

# Applications of a Dynamic Programming Approach to the Traveling Salesman Problem

Neil Simonetti

February 1998

## Abstract

Consider the following restricted (symmetric or asymmetric) traveling salesman problem: given an initial ordering of the  $n$  cities and an integer  $k > 0$ , find a minimum cost tour such that if city  $i$  precedes city  $j$  by at least  $k$  positions in the initial ordering, then city  $i$  precedes city  $j$  in any optimal tour. Balas [5] 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 [5] 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 1-1 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  $k(j)$ ,  $j = 1, \dots, n$ . One important feature of our implementation is that we construct in advance, without knowledge of any particular problem instance, a typical layer of an auxiliary supernetwork that can be stored and subsequently used to solve any instance with  $\max_j k(j) \leq K$  for some large  $K$  associated with this auxiliary structure. This advance construction, which needs to be done only once, absorbs the bulk of the computing time.

We discuss applications to, and computational experience with, TSP's 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. For TSP's that do not satisfy the postulated precedence constraints, 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  $k$ .

Finally, we include the specifics for the dynamic programming model when applied to the Prize Collecting TSP, highlighting the differences in the two models for implementation.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Background . . . . .	3
<b>2</b>	<b>The Algorithm</b>	<b>5</b>
2.1	Defining the Dynamic Program . . . . .	5
2.2	The Implementation Model . . . . .	6
2.3	Space Complexity Issues . . . . .	13
<b>3</b>	<b>Time Window Problems</b>	<b>15</b>
3.1	Defining the Precedence Constraints . . . . .	15
3.2	Minimizing Distance Instead of Time . . . . .	17
3.3	Computational Experience . . . . .	18
3.4	Traveling Salesman Problem with Target Times . . . . .	24
<b>4</b>	<b>Local Search for Arbitrary TSP's</b>	<b>26</b>
4.1	Iterating the Dynamic Program . . . . .	26
4.2	Combining the Dynamic Program and Interchange Heuristics . . . . .	28
4.3	Clustered TSP's . . . . .	35
<b>5</b>	<b>Single Machine Scheduling</b>	<b>39</b>
5.1	Scheduling with Set-up Times and Release Dates . . . . .	39
5.2	Applications for Job Shop Scheduling . . . . .	40

<b>6</b>	<b>Prize-Collecting TSP's</b>	<b>44</b>
6.1	Defining a New Dynamic Program . . . . .	44
6.2	Results from the New Definition . . . . .	46
6.3	Implementation Issues . . . . .	50
<b>7</b>	<b>Conclusions</b>	<b>54</b>
<b>8</b>	<b>The Dynamic Programming Code</b>	<b>56</b>
8.1	Building the Auxiliary Structure . . . . .	56
8.2	The Subroutines . . . . .	57
8.3	Essential Ingredients for Your Shell . . . . .	59
8.4	Sample Shell Programs . . . . .	59

# Chapter 1

## Introduction

Everything presented here is joint work with Egon Balas. The work in Section 5.2 is also with Alkis Vazacopoulos. The work in Chapter 6 is also with Subba Rao V. Majety.

### 1.1 Background

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 Hamilton 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 [10], [23].

In [5], Balas introduced a class of polynomially solvable TSP's with precedence constraints: Given an integer  $k > 0$  and an ordering  $\sigma := (1, \dots, n)$  of  $N$ , find a minimum cost tour, i.e. permutation  $\pi$  of  $\sigma$ , satisfying

- (i)  $\pi(i) < \pi(j)$  for all  $i, j \in \sigma$  such that  $i + k \leq j$ .

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

**Theorem 1.1.1** [5] *Any TSP with condition (i) can be solved in time  $O(k^2 2^{k-2} n)$ .  $\square$*

So for fixed  $k$ , we have a linear time algorithm for solving TSP's – whether symmetric or asymmetric – with this type of precedence constraints. 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 (i) becomes

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

and a result similar to Theorem 1.1.1 applies [5]. Frequently occurring practical problems, like TSP's with time windows, which play a prominent role in vehicle routing [20], can be formulated in this way.

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 [6], serves as a model for scheduling steel rolling mills.

Whenever the problem to be solved satisfies the required type of precedence constraints, the dynamic programming procedure finds an optimal solution. On the other hand, for problems that do not satisfy such precedence constraints, the dynamic programming procedure finds a local optimum over a neighborhood defined by those 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 exhaustively search an exponential size neighborhood.

# Chapter 2

## The Algorithm

### 2.1 Defining the Dynamic Program

The method proposed in [5] associates with a TSP satisfying (i), a network  $G^* := (V^*, A^*)$  with  $n + 1$  layers of nodes, one layer for each position in the tour, with the home city 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 explained below, is such as to create a 1-1 correspondence between tours in  $G$  satisfying condition (i) (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 three entities:

1.  $j$ , the city in position  $i$ ,
2.  $S^-$ , the set of cities numbered  $i$  or higher that are visited in one of the positions 1 through  $i - 1$ . ( $S^- := \{h \in N : h \geq i, \pi(h) < i\}$ )
3.  $S^+$ , the set of cities numbered below  $i$  that are not visited in one of the positions 1 through  $i - 1$ . ( $S^+ := \{h \in N : h < i, \pi(h) \geq i\}$ )

Nodes in  $G^*$  can be referenced by the notation  $(i, j, S^-, S^+)$ . When referencing a node in a fixed (possibly arbitrary) layer  $i$ , the notation will simply be  $(j, S^-, S^+)$ , where the elements are dependent on  $i$ . The reason this representation is economical is that

$$|S^-| = |S^+| \leq \lfloor k/2 \rfloor,$$

i.e. the set  $(1, \dots, i - 1)$  of order  $n$  is represented by order  $k$  entries.

A layer  $i$  will be referred to as *typical* if  $k + 1 \leq i \leq n - k + 1$ . A typical layer contains  $(k + 1)2^{k-2}$  nodes [5]. The first  $k$  and last  $k$  layers contain fewer nodes.

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 [5] for details). The cost assigned to the arc of  $G^*$  joining the two nodes is then  $c_{j\ell}$ , the cost of the arc  $(j, \ell)$  in the original TSP instance. No node of  $G^*$  has indegree greater than  $k$ , which bounds the number of arcs joining two consecutive layers at  $k(k + 1)2^{k-2}$ , and the total number of arcs of  $G^*$  at  $k(k + 1)2^{k-2}n$ . 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 arbitrary TSP as a heuristic, finds a local optimum over a neighborhood whose size is exponential in  $n$ . Indeed, we have

**Proposition 2.1.1** *The number of tours satisfying condition (i) is  $O\left(\left(\frac{k-1}{e}\right)^{n-1}\right)$ .*

**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  $\sigma$  that changes positions only within the subsequences yields a tour satisfying (i), 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 (i) is replaced by (ia), one can build an appropriate network  $G^{*(a)}$  with properties similar to those of  $G^*$ .

## 2.2 The Implementation Model

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  $G^*$ , the bulk of the effort going into (a). 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  $n$ . Notice, further, that the structure and size of each layer, i.e. node set  $V_i^*$ ,  $k + 1 \leq i \leq n - k + 1$ , is also the same; and the structure of each arc set  $A^* \cap (V_i^*, V_{i+1}^*)$ ,  $k + 1 \leq i \leq n - k$ , is 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 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 TSP's satisfying condition (i a) and the corresponding network  $G^{*(a)}$ . The upshot of this is the remarkable fact 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 2.1. 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  $G^{*(a)}$ . To avoid examining these when we use  $G_K^{**}$  in place of  $G^{*(a)}$ , one can replace the copy of  $W_K^*$  for layer  $i$  with a copy of  $W_{m_i}^*$  for some  $m_i \leq K^*$  which will not remove any  $s - t$  paths also in  $G^{*(a)}$ . Call  $G^{**}$  this slimmer auxiliary structure (dependent on the problem instance). Given the values  $m_i$ , the nodes in layer  $i$  of  $G^{**}$  are simply  $W_{m_i}^*$ . The arcs connecting two adjacent layers,  $i$  and  $i + 1$ , of  $G^{**}$ , are easily found from the list of compatible pairs  $(u, v)$  of  $W_K^* \times W_K^*$  by choosing those pairs with  $u \in W_{m_i}^*$  and  $v \in W_{m_{i+1}}^*$ .

Next we address the issue of calculating good values for  $m_i$  when choosing  $W_{m_i}^*$  to represent a layer of  $G^{**}$ , and finding a way to avoid examining those  $s - t$  paths in  $G^{**}$  that are

not also in  $G^{*(a)}$ .

Since  $W_h^* \subset W_m^*$  whenever  $h < m$ , there is a natural partition of  $W_K^*$  into *levels*. Let  $W_1^*$  be the first level, and define the  $h$ -th level of nodes ( $2 \leq h \leq W_K^*$ ) to be  $W_h^* \setminus W_{h-1}^*$ . Table 2.1 illustrates the breakdown of  $W_4^*$  into its levels. The set of arcs into and out of the nodes of  $W_4^*$  is shown as a list of compatible predecessors and successors, represented through their number in  $W_4^*$ . Notice that the neat layout of the levels of  $W_K^*$  is predicated upon listing the nodes in a sequence such that for  $m = 1, \dots, K-1$ , all nodes in  $W_m^*$  precede all nodes in  $W_{m+1}^* \setminus W_m^*$ .

Figure 2.1 shows the graphs  $G_K^{**}$ ,  $G^{**}$  and  $G^{*(a)}$  for a TSP with condition (ia) and with  $n = 9$ ,  $k(3) = 4$  and  $k(i) = 3$  for all  $i \neq 3$ . The  $s - t$  paths shown in the figure correspond to the feasible tour  $(1, 4, 2, 6, 3, 5, 7, 9, 8, 1)$ ; the infeasible tour  $(1, 2, 3, 4, 5, 8, 7, 9, 6, 1)$  which extends beyond  $G^{**}$ ; and the infeasible tour  $(1, 3, 2, 5, 7, 4, 8, 6, 9, 1)$ , which does not extend beyond  $G^{**}$  but must be avoided because it contains nodes not in  $G^{*(a)}$ .

Table 2.1: Nodes of  $W_4^*$  and its compatible pairs.

	No.	Node Label $(i, j, S^-, S^+)$	Compatible	
			Predecessors	Successors
Level 1:	1:	$(i, i, \emptyset, \emptyset)$	1,3,8,20	1,2,4,9
Level 2:	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
Level 3:	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
Level 4:	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

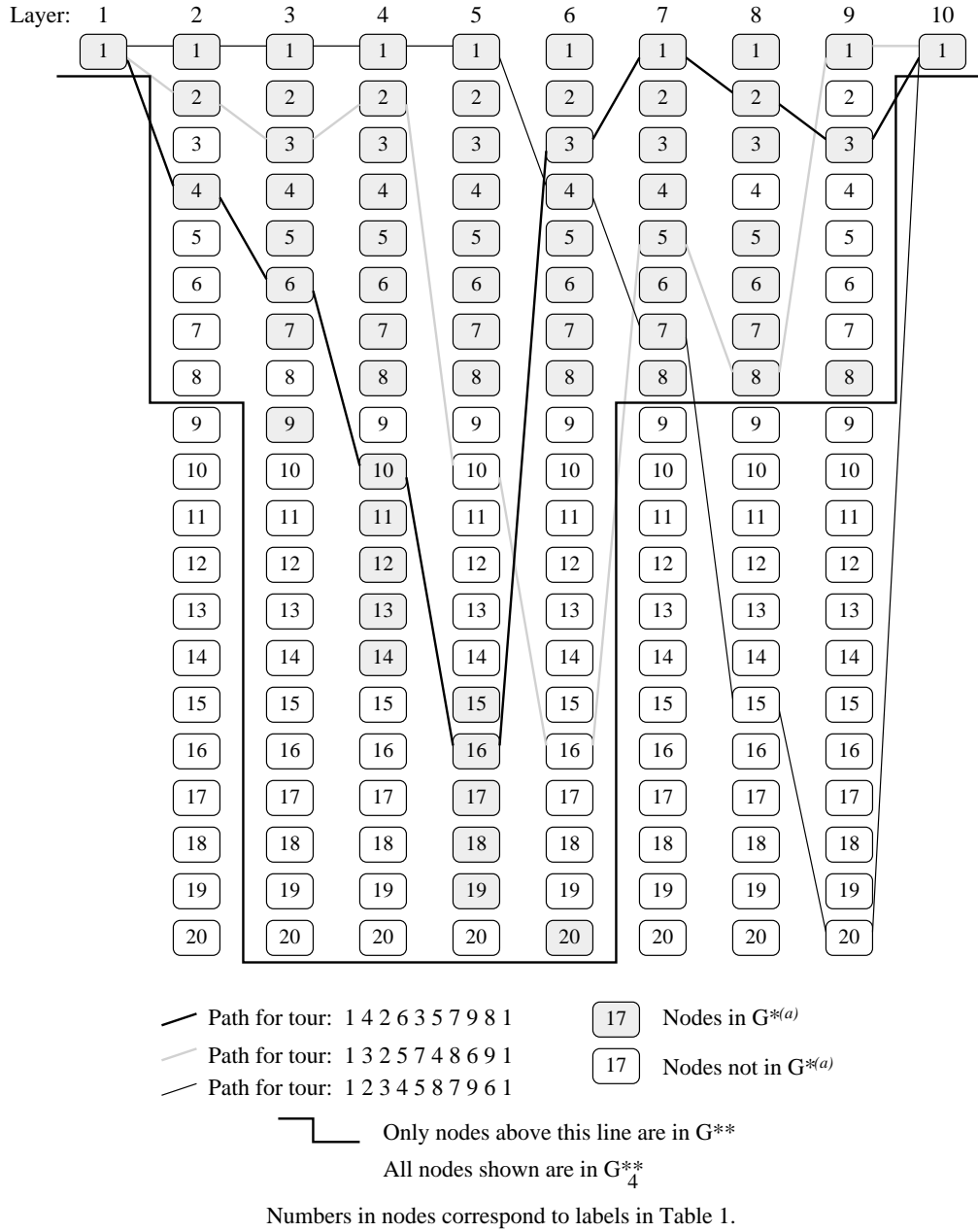


Figure 2.1:  $G_K^{**}$ ,  $G^{**}$  and  $G^{*(a)}$  for an instance with  $n = 9$ ,  $k(3) = 4$  and  $k(i) = 3$  for  $i \neq 3$ .

In Section 2.1, the nodes of  $G^*$  (or  $G^{*(a)}$ ) were denoted by  $(i, j, S^-, S^+)$ . We will use the same notation for the nodes of  $W_K^*$  and, whenever the discussion is about a generic layer  $W_K^*$ , we will omit the layer specifier  $i$ . W

We first address the problem of determining the level to which a given node of  $W_K^*$  belongs. Let  $L(v)$  be the level of  $W_K^*$  containing node  $v$ . Notice that  $L(v)$  is the smallest value of  $k$  for which  $W_K^*$  contains node  $v$ .

**Proposition 2.2.1** *Let  $v = (j, S^-, S^+) \in W_K^*$ . If  $W_K^*$  is used for layer  $i$ ,*

$$L(v) = 1 + \max\{|i - j|, \max(S^-) - \min(S^+), j - \min(S^+)\}. \quad (2.1)$$

**Proof.**

(i)  $L(v) > j - \min(S^+)$ . To see this, note that from the definition of  $S^+$ , we have  $\pi(j) < \pi(\min(S^+))$ ; hence if  $j \geq \min(S^+) + L(v)$ , the precedence constraint is violated and the claim follows.

(ii)  $L(v) > \max(S^-) - \min(S^+)$ . Indeed, again from the definitions of  $S^-$  and  $S^+$ ,  $\pi(\min(S^+)) > \pi(\max(S^-))$ ; hence if  $\max(S^-) \geq \min(S^+) + L(v)$ , the precedence constraint is again violated, and the claim follows.

(iii)  $L(v) > |i - j|$ , since otherwise either  $j \geq i + L(v)$  or  $j \leq i - L(v)$ , which violates the precedence constraint.

Since  $L(v)$  is integer, from (i) we have the  $\geq$  direction of the equality (2.1). To prove the other direction, we exhibit a tour whose associated  $s - t$  path contains the node  $v$ . Since  $L(v)$  is the smallest value of  $k$  for which node  $v$  exists, this proves the point. Let the tour visit the cities in the set  $\{1, 2, \dots, i - 1\} \setminus S^+$  in increasing numerical order, then the cities in  $S^-$  in increasing order, then city  $j$ , then the cities in  $S^+ \setminus \{j\}$  in increasing order, and finally the cities in  $\{i, i + 1, \dots, n\} \setminus S^-$  in increasing order. Clearly,  $v$  is on this path.  $\square$

The *depth*  $D(i)$  of layer  $i$  is the highest level of a node in  $W_K^*$  which appears in layer  $i$  of  $G^{*(a)}$ . For the example in Figure 2.1,  $D(1) = 1$ ,  $D(2) = 3$ ,  $D(3) = \dots = D(6) = 4$ , and so on.

The *reach*  $R(j)$  of a city  $j \in N$  is the set of layers whose depth may be affected by  $k(j)$ . For the example in Figure 2.1,  $R(3) = \{3, 4, 5, 6\}$ . This is explained by the following proposition.

**Proposition 2.2.2** *Given  $j \in N$ ,*

$$R(j) = \{j, \dots, j + k(j) - 1\}.$$

**Proof.** From Proposition 2.2.1, city  $j$  cannot be assigned position  $j + k(j)$  or higher. So the reach of  $j$  can go no further than position  $j + k(j) - 1$ . The reach does not fall below position  $j$  because the precedence constraints that allow city  $j$  to be visited in a position less than  $j$  involve only values of  $k(\ell)$  for  $\ell < j$ .  $\square$

**Remark 2.2.3** For any layer  $i$ ,

$$D(i) := \max_j \{k(j) : i \in R(j)\}.$$

$\square$

We now construct the layered digraph  $G^{**}$  using  $W_{D(i)}^*$  as the node set for layer  $i$ .  $G^{**}$  is an intermediate structure between  $G^{*(a)}$  and  $G_K^{**}$ . Since the complexity of solving any particular problem instance is a linear function of the number of arcs in  $G^{**}$ , the following result is of particular interest:

**Theorem 2.2.4** The number of arcs of  $G^{**}$  is bounded by

$$\sum_{i=2}^{n+1} D(i-1)(D(i)+1)2^{D(i)-2}.$$

**Proof.** The number of nodes in layer  $i$  of  $G^{**}$  is  $(D(i)+1)2^{D(i)-2}$  (see Section 2.1), so we need to show only that the maximum in-degree of a node in layer  $i$  of  $G^{**}$  is  $D(i-1)$ . In [5] it was also shown that for a given node  $v := (j, S^-, S^+)$  in layer  $i$  of  $G^{**}$ , at most one node  $u := (l, T^-, T^+)$  from layer  $i-1$  may be a predecessor of  $v$  for a fixed value of  $l$ , so we need to show that there are at most  $D(i-1)$  candidates for  $l$ . The depth of layer  $i-1$  limits the candidates for  $l$  to the set  $\{(i-1) - D(i-1) + 1, \dots, (i-1) + D(i-1) - 1\}$ . Also, by the compatibility of nodes  $u$  and  $v$ ,  $l$  must be restricted to the set  $(\{1, \dots, i-1\} \cup S^-) \setminus S^+$ . Intersecting these two sets, we conclude that  $l$  must be in the set  $P := (\{i - D(i-1), \dots, i-1\} \cup S^-) \setminus S^+$ . Since  $|S^+| = |S^-|$ ,  $|P| = D(i-1)$  as long as  $S^+ \subset \{i - D(i-1), \dots, i-1\}$ . If this were not the case, then  $\min(S^+) \leq (i-1) - D(i-1)$  since the elements in  $S^+$  only come from the set of cities less than  $i$ , and so, for a node  $u := (l, T^-, T^+)$  to be a potential predecessor of  $v$  from layer  $i-1$ , we must have  $\min(T^+) \leq \min(S^+) \leq (i-1) - D(i-1)$ , which would imply

$$L(u) \leq D(i-1) \leq (i-1) - \min(T^+) \leq \max(T^-) - \min(T^+)$$

contrary to Proposition 2.2.1.  $\square$

As illustrated by Figure 2.1, there may be paths in  $G^{**}$  which are not present in  $G^{*(a)}$ . Thus we need a test for the nodes of  $G^{**}$  to prevent these paths from being considered.

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  $\max(S^-)$ , 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 2.2.5** *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  $\pi$  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  $i$ th 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:

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

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 greater than or equal to  $i$ . This gives:

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

In Figure 2.1, 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 determining  $G^{**}$  and finding a shortest  $s - t$  path in the subgraph  $G^{*(a)}$  of  $G^{**}$ , where  $G^{*(a)}$  is the auxiliary graph associated with the specific problem instance one wants to solve. Determining  $G^{**}$ , extracting the nodes and arcs of  $G^{*(a)}$  from those of  $G^{**}$ , 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  $G^{**}$ .

## 2.3 Space Complexity Issues

Not only does the time complexity grow exponentially with our parameter,  $k$ , but the space required to store our auxiliary graph structure grows at the same rate. Since most workstations do not have unlimited memory available, saving even a constant factor of memory can be significant.

At first glance, it would appear that the space required to store the auxiliary structure must be  $O(k(k+1)2^{(k-2)})$ , since there are  $(k+1)2^{(k-2)}$  nodes having a maximum in-degree of  $k$ . However, close inspection to Table 2.1 will show that every node appears in exactly one distinct list of successors or predecessors. For example, node 8 appears in 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, and generate four pointers to this list from nodes 1, 2, 4, and 9. This allows us to store the graph with space  $O((k+1)2^{(k-2)})$ . Including the cost of these lists and pointers, this is still roughly a savings of a factor of  $\frac{k}{3}$  in the cost of storing the arcs.

Our other major concern for space is the reconstruction of the tour. When we calculate the cost of the shortest path for a given state represented by a node of the auxiliary graph, we need only the costs at the previous layer, so traversing the auxiliary graph to find the cost of the shortest path only requires 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 tour we have generated, we must trace our way backwards through the graph, which requires that we store, at every node of *every* layer, which node was the predecessor with least cost. Since nodes have in-degree at most  $k$ , this can be done with  $\log_2 k$  bits at each node rather than  $\log_2((k+1)2^{(k-2)})$ , provided we have the predecessor lists. Unfortunately, since this must be done for every layer, we have a linear space dependence on  $n$ , which puts a big strain on available memory even for relatively small problems.

The trick to avoiding this linear space dependence on  $n$  comes in realizing the pattern of most solutions to this problem. In many applications, our procedure can either be iterated, or used as a post processing step for another heuristic. When this is done, our algorithm is usually applied to a solution of “reasonably good” quality before it is started.

If our algorithm fails to find an improvement on an initial tour (i.e. the initial tour is the optimal tour for the precedence constraints), then the path through the auxiliary graph will be confined to the first node (level 1) of every layer. If our algorithm finds a few local improvements but otherwise leaves the initial ordering the same, the auxiliary graph will be confined to the first node of the layers representing the positions where the tour was

unchanged (see Figure 2.2). When this is the case, we can store just the path fragments where the path leaves the first level. A potential path fragment is identified at its right end, suppose it is level  $i$ , when the best predecessor for the level 1 node of layer  $i$  is not the level 1 node of layer  $i - 1$ . At this point, we can trace the path fragment back to its beginning, at level  $i - h$ .

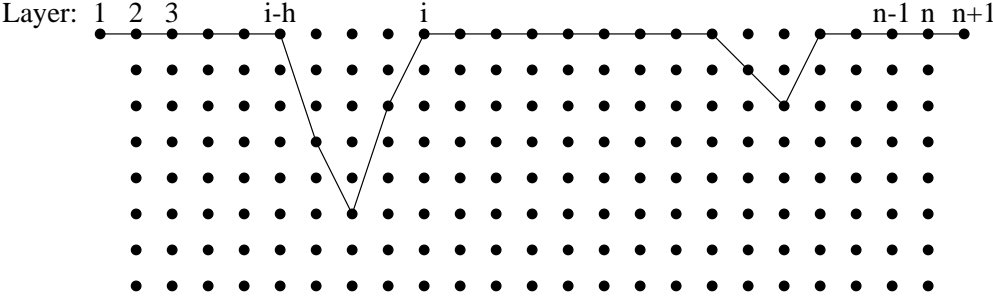


Figure 2.2: A path through  $G_K^{**}$  for a solution similar to the initial tour

When we reconstruct our tour, we trace backwards, inserting path fragments as needed. For example, if we encounter three path fragments, 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 path fragment (from 21 to 18) and then include the latest path fragment whose right end is at layer 18 or earlier, which would be the fragment from layer 10 back to layer 6. We would skip the fragment from layer 20 to layer 15, since this fragment would not be along the optimal path.

If no path fragment spans more than  $H$  layers, then we can replace our linear space dependence on  $n$  with a linear space dependence on  $H$ . For the applications of our algorithm as a heuristic on a general TSP (see Chapter 4), where problems sizes have reached into the thousands, I have only encountered one instance where  $H > 3k$ , and in this case  $H < 6k$ .

Note that if we allow for value of  $H$  that is insufficient, this can be detected by noticing that we run out of space before the path fragment 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 can restart the algorithm with a larger allowance for  $H$ .

# Chapter 3

## Time Window Problems

### 3.1 Defining the Precedence Constraints

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 traveling salesman problems with time windows (TSPTW), either symmetric or asymmetric (see, for instance, [4, 20]).

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. The time windows can be used to derive precedence constraints and the problem can be treated as a TSP with condition (i a). Given an initial ordering  $\{1, \dots, n\}$  of the cities, 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 obtained a TSP with condition (i a) 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.

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

Figure 3.1: Constructing Precedence Constraints from Time Windows

The initial ordering can be generated, for instance, by sorting the time-windows by their midpoint. Figure 3.1 shows an example of this, along with the constants  $k(i)$  derived from this ordering. 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. In the example, moving the time window for city 3 after that of city 4 reduces the largest  $k(i)$  by 1 (see Figure 3.2). 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

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

Figure 3.2: Improving the Bound on the Largest  $k(i)$

time windows, but also on the initial tour. In fact, it is not even necessary for the initial ordering to represent a feasible tour. All that is needed is an initial ordering for which the values  $k(i)$  derived by the above procedure define precedence constraints satisfied by every feasible tour.

### 3.2 Minimizing Distance Instead of Time

If the objective is the minimization of total time needed to complete the tour, then 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 (i a), if feasible with respect to the time windows, defines an optimal tour for the TSPTW.

However, the literature (see, for instance, [4, 20]) 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 3.3 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,

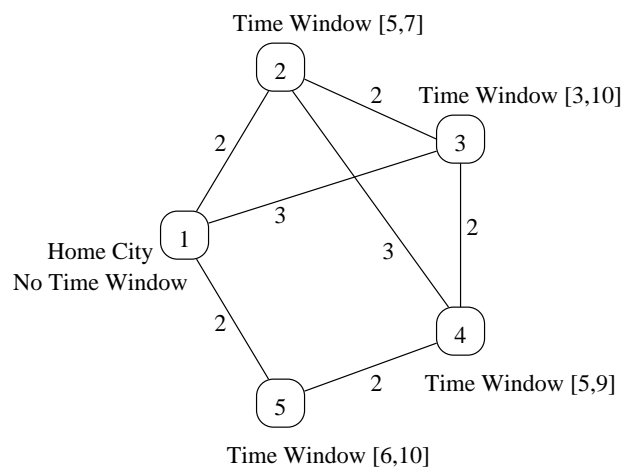


Figure 3.3: Illustrating the Difficulty of Minimizing Distance Instead of Time

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 addressed in [8], 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 such pairs are not unique, we clone the node whose state has multiple non-dominated time-distance pairs. To keep our polynomial bound, we restrict the number of pairs (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 Chapter 2, 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 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 much smaller than that required for guaranteed optimality.

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 improve the solution. Also, using some other heuristic between multiple applications of our algorithm has improved our results sometimes.

### 3.3 Computational Experience

All of our results in this and the following sections were obtained on a Sun Ultra I Workstation. Where our algorithm could not guarantee optimality, a simple Or-opt heuristic [15] was applied, followed by renewed application of our algorithm.

Using a method similar to that given by Baker [4], Tama [21] has generated a number of instances of one-machine scheduling problems with set-up times and minimum makespan objective, formulated as asymmetric TSP's 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 [21]. We ran our algorithm on these problems, using  $K = 13$  and  $K = 17$ , 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 3.1).

Table 3.1: Asymmetric TSP’s with Time Windows (One-Machine Scheduling)

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

\*Indicates that a guarantee of optimality (or infeasibility) was found

As to symmetric TSP’s with Time Windows, thirty instances of the RC2 problems proposed by Solomon [20] that were first studied by Potvin and Bengio [16] and then by Gendreau, Hertz, Laporte and Stan [9] were run with our algorithm, using  $K = 12$  and  $K = 15$  with the results shown in Table 3.2. Here the objective was to minimize total distance, i.e. occasionally nodes had to be cloned as explained in Section 3.2. The problem names listed in Table 3.2 refer to the code used by Solomon, followed by the route number. Neither Potvin et al. nor Gendreau et al. could guarantee the optimality of a solution even if their solution was optimal. The entry for “Required  $K$ ” in Table 3.2 indicates what value of  $K$  would be needed for a guarantee of optimality.

As can be seen from Table 3.2, our algorithm solved 14 of these problems with a guarantee of optimality. Of these 14, Potvin et al. had solved 2 to optimality and Gendreau et al. had solved 9 to optimality. Of the remaining 16 problems, our algorithm fared better than Potvin et al. 13 times, equalled Gendreau et al. once, and fared better than Gendreau et al. 8 times.

Table 3.2: Symmetric TSP's with Time Windows (Vehicle Routing):

Problem	$n$	Potvin [16]	Gendreau [9]	Req. $K$	Our Results (CPU seconds)	
					$K = 12$	$K = 15$
rc201.1	20	465.53	444.54 (9.86)	6	444.54* (2.49)	
rc201.2	26	739.79	712.91 (23.06)	7	711.54* (3.33)	
rc201.3	32	839.76	795.44 (46.91)	7	790.61* (4.51)	
rc201.4	26	803.55	793.64 (19.21)	7	793.63* (3.27)	
rc202.1	33	844.97	772.18 (33.65)	20	773.16 (18.42)	772.33 (223.30)
rc202.2	14	328.28	304.14 (7.76)	13	304.14* (2.92)	
rc202.3	29	878.65	839.58 (22.16)	15	837.72 (9.54)	837.72* (45.46)
rc202.4	28	852.73	793.03 (36.61)	18	809.29 (23.00)	799.18 (212.46)
rc203.1	19	481.13	453.48 (13.18)	13	453.48 (12.19)	453.48 <sup>+</sup> (96.75)
rc203.2	33	843.22	784.16 (49.26)	24	809.14 (33.06)	808.78 (404.42)
rc203.3	with initial tour from [9]: 789.04			32	784.16 (20.84)	784.16 (246.66)
	37	911.18	842.25 (50.90)	27	could not find feasible solution	
rc203.4	with initial tour from [9]: 842.03			36	834.90 (30.14)	834.90 (373.34)
	15	330.33	314.29 (9.88)	12	314.29* (3.58)	
rc204.1	with initial tour from [9]: 878.76			45	878.64 (74.82)	878.64 (1061.16)
	46	923.86	897.09 (84.41)	39	could not find feasible solution	
rc204.2	with initial tour from [9]: 664.14			32	664.14 (32.79)	664.14 (467.31)
	33	686.56	679.26 (53.25)	28	662.16 (77.22)	681.85 (830.17)
rc204.3	with initial tour from [9]: 460.24			22	475.59 (61.28)	466.21 (639.93)
	24	455.03	460.24 (37.86)	22	475.59 (61.28)	466.21 (639.93)
rc205.1	14	381.00	343.21 (3.58)	7	343.21* (2.02)	
rc205.2	27	796.57	755.93 (23.61)	11	755.93* (5.99)	
rc205.3	35	909.37	825.06 (138.38)	23	825.06 (42.78)	825.06 (384.03)
rc205.4	28	797.20	762.41 (21.13)	12	760.47* (5.25)	
rc206.1	4	117.85	117.85 (0.03)	3	117.85* (0.29)	
rc206.2	37	850.47	842.17 (108.03)	16	828.06 (33.92)	828.06 (315.10)
rc206.3	25	652.86	591.20 (22.30)	14	574.42 (17.40)	574.42 <sup>+</sup> (133.75)
rc206.4	38	893.34	845.04 (100.66)	16	831.67 (46.81)	831.67 (418.59)
rc207.1	34	797.67	741.53 (47.53)	19	735.82 (70.87)	735.82 (622.47)
rc207.2	31	721.39	718.09 (52.55)	21	701.25 (61.38)	701.25 (703.03)
rc207.3	with initial tour from [9]: 684.40			32	684.40 (50.31)	684.40 (545.69)
	33	750.03	684.40 (55.91)	23	746.27 (113.44)	697.93 (1128.64)
rc207.4	6	119.64	119.64 (0.16)	5	119.64* (0.50)	
rc208.1	with initial tour from [9]: 804.41			37	795.58 (54.90)	795.58 (982.90)
	38	812.23	799.19 (88.20)	37	852.24 (53.81)	793.61 (1141.24)
rc208.2	with initial tour from [9]: 543.41			28	543.41 (67.11)	543.41 (900.98)
	29	584.14	543.41 (67.06)	28	533.78 (59.78)	533.78 (905.16)
rc208.3	with initial tour from [9]: 654.27			35	655.50 (122.29)	659.07 (2472.65)
	36	691.50	660.15 (87.25)	35	654.27 (197.54)	649.11 (2420.97)

Notes:  $q = 15$  for our results

CPU times for [9] were found using a SPARCserver 330.

\*Indicates a guarantee of optimality.

<sup>+</sup>453.48 was verified to be optimal for rc203.1 with  $k=13$  and  $q=25$  in 15.56 seconds.

<sup>+</sup>574.42 was verified to be optimal for rc206.3 with  $k=14$  and  $q=20$  in 38.29 seconds.

On the 8 problems where our algorithm had the most difficulty, we then used the solution generated by Gendreau et al. as the initial tour, and we improved that solution in 5 of these 8 instances. Note that the solution we received from Gendreau et al. was sometimes different than the solution portrayed in the results found in [9]. As mentioned earlier, when a guarantee of optimality was not found, a heuristic based on the Or-Opt heuristic [15] was applied once, and then our algorithm was applied again. This is why the computation time can vary by a factor of two 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).

Since all of the problems in Table 3.2 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. Table 3.3 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.

Table 3.3: Symmetric TSP's with Time Windows (Vehicle Routing)

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

Notes: \*Indicates a guarantee of optimality.

+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

Asymmetric time window problems of a much larger size [3] were recently made available at <http://www.zib.de/ascheuer/ATSPTWinstances.html>. They are based on real-world instances of stacker crane routing problems. 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 [3], 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

Table 3.4: Asymmetric TSP's with Time Windows (Stacker Crane Routing)

Problem	$n$	Required $K$	Our Results (CPU seconds)		Lower Bound from [3]
			$K = 12$	$K = 15$	
rbg101a	11	5	149* (0.95)		
rbg016a	17	6	179* (1.72)		
rbg016b	17	12	142* (2.71)		
rbg017	16	7	148* (2.51)		
rbg017.2	16	11	107 <sup>1</sup> (9.80)		
rbg017a	18	12	146* (3.23)		
rbg019a	20	4	217* (2.03)		
rbg019b	20	9	182* (3.42)		
rbg019c	20	13	190 <sup>2</sup> (7.64)		
rbg019d	20	5	344* (2.39)		
rbg020a	21	13	210 <sup>2</sup> (3.59)		
rbg021	20	13	190 <sup>2</sup> (7.82)		
rbg021.2	20	13	182 <sup>2</sup> (9.00)		
rbg021.3	20	13	182 <sup>3</sup> (9.60)		
rbg021.4	20	14	179 <sup>3</sup> (11.52)		
rbg021.5	20	15	169 (12.35)	169 <sup>1</sup> (127.97)	
rbg021.6	20	15	147 (14.98)	134 <sup>1</sup> (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*
rbg027a	28	19	268 (11.28)	268 (137.66)	268*
rbg031a	32	8	328 <sup>1</sup> (11.13)		
rbg033a	34	9	433* (5.66)		
rbg034a	35	10	401 <sup>1</sup> (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 <sup>1</sup> (20.08)		
rbg041a	42	11	402 <sup>1</sup> (24.57)		
rbg042a	43	13	411 <sup>4</sup> (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: \*Indicates a guarantee of optimality

<sup>1</sup>Indicates value guaranteed 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)

<sup>2</sup>Indicates value guaranteed to be optimal with required  $K$  and  $q = 15$   
 rbg019c (8.52), rbg020a (8.12), rbg021 (8.68)

<sup>3</sup>Indicates value guaranteed to be optimal with required  $K$  and  $q = 20$   
 rbg021.2 (11.35), rbg021.3 (11.83), rbg021.4 (29.09)

<sup>4</sup>rbg042a was verified optimal with  $K = 13$  and  $q = 30$  (61.94)

Table 3.5: Asymmetric TSP's with Time Windows (Stacker Crane Routing)

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

Notes:  $q = 15$  unless specified

\*Indicates a guarantee of optimality

<sup>1</sup>Indicates value guaranteed to be optimal with  $q = 20$  (CPU sec)  
rbg055a (16.21), rbg067a (18.83), rbg092a (30.33), rbg132 (39.11)

<sup>4</sup>Value generated with  $q = 4$

<sup>6</sup>Value generated with  $q = 6$

we could guarantee optimality was a problem with 152 nodes. Tables 3.4 and 3.5 list our results with a value of  $q = 15$ , compared with the lower bounds in [3] 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  $55 \leq n \leq 232$ , guaranteed optimal solutions were found in half of all the cases in less than one minute.

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

### 3.4 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 3.6 shows the results of using our algorithm on a set of TSP's with Target Times. To generate instances of this problem, we used the same data studied by Potvin et al. [16] and Gendreau et al. [9], and used the time window midpoints for the target times. As can be seen from Table 3.6, 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. Arc costs for the problems in Table 3.6 generally ranged from 10 to 100 units, and 1 time unit is required to travel one distance unit.

As Table 3.6 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 3.6: Symmetric TSP's with Target Times

Problem	$n$	Req. $K$	Results for $K = 10$			Results for $K = 14$			Results for $K = 17$		
			$d$	cost	sec.	$d$	cost	sec.	$d$	cost	sec.
rc201.1	20	5	81.92*	611.10	1						
rc201.2	26	6	99.56*	875.73	1						
rc201.3	32	5	93.53*	869.68	2						
rc201.4	26	7	114.50*	891.93	2						
rc202.1	33	$\approx 19$	277.38	875.97	7	254.08	864.32	139	254.08	864.32	1337
rc202.2	14	8	118.12*	552.18	2						
rc202.3	29	8	134.62*	889.27	4						
rc202.4	28	17	244.41	925.77	6	237.26	900.66	103	273.26*	900.66	871
rc203.1	19	14	220.23	610.81	3	220.23*	610.81	56			
rc203.2	33	$\approx 25$	387.57	974.42	8	387.57	974.42	174	383.87	972.58	1976
rc203.3	37	$\approx 27$	535.94	975.11	9	482.71	957.49	221	403.46	950.40	2452
rc203.4	15	11	211.98	598.25	2	211.98*	598.25	30			
rc204.1	46	$\approx 43$	770.78	1078.49	14	658.42	1022.31	335	623.43	966.05	4101
rc204.2	33	$\approx 30$	453.11	781.41	9	434.94	793.15	221	427.36	789.36	2332
rc204.3	24	$\approx 23$	321.52	659.22	7	321.52	659.22	136	321.52	659.21	1122
rc205.1	14	3	53.03*	473.12	1						
rc205.2	27	12	186.27	816.47	5	186.27*	816.47	72			
rc205.3	35	15	240.43	965.58	7	240.43	965.58	135	240.43*	965.58	1117
rc205.4	28	11	159.55	883.81	4	159.55*	883.81	64			
rc206.1	4	3	22.08*	227.04	1						
rc206.2	37	13	202.38	902.06	8	202.38*	902.06	136			
rc206.3	25	11	145.93	697.63	3	145.93*	697.63	51			
rc206.4	38	14	204.53	941.35	8	204.53*	941.35	156			
rc207.1	34	17	234.99	906.66	7	234.99	906.66	139	234.99*	906.66	1085
rc207.2	31	$\approx 20$	241.18	798.50	7	241.18	798.50	126	241.18	798.50	1348
rc207.3	33	$\approx 18$	275.88	896.67	8	259.51	880.36	163	259.51	880.36	1793
rc207.4	6	1	12.08*	309.85	1						
rc208.1	38	$\approx 28$	449.78	936.43	10	406.04	914.56	242	375.77	899.40	2703
rc208.2	29	$\approx 21$	301.41	779.39	6	267.46	762.42	113	267.46	762.42	1072
rc208.3	36	$\approx 24$	297.89	824.90	8	285.85	818.67	177	285.85	818.67	2082

Notes: \*Indicates a guarantee of optimality (within .01).  
 $d$  is the value for the minimum window size returned by the algorithm.  
Blanks indicate there was no run since optimality was proved with a lower  $K$ .

# Chapter 4

## Local Search for Arbitrary TSP's

### 4.1 Iterating the Dynamic Program

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 (i). 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  $n$ . Although this property is not unique for our heuristic, it nevertheless bodes well for its approach. On the other hand, the outcome of a run of our procedure depends crucially on the starting tour, more so – it seems – than in the case of the interchange procedures. Which suggests that the best use of our approach as a heuristic is to apply it to the best tour found by some interchange (or other) heuristic, based on searching a different neighborhood.

On the other hand, when used as a heuristic, our procedure can be iterated as follows. Suppose we run the algorithm on a graph  $G_0$  with the starting tour  $T_0$  given by the sequence  $\{1, \dots, n\}$  and with a single value  $k$  for all cities. Let  $T_0^*$  be the tour found by the algorithm.  $T_0^*$  represents a local optimum over the neighborhood of  $T_0$  defined by condition (i). Suppose now that we have some heuristic measure of how “good” each arc of  $T_0^*$  is, i.e. how likely it is to belong to an optimal tour, and we select (either deterministically, or by a biased random choice) a subset  $S_0$  of size  $\alpha n$  (for some  $0 < \alpha < 1$ ) of the arcs of  $T_0^*$ . For each  $(i, j) \in S_0$ , we contract the arc  $(i, j)$  into a single node  $ij$ , joined to each node  $\ell \in N \setminus \{i, j\}$  by a pair of directed arcs  $(\ell, ij)$  and  $(ij, \ell)$ , with costs  $c_{\ell, ij} := c_{\ell i}$  and  $c_{ij, \ell} := c_{j \ell}$ . Let  $G_1$  be the graph obtained from  $G_0$  this way, and let  $T_1$  be the tour in  $G_1$  corresponding to the tour  $T_0^*$  in  $G_0$  (i.e. the arcs of  $T_1$  are those arcs of  $T_0^*$  that have not been contracted). Note that, irrespective of whether  $G_0$  is a directed or undirected graph,  $G_1$  is a digraph: if  $G_0$  is

undirected, its edges – other than those incident with  $i$  and  $j$  – are replaced in  $G_1$  by a pair of arcs with the same cost as the corresponding edge of  $G_0$ .

Now apply our procedure to  $G_1$ , with  $T_1$  as the starting tour and with the same  $k$  that has been used for  $G_0$ . Since  $T_1$  is shorter than  $T_0^*$  but  $k$  is the same, the neighborhood of  $T_1$  defined by (i) will contain arcs not contained in the neighborhood of  $T_0^*$  defined the same way. Thus while  $T_0^*$  is optimal over the neighborhood of  $T_0$  and therefore is likely to be optimal over its own neighborhood,  $T_1$  is less likely to be, and if it is not, then running the algorithm on  $G_1$  will improve upon the tour  $T_1$  and find a locally optimal tour  $T_1^*$ . Clearly,  $T_1^*$  can be expanded into a tour  $T_1^{**}$  in  $G_0$  by “decontracting” the contracted arcs of  $T_0^*$ , i.e. by setting

$$T_1^{**} := T_1^* \cup (T_0^* \setminus T_1),$$

where the different tours are viewed as arc sets. While the decontracted arcs in  $T_0^* \setminus T_1$  have the same cost in  $T_1^{**}$  as in  $T_0^*$ , the arcs of  $T_1^{**}$  coming from  $T_1^*$  have a lower total cost than the corresponding arcs of  $T_1$  whenever  $T_1^*$  differs from  $T_1$ . Thus  $T_1^{**}$  is a guaranteed improvement over  $T_0^*$ , unless  $T_1^*$  happens to be the same as  $T_1$ . This procedure can be applied iteratively.

To determine which arcs should be contracted, we first randomly choose a number  $t$ ,  $0 \leq t \leq n - k$  to represent approximately how many arcs should be contracted. Then we assign to each arc  $a$ , a value  $good(a)$  between 0 and 1, representing the “goodness” of the arc (more on this below). These values are then scaled to make their sum equal to  $t$ . These new values are used as probabilities to determine whether an arc should be contracted or not. It is possible for some of these probabilities to be greater than 1 (meaning automatic contraction), but in general, the number of arcs contracted is close to the randomly selected  $t$ .

We experimented with five rules for assigning the “goodness” value  $good(a)$  for an arc  $a$ . (1) After generating  $n$  least cost rooted arborescences, where each node is used as a root, we assign  $good(a)$  to be the fraction of arborescences in which the arc  $a$  is used. (2) Squaring the value in (1). (3) We assign  $good(a)$  to be the fraction of the cost of  $a$  divided by the cost of the most expensive arc in the tour. (4) Squaring the value in (3). (5) We assign  $good(a) = 1$  to every arc  $a$ . This last choice makes the random selection of arcs to be contracted unbiased.

In our tests, we noticed that in most cases, improvements were found when less than half the arcs were contracted, in our experiments we choose  $t$  by selecting the lesser of two random numbers between 0 and  $n - k$ . Table 4.1 shows the results of our tests. The methods based on squared values tended to be the least promising, probably because this increases the

Table 4.1: Experimenting with Contraction Rules

TSPLIB problem	starting value	Arborescences		Arc Costs		Random rule (5)
		rule (1)	rule (2)	rule (3)	rule (4)	
fl417	0.23	0.05	0.05	0.05	0.05	0.05
d493	1.46	1.16	1.39	1.30	1.39	1.30
att532	1.57	1.26	1.48	1.38	1.46	1.40
pa561	2.71	2.57	2.57	2.57	2.57	2.57
u574	2.19	1.97	1.99	1.97	1.97	1.97
rat575	2.72	2.39	2.72	2.42	2.07	2.66

Notes: Numbers given are percent over optimal value given in the TSPLIB  
The starting value was obtained by the best of 500 iterations of 3-opt.  
The rule values were generated with  $k = 8$ .  
The code was allowed to run for 300 CPU seconds.

bias to such a large degree that the randomness is lost. The method based on arborescences (unsquared) does best, but the improvement over using the arc costs for our bias did not seem large enough to justify the extra time required to find these arborescences (which is at best  $O(n^2)$ ). For this reason, we chose to use the bias based on arc costs for our results in this chapter.

## 4.2 Combining the Dynamic Program and Interchange Heuristics

To examine how our iterated dynamic programming algorithm would perform in combination with some other heuristic using a different neighborhood, we implemented a simple linear-time version of 3-opt, using a randomized nearest-neighbor tour as the starting solution, and with neighbor lists of size 40. We ran the 3-opt routine alone for 10,000 iterations, and recorded the solution and the time required. We then combined the 3-opt routine with our iterated algorithm in the following way: after 1000 iterations of 3-opt, we alternated running our iterated algorithm and the 3-opt heuristic for 30 second intervals until a comparable total time to the 10,000 3-opt iterations was reached. The 3-opt routine always worked from the randomized nearest-neighbor tour. Our algorithm would always choose to work on the best available solution given equal amounts of time from our algorithm; so if one 3-opt tour was improved to find a tour of cost  $x$  in time  $t$ , if the next best 3-opt tour had failed to be improved to a tour of cost  $x$  or less in time  $t$ , our algorithm would turn its effort back to the first tour. Results in Tables 4.2 through 4.4, show our results with  $K = 6, 8,$  and  $10,$

Table 4.2: Symmetric Problems from TSPLIB ( $100 \leq n \leq 400$ )

Problem	3-opt alone	$K = 6$	$K = 8$	$K = 10$
kroA100	0.00 (0.83)	0.00 (0.80)	0.00 (0.80)	0.00 (0.78)
kroB100	0.00 (0.44)	0.00 (0.40)	0.00 (0.40)	0.00 (0.39)
kroC100	0.00 (0.07)	0.00 (0.04)	0.00 (0.04)	0.00 (0.03)
kroD100	0.00 (0.66)	0.00 (0.63)	0.00 (0.63)	0.00 (0.63)
kroE100	0.00 (0.99)	0.00 (0.96)	0.00 (0.97)	0.00 (0.97)
rd100	0.00 (1.67)	0.00 (1.64)	0.00 (1.66)	0.00 (1.63)
eil101	0.00 (29.17)	0.00 (172.65)	0.00 (34.94)	0.00 (205.81)
lin105	0.00 (0.13)	0.00 (0.09)	0.00 (0.10)	0.00 (0.09)
pr107	0.00 (3.47)	0.00 (3.42)	0.00 (3.47)	0.00 (3.37)
gr120	0.00 (61.96)	0.00 (95.15)	0.00 (158.76)	0.00 (83.25)
pr124	0.00 (3.67)	0.00 (3.63)	0.00 (3.67)	0.00 (3.61)
bier127	0.00 (182.24)	0.00 (127.73)	0.00 (249.04)	0.00 (253.18)
ch130	0.00 (64.13)	0.00 (74.69)	0.00 (36.03)	0.00 (72.31)
pr136	0.09 (375.11)	0.00 (40.63)	0.03 (400.95)	0.00 (233.52)
pr144	0.00 (0.54)	0.00 (0.46)	0.00 (0.47)	0.00 (0.46)
ch150	0.00 (47.99)	0.00 (87.27)	0.00 (75.94)	0.00 (201.27)
kroA150	0.00 (19.73)	0.00 (19.56)	0.00 (19.91)	0.00 (19.41)
kroB150	0.00 (7.98)	0.00 (7.84)	0.00 (8.02)	0.00 (7.81)
pr152	0.00 (25.16)	0.00 (24.91)	0.00 (25.38)	0.00 (24.81)
u159	0.00 (1.55)	0.00 (1.46)	0.00 (1.46)	0.00 (1.45)
si175	0.00 (13.50)	0.00 (13.39)	0.00 (13.45)	0.00 (13.27)
brg180	0.00 (0.44)	0.00 (0.31)	0.00 (0.32)	0.00 (0.31)
rat195	0.34 (377.5)	0.47 (398.23)	0.56 (402.73)	0.56 (383.83)
d198	0.01 (1384.38)	0.09 (1401.20)	0.09 (1384.13)	0.09 (1413.32)
kroA200	0.00 (301.59)	0.00 (112.46)	0.00 (112.85)	0.00 (134.01)
kroB200	0.05 (482.37)	0.03 (482.00)	0.00 (359.27)	0.08 (495.41)
ts225	0.00 (0.77)	0.00 (0.65)	0.00 (0.65)	0.00 (0.63)
tsp225	0.28 (559.71)	0.00 (315.01)	0.23 (559.77)	0.00 (525.96)
pr226	0.00 (430.19)	0.00 (133.30)	0.00 (136.06)	0.00 (134.09)
gil262	0.29 (732.76)	0.08 (735.46)	0.46 (746.32)	0.29 (732.51)
pr264	0.00 (30.69)	0.00 (30.20)	0.00 (30.48)	0.00 (30.15)
a280	0.54 (710.63)	0.50 (732.23)	0.43 (711.01)	0.00 (711.00)
pr299	0.23 (960.15)	0.35 (959.06)	0.19 (959.64)	0.09 (961.43)
lin318	0.49 (1357.70)	0.10 (1357.33)	0.43 (1360.74)	0.20 (1381.54)
rd400	0.96 (1219.73)	0.96 (1219.02)	0.96 (1230.19)	0.96 (1247.63)

Numbers given are percent over optimal value given in the TSPLIB

Table 4.3: Symmetric Problems from TSPLIB ( $417 \leq n \leq 3038$ )

Problem	3-opt alone	$K = 6$	$K = 8$	$K = 10$
fl417	0.07 (3629.40)	0.05 (3629.06)	0.01 (3630.94)	0.01 (3633.05)
pr439	0.45 (2340.52)	0.36 (2343.04)	0.45 (2340.56)	0.37 (2350.29)
pcb442	1.00 (1288.24)	0.87 (1287.16)	0.91 (1311.73)	0.59 (1289.84)
d493	1.04 (2263.76)	0.87 (2279.62)	0.79 (2263.74)	0.98 (2268.01)
att532	1.02 (2245.39)	1.26 (2244.23)	0.95 (2264.51)	1.04 (2247.03)
si535	0.16 (4008.80)	0.11 (4018.34)	0.06 (4036.56)	0.10 (4009.57)
pa561	2.24 (1703.03)	1.74 (1701.56)	1.85 (1716.53)	1.95 (1714.81)
u574	1.40 (2495.02)	1.51 (2494.49)	1.19 (2495.83)	1.26 (2498.64)
rat575	1.85 (1515.63)	1.98 (1538.02)	1.68 (1519.31)	1.79 (1532.23)
p654	0.17 (6213.15)	0.10 (6212.24)	0.05 (6232.55)	0.00 (1999.27)
d657	1.61 (2887.52)	1.40 (2894.52)	1.10 (2915.24)	1.16 (2893.58)
u724	1.82 (2355.20)	1.59 (2354.54)	1.39 (2356.40)	1.59 (2368.36)
rat783	2.56 (2235.65)	2.24 (2233.62)	2.29 (2237.51)	2.21 (2244.87)
dsj1000	2.16 (6685.92)	1.76 (6684.74)	2.10 (6700.42)	2.05 (6730.99)
pr1002	2.26 (4985.73)	1.99 (4993.01)	2.17 (4983.83)	1.72 (4692.03)
si1032	0.10 (4258.06)	0.09 (4254.68)	0.10 (4262.51)	0.09 (4271.17)
u1060	1.81 (6245.76)	1.28 (6244.12)	1.48 (6243.68)	1.43 (6264.65)
vm1084	1.28 (4905.07)	1.55 (4917.53)	1.50 (4905.50)	1.73 (4921.40)
pcb1173	2.63 (4156.50)	1.62 (4154.39)	2.26 (4159.65)	2.20 (4182.38)
d1291	1.59 (7866.47)	1.09 (7879.41)	1.35 (7881.15)	1.35 (7900.07)
rl1304	0.98 (7104.42)	0.77 (7125.00)	0.63 (7109.00)	0.63 (7114.63)
rl1323	1.41 (7016.09)	1.61 (7015.67)	1.47 (7031.85)	1.36 (7030.78)
nrw1379	2.59 (4931.61)	2.10 (4946.48)	2.12 (4933.30)	2.12 (4959.66)
fl1400	0.79 (12817.40)	0.72 (12816.26)	0.49 (12819.74)	0.59 (12833.72)
u1432	3.01 (4209.54)	2.58 (4206.06)	2.36 (4207.47)	2.78 (4217.55)
fl1577*	1.45 (13584.78)	1.62 (13602.17)	1.37 (13590.58)	1.36 (13602.41)
d1655	2.82 (9676.05)	2.08 (9685.27)	2.57 (9680.79)	1.82 (9674.02)
vm1748	2.16 (9743.38)	1.63 (9740.46)	1.90 (9740.83)	1.91 (9740.17)
u1817	2.95 (8395.59)	3.08 (8393.72)	2.21 (8400.01)	2.55 (8430.50)
rl1889	2.54 (11852.84)	1.88 (11855.33)	2.17 (11853.78)	2.61 (11874.56)
d2103*	3.90 (14117.94)	4.16 (14240.48)	4.00 (14241.26)	4.83 (14271.90)
u2152	3.51 (9315.85)	3.10 (9309.12)	2.96 (9328.00)	3.21 (9376.30)
u2319	1.79 (6342.06)	1.54 (6336.55)	1.48 (6335.47)	1.41 (6383.60)
pr2392	3.41 (11375.29)	2.88 (11368.84)	2.83 (11385.66)	2.79 (11443.76)
pcb3038	3.48 (13906.65)	2.84 (13921.05)	2.92 (13904.75)	3.20 (13977.70)

Notes: Numbers given are per cent over optimal value given in the TSPLIB  
 \*Indicates percentages are over the lower bound given in the TSPLIB

Table 4.4: Asymmetric Problems from TSPLIB

Problem	3-opt alone	$K = 6$	$K = 8$	$K = 10$
kro124p*	0.56 (114.84)	1.17 (129.34)	0.15 (129.41)	1.19 (130.07)
ftv170	2.87 (190.53)	2.00 (199.55)	2.87 (200.24)	2.87 (201.37)
rbg323	0.53 (6130.57)	0.15 (3600.05)	0.23 (3600.24)	0.38 (3602.26)
rbg358	1.12 (5003.40)	0.34 (3600.13)	0.52 (3600.09)	0.52 (3600.40)
rbg403	0.04 (21306.78)	0.00 (2138.93)	0.00 (2164.69)	0.00 (2178.55)
rbg443	0.15 (26340.59)	0.11 (3626.81)	0.11 (3600.76)	0.11 (3621.94)

Notes: Numbers given are per cent over optimal value given in the TSPLIB  
 \*Problem kro124p has 100 cities

compared to 3-opt alone. The data in Tables 4.2 and 4.3 are symmetric TSP instances from the TSPLIB. Those in Table 4.4 are asymmetric TSP instances from the TSPLIB.

Notice that in most instances the combined algorithm performed better than 3-opt by itself. Table 4.5 shows the improvement our iterated heuristic finds from the base tour generated by 3-opt and reveals an important phenomenon: the improvement *depends very little on the quality of the 3-opt solution*, even for the largest problems (more than 1000 nodes)! This can be attributed to the fact that the neighborhood our algorithm examines is very different from the neighborhoods examined by interchange heuristics. This makes us believe that if our algorithm were combined with a higher quality interchange heuristic (like iterated

Table 4.5: Improvement from 3-opt solutions

Average improvement:	Improvement based on 3-opt solution after:			
	1 minute	5 minutes	10 minutes	15 minutes
<b>all problems</b>				
during first minute:	0.186%	0.137%	0.118%	0.136%
from 1st to 5th minute:	0.047%	0.039%	0.045%	0.045%
from 5th to 10th minute:	0.032%	0.046%	0.023%	0.018%
<b>problems with <math>n &lt; 1000</math></b>				
during first minute:	0.216%	0.167%	0.134%	0.143%
from 1st to 5th minute:	0.022%	0.024%	0.043%	0.041%
from 5th to 10th minute:	0.027%	0.023%	0.020%	0.011%
<b>problems with <math>n \geq 1000</math></b>				
during first minute:	0.141%	0.095%	0.096%	0.125%
from 1st to 5th minute:	0.083%	0.061%	0.048%	0.051%
from 5th to 10th minute:	0.039%	0.077%	0.027%	0.028%

Lin-Kernighan) we would obtain improvements that would surpass the interchange heuristic alone. In order to test this idea, we asked Gerhard Reinelt for the tours he obtained with the iterated Lin-Kernighan algorithm he used for the results published in [11]. Unfortunately those tours are no longer available, but Reinelt [17] kindly sent us 10 tours which are actually of higher quality than those in [11]. In Table 4.6 we used these tours as starting points for our algorithm. Of the 10 tours sent by Reinelt, there were 7 on problems from the set we were studying. Of these, there were 4 that were not within .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.

Table 4.6: Asymmetric Problems from TSPLIB

Problem	Reinelt's tour	Improvements (CPU time)
rl1323	0.377	0.372 (5.07)
fl1400	0.263	No Improvement
d1655	0.034	0.031 (8.07)
d2103	0.690*	0.685* (10.39); 0.664* (38.17)

Notes:

Our Algorithm was run with  $k = 8$  for one hour of CPU time.

Numbers are per cent over optimal value.

\*Indicates per cent over best know lower bound

Reinelt's tour for d2103 is 0.067% over the best known upper bound.

Our best tour for d2103 is 0.041% over the best known upper bound.

Recently, the state-of-the-art chained Lin-Kernighan (CLK) code of Applegate, Bixby, Chvatal and Cook [2] was made available. This is a sophisticated implementation of the algorithm described in [14]. We ran that code on the 22 symmetric TSPLIB problems between 1000 and 3500 nodes for 100, 1000, and 10,000 “kicks” (a “kick” is an attempt to push a solution out of a local optimum). We then applied our algorithm to the resulting tours with  $k = 6, 8,$  and  $10$ .

In 20 of the 22 instances, we found improvements over the solution obtained by the CLK code after 100 kicks with each value of  $k$ ; however, at this point, the rate of improvement was no better than continuing the CLK code. In 10 of the 22 instances, we found improvements on the solution found after 1000 kicks with each value of  $k$ , and in over half of these cases, an improvement was found after just the first iteration of our algorithm. However, outside of this initial surge, the rate of gain from our algorithm did not exceed that of continuing the CLK code, with one notable exception mentioned below. In 4 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 our algorithm. At

Table 4.7: Using CLK Solution as Starting Tour.

Problem	CLK value after 10,000 kicks	Our Improvements with $k = 6$ (CPU seconds)
rl1304	0.0004	optimal (0.52)
d1655	0.0338	0.0225 (0.67)
vm1748	0.1408	0.1405 (0.70)
rl1889	0.1254	0.1238 (0.76); 0.1235 (5.87); 0.1147 (11.15)

Numbers are per cent over optimal value.

this point, the CLK code was no longer finding improvements consistently (in fewer than half of the 22 instances were improvements found between the 5,000th and 10,000th kick, and in only 4 of the 22 instances were improvements found between the 9,000th and 10,000th kick). In one of these four cases, the CLK code had found no additional improvements after the 1,000th 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 9,000 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 4.7.

For the three cases where solutions were not yet optimal, we fed our improved tour back to the CLK code for 10,000 more kicks, and compared these numbers to those found by simply applying the CLK code alone for 20,000 kicks. For problem rl1304, as mentioned above, the optimal solution was found much more quickly by combining the algorithms. For problem vm1748, the CLK code was able to find an additional improvement over our

Table 4.8: CLK Improvements Between 10,000 - 20,000 Kicks

Problem	Value at 10,000th kick	Improvements
rl1304	0.0004 [954] (23.81)	optimal [10632] (255.64)
d1655	0.0338 [7742] (155.04)	0.0177 [14517] (286.01); 0.0145 [14522] (286.06); 0.0113 [15238] (300.16)
vm1748	0.1408 [2739] (115.70)	0.1168 [11096] (469.38); 0.1144 [11109] (469.87)
rl1889	0.1254 [8186] (349.52)	0.1242 [10793] (460.71); 0.1235 [12531] (535.25); 0.1197 [16120] (683.02); 0.1147 [16121] (683.03)

Notes: Numbers are per cent over optimal value.  
Number in square brackets indicates on which kick the improvement was found.  
Number in parentheses indicate the cumulative CPU seconds needed to find the improvement.

Table 4.9: CLK Improvements After Given our Tour

Problem	Our value	Further improvements by CLK
rl1304	optimal	Already optimal after our application
d1655	0.0225	No further improvement found
vm1748	0.1405	0.1385 [400] (17.38)
rl1889	0.1238	0.1229 [7199] (287.11); 0.1200 [7239] (289.02); 0.0875 [7249] (289.25); 0.0872 [7265] (289.87); 0.0847 [7267] (289.92)

Notes: Numbers are per cent over optimal value.  
Number in square brackets indicates on which kick the improvement was found.  
Number in parentheses indicate the cumulative CPU seconds needed to find the improvement.

solution much more quickly than without using our solution. For problem rl1889, while the additional improvement the CLK code found to our solution took longer than without using our solution, the improvement found after an application of our algorithm quickly led to a much bigger improvement than without the application of our algorithm. These results can be found in Tables 4.8 and 4.9.

Our explanation for the phenomenon described above is simple: the neighborhood over which our algorithm finds a local optimum is sufficiently different from the neighborhood searched by the CLK algorithm to make a local optimum for one of the algorithms be a nonoptimal point for the other, improvable within its neighborhood. Based on these results, it is likely that the chained Lin-Kernighan code could be improved by applying an iteration of our algorithm, even with only a small value of  $k$ , whenever the Lin-Kernighan code gets stuck. This may be an area of future study, which we think holds great promise.

Our algorithm seems to perform better on Euclidean or nearly Euclidean problems. Table 4.10 illustrates this, as we generated four asymmetric 500 node problems using the *genlarge* problem generator, which Bruno Repetto [18] kindly gave to us. Repetto solved these problems with an open-ended heuristic approach, where he would produce an initial tour by randomly choosing one of the nearest two neighbors, and then apply his implementation [19] of the Kanellakis & Papadimitriou heuristic [12], an adaptation to asymmetric TSP's of the Lin-Kernighan heuristic for the symmetric TSP [13]. This process would continue for 1200 seconds and the best solution found would be returned. The tours generated were used as initial tours for our algorithm.

We suspect this behavior has to do with the fact that for Euclidean problems, if two cities

Table 4.10: Random versus Almost-Euclidean Asymmetric TSP's

Problem	Open-Ended K&P Result	Our Results (seconds)		
		$K = 10$	$K = 14$	$K = 17$
500.aa	2964	no improvement		
500.ba	7495	no improvement		
500.ca	21703	21690 (13)	21632 (290)	21618 (3641)
500.da	9950	9886 (14)	9857 (298)	9844 (3605)

Notes: The Kanellakis & Papadimitriou heuristic was run for 1200 seconds.  
 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.

$i$  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 a low cost. For 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. The geographic problems are even better suited for our algorithm because cities tend to cluster in metropolitan areas, which would tend to imply precedence constraints of the type (i). But this takes us to our next section.

### 4.3 Clustered TSP's

It is not infrequent to encounter TSP's in practice whose cities display an uneven spatial distribution in the sense of forming clusters. If this property is sufficiently pronounced, a special class of TSP's emerges that are solvable by our approach. We formalize the notion of clustering as follows.

A partition  $(N_1, \dots, N_q)$  of the set  $N$  of cities into  $q$  subsets,  $2 \leq q \leq n$ , will be called a *proper clustering* of  $N$  if every optimal tour in  $G$  enters (exits) exactly once each cluster  $N_i$ ,  $i = 1, \dots, q$ . The ordering  $N_1, \dots, N_q$  of the clusters will be termed *optimal* if every optimal tour visits the clusters in that sequence. It is not hard to give polynomially verifiable sufficient conditions (see [4]) for a partition of  $N$  to be a proper clustering, and for the latter

to be optimally ordered.

**Proposition 4.3.1** *Let  $N_1, \dots, N_q$  be an optimally ordered proper clustering of  $N$ , and let  $T$  be a tour in  $G$  with node sequence  $1, \dots, n$  such that  $T$  enters (exits) each cluster  $N_i$  exactly once, in the order  $i = 1, \dots, q$ . For  $j = 1, \dots, n$ , define  $k(j) := |N_\ell|$ , where  $\ell$  is the unique index such that  $j \in N_\ell$ . Then for every optimal tour in  $G$  defined by a permutation  $\pi$  on  $\{1, \dots, n\}$ ,  $j \geq i + k(i)$  implies  $\pi(i) < \pi(j)$ .*

**Proof.** We show that any tour for which the claim is false is not optimal. Indeed, let  $T_0$  be a tour in  $G$  defined by a permutation  $\pi_0$  on  $\{1, \dots, n\}$ , such that for some  $i, j \in \{1, \dots, n\}$ ,  $j \geq i + k(i)$  but  $\pi_0(i) > \pi_0(j)$ . Then  $i$  and  $j$  belong to different clusters, say  $N_s$  and  $N_t$ , respectively, with  $s < t$ . If  $\pi_0(k) > \pi_0(\ell)$  for all  $k \in N_s, \ell \in N_t$ , then  $T_0$  traverses  $N_t$  before  $N_s$ , hence it cannot be optimal. Otherwise  $T_0$  enters  $N_t$  to visit node  $j$  (and possibly other, but not all nodes of  $N_t$ ), then enters  $N_s$  to visit  $i$ , and then returns to  $N_t$  to visit (some or all of) the remaining nodes of that cluster. But from the definition of a proper clustering, this implies that  $T_0$  is not optimal.  $\square$

Thus TSP's (symmetric or asymmetric) satisfying the above definition of a proper clustering can be solved by our approach in time linear in  $n$ , exponential only in the size of the largest cluster.

Although the requirements for a proper clustering as defined here may not too often be satisfied, most “geographic” TSP's, i.e. TSP's in which one seeks a tour of the cities of some specified geographic area that minimizes total travel costs based on inter-city distance, do display some degree of clustering; which should make our approach a promising heuristic for this class of problems (see Figure 4.1). For example, 100 of the largest 1000 places in the United states are within 100 kilometers of Long Beach, CA. More than 50 of the largest 1000 places in the United states are within 100 kilometers of each of the following cities: New York, NY; Oakland, CA; Waukegan, IL; Worcester, MA.

To test this hypothesis, we generated symmetric TSP's of various sizes using the  $t$  largest cities of the United States, for  $t = 200, 300, 400, 500, 600, 750, 1000$ . Population figures and coordinates were obtained from the United States Census Bureau [22]. Distances were calculated based on these coordinates, assuming a perfectly spherical earth with a radius of 6378.15 kilometers. All distances were rounded to the nearest tenth of a kilometer. These problems instances, as well as those of Table 4.10, can be found at <http://www.contrib.andrew.cmu.edu/~neils/testdata>.

We ran on these problems our 3-opt routine for 10,000 iterations, and then ran on the



Figure 4.1: Clustering of the 1000 Largest Cities in the United States

Table 4.11: Symmetric Problems from US geography

Problem	3-opt alone	$K = 6$	$K = 8$	$K = 10$
city200*	0.00 (12.67)	0.00 (12.62)	0.00 (12.67)	0.00 (12.68)
city300	0.18 (1945.35)	0.18 (1945.04)	0.20 (1960.02)	0.16 (1972.58)
city400	0.37 (2960.91)	0.46 (2960.03)	0.42 (2962.33)	0.46 (2964.15)
city500*	0.27 (4103.18)	0.18 (4103.55)	0.23 (4109.24)	0.20 (4115.15)
city600	1.74 (5165.54)	1.80 (5165.24)	1.54 (5186.22)	1.51 (5178.69)
city750	2.13 (6280.25)	1.88 (6280.83)	1.75 (6280.54)	1.65 (6298.61)
city1000	2.00 (9112.67)	1.66 (9112.01)	1.83 (9115.26)	1.70 (9121.10)

Notes: Numbers given are per cent over Held-Karp lower bound  
 \*Indicates numbers given are per cent over optimal value.

same problems the combined dynamic programming/3-opt procedure for the same amount of time, with  $K = 6, 8$  and  $10$ . As Table 4.11 shows, in all the problems with at least 500 cities the combined algorithm performed significantly better.

# Chapter 5

## Single Machine Scheduling

### 5.1 Scheduling with Set-up Times and Release Dates

As mentioned above in Section 3.1, there are many scheduling problems in the literature which can be modelled on the TSP. Those with set-up times can be modeled on an unconstrained TSP. Those with set-up times, release dates and due dates can be modelled with the TSP with time windows. In this section we discuss two additional types of problems, those with set-up times and release dates, but not due dates, and those with set-up times, release dates, and delivery times.

For problems that have release dates without due dates, there would be no cases where the number of feasible tours is polynomial, as we found when we had release dates and due dates forming tight time windows. However, many problems with release dates and not due dates tend to discourage scheduling jobs with early release dates late in the schedule. For example, if, in the process of scheduling the demands of many clients, you schedule a client's job with an early request at the end of your schedule in order to minimize the makespan, finishing many later-received jobs before this client's job, you are likely to lose future business of this client. For this kind of reason, it is generally desired to find a reasonably good schedule which does not upset the first-come first-served ordering too much. Such a soft constraint is tailor-made for our dynamic programming algorithm by varying our parameter  $k$ . One can even customize the solution with client dependent values of  $k$ , giving important clients small values and less important clients large values.

One such real world example of this kind of problem is the Aircraft Sequencing Problem (ASP), studied in [7]. In this problem there is a single runway and various types of aircraft. For every pair of aircraft types, there is a minimum required interval of time between landings to ensure safety. Also, each individual aircraft has a release time, representing the time it

arrives at the runway.

Bianco et al. [7] constructed two realistic problems involving four types of aircraft and a total of 30 and 44 airplanes. Using a branch and bound algorithm on a Vax 11/780, they solved these problems to optimality in 373 and 1956 seconds of CPU time, respectively. Our code not only found optimal solutions to these problems in seconds, but both the average waiting time and the maximum waiting time for the aircraft were lower with our solution ( $K = 12$ ), as Table 5.1 shows.

Table 5.1: Aircraft Sequencing Problem

Number of Aircraft		FCFS*	Bianco [7]	Our Results		
				$K = 8$	$K = 10$	$K = 12$
30	Makespan:	3266	3151	3151	3151	3151
	CPU Time:	–	373	0.04	0.30	1.43
	Avg. Wait:	267.6	214.8	129.3	129.3	129.3
	Max. Wait:	598	896	580	580	580
	Runway Idle:	0	68	177	177	177
44	Makespan:	4952	4064	4160	4072	4064
	CPU Time:	–	1956	0.09	0.50	2.47
	Avg. Wait:	881.4	494.5	499.3	485.3	483.6
	Max. Wait:	1799	1310	1363	1111	1103
	Runway Idle:	0	0	0	0	0

Notes: CPU times for [7] were found using a Vax 11/780.  
 \*First-Come First-Served ordering.

Because our algorithm works so quickly on this type of problem, it could be used on-line by rerunning the optimization code for every unexpected change in schedule, a frequent occurrence for any major airport.

## 5.2 Applications for Job Shop Scheduling

Another common scheduling problem consists of set-up times, release dates and delivery times. Rather than insisting that a job  $j$  is finished by some predetermined time  $t_j$ , we require an amount of time  $q_j$  for post-processing (or delivery) on every job which is unrelated to our machine, but still must be considered when evaluating the makespan. One important application of this method is to optimize single machines which are part of a multiple machine job shop scheduling problem. Such an approach has been called the shifting bottleneck procedure[1].

Given a set of jobs  $J$  to be scheduled on a single machine, where each job  $j \in J$  has a release date  $r_j$ , a processing time  $p_j$ , and a delivery time  $q_j$ , and given an upper bound  $M$  on the time allowed for every job to be finished and delivered, we can derive a time window for each  $j$ , namely  $[r_j, M - p_j - q_j]$ . Given an algorithm that can solve this time window problem, we would have the ability to answer the decision problem, “Is there a solution with makespan  $< M$ ?” for the original single machine scheduling problem. Implementing a binary search, one could find the schedule with the optimal makespan. In cases where these time windows are narrow enough at the optimal makespan, we can solve this problem to optimality with our algorithm, otherwise our algorithm can be used as a heuristic, or combined with other heuristics.

In our computational experience, we have replaced the binary search with a stepping procedure. Setting  $M := \infty$ , we solve the above decision problem (which in this case will always have a feasible solution since no time window will ever close), and find a schedule with an actual makespan of  $M'$ . Then, setting  $M := M' - 1$ , we repeat this procedure until no feasible solution can be found. While the worst case complexity of this approach is worse, in general this method has converged faster in our experience. Furthermore, it allows us to use cheaper methods, (i.e. smaller values of  $K$ ) in the early iterations of the stepping procedure.

Alkis Vazacopoulos has been working on these problems using an Or-opt approach [15]. After creating a tour based on a randomized insertion technique, Or-opt with strings up to some constant size  $\ell$  is used until no further improvement is found. This is iterated 500 times in his code. Vazacopoulos has kindly made his problems and code available to us [24]. Table 5.2 shows our performance on small problems. Table 5.3 compares our results with those of Vazacopoulos for larger problems. A large variance in running times for problems of the same size can occur when convergence in the stepping procedure varies.

We noticed that, for the small problems, optimal solutions were found with values of  $K$  much smaller than  $n$ . In roughly half of the small problems, optimal solutions were found with  $K = \frac{3}{5}n$ , which leads us to believe we may frequently get optimal solutions even when there is no guarantee. On the larger problems ( $n = 50$ ), our code performed badly compared to the iterated Or-opt, which prompted an effort to combine the algorithms, as was done for arbitrary TSP’s in Chapter 4.

We combined to two algorithms in the following way: Anytime an Or-opt iteration found a solution better than the previous best Or-opt solution, that solution and makespan value were fed as starting tour and value for our algorithm, instead of starting with the tour sorted by time window midpoint and a target makespan of infinity. Even if a new Or-opt solution is not better than our best combined solution at some point, as long as the Or-opt solution

Table 5.2: Single Machine Scheduling,  $n \leq 20$ 

Problem	Our Results (CPU seconds)			
$n = 10$	$K = 10$	$K = 8$	$K = 6$	$K = 4$
ps01a	689 (0.08)	689 (0.08)	700 (0.05)	739 (0.00)
ps01b	656 (0.18)	656 (0.07)	656 (0.04)	656 (0.01)
ps01c	649 (0.18)	649 (0.13)	664 (0.02)	669 (0.01)
ps01d	598 (0.08)	598 (0.10)	598 (0.02)	598 (0.01)
ps01e	795 (0.32)	795 (0.15)	796 (0.03)	800 (0.01)
ps02a	710 (0.29)	710 (0.11)	710 (0.01)	729 (0.02)
ps02b	563 (0.24)	563 (0.20)	563 (0.01)	577 (0.02)
ps02c	563 (0.12)	563 (0.12)	573 (0.03)	573 (0.02)
ps02d	735 (0.11)	735 (0.10)	735 (0.03)	735 (0.00)
ps02e	602 (0.12)	602 (0.05)	625 (0.01)	625 (0.03)
$n = 15$	$K = 15$	$K = 12$	$K = 9$	$K = 6$
ps06a	1006 (9.68)	1006 (3.32)	1026 (0.48)	1056 (0.04)
ps06b	838 (7.43)	838 (3.46)	842 (0.50)	862 (0.04)
ps06c	855 (4.80)	855 (3.43)	875 (0.46)	905 (0.02)
ps06d	817 (22.09)	827 (3.39)	827 (0.37)	837 (0.10)
ps06e	895 (9.95)	895 (4.34)	895 (0.43)	925 (0.08)
ps07a	955 (4.93)	955 (4.26)	955 (0.53)	955 (0.07)
ps07b	784 (7.36)	784 (3.43)	784 (0.48)	807 (0.07)
ps07c	810 (9.69)	810 (4.31)	810 (0.41)	810 (0.10)
ps07d	890 (7.41)	890 (3.45)	890 (0.43)	910 (0.09)
ps07e	742 (26.57)	742 (4.49)	742 (0.37)	777 (0.04)
$n = 20$	$K = 17$	$K = 16$	$K = 12$	$K = 8$
ps11a	1393 (230.71)	1402 (110.71)	1402 (6.31)	1402 (0.39)
ps11b	1223 (205.76)	1223 (111.26)	1280 (6.36)	1280 (0.24)
ps11c	1190 (280.73)	1190 (151.98)	1210 (6.25)	1247 (0.51)
ps11d	1059 (204.02)	1099 (111.24)	1129 (6.25)	1129 (0.27)
ps11e	1299 (231.34)	1299 (125.51)	1299 (7.03)	1344 (0.27)
ps12a	1157 (281.22)	1159 (207.34)	1197 (6.27)	1267 (0.27)
ps12b	1155 (378.19)	1155 (204.51)	1199 (6.08)	1239 (0.33)
ps12c	1148 (254.53)	1148 (138.01)	1169 (7.61)	1176 (0.35)
ps12d	1078 (198.60)	1078 (106.82)	1150 (6.80)	1170 (0.32)
ps12e	871 (198.22)	871 (107.30)	891 (5.95)	917 (0.28)

Instances where  $K = n$ , optimality is guaranteed.

Table 5.3: Single Machine Scheduling,  $n = 20, 50$ 

Problem	Iterated Or-opt [24]	Our Results (CPU seconds)			
		$K = 17$	$K = 16$	$K = 12$	$K = 8$
ps20a	626 (154.78)	627 (287.34)	627 (149.06)	639 (12.70)	657 (0.38)
ps20b	644 (165.53)	644 (337.13)	644 (161.18)	658 (7.06)	678 (0.40)
ps20c	691 (149.48)	691 (482.44)	701 (195.83)	705 (9.04)	712 (0.57)
ps20d	623 (171.59)	623 (288.16)	623 (149.90)	637 (9.74)	638 (0.43)
ps20e	726 (157.51)	723 (371.61)	723 (196.44)	739 (12.17)	758 (0.45)
ps50a	1436 (11699.20)	1534 (2182.10)	1534 (1070.46)	1574 (37.74)	1654 (1.08)
ps50b	1685 (11917.66)	1726 (1883.92)	1740 (786.80)	1771 (44.84)	1840 (1.33)
ps50c	1549 (11233.05)	1592 (2502.88)	1592 (1293.06)	1626 (28.17)	1668 (0.90)
ps50d	1508 (12746.32)	1606 (2857.37)	1606 (1163.11)	1645 (37.80)	1744 (1.32)
ps50e	1509 (11740.45)	1541 (1975.50)	1541 (674.71)	1584 (34.11)	1701 (1.04)

Notes: Or-opt was run with 500 iterations and string lengths up to size 5 for  $n = 20$ .  
Or-opt was run with 500 iterations and string lengths up to size 10 for  $n = 50$ .

is better than the previous best Or-opt solution, the information is fed to our algorithm. Rather than use a fixed  $K$  for the stepping procedure, we began with  $K = 10$  and increased  $K$  by two when no feasible solution was found, to maximum of  $K = 16$ . The results of this merger look promising, as Table 5.4 indicates. In many cases, the combined solution value after ten iterations was better than the iterated Or-opt alone after 500 iterations.

Table 5.4: Single Machine Scheduling, Combined Algorithm

Problem	Iterated Or-opt [24]	Combined Algorithm		
		10 iterations	100 iterations	500 iterations
ps50a	1436 (11699.20)	1434 (2526.70)	1432 (5958.48)	1432 (14760.79)
ps50b	1685 (11917.66)	1692 (1921.79)	1673 (4854.82)	1673 (13934.63)
ps50c	1549 (11233.05)	1538 (3905.78)	1538 (5922.63)	1538 (14333.79)
ps50d	1508 (12746.32)	1514 (1903.79)	1496 (5558.93)	1496 (15901.16)
ps50e	1509 (11740.45)	1496 (3464.83)	1496 (8180.72)	1487 (18371.35)

Notes: CPU times in parentheses.  
Or-opt was run with 500 iterations and string lengths up to size 10.  
 $K$  was stepped from 10 to 16 by 2 in the combined algorithm.

Further study can be done by implementing our routine into a shifting bottleneck procedure for a general job shop problem [1].

# Chapter 6

## Prize-Collecting TSP's

### 6.1 Defining a New Dynamic Program

Another direction in which our model can be generalized is to require a tour of  $m < n$  cities rather than to require a tour visiting every city. This type of problem, known as the Prize Collecting TSP [6], serves as a model for some scheduling problems, such as steel rolling mills.

One way to model our precedence constraints, given our integer parameter  $k > 0$ , is to partition the set of cities, excluding the home city, into  $m - 1$  sets,  $C_2, C_3, \dots, C_m$ , one for each position in the tour, excluding the first and last positions which are reserved for the home city. We then require that:

- (a) if city  $i$  from set  $C_a$  and city  $j$  from set  $C_b$  are both visited in the tour, then city  $i$  must follow city  $j$  if  $b \leq a - k$ .

It was shown in [5] that, as consequence of the precedence constraint in our original model, the set of cities eligible to be put in an arbitrary position  $i$  of the tour is  $\{i - k + 1, \dots, i + k - 1\}$ . No similar result is implied by our precedence constraint, so in order to ensure that we have a polynomially sized dynamic program, we must impose a similar constraint, namely:

- (b) the set of cities eligible to be placed in position  $i$  of the tour is  $C_{i-k+1} \cup \dots \cup C_{i+k-1}$ .

Define  $p := \max_i |C_i|$ . If  $p = 1$ , then we have a 1-1 correspondence of cities to positions, and so our previous model would apply, hence I will assume  $p \geq 2$ .

To build our auxiliary structure efficiently, we need an economical representation for each node, similar to three entities we identified in Section 2.1

1.  $j$ , the city in position  $i$ ,
2.  $S^-$ , the set of cities from groups  $C_h, h \geq i$  that are visited in one of the positions 1 through  $i - 1$ . ( $S^- := \{h \in N : h \in \bigcup_{\ell=i}^m C_\ell, h \in M \text{ and } \pi(h) < i\}$ )
3.  $S^+$ , the set of cities from groups  $C_h, h < i$  that are not visited in one of the positions 1 through  $i - 1$ . ( $S^+ := \{h \in N : h \in \bigcup_{\ell=1}^{i-1} C_\ell, h \notin M \text{ or } \pi(h) \geq i\}$ )

$M$  is the set of cities actually visited in the tour, and  $\pi$  is the mapping from these cities to their positions in the tour.

Unfortunately, there is one severe problem with this definition for  $S^+$ . In the case where  $m = \frac{n}{2}$ ,  $S^+$  will grow to size  $\frac{n}{2}$  as we approach the end of our auxiliary structure, since half of the cities will satisfy  $h \notin M$ . However, if we change our definition to reflect that cities in early groups will never be eligible to be visited late in our tour, we can avoid this.

Let  $U$  be the set of cities visited in positions 1 through  $i - 1$  of the tour, and define the following to represent a node  $(i, j, S_{ijr}^-, S_{ijr}^{++})$  of the auxiliary structure:

1.  $j$ , the city in position  $i$ ,
2.  $S_{ijr}^-$ , the set of cities from groups  $C_h, h \geq i$  that are visited in one of the positions 1 through  $i - 1$ . ( $S_{ijr}^- := \{h \in N : h \in \bigcup_{\ell=i}^m C_\ell, h \in U\}$ )
3.  $S_{ijr}^{++}$ , the set of cities from groups  $C_h, h < i$  that are still eligible to be visited in one of the positions  $i + 1$  and later, given the information in  $U$  and  $j$ .

We will prove there is a more rigorous equivalent definition for  $S_{ijr}^{++}$  later. At this point, notice that our definition of  $S_{ijr}^{++}$  considers only elements eligible for positions  $i + 1$  and beyond, rather than  $i$  and beyond. The reason for this is to reduce the potential number of states in a layer. In the old model, knowing the set of cities visited before and after position  $i$ , one could deduce the only candidate for position  $i$ , thus creating no additional states for encoding this information in  $S^+$ . However, in the prize-collecting TSP, knowing the set of cities visited before and after position  $i$  still may leave several candidates for position  $i$ , and so encoding this information in  $S_{ijr}^{++}$  unnecessarily increases the number of states. The use of the subscript  $r$  instead of using the subscript  $U$  was implemented because, given a fixed  $i$  and  $j$ , different partial tours  $U$  may have identical pairs  $S_{ijr}^-$  and  $S_{ijr}^{++}$ .

## 6.2 Results from the New Definition

**Remark 6.2.1**  $S_{ijr}^- = \{h \in \bigcup_{\ell=i}^{i+k-2} C_\ell : h \in U\}$

This is a direct consequence of constraint (b).  $\square$

**Proposition 6.2.2**  $S_{ijr}^{++} = \{h \in \bigcup_{\ell=u}^{i-1} C_\ell : h \notin (U \cup \{j\})\}$  where  $u = \max\{i-k+2, c-k+1\}$  and  $c$  is defined so that  $\max(U \cup \{j\}) \in C_c$

**Proof.** Since  $S_{ijr}^{++}$  only contains cities eligible to be visited at or after position  $i+1$ , constraint (b) tells us the smallest indexed group which could contribute to  $S_{ijr}^{++}$  is  $C_{i-k+2}$ . Since  $S_{ijr}^{++}$  only contains cities eligible to be visited based on information in  $U$  and  $\{j\}$ , constraint (a) tells us the smallest indexed group which could contribute to  $S_{ijr}^{++}$  is  $C_{c-k+1}$ , where  $c$  is defined as above.  $\square$

Note that in the case where  $u = i$ ,  $S_{ijr}^{++}$  will be empty.

**Proposition 6.2.3** Given  $i$ ,  $2k < i < m - 2k$ , and given  $S^- \subset \bigcup_{\ell=i}^{i+k-2} C_\ell$ ,  $S^- = S_{ijr}^-$  for some  $j, r$  if and only if

$$\text{for each } q, 1 \leq q \leq k-1, \left| S^- \cap \left( \bigcup_{\ell=i+k-1-q}^{i+k-2} C_\ell \right) \right| \leq q. \quad (6.1)$$

**Proof.** To show the condition is necessary, let  $1 \leq q \leq k-1$ . By constraint (b), cities in  $\bigcup_{\ell=i+k-1-q}^{i+k-2} C_\ell$  must be assigned to positions at or above  $i-q$ , so at most  $(i-1) - (i-q) + 1 = q$  cities can be visited before position  $i$  in the tour.  $U$  describes the before position  $i$ , and so  $|S^- \cap \bigcup_{\ell=i+k-1-q}^{i+k-2} C_\ell| \leq q$ .

To show the condition is sufficient, construct a tour  $T$  such that  $S^- = S_{ijr}^-$  for some  $j$  and  $r$ .

For positions  $1 \leq h \leq i - |S^-| - 1$ , choose the first element of  $C_h$ .

For positions  $i - |S^-| \leq h \leq i - 1$ , choose the  $(i-h)$ th largest indexed city in  $S^-$ .

Note that this is feasible because (6.1) ensures that the  $q$ th largest indexed city in  $S^-$  comes from  $\bigcup_{\ell=i}^{i+k-1-q} C_\ell$ .

For positions  $i \leq h \leq i+k-1$ , choose the second element of  $C_{h+k-1}$ .

For positions  $i+k \leq h \leq m$ , choose the first element of  $C_h$ .  $\square$

**Proposition 6.2.4** *Given  $i$ ,  $2k < i < m - 2k$ , and given  $S^-$  satisfying Proposition 6.2.3,  $S^- = S_{ijr}^-$  for some  $r$  if and only if*

$$j \in \left( \bigcup_{\ell=u}^{i+k-1} C_\ell \right) \setminus S^-, \quad \text{where } u = \max\{i - k + 1, c - k + 1\} \quad (6.2)$$

*and  $c$  is such that  $\max(S^-) \in C_c$ .*

*(If  $S^-$  is empty, define  $c := -\infty$ .)*

**Proof.** By constraint (b), for city  $j$  to be in position  $i$ , we must have  $j \in \bigcup_{\ell=i-k+1}^{i+k-1} C_\ell$ . If  $S^- \neq \emptyset$ , then if  $\max(S^-) \in C_c$ ,  $j$  must come from a group indexed at or higher than  $c - k + 1$  by constraint (a). Since  $S^-$  contains cities already visited before position  $i$ , these are obviously not candidates for  $j$ .  $\square$

**Proposition 6.2.5** *Given  $i$ ,  $2k < i < m - 2k$ , given  $S^-$  satisfying Proposition 6.2.3, given  $j$  satisfying Proposition 6.2.4, and given  $S^{++} \subset (\bigcup_{\ell=u}^{i-1} C_\ell) \setminus \{j\}$  where  $u = \max\{i - k + 2, c - k + 1\}$  and  $c$  is such that  $\max(S^- \cup \{j\}) \in C_c$ ,  $S^{++} = S_{ijr}^{++}$  for some  $r$  if and only if*

$$\text{for each } q, 1 \leq q \leq i - u, |S^-| + \left| \left( \bigcup_{\ell=i-q}^{i-1} C_\ell \right) \setminus (S^{++} \cup \{j\}) \right| \leq q + k - 1. \quad (6.3)$$

**Proof.** Notice if  $i = u$ , then there is no  $q, 1 \leq q \leq i - u$ , and so (6.3) vacuously holds.

To show the condition is necessary when  $i > u$ , let  $1 \leq q \leq i - u$ . Notice that all cities in  $(\bigcup_{\ell=u}^{i-1} C_\ell) \setminus (S^{++} \cup \{j\})$  must be in  $U$ , otherwise the cities would be in  $(S^{++} \cup \{j\})$ . By constraint (b), cities in  $\bigcup_{\ell=i-q}^{i-1} C_\ell$  must be assigned to positions at or above  $i - q - k + 1$ , so at most  $(i - 1) - (i - q - k + 1) + 1 = q + k - 1$  cities can be visited before position  $i$  in the tour.  $U$  describes the before position  $i$ , and so  $\bigcup_{\ell=i-q}^{i-1} C_\ell \setminus (S^{++} \cup \{j\}) \leq q + k - 1$ .

To show the condition is sufficient, construct a tour  $T$  such that  $S^- = S_{ijr}^-$  and  $S^{++} = S_{ijr}^{++}$  for some  $r$ . Let  $H := (S^- \cup \bigcup_{\ell=u}^{i-1} C_\ell) \setminus (S^{++} \cup \{j\})$ . (Note that if  $S^-$  is empty and  $u = i$ , then  $H = \emptyset$ .)

For positions  $1 \leq h \leq \min\{i - k + 1, i - |H| - 1\}$ , choose the first element of  $C_h$ .

If  $i - k + 1 < i - |H| - 1$ ,

for positions  $i - k + 2 \leq h \leq i - |H| - 1$ , choose the second element of  $C_{h+|H|-k+2}$ .

Note that this is feasible because  $i - k + 1 < i - |H| - 1$  implies  $|H| < k - 2$ .

If  $H \neq \emptyset$ ,

for positions  $i - |H| \leq h \leq i - 1$ , choose the  $(i - h)$ th largest indexed city in  $H$ .

Note that this is feasible because (6.3) ensures that for  $q, k + 1 \leq q \leq k + i - u$  the  $q$ th largest indexed city in  $H$  comes from  $\bigcup_{\ell=u}^{i-k+1-q} C_\ell$ .

For position  $i$ , choose city  $j$ .

For positions  $i \leq h \leq i + k - 1$ , choose the second element of  $C_{h+k-1}$ .

For positions  $i + k \leq h \leq m$ , choose the first element of  $C_h$ .  $\square$

**Theorem 6.2.6** *Given  $i, j, \ell, r, r'$ , let  $S^- := S_{ijr}^-$ ,  $S^{++} := S_{ijr}^{++}$ ,  $T^- := S_{(i+1)\ell r'}^-$ ,  $T^{++} := S_{(i+1)\ell r'}^{++}$ , and define  $c$  such that  $\max(T^- \cup \{\ell\}) \in C_c$ . The two nodes  $(i, j, S^-, S^{++})$  and  $(i + 1, \ell, T^-, T^{++})$  of the auxiliary structure are compatible if and only if*

$$\ell \neq j, \text{ and} \tag{6.4}$$

$$\ell \in \left( S^{++} \cup \bigcup_{q=i}^m C_q \right) \setminus S^-, \text{ and} \tag{6.5}$$

$$S^- \setminus C_i = T^- \setminus \{j\}, \text{ and} \tag{6.6}$$

$$\left( S^{++} \cup (C_i \setminus S^-) \right) \setminus \left( \{j, \ell\} \cup \bigcup_{q=i-k+2}^x C_q \right) = T^{++}, \tag{6.7}$$

$$\text{where } x := \max(i - k + 2, c - k + 1)$$

**Proof.** According to the information in node  $(i, j, S^-, S^{++})$ , the set of cities which are eligible to be visited in the tour beyond position  $i$  is

$$A := \left( S^{++} \cup C_i \cup \bigcup_{q=i+1}^m C_q \right) \setminus (S^- \cup \{j\})$$

According to the information in node  $(i + 1, \ell, T^-, T^{++})$ , city  $\ell$  is visited in position  $i + 1$  and the set of cities eligible to be visited beyond position  $i + 1$  is

$$B := \left( T^{++} \cup \bigcup_{q=i+1}^m C_q \right) \setminus (T^- \cup \{\ell\})$$

Furthermore, the information in node  $(i + 1, \ell, T^-, T^{++})$  has eliminated

$$S^{++} \cap \left( \{\ell\} \cup \bigcup_{q=i-k+2}^x C_q \right)$$

from being eligible for positions  $i + 2$  and beyond.

It is clear that the two nodes will be compatible if and only if

$$\ell \in A, \text{ and} \tag{6.8}$$

$$A \setminus \left( \{\ell\} \cup \bigcup_{q=i-k+2}^x C_q \right) = B \tag{6.9}$$

Obviously (6.4) and (6.5) hold if and only if  $\ell \in A$ , so we must show that (6.6) and (6.7) hold if and only if (6.9) holds.

$$(6.9) \iff \left( S^{++} \cup C_i \cup \bigcup_{q=i+1}^m C_q \right) \setminus \left( S^- \cup \{j, \ell\} \cup \bigcup_{q=i-k+2}^x C_q \right) = \left( T^{++} \cup \bigcup_{q=i+1}^m C_q \right) \setminus (T^- \cup \{\ell\}) \quad (6.10)$$

From  $T^{++} \cap \bigcup_{q=i+1}^m C_q = \emptyset$ ,  $T^{++} \cap T^- = \emptyset$ , and  $\ell \notin T^{++}$ , we get

$$(6.10) \iff \left( S^{++} \cup C_i \right) \setminus \left( S^- \cup \{j, \ell\} \cup \bigcup_{q=i-k+2}^x C_q \right) = T^{++} \quad \text{and} \quad (6.11)$$

$$\bigcup_{q=i+1}^m C_q \setminus (S^- \cup \{j, \ell\}) = \bigcup_{q=i+1}^m C_q \setminus (T^- \cup \{\ell\}) \quad (6.12)$$

Notice that (6.11) is (6.7), so all that remains is to show (6.6)  $\Leftrightarrow$  (6.12).

$$(6.12) \iff \bigcup_{q=i+1}^m C_q \setminus \left( \left( S^- \cap \bigcup_{q=i+1}^m C_q \right) \cup \{j, \ell\} \right) = \bigcup_{q=i+1}^m C_q \setminus (T^- \cup \{\ell\}) \quad (6.13)$$

Since  $S^- \subset \bigcup_{q=i}^m C_q$ , we have  $S^- \cap \bigcup_{q=i+1}^m C_q = S^- \setminus C_i$ , and hence

$$(6.13) \iff \bigcup_{q=i+1}^m C_q \setminus \left( (S^- \setminus C_i) \cup \{j, \ell\} \right) = \bigcup_{q=i+1}^m C_q \setminus (T^- \cup \{\ell\}) \quad (6.14)$$

Adding  $\{j, \ell\}$  to both sides gives

$$(6.14) \iff \left( \{j, \ell\} \cup \bigcup_{q=i+1}^m C_q \right) \setminus (S^- \setminus C_i) = \left( \{j, \ell\} \cup \bigcup_{q=i+1}^m C_q \right) \setminus (T^- \setminus \{j\}) \quad (6.15)$$

Since  $S^- \setminus C_i \subset \{j, \ell\} \cup \bigcup_{q=i+1}^m C_q$  and  $T^- \setminus \{j\} \subset \{j, \ell\} \cup \bigcup_{q=i+1}^m C_q$ , we have

$$(6.15) \iff S^- \setminus C_i = T^- \setminus \{j\} \quad (6.16)$$

and notice that (6.16) is (6.6).  $\square$

**Theorem 6.2.7** *The shortest path in this auxiliary graph can be found in  $O(mk^3p^22^{kp})$  time.*

**Proof.** We need only show that the auxiliary graph has at most  $mk^3p^22^{(kp+2)}$  arcs.

Given a node  $(i, j, S_{ijr}^-, S_{ijr}^{++})$ , if  $\max(S_{ijr}^-) \in C_{i+\ell}$ , then  $S_{ijr}^{++} \subset \bigcup_{q=i+\ell-k+1}^{i-1} C_q$ , by constraint (a). Given  $i$ , the number of candidates for  $j$  is less than  $2kp$  by constraint (b). Given

$i$  and  $j$ , for each  $\ell, 0 \leq \ell \leq k-1$  there are no more than  $2^{p(\ell+1)}$  distinct possible sets for  $S_{ijr}^-$  that satisfy  $\max(S_{ijr}^-) \in C_{i+\ell}$ , since this would imply that  $S_{ijr}^- \subset \bigcup_{q=i}^{i+\ell} C_q$ , and there are no more than  $2^{p(k-\ell-1)}$  distinct possible sets for  $S_{ijr}^{++}$  since  $S_{ijr}^{++} \subset \bigcup_{q=i+\ell-k+1}^{i-1} C_q$ .

Therefore, the number of distinct nodes in a layer of the auxiliary graph is no more than

$$\begin{aligned} (2kp) \sum_{q=0}^{k-1} \left( 2^{p(q+1)} 2^{p(k-q-1)} \right) &= (2kp) \sum_{q=0}^{k-1} (2^{kp}) \\ &= (2kp)(k)(2^{kp}) \\ &= k^2 p 2^{kp+1} \end{aligned}$$

Each node will have out-degree less than  $2kp$  since, given  $v := (i, j, S_{ijr}^-, S_{ijr}^{++})$ , there are less than  $2kp$  candidates for position  $i+1$ , and once the city for position  $i+1$  is fixed, call it  $\ell$ , there is a unique pair  $(S_{(i+1)\ell r}^-, S_{(i+1)\ell r}^{++})$  that satisfies the compatibility conditions with  $v$ . Thus the number of arcs from one layer to another is no greater than  $k^3 p^2 2^{p(k+2)}$ . Since there are  $m+1$  layers in the graph, we have  $m$  sets of arcs, bounding the number of arcs in the entire graph by  $mk^3 p^2 2^{(kp+2)}$ .  $\square$

### 6.3 Implementation Issues

Some of the structure of the earlier auxiliary graph does not carry over so easily to this model. Nodes of the auxiliary graph for a given  $k$  may not be a subset of the nodes for a larger  $k$ , Nodes may appear in multiple distinct successor lists, in-degrees of nodes may be very large, and city dependent values of  $k$  may be tied to a group of cities  $C_i$  or individual cities, creating different approaches to weeding out infeasible tours.

The reason that nodes of the auxiliary graph for a given  $k$  may not be a subset of the nodes for a larger  $k$ , lies in the definition of  $S_{ijr}^{++}$ . According to Proposition 6.2.5, there is a minimum size to the set  $S_{ijr}^{++}$  which is linked to  $k$ . For example, when  $j \in C_i$ , and  $S_{ijr}^- = \emptyset$ , the condition in Proposition 6.2.5 for  $q = k-2$  mandates that  $|S_{ijr}^{++}| \geq (p-2)(k-2) - 1$ . Proposition 6.2.2 implies that  $|S_{ijr}^{++}| \leq p(k-2)$ , and so one can see that no node with  $j \in C_i$ , and  $S_{ijr}^- = \emptyset$  can be in the set of nodes for both  $k = p$  and  $k = p+4$  if  $p \geq 3$ .

However, there definitely is some common structure that can be used, if one looks at the upper bound on the complement of  $S_{ijr}^{++} \cup \{j\}$  among those groups which are eligible to contribute to  $S_{ijr}^{++}$  given values of  $j$  and  $S_{ijr}^-$ . Using the same example, Proposition 6.2.5

states that:

$$\left| \left( \bigcup_{\ell=i-k+2}^{i-1} C_\ell \right) \setminus (S_{ijr}^{++} \cup \{j\}) \right| \leq 2k - 3.$$

In general, if we define  $u := \max\{i-k+2, c-k+1\}$ , where  $c$  is such that  $\max(S_{ijr}^- \cup \{j\}) \in C_c$ , then the groups eligible to contribute to  $S_{ijr}^{++}$  are  $C_u$  through  $C_{i-1}$  and so Proposition 6.2.5 gives:

$$\left| \left( \bigcup_{\ell=u}^{i-1} C_\ell \right) \setminus (S_{ijr}^{++} \cup \{j\}) \right| \leq (i - u) + (k - 1) - |S_{ijr}^-|.$$

And so, given a node in the auxiliary graph for a fixed  $k$ , we can find a node with the same  $j$ ,  $S_{ijr}^-$ , and complement set to  $S_{ijr}^{++} \cup \{j\}$  for an auxiliary graph with a larger  $k$ . For uniqueness, we must consider the complement set among  $\bigcup_{\ell=u}^{i-1} C_\ell$ , where  $u$  is defined according to the larger  $k$ . This would imply that for the larger  $k$ ,  $\bigcup_{\ell=u}^{u+\Delta k} C_\ell \cap S_{ijr}^{++} = \emptyset$ , where  $\Delta k$  represents the difference in the values of  $k$ . We can now define the level of a node  $v := (i, j, S_{ijr}^-, S_{ijr}^{++})$  in a similar way we did for the original model, by declaring the level of node  $v$  to be the smallest value of  $k$  for which Propositions 6.2.3 through 6.2.5 hold. And in the case of Proposition 6.2.5, we are considering the complement set of  $S_{ijr}^{++} \cup \{j\}$  according to the  $u$  defined by the  $k$  for which the original auxiliary structure was built.

In Section 2.3 we saved space in the storage of the auxiliary graph by noting that every node appeared in exactly one distinct successor or predecessor list. This is no longer the case in our new model, as can be seen in Table 6.1, which represents the entire list of nodes with  $S_{ijr}^- = \emptyset$  in the auxiliary graph for  $k = 3$  and  $p = 2$ . However, looking closely at all the successor lists that contain node 1, we see they are identical, except for the appearance or disappearance of nodes 7 and 11. The reason for this anomaly is that when testing the compatibility of two nodes  $(i, j, S^-, S^{++})$  and  $(i+1, \ell, T^-, T^{++})$  the test between  $S^{++}$  and  $T^{++}$  first “clips off” the elements of  $S^{++}$  from group  $C_{i-k+2}$ , since these cities are never eligible for position  $i+2$  or beyond. However, if  $\ell$  comes from group  $C_{i-k+2}$ , it must also be in  $S^{++}$ , so even if  $T^{++}$  no longer needs the information, the presence of  $j \in C_{i-k+2}$  can create slightly different successor lists. The original model had no such discrepancy since no city could be “clipped off”, i.e. every city had to be visited in the tour at some point in the sequence.

Since our anomaly is limited to the case where  $j \in C_{i-k+2}$ , keeping a bit vector of size  $p$  is enough to distinguish among these near identical lists, so each near identical list needs to be stored only once, with each node carrying a pointer to one of these lists, and a bit vector of size  $p$  to determine which of the nodes with  $j \in C_{i-k+2}$  are actually successors. Among the 30 nodes in Table 6.1, nodes 3, 4, and 7-16, have nearly identical successor lists.

Table 6.1: Nodes of the auxiliary structure with  $S^- = \emptyset$  for  $k = 3, p = 2$ .

Level	No.	$j$	$S^{++}$	In-degree	Compatible Successors
1	1:	$C_i^{(1)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	12	8, 12, 16, 17, 20, 23, 26, 29, 30
	2:	$C_i^{(2)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	12	9, 13, 15, 18, 21, 24, 27, 29, 30
2	3:	$C_{i-1}^{(1)}$	$\{C_{i-1}^{(2)}\}$	12	1, 2, 3, 4, 5, 6, 11, 29, 30
	4:	$C_{i-1}^{(2)}$	$\{C_{i-1}^{(1)}\}$	12	1, 2, 3, 4, 5, 6, 7, 29, 30
	5:	$C_{i+1}^{(1)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	12	*out-degree = 9
	6:	$C_{i+1}^{(2)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	12	*out-degree = 9
3	7:	$C_{i-2}^{(1)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	5	1, 2, 3, 4, 5, 6, 7, 11, 29, 30
	8:	$C_{i-2}^{(1)}$	$\{C_{i-1}^{(2)}\}$	7	1, 2, 3, 4, 5, 6, 11, 29, 30
	9:	$C_{i-2}^{(1)}$	$\{C_{i-1}^{(1)}\}$	7	1, 2, 3, 4, 5, 6, 7, 29, 30
	10:	$C_{i-2}^{(1)}$	$\emptyset$	9	1, 2, 3, 4, 5, 6, 29, 30
	11:	$C_{i-2}^{(2)}$	$\{C_{i-1}^{(1)}, C_{i-1}^{(2)}\}$	5	1, 2, 3, 4, 5, 6, 7, 11, 29, 30
	12:	$C_{i-2}^{(2)}$	$\{C_{i-1}^{(2)}\}$	7	1, 2, 3, 4, 5, 6, 11, 29, 30
	13:	$C_{i-2}^{(2)}$	$\{C_{i-1}^{(1)}\}$	7	1, 2, 3, 4, 5, 6, 7, 29, 30
	14:	$C_{i-2}^{(2)}$	$\emptyset$	9	1, 2, 3, 4, 5, 6, 29, 30
	15:	$C_{i-1}^{(1)}$	$\emptyset$	16	1, 2, 3, 4, 5, 6, 29, 30
	16:	$C_{i-1}^{(2)}$	$\emptyset$	16	1, 2, 3, 4, 5, 6, 29, 30
	17:	$C_i^{(1)}$	$\{C_{i-1}^{(2)}\}$	16	12, 16, 17, 20, 23, 26, 29, 30
	18:	$C_i^{(1)}$	$\{C_{i-1}^{(1)}\}$	16	8, 16, 17, 20, 23, 26, 29, 30
	19:	$C_i^{(1)}$	$\emptyset$	18	16, 17, 20, 23, 26, 29, 30
	20:	$C_i^{(2)}$	$\{C_{i-1}^{(2)}\}$	16	13, 15, 18, 21, 24, 27, 29, 30
	21:	$C_i^{(2)}$	$\{C_{i-1}^{(1)}\}$	16	9, 15, 18, 21, 24, 27, 29, 30
	22:	$C_i^{(2)}$	$\emptyset$	18	15, 18, 21, 24, 27, 29, 30
	23:	$C_{i+1}^{(1)}$	$\{C_{i-1}^{(2)}\}$	16	*out-degree = 8
	24:	$C_{i+1}^{(1)}$	$\{C_{i-1}^{(1)}\}$	16	*out-degree = 8
	25:	$C_{i+1}^{(1)}$	$\emptyset$	18	*out-degree = 7
	26:	$C_{i+1}^{(2)}$	$\{C_{i-1}^{(2)}\}$	16	*out-degree = 8
	27:	$C_{i+1}^{(2)}$	$\{C_{i-1}^{(1)}\}$	16	*out-degree = 8
	28:	$C_{i+1}^{(2)}$	$\emptyset$	18	*out-degree = 7
	29:	$C_{i+2}^{(1)}$	$\emptyset$	62	*out-degree = 7
	30:	$C_{i+2}^{(2)}$	$\emptyset$	62	*out-degree = 7

Notes:  $C_a^{(b)}$  indicates the  $b$ th city from group  $C_a$ .

\*Successors for these arcs do not appear in this list.

The total number of nodes in the auxiliary structure for  $k = 3, p = 2$  is 184:  
 2 nodes in level 1, 14 nodes in level 2, and 168 nodes in level 3.

Another space saving device from Section 2.3 was the use of predecessor lists, allowing us to store a predecessor as being the  $\ell$ th predecessor among less than  $k$  total predecessors. This could be done with  $\log_2 k$  bits instead of  $\log_2((k+1)2^{k-2})$  bits, which would be needed to point to an arbitrary node in a fixed layer. As Table 6.1 shows, however, nodes 29 and 30 have an in-degree of 62, more than  $\frac{1}{3}$  of the total number of nodes (184) for the graph. This prevents us from using the same trick to save memory in this fashion during the optimization process.

# Chapter 7

## Conclusions

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

We have applied our procedure to solving TSP's 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 also applied our procedure to directly solving scheduling problems with setup times, involving release dates alone, or release dates with delivery times. Further work can be done to test the routine as part of a package to solve job shop scheduling problems via the shifting bottleneck procedure.

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.

For TSP's that do not obey the postulated precedence constraints, 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, which is competitive with  $k$ -opt interchange procedures. Although by itself our algorithm is out-performed by the most sophisticated version of chained Lin-Kernighan procedure, on the other hand it is sometimes 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.

Finally, we have developed a new auxiliary structure that models similar constraints for the Prize-Collecting TSP, with applications in steel rolling mills, among others. It is possible that this procedure could be adapted as a heuristic like the other for unconstrained PC-TSP's.

# Chapter 8

## The Dynamic Programming Code

The source code for the dynamic programming routines presented in this paper can be found at <http://www.contrib.andrew.cmu.edu/~neils/tsp/index.html>. This appendix will explain how to use these routines.

### 8.1 Building the Auxiliary Structure

The program `auxgraph.c` will produce the files used to store the auxiliary structure  $W_K^*$  and its compatible pairs, as described in Section 2.2. The program takes one parameter, which is the value of  $K$ .

The program produces eight files (stored in binary except for `auxgraph.lim`):

<code>auxgraph.j</code>	This file has an array of $j$ values, indexed by each node.
<code>auxgraph.lim</code>	This file holds the number $K$ .
<code>auxgraph.loc</code>	This file has an array, indexed by each node, which specifies where the node appears in the unique predecessor list in which it is found.
<code>auxgraph.mk</code>	This file has an array of $k$ -threshold values, indexed by each node.
<code>auxgraph.p.arc</code>	This file has an array holding the predecessor lists. lists are separated by the number $2^{30}$ .
<code>auxgraph.p.inx</code>	This file has an array, indexed by each node, which specifies where in <code>auxgraph.p.arc</code> a nodes predecessor list begins.
<code>auxgraph.s.arc</code>	This file has an array holding the successor lists.
<code>auxgraph.s.inx</code>	This file has an array with a similar function to <code>auxgraph.p.inx</code> .

This program can take a very long time to run for large values of  $K$ .  $K = 15$  takes about a half hour on a Sun Sparc5. Every increase in one unit for  $K$ , roughly doubles the required running time. Storage requirements also increase exponentially. While  $K = 8$  requires less than 12 kilobytes of space,  $K = 12$  requires close to 270 kilobytes,  $K = 15$  requires nearly 2.6 megabytes, and  $K = 17$  demands almost 12 megabytes of space.

## 8.2 The Subroutines

There are two subroutines in `solver.h` which you will need to use:

*GetAuxgraph*( $k, n, h, q$ )

This subroutine allocates memory for the auxiliary graph and the dynamic programming routine, and reads those parts of the auxiliary structures from the files created by `auxgraph.c`. The program will look for a file titled `auxgraph.where` and, if the file is present, it will look in the directory named in the file for the auxiliary structure files. If no `auxgraph.where` file is found, it will look in the current directory. The largest memory allocation statement requires roughly  $hq(k+1)2^{k-2}$  bytes.

- $k$  - the value of  $k$  you wish to use in the program.  
This value must be no larger than the number  $K$  found in `auxgraph.lim`.
- $n$  - a number equal to or greater than the problem size.
- $h$  - the depth of the working environment for the dynamic program.  
(see Section 2.3 for a description of this parameter.)
- $q$  - the thickness of the working environment for the dynamic program.  
(see Section 3.2 for a description of this parameter.)

For regular TSP problems or time window problems minimizing time,  $q$  should be 1. If you wish to use edge contraction on a symmetric cost matrix,  $q$  should be 2, to allow for the contracted edges to be rehooked in either the original order or the reverse order. For time window problems minimizing distance,  $q$  should be set higher (see Section 3.2).

*DynOpt*

This subroutine executes the dynamic program. It has three versions, one for unconstrained TSP's (with or without edge contraction), another for time window problems minimizing time, and a third for time window problems minimizing distance. This procedure returns the cost of the best tour it can find. If it fails to find a feasible tour (in the case of time windows), the value  $2^{30}$  is returned.

Regular TSP: *DynOpt*( $k, n, h, contRule, targ, tourIn, tourOut, kval$ )

- $k$  - the largest value of  $k$  you wish to use in the program.  
This value must be no larger than the number  $k$  used in *GetAuxgraph*.
- $n$  - the number of cities in the problem.
- $h$  - see *GetAuxgraph* for description.
- contRule* - the rule for contracting arcs (see Section 4.1).  
0 - no contraction.  
1 - random contraction.  
3 - contraction probability related to arc cost.  
8 - contraction probabilities related to the squares of those for rule 3.  
Add 10 to any contraction rule for symmetric cost matrices. This will allow contracted segments to be inserted in the original or reversed order.
- targ* - target number of arcs to contract (actual number may vary).  
This parameter should be zero if no contraction rule is used.
- tourIn* - an array containing the initial tour. This must be specified for unconstrained TSP's. The cities must be numbered in the range of 0 to  $n - 1$ .
- tourOut* - an array used to return the final tour.  
The cities will again be in the range of 0 to  $n - 1$ . If the parameter  $h$  is too small for the final tour to be recovered, *tourOut*[0] will be equal to  $n$ .
- kval* - an array of values  $k(i)$  for each node  $i$  (see Section 1.1)  
If a null pointer is passed as this parameter, the program will use the value in the parameter  $k$  for every city.

TSPTW - Time: *DynOpt*( $k, n, h, targ, tourIn, tourOut, r, d, p, guar$ )

The first six parameters are the same as above. At this point, contraction has not been implemented for time window problems, so *targ* should be zero. An initial tour does not have to be specified for time window problems, if *tourIn*[0] =  $n$  then the program will build an initial tour based on the time window midpoints.

- $r$  - an array of the release dates or the starts of the time windows for each node.
- $d$  - an array of the due dates or the ends of the time windows for each node.
- $p$  - an array of the process times or service times for each node.
- guar* - a pointer to a character which will receive TRUE if the solution is known to be optimal, and FALSE if no guarantee can be made.

TSPTW - Distance: *DynOpt*( $k, n, h, q, targ, tourIn, tourOut, r, d, p, guar$ )

All parameters are the same as those for time window problems minimizing time, with the addition of the parameter  $q$ , mentioned above in *GetAuxgraph*. Note that even if the graph was constructed to handle a large value of  $q$ , a small value of  $q$  may be used to speed up the execution, and the large value used only when no guarantee could be made with the small value.

## 8.3 Essential Ingredients for Your Shell

For you to be able to compile these subroutines you need to include `stdio.h`, `math.h`, `string.h` from the standard library.

Then you must create a macro with the name `theNorm(a,b)`, which will tell the program how to compute the distance from city *a* to city *b*. This can be a macro for a reference to a global matrix, for example:

```
#define theNorm(a,b) (myMatrix[a][b])
```

or it can be an expression involving other global structures, like Euclidean distance:

```
#define theNorm(a,b) ((int)(0.5 + sqrt((x[a]-x[b])**2 + (y[a]-y[b])**2)))
```

or it can be a reference to a function:

```
#define theNorm(a,b) (ReturnDistance(a,b))
```

Next, if you are solving a time window problem, you must specify your objective. If you want to minimize time, insert the line:

```
#define _twTime 1
```

If you want to minimize distance, insert the line:

```
#define _twDist 1
```

If you are not using time windows, but you wish to use the arc contraction feature, insert the line:

```
#define _shrink 1
```

If you are not using time windows, and not using the contraction feature, no extra line needs to be inserted.

At this point you can include the file `"solver.h"`, the other files (`"ag.h"`, `"agtw.h"`, and `"dynopt.h"`) are included from this file, so you do not need an include statement for them. `"agtw.h"` is not needed for regular TSP problems.

## 8.4 Sample Shell Programs

For examples of how to coordinate these specifications, the web site has four shell programs that illustrate their use. Test data for the programs can be found from links on the web site.

`regtsp h k filename`

*regtsp* needs as input the parameters  $h$  and  $k$  (see above) and a filename containing a TSPLIB problem of type EUC\_2D.

`twdist h k q filename`

*twdist* needs as input the parameters  $h$ ,  $k$ , and  $q$  (see above) and a filename containing one of the stacker crane problems developed by N. Ascheuer [3].

`twtime h k filename`

*twtime* needs as input the parameters  $h$  and  $k$  (see above) and a filename containing one of the single machine instances like those constructed by J. Tama [21].

`uscity h k i filename`

*uscity* needs as input the parameters  $h$  and  $k$  (see above), a parameter  $i$  which specifies the number of iterations to perform and a filename containing one of the symmetric TSP based on the largest United States Cities. This program is an example of how to use the contraction feature in an iterative setting.

# Bibliography

- [1] J. Adams, E. Balas, D Zawack, “The Shifting Bottleneck Procedure for Job Shop Scheduling.” *Management Science*, 34, 1988, 391-401.
- [2] D. Applegate, R.E. Bixby, V. Chvatal and W.J. Cook, “A New Paradigm for Finding Cutting Planes in the TSP.” Presented at the International Symposium on Mathematical Programming, Lausanne, August 1997.
- [3] N. Ascheuer, M. Fischetti, and M. Grötschel, “Solving ATSP with Time Windows by Branch-and-Cut.” Technical report, ZIB Berlin, 1997.
- [4] E. Baker, “An Exact Algorithm for the Time-Constrained Traveling Salesman Problem.” *Operations Research*, 31, 1983, 938-945.
- [5] E. Balas, “New Classes of Efficiently Solvable Generalized Traveling Salesman Problems.” MSRR No. 615, GSIA, Carnegie Mellon University, March 1995 (revised May 1996). Presented at CO96, London, March 1996.
- [6] E. Balas, “The Prize Collecting Traveling Salesman Problem.” *Networks*, 19, 1989, 621-636.
- [7] L. Bianco, G. Rinaldi, A Sassano, “A Combinatorial Optimization Approach to Aircraft Sequencing Problem.” NATO ASI Series, Vol. F38, Ed. A. R. Odoni et al. 1987.
- [8] Y. Dumas, J. Desrosiers, E. Gelinas and M.Solomon, “An Optimal Algorithm for the Traveling Salesman Problem with Time Windows.” *Operations Research*, 43, 1995, 367-371.
- [9] M. Gendreau, A. Hertz, G. Laporte and M. Stan, “A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows.” CRT-95-07, Centre de Recherche sur les Transports, Montreal, 1995.
- [10] P.C. Gilmore, E.L. Lawler and D. Shmoys, “Well-Solved Special Cases.” Chapter 4 in E.L. Lawler, J.K Lenstra, A.H.G. Rinnooy Kan and D. Shmoys (editors), *The Traveling Salesman Problem: A Guided Tour to Combinatorial Optimization*, Wiley, 1985.
- [11] M. Jünger, G. Reinelt, G. Rinaldi, “The Traveling Salesman Problem,” *Network Models*, M. Ball, T. Magnanti, C. Monma, G. Nemhauser (editors), 225-230.
- [12] P. Kanellakis and C. Papadimitriou, “Local Search for the Traveling Salesman Problem.” *Operations Research*, 28, 1980, 1086-1099.
- [13] S. Lin and B.W. Kernighan, “An Effective Heuristic Algorithm for the Traveling Salesman Problem.” *Operations Research*, 21, 1973, 495-516.

- [14] O. Martin, S.W. Otto and E.W. Felten, "Large Step Markov Chains for the TSP Incorporating Local Search Heuristics." *Operations Research Letters*, 11, 1992, 219-225.
- [15] I. Or, "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, 1976.
- [16] J.Y. Potvin and S. Bengio, "A Genetic Approach to the Vehicle Routing Problem with Time Windows." CRT-93-5, Centre de Recherche sur les Transports, Montreal, 1993.
- [17] G. Reinelt, personal communication.
- [18] B. Repetto, personal communication.
- [19] B. Repetto, *Upper and Lower Bounding Procedures for the Asymmetric Traveling Salesman Problem*. Ph.D. Thesis, GSIA, Carnegie Mellon University, April 1994.
- [20] M. Solomon, "Algorithms for Vehicle Routing and Scheduling with Time Window Constraints." *Operations Research*, 35, 1987, 254-265.
- [21] J. Tama, *Polyhedral Aspects of Scheduling Problems with an Application to the Time-Constrained Traveling Salesman Problem*. Ph.D. thesis, GSIA, Carnegie Mellon University, November 1990.
- [22] United States Census Bureau, <http://www.census.gov/cgi-bin/gazetteer>
- [23] A.A. Van der Veen, *Solvable Cases of the Traveling Salesman Problem with Various Objective Functions* Ph.D. Dissertation, University of Groningen, 1992.
- [24] A. Vazacopoulos, personal communication.