\})$	4,12	8,15
8:	$(i,i-2,\{i\},\{i-2\})$	5,7,19	1,2,4,9
9:	$(i,i+3,\emptyset,\emptyset)$	1,3,8,20	12,13,14
10:	$(i,i+2,\{i\},\{i-1\})$	2,6,16	16,17
11:	$(i,i+2,\{i+1\},\{i-1\})$	4,12	18,19
12:	$(i,i-1,\{i+2\},\{i-1\})$	9	6,7,11
13:	$(i,i,\{i+2\},\{i-1\})$	9	16,17
14:	$(i,i+1,\{i+2\},\{i-1\})$	9	18,19
15:	$(i,i+1,\{i\},\{i-2\})$	5,7,19	20
16:	$(i,i-2,\{i+1\},\{i-2\})$	10,13	3,5,10
17:	$(i,i,\{i+1\},\{i-2\})$	10,13	20
18:	$(i,i-1,\{i,i+1\},\{i-2,i-1\})$	11,14	20
19:	$(i,i-2,\{i,i+1\},\{i-2,i-1\})$	11,14	8,15
20:	$(i,i-3,\{i\},\{i-3\})$	15,17,18	1,2,4,9

Table 1 lists the nodes of W_K^*, a layer of G_K^{**} for $K=4$, in the notation (i,j,S^-,S^+) used before, along with their incoming and outgoing arcs shown in the form of predecessor and successor lists. Notice that the nodes are ordered so that for $h=1,\ldots,K-1$, the nodes of W_h^* are contiguous and precede those of W_{h+1}^*. Such a nested ordering of the nodes is always possible, and using it enables us to deal with any problem such that $k^* := \max_j k(j) < K$ by simply ignoring all nodes below those of $W^*_{k^*}$. On the other hand, this ordering does not make the nodes of $V_i^{*(a)}$, the i-th layer of $G^{*(a)}$, a contiguous subset of those of W^*_K, since $V_i^{*(a)}$ does not depend just on $k(i)$, but on several $k(j)$ (those such that $i-k(j)+1 \leq j \leq i$). This is illustrated in Figure 1 featuring G^{**}_K for $K = 4$ and $n = 9$, along with $G^{*(a)}$ (superimposed, with shaded nodes) for a TSP with 9 cities satisfying condition (1a) with $k(3) = 4$ and $k(j) = 3$ for all $j \neq 3$. The figure shows two $s-t$ paths in G^{**}_K, one of which is also an $s-t$ path in $G^{*(a)}$, and hence defines a feasible tour for our problem, namely $(1, 4, 2, 6, 3, 5, 7, 9, 8, 1)$, while the other one contains some nodes not in $G^{*(a)}$, hence defines a tour that is infeasible: $(1, 3, 2, 5, 7, 4, 8, 6, 9, 1)$, with $4 + k(5) \leq 7$.

In order to avoid examining paths of G^{**}_K that contain nodes not in $G^{*(a)}$, we use a test.

The *k-threshold* for a node $v := (j, S^-, S^+) \in W^*_K$, denoted kthresh$(v)$, is the smallest value of $k(j)$ compatible with the condition $v \in G^{*(a)}$. Thus $k(j) <$ kthresh(v) implies $v \notin G^{*(a)}$.

To calculate the k-threshold for a node $v := (j, S^-, S^+) \in W^*_K$, we notice that $v \in G^{*(a)}$ implies that $k(j)$ is larger than the difference between j and all higher-numbered cities visited before j in the tour. The highest-numbered city visited before j is $\max(S^-)$, the largest element of S^-, unless S^- is empty, in which case no higher-numbered city is visited before j. From this we have $k(j) > \max\{0, \max(S^-) - j\}$, so kthresh$(v) = 1 + \max\{0, \max(S^-) - j\}$. This information by itself does not tell us how to remove every node of G^{**} not in $G^{*(a)}$, but we have the following proposition, which offers the test we need.

Proposition 3.1 *Every path in G^{**} corresponding to an infeasible tour contains at least one node $v := (i, j, S^-, S^+)$, such that kthresh$(v) > k(j)$.*

Proof. Let π be the permutation for an infeasible tour T. Then there exist two cities, q and j, such that $\pi(j) > \pi(q)$ and $q \geq j + k(j)$. Let $i := \pi(j)$, (i.e. j is in the ith position). Let $v := (i, j, S^-, S^+)$ be the node used in layer i of G^{**} for the path corresponding to T.

If $q \geq i$, then $q \in S^-$, and so:

$$\begin{aligned} \text{kthresh}(v) &= 1 + \max\{0, \max(S^-) - j\} \\ &\geq 1 + \max(S^-) - j \geq 1 + q - j > q - j \geq k(j). \end{aligned}$$

If $q < i$, then $j < q < i$, and so $j \in S^+$. Since $|S^+| = |S^-|$, S^- cannot be empty, and so $\max(S^-) \geq i$ since the elements in S^- only come from the set of cities numbered greater

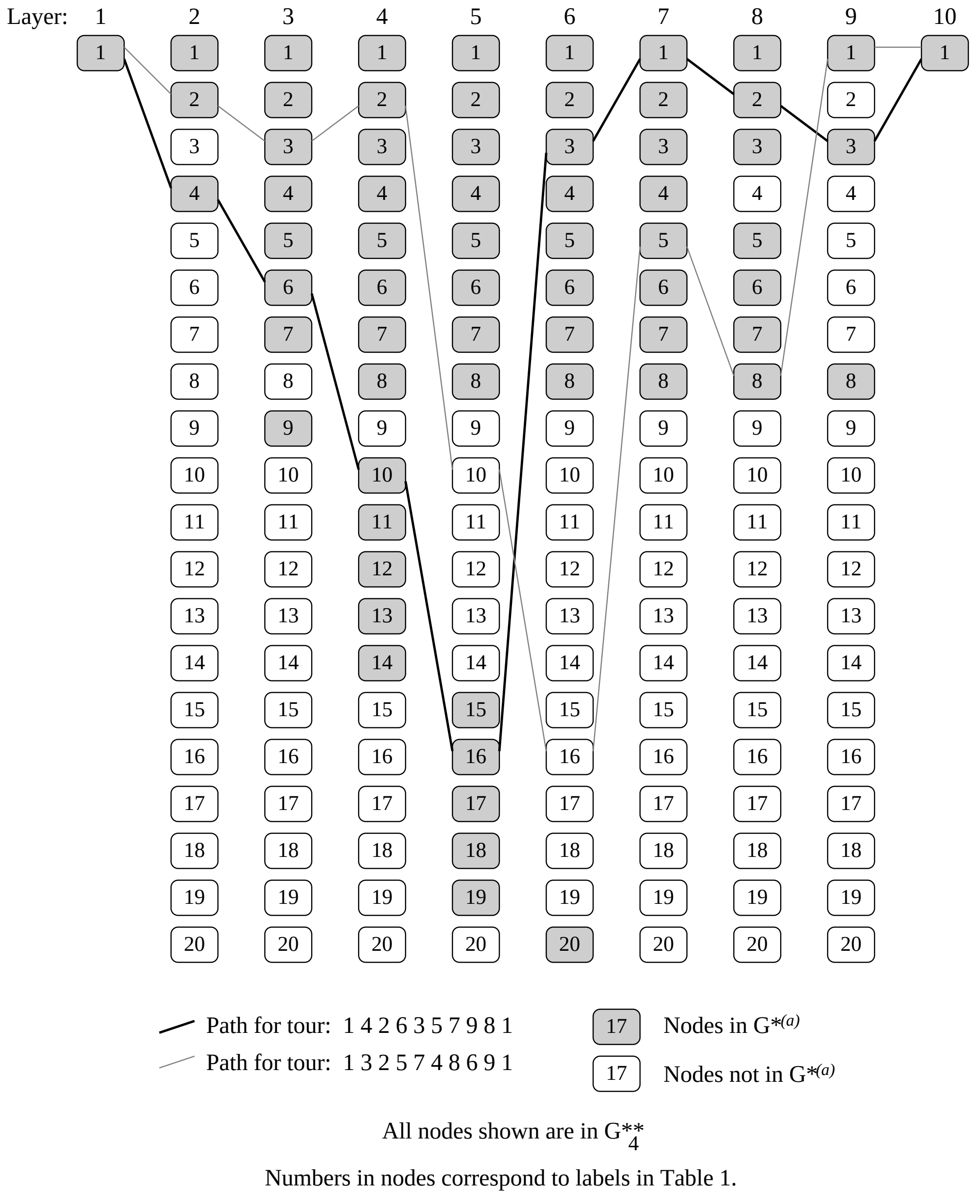
Layer: 1 2 3 4 5 6 7 8 9 10
Path for tour: 1 4 2 6 3 5 7 9 8 1
Path for tour: 1 3 2 5 7 4 8 6 9 1
Nodes in G*(a)
Nodes not in G*(a)
All nodes shown are in G**4
Numbers in nodes correspond to labels in Table 1.

Figure 1: G_K^{**} and $G^{*(a)}$ for an Instance with $n = 9$, $k(3) = 4$ and $k(j) = 3$ for $j \neq 3$

than or equal to i. This gives:

$$\text{kthresh}(v) = 1 + \max\{0,\ \max(S^-) - j\} \geq 1 + i - j > 1 + q - j > q - j \geq k(j).\square$$

In Figure 1, the k-threshold of node 16 in layer 6, which is 4, is higher than $k(j)$, which is 3. (j for node 16 in layer 6 is 4.)

Once W_K^* has been built for a certain K, *any problem instance* with $k(j) \leq K$ for all $j \in N$ can be solved to optimality (with a guarantee of optimality) by finding a shortest $s-t$ path in the subgraph $G^{*(a)}$ of $G_{k^*}^{**}$, where $G^{*(a)}$ is the auxiliary graph associated with the specific problem instance one wants to solve, and $G_{k^*}^{**}$ is the graph obtained from G_K^{**} by removing the nodes of $W_K^{**} \setminus W_{k^*}^{**}$ (with $k^* := \max\limits_j k(j)$). Extracting the nodes and arcs of $G^{*(a)}$ from those of $G_{k^*}^{**}$, and putting the appropriate costs on the arcs, does not increase the complexity of finding a shortest $s-t$ path in $G^{*(a)}$, which remains linear in the number of arcs of $G_{k^*}^{**}$.

As stated in Section 1, the time complexity of our algorithm is linear in n, but exponential in k. As is often the case with dynamic-programming algorithms, however, the real bottleneck for the efficiency of our algorithm is its space complexity. There are two aspects to this issue: the complexity of storing the auxiliary structure generated in advance, and the complexity of storing the information needed to construct an optimal tour. We address them in turn.

At first glance, it would seem that the space required to store the auxiliary structure G_K^{**}, even though we restrict ourselves to a single layer of nodes and its incident arcs, is $O(K(K+1)2^{K-2})$. In reality, we can take advantage of the arc structure, which is such that every node appears on exactly one distinct predecessor list (possibly repeated many times), and the same is true for the successor lists. For instance, in Table 1, node 8 appears on the predecessor list for nodes 1, 2, 4 and 9, but for each of these nodes the predecessor list is the same, namely, 1, 3, 8 and 20. Therefore we store this list once, along with four pointers to it from nodes 1, 2, 4 and 9. This allows us to store the graph in space $O((K+1)2^{K-2})$.

As to the construction of an optimal tour, here the linear dependence on n, which is a blessing in terms of time complexity, becomes a curse in terms of space complexity. When we calculate the cost of a shortest path from the source to a given node of G_K^{**}, we need only the costs of the shortest paths to the nodes of the previous layer; so traversing the auxiliary graph to find the cost of a shortest $s-t$ path requires only two "levels" of additional storage for these costs, a space requirement of an additional $2(K+1)2^{K-2}$. However, in order to recover the shortest path we have identified, we must trace our way backwards through G_K^{**},

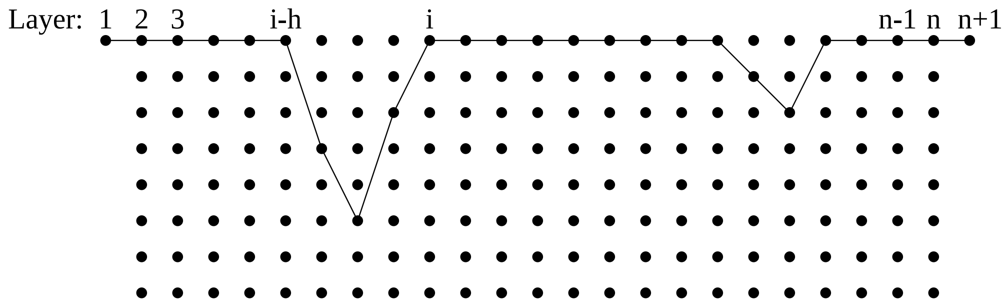

Figure 2: A Path Through G_K^{**} for a Solution Not Very Different from the Initial Tour

which requires that we store, at every node of *every* layer, which node was the predecessor with least cost; thus we have a linear space dependence on n. Fortunately, there is a way to circumvent this feature, by using the typical pattern of the solutions generated by our procedure.

The easiest way to explain this is by first assuming that the dynamic-programming procedure finds no improvement over the starting tour, i.e. the initial ordering σ. In this case a shortest $s - t$ path in G_K^{**} will contain the first node of every layer, corresponding to $(i, j, S^-, S^+) = (i, i, \emptyset, \emptyset)$. If our algorithm finds some local improvements but otherwise leaves the initial sequence unchanged, the shortest $s - t$ path in G_K^{**} will traverse the first node of each layer in those portions where there was no change. We can then store the shortest $s - t$ path by storing just those path segments that leave the top nodes, which we will call *deviating path segments* (DPS); see Figure 2. On the forward pass, a DPS is detected at its right end, when the optimal arc into the first node of layer i, say, does not come from the first node of layer $i - 1$. At this point the DPS is traced back to the point where it again reaches the top level, say at layer $i - h$, and is stored along with the value i. When we reconstruct our tour, we trace backwards, inserting DPSs as needed. For example, if we encounter three DPSs, one from layer 10 back to layer 6, one from layer 20 back to layer 15, one from layer 21 back to layer 18, we reconstruct our tour by first including the last DPS (from 21 to 18) and then including the latest DPS whose right end is at layer 18 or earlier, which would be the one from layer 10 back to layer 6. We would skip the DPS from layer 20 to layer 15, since it would not be along the optimal path.

Let H denote the length of the longest DPS. The above-described procedure then adds at most H times the number of DPSs to our running time, but it reduces the space requirement from $(K+1)2^{K-1}n$ to $(K+1)2^{K-1}H + Hn$, where the final term is the space needed to store at most n DPSs. For the applications of our algorithm as a heuristic for a general TSP (see Section 6), where problem sizes have reached into the thousands, we have only encountered one instance where $H > 3K$, and in this case $H < 6K$. This allows us to choose a value of H proportional to K. Note that if we allow for a value of H that is insufficient, this can be detected by noticing that we run out of space before the DPS returns to the first level. In such a case, we know not to attempt to reconstruct the tour after obtaining the optimal value, and we restart the algorithm with a larger allowance for H.

Our code can be found through the internet at `http://www.andrew.cmu.edu/~neils/TSP`.

4. Time-Window Problems

Consider a scheduling problem with sequence-dependent setup times and a time window for each job, defined by a release time and a deadline. Or consider a delivery problem, where a vehicle has to deliver goods to clients within a time window for each client. These types of problems have been modeled as TSPs with time windows (TSPTW), either symmetric or asymmetric (see, for instance, Baker 1983 or Solomon 1987).

In such a problem, we are given a set N of cities or customers, a distance d_{ij}, a travel time t_{ij} between cities i and j for all pairs $i, j \in N$, and a time window $[a_i, b_i]$ for each city i, with the interpretation that city i has to be visited (customer i has to be serviced) not earlier than a_i and not later than b_i. If city i is reached before a_i, there is a waiting time w_i until a_i; but if city i is visited after b_i, the tour is infeasible. The objective is to find a minimum-cost tour, where the cost of a tour may be the total distance traveled (in which case the waiting times are ignored) or the total time it takes to complete the tour (in which case the waiting times w_i are added to the travel times t_{ij}).

Here we discuss how the TSPTW can be solved by our approach. We first consider the total-time criterion. Given an initial ordering of the cities, the time windows can be used to derive precedence constraints and the problem can be treated as a TSP with condition (1a): city i has to precede in any feasible tour any city j such that $a_j + t_{ji} > b_i$. If j_0 is the smallest index such that $a_j + t_{ji} > b_i$ for all $j \geq j_0$, then we can define $k(i) := j_0 - i$. Doing this for every i, we obtain a TSP with condition (1a) whose feasible solutions include all feasible tours for the TSP with time windows $[a_i, b_i]$. Since the converse is not necessarily true, the construction of a shortest path in G^{**} must be amended by a feasibility check:

while traversing G^{**}, paths that correspond to infeasible tours (with respect to the upper limit of the time windows) must be weeded out whereas paths that violate a lower limit of a time window must have their lengths (costs) increased by the waiting time. This check is a simple comparison that does not affect the complexity of the algorithm.

The initial ordering can be generated, for instance, by sorting the time windows by their midpoint. Table 2 shows an example of this, along with the constants $k(i)$ derived from this ordering. To keep things simple we take all t_{ji} to be zero. This is not necessarily an optimal ordering for minimizing the largest $k(i)$, but it appears to be a good start. From this ordering, a local search can be used to lower the largest $k(i)$ by moving a time window to a different location (minimizing the largest $k(i)$ may be NP-complete). In the example, moving the time window for city 3 after that of city 4 reduces the largest $k(i)$ by 1 (see Table 3). The important thing to bear in mind is that *any* initial ordering can be used to derive precedence constraints from the time windows, and the resulting constants $k(i)$ depend not only on the time windows, but also on the initial tour. In fact, it is not even necessary for the initial ordering to represent a feasible tour; given that the values $k(i)$ derived by the above procedure define precedence constraints satisfied by every feasible tour.

Table 2: Constructing Precedence Constraints from Time Windows

i	Time Window	$k(i)$
1	Home City	1
2	[.]	3
3	[.]	4
4	[.]	1
5	[.]	3
6	[.]	2
7	[.]	1

Now since the objective is the minimization of the total time needed to complete the tour, the time spent in waiting is simply added to the time spent in traveling as a shortest path is constructed in G^{**}, and thus waiting can be taken into account without any complication: an optimal solution to the TSP with condition (1a), if feasible with respect to the time windows, defines an optimal tour for the TSPTW.

Table 3: Improving the Bound on the Largest $k(i)$

i	Time Window	$k(i)$
1	Home City	1
2	[.]	3
3	[.]	2
4	[.]	3
5	[.]	3
6	[.]	2
7	[.]	1

However, the literature (see, for instance, Baker 1983 or Solomon 1987) typically considers the TSPTW with the objective of minimizing the distance traveled. This objective is not affected by waiting time and is therefore different from the first objective even when $t_{ij} = d_{ij}$ for all i, j. In order to model this second objective, it is necessary to keep track both of distance and travel time (with waiting time included). However, this is not enough, since a tour that waits in one place to gain an advantage in distance, may be unable to satisfy a time window later in the tour. Figure 2 illustrates this point. The route 1-2-3-4 has distance 6, but requires a time of 9 because there is a wait at city 2. This wait prevents the route from continuing to city 5 before its window closes. The route 1-3-2-4 has distance 8, but also has a total time of 8, so this route can continue through city 5. If the algorithm is not modified to keep this other partial tour, this solution will not be found.

This dilemma was handled in Dumas et al. (1995) by adding a time parameter to the state. Unfortunately, this addition can cause the number of potential states to grow with the length of the arcs in the problem. We address this issue by storing only non-dominated time-distance pairs, and when a state has multiple non-dominated pairs, we clone the corresponding node. To keep our polynomial bound, we restrict the number of copies (clones) that can be stored for any state by a constant parameter q, which we call the *thickness* of the auxiliary graph. Note that our procedure uses the same auxiliary structure G_K^{**} built in advance as in the standard case discussed in Section 3, and the cloning of nodes, when needed, is done in the process of constructing a shortest source-sink path in G^{**}.

If we never generate more than q nondominated time-distance pairs, the algorithm is

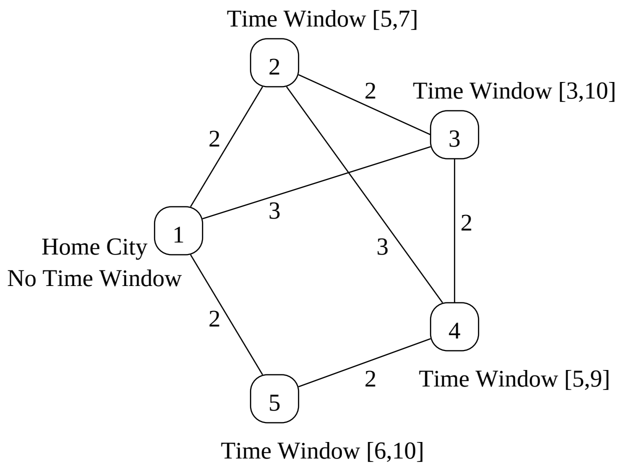

Figure 3: Illustrating the Difficulty of Minimizing Distance Instead of Time

guaranteed to find an optimal solution to the TSPTW with the distance objective. If more than q pairs are generated, we store only the q best with respect to distance. When this happens, we can no longer guarantee optimality; but as the computational results below show, optimal solutions are often found with a graph thickness smaller than that required for guaranteed optimality.

The largest value of $k(i)$ that we were able to accommodate on the computers available to us was 17. As discussed in Section 3, this limit is given by space rather than time considerations. In the case where the precedence constraints implied by the time windows require values of $k(i)$ too large to make the algorithm practical, the algorithm can still be run by truncating the auxiliary graphs to smaller values of $k(i)$; but once again, in this case we can no longer guarantee an optimal solution. In such situations, using the initial sequence mentioned above may not yield good results; but starting from the heuristic solution of some other method, the algorithm can often improve that solution. Also, using some other heuristic between multiple applications of our algorithm has improved our results sometimes.

Next we discuss our computational experience with time-window problems. All of our results in this and the following sections were obtained with a code written in C and run on a Sun Ultra I Workstation with 64MB of memory. Where our algorithm could not guarantee optimality, a simple Or-opt heuristic (Or 1976) was applied, followed by renewed application of our algorithm.

Using a method similar to that given by Baker (1983), Tama (1990) has generated a number of instances of one-machine scheduling problems with set-up times and minimum makespan objective, formulated as asymmetric TSPs with Time Windows. He kindly made available to us 13 problem instances, eight 20-city and five 26-city problems, which he was unable to solve using cutting planes followed by a branch and bound algorithm (Tama 1990). We ran our algorithm on these problems, using $K = 13$ and $K = 16$, with the following outcome: Of the eight 20-city problems, 7 were *proved* to be infeasible, and the remaining 1 was solved to optimality. Of the five 26-city problems, 1 was *proved* to be infeasible, 3 were solved to optimality, and the remaining 1 was "solved" without a guarantee of optimality because the required value of K was too high (see Table 4).

Table 4: Asymmetric TSPs with Time Windows (One-Machine Scheduling)

			Solution Value (seconds)	
	Problem Name	Required K	$K = 13$	$K = 16$
(20-city problems)	p192358	13	Infeasible* (0.36)	
	p192422	10	Infeasible* (0.49)	
	p192433	12	Infeasible* (0.58)	
	p192452	10	Infeasible* (0.47)	
	p192572	11	Infeasible* (0.48)	
	p192590	12	Infeasible* (0.52)	
	p193489	13	Infeasible* (0.96)	
	p194450	9	936* (0.86)	
(26-city problems)	p253574	16	Infeasible (14.92)	Infeasible* (8.83)
	p253883	18	1188 (3.94)	1188 (36.45)
	p254662	16	1140 (4.22)	1140* (18.97)
	p255522	13	1373* (1.81)	
	p256437	13	1295* (1.77)	

Notes:
*indicates that a guarantee of optimality (or a guarantee of the infeasibility of the problem) was found.

As to symmetric TSPs with Time Windows, thirty instances of the RC2 problems proposed by Solomon (1987) that were first studied by Potvin and Bengio (1993) and then by Gendreau et al. (1995) were run with our algorithm, using $q = 15$ and $K = 12, 15$ with the results shown in Table 5. These problems, kindly provided to us by M. Stan, originate in vehicle-routing problems (also available at `http://dmawww.epfl.ch/~rochat/rochat_data/solomon.html`) and are the outcome of a first phase procedure in which clients are allocated to vehicles (the second phase task being the routing). The extension in the

Table 5: Symmetric TSPs with Time Windows (Vehicle Routing)

Problem	n	Potvin and Bengio (1993)	Gendreau et al. (1995)		Required K	Our Results (CPU seconds) $K=12$		$K=15$	
rc206.1	4	117.85	117.85	(0.03)	3	117.85*	(0.29)		
rc207.4	6	119.64	119.64	(0.16)	5	119.64*	(0.50)		
rc202.2	14	328.28	304.14	(7.76)	13	304.14*	(2.92)		
rc205.1	14	381.00	343.21	(3.58)	7	343.21*	(2.02)		
rc203.4	15	330.33	314.29	(9.88)	12	314.29*	(3.58)		
rc203.1	19	481.13	453.48	(13.18)	13	453.48	(12.19)	453.48$^+$	(96.75)
rc201.1	20	465.53	444.54	(9.86)	6	444.54*	(2.49)		
rc204.3	24	455.03	460.24	(37.86)	22	475.59	(61.28)	466.21	(639.93)
rc206.3	25	652.86	591.20	(22.30)	14	574.42	(17.40)	574.42$^+$	(133.75)
rc201.2	26	739.79	712.91	(23.06)	7	711.54*	(3.33)		
rc201.4	26	803.55	793.64	(19.21)	7	793.63*	(3.27)		
rc205.2	27	796.57	755.93	(23.61)	11	755.93*	(5.99)		
rc202.4	28	852.73	793.03	(36.61)	18	809.29	(23.00)	799.18	(212.46)
rc205.4	28	797.20	762.41	(21.13)	12	760.47*	(5.25)		
rc202.3	29	878.65	839.58	(22.16)	15	837.72	(9.54)	837.72*	(45.46)
rc208.2	29	584.14	543.41	(67.06)	28	533.78	(59.78)	533.78	(905.16)
rc207.2	31	721.39	718.09	(52.55)	21	701.25	(61.38)	701.25	(703.03)
rc201.3	32	839.76	795.44	(46.91)	7	790.61*	(4.51)		
rc202.1	33	844.97	772.18	(33.65)	20	773.16	(18.42)	772.33	(223.30)
rc203.2	33	843.22	784.16	(49.26)	24	809.14	(33.06)	808.78	(404.42)
with initial tour of length 789.04 from Stan (1997)					32	784.16	(20.84)	784.16	(246.66)
rc204.2	33	686.56	679.26	(53.25)	28	662.16	(77.22)	681.85	(830.17)
rc207.3	33	750.03	684.40	(55.91)	23	746.27	(113.44)	697.93	(1128.64)
with initial tour of length 684.40 from Stan (1997)					32	684.40	(50.31)	684.40	(545.69)
rc207.1	34	797.67	741.53	(47.53)	19	735.82	(70.87)	735.82	(622.47)
rc205.3	35	909.37	825.06	(138.38)	23	825.06	(42.78)	825.06	(384.03)
rc208.3	36	691.50	660.15	(87.25)	35	655.50	(122.29)	659.07	(2472.65)
with initial tour of length 654.27 from Stan (1997)					35	654.27	(197.54)	649.11	(2420.97)
rc203.3	37	911.18	842.25	(50.90)	27	could not find feasible solution			
with initial tour of length 842.03 from Stan (1997)					36	834.90	(30.14)	834.90	(373.34)
rc206.2	37	850.47	842.17	(108.03)	16	828.06	(33.92)	828.06	(315.10)
rc206.4	38	893.34	845.04	(100.66)	16	831.67	(46.81)	831.67	(418.59)
rc208.1	38	812.23	799.19	(88.20)	37	852.24	(53.81)	793.61	(1141.24)
rc204.1	46	923.86	897.09	(84.41)	39	could not find feasible solution			
with initial tour of length 878.76 from Stan (1997)					45	878.64	(74.82)	878.64	(1061.16)

Notes:
$q = 15$ for our results
CPU times for Gendreau's solutions were obtained using a SPARCserver 330 according to Gendreau et al. (1995).
*value guaranteed to be optimal.
$^+$453.48 was verified to be optimal for rc203.1 with K=13 and q=25 in 15.56 seconds.
$^+$574.42 was verified to be optimal for rc206.3 with K=14 and q=20 in 38.29 seconds.

names of the problems indicates the route. Here the objective was to minimize total distance, i.e. occasionally nodes had to be cloned as explained above. Neither Potvin and Bengio (1993) nor Gendreau et al. (1995) could guarantee the optimality of a solution even if their solution was optimal. The entry for "Required K" in Table 5 indicates what value of K would be needed for a guarantee of optimality.

As can be seen from Table 5, our algorithm solved 14 of these problems with a guarantee of optimality. Of these 14, Potvin et al. (1995) had solved two to optimality and Gendreau and Bengio (1993) had solved nine to optimality. Of the remaining sixteen problems, our algorithm fared better than Potvin et al. (1995) thirteen times, equaled Gendreau and Bengio (1993) once, and fared better than Gendreau and Bengio (1993) eight times. On the five problems for which we obtained through personal communication (Stan, 1997) feasible tours, we then used these as initial tours, and we improved those tours in four of these five instances. Note that the solutions we received from Stan (1997) were often slightly different than the solution portrayed in the results found in Gendreau et al.(1995). As mentioned earlier, when a guarantee of optimality was not found, a heuristic based on the Or-opt heuristic (Or 1976) was applied once, and then our algorithm was applied again. This is why the computation time can vary widely for problems of roughly the same size (e.g. rc202.1 and rc202.3). Furthermore a worse initial solution might lead to a better final solution if the Or-opt routine improves the worse solution (e.g. rc204.2).

Table 6: Symmetric TSPs with Time Windows (Vehicle Routing)

		Scale	Required	Our Results (CPU seconds)			
Problems Combined	n	Factor	K	$K = 12$		$K = 14$	
rc201.1 & rc201.2	45	2.0	13	1198.74	(31.54)	1198.74$^+$	(129.76)
rc201.1 & rc201.4	45	2.2	12	1263.57*	(13.43)		
rc201.1 & re201.3	51	2.2	13	1420.77	(28.44)	1420.77*	(65.43)
rc201.2 & rc201.4	51	2.2	13	1512.79	(29.64)	1512.79*	(65.40)
rc201.2 & rc201.3	57	2.0	14	1531.20	(35.93)	1531.20*	(83.28)
rc201.3 & rc201.4	57	2.0	14	1699.69	(42.23)	1699.69$^+$	(176.40)

Notes:
$q = 15$
*value guaranteed to be optimal.
$^+$1198.74 was verified to be optimal for rc201.1-.2 with $K = 13$ and $q = 18$ in 40.10 seconds
$^+$1699.69 was verified to be optimal for rc201.3-.4 with $K = 14$ and $q = 18$ in 107.77 seconds

Since all of the problems in Table 5 are quite small (average size of 28 cities), we generated larger problems by combining the cities of two problems from the first class (rc201.x). To make the problem feasible for a single vehicle, every time window was multiplied by a scaling factor. If the problem was feasible using a scaling factor of 2, this was used. Otherwise, the scaling factor used was 2.2. Increasing the scaling factor does not affect the time-window structure in terms of overlaps, but may decrease the number of precedence constraints and thereby require a larger K. Table 6 shows the results of these problems, having an average size of 51 cities. Optimal solutions were found in every case with $K = 12$, but for many, optimality could not be guaranteed with K less than 14.

Asymmetric time window problems of a much larger size (Ascheuer et al. 1997) were recently made available at `http://www.zib.de/ascheuer/ATSPTWinstances.html`. They are based on real-world instances of stacker-crane-routing problems. The problem names in principle contain the information on the problem size; where n differs from the number in the name, this is presumably due to inconsistencies in accounting for the dummy node. Even though problem sizes extend beyond 200 nodes, because the time windows were tight, the value of K required by these problems was small. This makes the problem difficult for the branch and cut code in Ascheuer et al. (1997), which could only solve one of the problems with more than 70 nodes when allowed to run for $5 \times \lceil \frac{n}{100} \rceil$ CPU hours, but makes the problem easy for our algorithm. The largest instance for which we could guarantee optimality was a problem with 151 nodes. Tables 7 and 8 list our results with a value of $q = 15$, compared with the lower bounds in Ascheuer et al. (1997) in those cases where optimality was not guaranteed with our algorithm.

As can be seen from these tables, our procedure performs quite remarkably on this set of problems. For instances with $n \leq 45$, guaranteed optimal solutions were found in less than a minute in all but three cases. In two of those three cases optimal solutions were found without a proof of optimality, while in the third case a solution was found within less than 1% of optimality, in less than four minutes. For problems with $55 \leq n \leq 151$, guaranteed optimal solutions were found in all but three cases in less than one minute.

We conclude that for time-window problems our approach clearly advances the state of the art.

Table 7: Asymmetric TSPs with Time Windows (Stacker-Crane Routing)

Problem	n	Required K	Our Results (CPU seconds) $K = 12$		$K = 15$		Lower Bound from Ascheuer et al. (1997)
rbg010a	11	5	149*	(0.95)			
rbg017	16	7	148*	(2.51)			
rbg017.2	16	11	107^1	(9.80)			
rbg016a	17	6	179*	(1.72)			
rbg016b	17	12	142*	(2.71)			
rbg017a	18	12	146*	(3.23)			
rbg019a	20	4	217*	(2.03)			
rbg019b	20	9	182*	(3.42)			
rbg019c	20	13	190^2	(7.64)			
rbg019d	20	5	344*	(2.39)			
rbg021	20	13	190^2	(7.82)			
rbg021.2	20	13	182^2	(9.00)			
rbg021.3	20	13	182^3	(9.60)			
rbg021.4	20	14	179^3	(11.52)			
rbg021.5	20	15	169	(12.35)	169^1	(127.97)	
rbg021.6	20	15	147	(14.98)	134^1	(161.66)	
rbg021.7	20	18	134	(20.79)	133	(224.54)	132*
rbg021.8	20	19	133	(23.43)	132	(267.26)	132*
rbg021.9	20	19	133	(24.26)	132	(285.19)	132*
rbg020a	21	13	210^2	(3.59)			
rbg027a	28	19	268	(11.28)	268	(137.66)	268*
rbg031a	32	8	328^1	(11.13)			
rbg033a	34	9	433*	(5.66)			
rbg034a	35	10	401^1	(18.03)			
rbg035a	36	9	254*	(7.67)			
rbg035a.2	36	24	187	(73.78)	169	(650.31)	166*
rbg038a	39	10	466*	(8.64)			
rbg040a	41	10	386^1	(20.08)			
rbg041a	42	11	402^1	(24.57)			
rbg042a	43	13	411^4	(47.38)			
rbg048a	49	35	none found		none found		456
rbg049a	50	29	none found		486	(281.50)	420
rbg050a	51	34	441	(107.05)	431	(1123.35)	414*
rbg050b	51	30	none found		518	(360.97)	453
rbg050c	51	30	none found		none found		508

Notes:
$q = 15$
*value guaranteed to be optimal
[1]value verified to be optimal with $q = 20$ (CPU sec)
rbg017.2 (5.88), rbg021.5 (76.42), rbg021.6 (92.94), rbg031a (7.27), rbg034a (11.57), rbg040a (13.01), rbg041a (15.63)
[2]value verified to be optimal with required K
rbg019c (8.52), rbg020a (8.12), rbg021 (8.68)
[3]value verified to be optimal with required K and $q = 20$
rbg021.2 (11.35), rbg021.3 (11.83), rbg021.4 (29.09)
[4]rbg042a was verified optimal with $K = 13$ and $q = 30$ (61.94)

Table 8: Asymmetric TSPs with Time Windows (Stacker-Crane Routing)

		Required	Our Results (CPU seconds)				Lower Bound
Problem	n	K	$k = 12$		$K = 15$		from Ascheuer et al. (1997)
rbg055a	56	10	814[1]	(25.56)			
rbg067a	68	10	1048[1]	(29.14)			
rbg086a	87	9	1051*	(18.70)			
rbg088a	89	10	1153*	(22.01)			
rbg092a	93	10	1093[1]	(48.13)			
rbg125a	126	10	1409*	(31.93)			
rbg132	131	9	1360[1]	(61.28)			
rbg132.2	131	17	1085	(137.80)	1083	(1135.49)[6]	1069
rbg152	151	10	1783*	(37.90)			
rbg152.2	151	17	1628	(216.40)	1628	(1738.56)[6]	1598
rbg152.3	151	24	1556	(316.50)	1542	(2765.42)[6]	1526
rbg172a	173	16	1799	(110.64)	1799	(812.17)[6]	1788
rbg193	192	15	2414	(128.82)	2414	(807.50)[4]	2388
rbg193.2	192	21	2018	(408.29)	2022	(2138.72)[4]	1969
rbg201a	202	16	2189	(126.69)	2189	(809.71)[4]	2158
rbg233	232	15	2689	(153.12)	2689	(975.22)[4]	2635
rbg233.2	232	21	2190	(446.88)	2193	(2505.86)[4]	2146

Notes:

$q = 15$ unless specified

*value guaranteed to be optimal

[1]value verified to be optimal with $q = 20$ (CPU sec)
rbg055a (16.21), rbg067a (18.83), rbg092a (30.33), rbg132 (39.11)

[4]value obtained with $q = 4$

[6]value obtained with $q = 6$

5. The Traveling Salesman Problem with Target Times

A new variant of the TSPTW, which we call the *traveling salesman problem with target times* (TSPTT), defines a target time for each city, rather than a time window. The objective is to minimize the maximum deviation between the target time and the actual service time over all cities. A secondary objective (subject to optimality according to the first objective) may be the minimization of the total time needed to complete the tour. Applications of this problem include delivery of perishable items (such as fresh fruit) to places without storage facilities, or routing a repair vehicle to customers with busy schedules who do not like waiting.

We can solve the TSPTT to optimality with our algorithm, by first constructing windows of a fixed size d centered at each city's target time, and then using binary search to find the smallest d (to within a predetermined accuracy) for which the problem is feasible. This of course implies repeated applications of the algorithm, but the number of repetitions is bounded by $\log_2 d$, where d is the largest time window one needs to consider. The auxiliary structure built in advance is again the same as in the standard case, and for each value of d the problem is solved as a TSPTW with the objective of minimizing the total time needed for completing the tour.

Table 9 shows the results of using our algorithm on a set of TSPs with Target Times, where $d_{\min}$ is the smallest d for which the problem is feasible, and cost is the total time needed to complete the tour. To generate instances of this problem, we used the same data studied by Potvin and Bengio (1993) and Gendreau et al. (1995), and used the time-window midpoints for the target times. As can be seen from Table 9, in many cases the optimal solution was found with a value of K much smaller than that needed to guarantee optimality, which may indicate that some solutions given to problems without a guarantee of optimality are optimal. In cases where no guarantee was achieved, the exact value of K needed for a guarantee is not known. The value in the table is the K required to solve the problem optimally with the best known feasible d. This is an upper bound on the K required to find the optimal d, i.e. $d_{\min}$, because decreasing the time-window diameter may introduce additional precedence constraints, which in turn may require a smaller (but not a larger) K. Arc costs for the problems in Table 9 ranged in most cases from 10 to 100 units, and 1 time unit is required to travel one distance unit.

As the table shows, our procedure found solutions guaranteed to be within 0.01 of optimality in 19 of 30 instances, in most cases in a few seconds, in other cases in a few minutes.

Table 9: Symmetric TSPs with Target Times

		Req'd	Results for $K = 10$			Results for $K = 14$			Results for $K = 17$		
Problem	n	K	$d_{\min}$	cost	sec.	$d_{\min}$	cost	sec.	$d_{\min}$	cost	sec.
rc206.1	4	3	22.08*	227.04	1						
rc207.4	6	1	12.08*	309.85	1						
rc202.2	14	8	118.12*	552.18	2						
rc205.1	14	3	53.03*	473.12	1						
rc203.4	15	11	211.98	598.25	2	211.98*	598.25	30			
rc203.1	19	14	220.23	610.81	3	220.23*	610.81	56			
rc201.1	20	5	81.92*	611.10	1						
rc204.3	24	$\leq$23	321.52	659.22	7	321.52	659.22	136	321.52	659.21	1122
rc206.3	25	11	145.93	697.63	3	145.93*	697.63	51			
rc201.2	26	6	99.56*	875.73	1						
rc201.4	26	7	114.50*	891.93	2						
rc205.2	27	12	186.27	816.47	5	186.27*	816.47	72			
rc202.4	28	17	244.41	925.77	6	237.26	900.66	103	237.26*	900.66	871
rc205.4	28	11	159.55	883.81	4	159.55*	883.81	64			
rc202.3	29	8	134.62*	889.27	4						
rc208.2	29	$\leq$21	301.41	779.39	6	267.46	762.42	113	267.46	762.42	1072
rc207.2	31	$\leq$20	241.18	798.50	7	241.18	798.50	126	241.18	798.50	1348
rc201.3	32	5	93.53*	869.68	2						
rc202.1	33	$\leq$19	277.38	875.97	7	254.08	864.32	139	254.08	864.32	1337
rc203.2	33	$\leq$25	387.57	974.42	8	387.57	974.42	174	383.87	972.58	1976
rc204.2	33	$\leq$30	453.11	781.41	9	434.94	793.15	221	427.36	789.36	2332
rc207.3	33	$\leq$18	275.88	896.67	8	259.51	880.36	163	259.51	880.36	1793
rc207.1	34	17	234.99	906.66	7	234.99	906.66	139	234.99*	906.66	1085
rc205.3	35	15	240.43	965.58	7	240.43	965.58	135	240.43*	965.58	1117
rc203.3	37	$\leq$27	535.94	975.11	9	482.71	957.49	221	403.46	950.40	2452
rc206.2	37	13	202.38	902.06	8	202.38*	902.06	136			
rc206.4	38	14	204.53	941.35	8	204.53*	941.35	156			
rc208.1	38	$\leq$28	449.78	936.43	10	406.04	914.56	242	375.77	899.40	2703
rc208.3	36	$\leq$24	297.89	824.90	8	285.85	818.67	177	285.85	818.67	2082
rc204.1	46	$\leq$43	770.78	1078.49	14	658.42	1022.31	335	623.43	966.05	4101

Notes:
*value guaranteed to be within .01 of the optimum.

6. Local Search for Arbitrary TSPs

If applied as a heuristic for an arbitrary TSP, our procedure finds a local optimum over the neighborhood of the starting tour defined by the precedence constraint (1). If applied repeatedly until there is no change, our procedure finds a tour that is a local optimum over its own neighborhood defined by (1). Unlike the standard interchange heuristics like 2-opt, 3-opt, etc., where computing time is proportional to the size of the neighborhood, our procedure finds a local optimum over a neighborhood of size exponential in n, in time linear in n. (See Johnson and McGeoch 1997 for a comprehensive survey of TSP heuristics.)

In our first tests of the algorithm run as a heuristic for arbitrary TSPs, we experimented with some 500-city asymmetric TSPs that we produced using the *genlarge* problem generator, which Repetto (1996) kindly gave to us and which can generate instances of various kinds (like those listed in the notes to Table 10). Repetto solved these problems with an open-ended heuristic approach, where he produces an initial tour by randomly choosing one of the nearest two neighbors, and then applies his implementation (Repetto 1994) of the Kanellakis and Papadimitriou (1980) heuristic, an adaptation to asymmetric TSPs of the Lin-Kernighan (1973) heuristic for the symmetric TSP. We ran Repetto's code for 1200 seconds and used the best solution found as the initial tour for our algorithm, which was then run with $K = 10, 14$ and 17. The results are shown in Table 10.

Table 10: Random versus Almost-Euclidean Asymmetric TSPs:
Percent over Held-Karp Lower Bound

	Kanellakis -	Our Results (seconds)		
Problem Name	Papadimitriou	$K = 10$	$K = 14$	$K = 17$
500.aa	57.74	no improvement		
500.ba	29.45	no improvement		
500.ca	4.67	4.61 (13)	4.33 (290)	4.26 (3641)
500.da	3.96	3.29 (14)	2.99 (298)	2.85 (3605)

Notes:
500.aa is a truly random asymmetric TSP.
500.ba is a random symmetric TSP perturbed 2% for asymmetry.
500.ca is a random Euclidean TSP perturbed 2% for asymmetry.
500.da is a random clustered Euclidean TSP perturbed 2% for asymmetry.

While no improvements were obtained for the truly random asymmetric and the randomly perturbed symmetric problems, there were repeated improvements for the random

Euclidean and clustered random Euclidean problems perturbed for asymmetry, even though these starting solutions were much better than those for the random instances.

The reason why our algorithm tends to perform better on almost-Euclidean problems may have to do with the fact that for such problems, if cities i and j are near each other (not necessarily adjacent) in a tour generated by a good heuristic, then there is a high probability that the arc connecting these cities has low cost. For arbitrary random instances, there is no such correlation. Since our algorithm looks at the arcs specified by the precedence constraint applied to the starting tour, for Euclidean instances these arcs would generally all have a reasonably low cost, whereas for non-Euclidean problems, many of these arcs would be of no interest to us. In some sense, there is more overlap of the neighborhood our algorithms examines and the space of "good tours" when the problem is Euclidean. Geographic problems are even better suited for our algorithm because cities tend to cluster in metropolitan areas, which implies precedence constraints of the type (1).

Early tests pointed out a basic weakness in the iterative aspect of our algorithm — and of course this is a crucial issue when it comes to the use of our procedure as a heuristic. In our experience, a local optimum is usually found within two to three applications of the procedure, if not on the first one. After that, some other starting point must be chosen if the procedure is to be reapplied. Starting every time from a random tour is inefficient. However, after a while we found a better way of iterating our algorithm; namely, by contracting some arcs of the tour found by the procedure, and then reapplying it to the shrunken instance: this has the effect of extending the reach of the precedence constraints beyond the initial value of K. To do this, we mark the arcs selected for contraction (more below on how the selection is made), and then contract each set of contiguous marked arcs into a single supernode S in the case of an asymmetric TSP, or into a pair of supernodes S', S'' in the case of a symmetric TSP. Suppose the sequence of arcs to be contracted is $(i_1, i_2), \ldots, (i_{m-1}, i_m)$. In the case of an asymmetric TSP, the cost of the arc (j, S) from some node j to S is the cost of (j, i_1), whereas the cost of (S, j) is that of (i_m, j). (Contracting a single arc is of course a special case, with $m = 2$.) In the case of a symmetric TSP, we want the contraction to allow for traversing the contracted segment in either direction. This can be achieved by creating a pair of supernodes S', S'', which must be traversed consecutively (i.e. must be adjacent in any tour). For an arbitrary node j, the costs (lengths) of the arcs to and from S' and S'' are defined as $c(j, S') = c(j, i_1)$, $c(S', j) = c(i_1, j)$, $c(j, S'') = c(j, i_m)$, $c(S'', j) = c(i_m, j)$. Then if a tour enters S' from a node j and exits S'' to a node k, the cost of this tour segment is

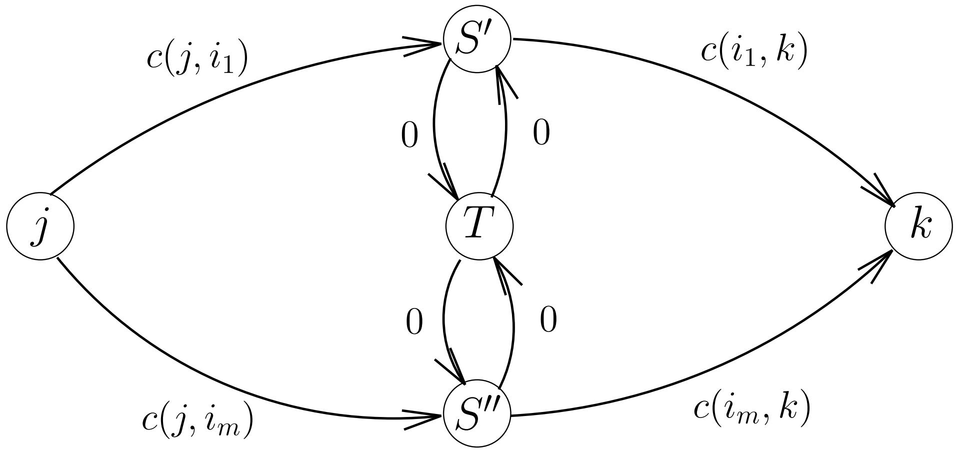

Figure 4: Construction of Supernodes for an Asymmetric TSP

$c(j,S') + c(S'',k) = c(j,i_1) + c(i_m,k)$; and if it enters S'' from a node j and exits S' to a node k, the cost is $c(j,S'') + c(S'',k) = c(j,i_m) + c(i_1,k)$. The condition that S' and S'' must be adjacent in any tour can be imposed, for instance, by placing an artifical node T between them, joined to each of them in both directions by zero-length arcs, and not joined to any other node. Figure 4 illustrates the construction.

If we now apply our procedure to the TSP with condition (1) defined on the contracted graph, with the same K as before, the neighborhood over which we optimize will have been enriched and the contracted tour may possibly be improved. The resulting locally optimal tour can then be decontracted to provide a tour for the original graph, better than the one whose arcs we had contracted.

This way our algorithm can be iterated arbitrarily many times by selecting a random collection of tour arcs to be contracted at each iteration, and then decontracting them before the next iteration. We select arcs for contraction randomly, with the probability of any arc being selected biased towards lower cost. We solve the TSP with condition (1) on the resulting contracted graph by our algorithm, then decontract the graph and retrieve the corresponding tour on G. This ends the iteration. Table 11 illustrates the performance of the iterative contraction procedure — which we call *Dynamic Programming with Arc Contraction* — on a collection of TSPLIB problems (`http://www.crpc.rice.edu/softlib/tsplib`). DP after 1000 3-opt means that DP is applied to the best of 1000 solutions obtained by

Table 11: Performance of DP with Iterated Arc Contraction

	Percent over Optimum		
TSPLIB Problem	DP after 1,000 3-opt (local optimum)	DP with Arc Contraction Iterated 20 times	(CPU sec)
pr1002	2.59	2.43	(84.43)
u1060	2.04	2.02	(77.45)
pcb1173	2.52	2.28	(102.82)
nrw1379	2.83	2.78	(121.01)
fl1577	1.89	1.83	(118.88)
dl655	3.00	2.96	(130.64)
vm1748	2.55	2.48	(136.94)
rl1889	2.65	2.60	(129.70)
Average	2.51	2.42	

starting in each case from a randomized nearest neighbor tour and using 3-opt until a local optimum is reached. DP in turn is iterated until its local optimum is reached. DP with Arc Contraction uses as starting solution the result from DP after 1000 3-opt.

While the arc-contraction procedure offers a good way to iterate our procedure without starting every time from scratch — an alternative made unattractive by the relatively high cost of our procedure in comparison to simpler heuristics — nevertheless the dynamic-programming procedure iterated by arc contraction cannot by itself beat the best interchange procedures, like iterated Lin-Kernighan (1973). On the other hand, we noticed that no matter how good a solution these other procedures are able to find, as long as that solution is not optimal, our procedure usually is able to improve upon it. Furthermore, the amount of improvement seems to depend very little, if at all, on the quality of the solution. To illustrate this, in Table 12 we present the improvement (in % of solution value) found by DP with Arc Contraction after running for one minute, when applied to the best tour found by 3-opt after 1000, 5000 and 10,000 iterations, respectively. The numbers are averages over all TSPLIB instances in the given range (`http://www.crpc.rice.edu/softlib/tsplib`).

All this suggests the idea of combining the dynamic-programming procedure with an interchange heuristic and applying them in an alternating fashion. The promise of combining in this way local search procedures based on different neighborhood definitions can be expected to depend on how different — or, mathematically speaking, how close to orthogonal — those neighborhoods are. To be more specific, consider combining our procedure

Table 12: Improvement (%) Obtained in One Minute when Starting from Best 3-opt Tour after Specified Number of Iterations

Problem size range	1000 it.	5000 it.	10,000 it.
$500 \leq n \leq 999$	0.116	0.143	0.076
$1000 \leq n \leq 1499$	0.170	0.203	0.182
$1500 \leq n \leq 3038$	0.239	0.189	0.195
Overall average	0.177	0.193	0.168

with 3-opt interchange. The dynamic-programming procedure finds a local optimum over a neighborhood whose size is exponential in n (Proposition 2.1), whereas the neighborhood searched for the execution of one 3-interchange is of size $O(n^3)$. To calculate the overlap between the two neighborhoods, we note that a 3-opt interchange applied to a tour of a TSP with condition (1) has to consider a neighborhood of size $O(K^3 n)$. In other words, the intersection of the two neighborhoods is a tiny fraction of the smaller one of the two. This suggests that such a combination might be pretty powerful.

In order to test this idea, we asked Gerhard Reinelt for the tours he obtained with the iterated Lin-Kernighan (1973) algorithm he used for the results of Jünger et al. (1995). Unfortunately those tours are no longer available, but Reinelt (1997) kindly sent us ten tours that are actually of higher quality than those in Jünger et al. (1995). We used these tours as starting solutions for the following algorithm, which we will call DPC: Our dynamic program routine (DP) is run until a local optimum is reached, then a simple 3-opt routine is run until a local optimum is reached. These two are alternated until the solution is optimal in both neighborhoods. At this point, the dynamic-programming routine with the arc contraction is run for a specified time.

Of the ten tours sent to us by Reinelt, there were seven on problems from the set we were studying. Of these, there were four that were not within 0.01% of the optimum. For these tours, our algorithm found improvements to three out of four instances in less than fifteen seconds of CPU time, as shown in Table 13.

Recently, the state-of-the-art chained Lin-Kernighan (CLK) code of Applegate et al. (1997) was made publicly available. This is a sophisticated implementation of the algorithm by Martin et al. (1992). We ran that code on the 22 symmetric TSPLIB problems between 1000 and 3500 nodes for 1000 and 10000 “kicks” (a “kick” is an attempt to push a solution

Table 13: Improving Near-Optimal Tours with DPC

	Per cent over optimum	
Problem	Reinelt's tour	Our improved tour (CPU time)
rl1323	0.377	0.372 (5.07)
fl1400	0.263	No Improvement
d1655	0.034	0.031 (8.07)
d2103	0.067	0.062 (10.39); 0.041 (38.17)

Notes:
DPC was run with $k = 8$ for one hour of CPU time.

out of a local optimum). We then applied DPC to the resulting tours with $K = 6$, 8 and 10.

In ten of the 22 instances, we found improvements over the solution obtained by the CLK code after 1000 kicks with each value of K, and in over half of these cases, an improvement was found on the first iteration of DP with the DPC algorithm. However, outside of this initial surge, the rate of gain from DPC did not exceed that of continuing the CLK code, with one notable exception mentioned below. In four of the 22 instances, we found improvements to the solution obtained by CLK after 10,000 kicks with each value of K, and in each case an improvement was found as a result of the first iteration of DP within DPC. At this point, the CLK code was no longer finding improvements consistently (in fewer than half of the 22 instances were improvements found between the 5,000$^{\text{th}}$ and 10,000$^{\text{th}}$ kick). In one of these four cases, the CLK code had found no additional improvements after the 1,000$^{\text{th}}$ kick, probably because the solution was a mere 0.0004% above optimal. However, our code was able to find, in less than a second, an improvement that led to an optimal solution that the CLK code failed to find over the next 9000 kicks, which took over 200 seconds on the same machine. The details of the cases based on tours found after 10,000 kicks are in Table 14.

Table 14: Using the CLK solution as starting tour for DPC

	Percent over optimum		
	CLK after	Our improvements with $K = 6$ (CPU time)	
Problem	10,000 kicks	First iteration of DP	Additional improvements with DPC
rl1304	0.0004	optimal (0.52)	
d1655	0.0338	0.0225 (0.67)	none
vm1748	0.1408	0.1405 (0.70)	none
rl1889	0.1254	0.1238 (0.76)	0.1147 (11.15)

Table 15: CLK with and without DP after the 10,000th kick

	Percent over optimum			
	CLK alone with		CLK with 10,000 kicks + DP + CLK with	
Problem	11,000 kicks	20,000 kicks	1,000 kicks	10,000 kicks
rl1304	optimal	optimal	optimal	optimal
dl655	0.0338	0.0113	0.0225	0.0225
vm1748	0.1408	0.1144	0.1385	0.1385
rl1889	0.1242	0.1147	0.1238	0.0847

Since most of the improvements came on the first iteration of DP within DPC, we wondered about using a single iteration of DP (less than one second with $K = 6$), and then feeding the resulting tour back to CLK for 1,000 and 10,000 additional kicks in those cases where the solution was not yet optimal. We then compared these results with running the CLK code for 11,000 and 20,000 kicks respectively (see Table 15). For problem rl1304, the optimal solution was found much more quickly by combining the algorithms. In the other three cases, there was always a short-term benefit (compare the additional 1,000 kicks after DP with 11,000 kicks alone), and there was a long-term benefit in one of the three cases (compare the additional 10,000 kicks after DP with 20,000 kicks). This makes us believe that using DP to "kick" a solution periodically (more than once) would be effective in enough instances to warrant the addition of this feature to CLK. We plan to incorporate this idea into a version of CLK code in the near future.

7. Conclusions

We have discussed an implementation of a linear-time dynamic-programming algorithm for solving TSPs with a special type of precedence constraints.

We have applied our procedure to solving TSPs with time windows, originating in scheduling problems with setup, release and delivery times, in truck delivery problems, and in stacker-crane routing. We present extensive computational evidence for our claim that on these classes of problems our procedure brings an improvement to the state of the art.

We have formulated a new type of TSP with a time-related objective, the TSP with target times, with important potential applications, and demonstrated that our algorithm can solve optimally problems of realistic size in this class. We don't know how other algorithms would perform on this class.

Finally, for TSPs without precedence constraints or time windows, we have shown how to use our algorithm as a local search procedure that finds in linear time a local optimum over an exponential-size neighborhood. For this purpose we developed an iterative version of our algorithm, Dynamic Programming with Arc Contraction. Although by itself this algorithm is outperformed by the most sophisticated version of chained Lin-Kernighan (1973) procedure, on the other hand it is often able to improve in its first iteration upon the best solution found by the latter in 10,000 iterations. This and other evidence points to the potential usefulness of combining these two approaches based on sharply differing neighborhood definitions.

References

Applegate, D., R.E. Bixby, V. Chvatal, W.J. Cook. 1997. A new paradigm for finding cutting planes in the TSP. International Symposium on Mathematical Programming, Lausanne, Switzerland.

Ascheuer, N., M. Fischetti, M. Grötschel. 1997. Solving ATSP with time windows by branch-and-cut. Technical report, ZIB Berlin.

Baker, E. 1983. An exact algorithm for the time-constrained traveling salesman problem. *Operations Research* **31** 938-945.

Balas, E. 1999. New classes of efficiently solvable generalized traveling salesman problems. *Annals of Operations Research* **86** 529-558.

Balas, E. 1989. The prize collecting traveling salesman problem. *Networks* **19** 621-636.

Burkard, R.E., V.G. Deineko, R. van Dal, J.A.A. van der Veen, G.J. Woeginger. 1997. Well solvable special cases of the TSP: a survey. SFB Report 52, Technical University, Graz, Austria.

Dumas, Y., J. Desrosiers, E. Gelinas, M.Solomon. 1995. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research* **43** 367-371.

Gendreau, M., A. Hertz, G. Laporte, M. Stan. 1995. A generalized insertion heuristic for the traveling salesman problem with time windows. CRT-95-07, Centre de Recherche sur les Transports, Université de Montréal, Montréal, PQ, Canada.

Gilmore, P.C., E.L. Lawler, D. Shmoys. 1985. Well-solved special cases. E.L. Lawler, J.K Lenstra, A.H.G. Rinnooy Kan, D. Shmoys eds. *The Traveling Salesman Problem: A*

Guided Tour to Combinatorial Optimization. Wiley, New York. 87-145.

Johnson, D.S., L.A. McGeoch. 1997. The traveling salesman problem: a case study in local optimization. E.H.L. Aarts, J.K.L. Lenstra eds. *Local Search in Combinatorial Optimization.* Wiley, New York. 215-311.

Jünger, M., G. Reinelt, G. Rinaldi. 1995. The traveling salesman problem. M. Ball, T. Magnanti, C. Monma, G. Nemhauser eds. *Network Models.* Elsevier 225-330.

Kanellakis, P., C. Papadimitriou. 1980. Local search for the traveling salesman problem. *Operations Research* **28** 1086-1099.

Lin, S., B.W. Kernighan. 1973. An effective heuristic algorithm for the traveling salesman problem. *Operations Research* **21** 495-516.

Martin, O., S.W. Otto, E.W. Felten. 1992. Large step markov chains for the TSP incorporating local search heuristics. *Operations Research Letters* **11**. 219-225.

Or, I. 1992. Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking. Ph.D. Thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL.

Potvin, J.Y., S. Bengio. 1993. A genetic approach to the vehicle routing problem with time windows. CRT-93-5, Centre de Recherche sur les Transports, Université de Montréal, Montreal, PQ, Canada.

Reinelt, G. 1997. Personal communication.

Repetto, B. 1996. Personal communication.

Repetto, B. 1994. *Upper and Lower Bounding Procedures for the Asymmetric Traveling Salesman Problem.* Ph.D. Thesis, GSIA, Carnegie Mellon University, Pittsburgh, PA.

Solomon, M. 1987. Algorithms for vehicle routing and scheduling with time window constraints. *Operations Research* **35**. 254-265.

Stan, M. 1997. Personal communication.

Tama, J. 1990. *Polyhedral Aspects of Scheduling Problems with an Application to the Time-Constrained Traveling Salesman Problem.* Ph.D. thesis, GSIA, Carnegie Mellon University, Pittsburgh, PA.

Van der Veen, A.A. 1992. *Solvable Cases of the Traveling Salesman Problem with Various Objective Functions.* Ph.D. Dissertation, University of Groningen, Gronigen, The

Netherlands.

Accepted by Jan Karel Lenstra; received October 1997; revised December 1998, April 2000; accepted August 2000.