Sequential and parallel local search for the time-constrained traveling salesman problem

Gerard Kindervater
Erasmus University, Rotterdam

Jan Karel Lenstra
Eindhoven University of Technology
CWI, Amsterdam

Martin Savelsbergh
Eindhoven University of Technology

In memory of Paolo Camerini

Local search has proven to be an effective solution approach for the traveling salesman problem. We consider variants of the TSP in which each city is to be visited within one or more given time windows. The travel times are symmetric and satisfy the triangle inequality; the objective is to minimize the tour duration. We develop efficient sequential and parallel algorithms for the verification of local optimality of a tour with respect to k-exchanges.

1980 Mathematics Subject Classification (1985 Revision): 90C27, 90B35.
Key Words & Phrases: traveling salesman problem, time window, local search, parallel computing.

1. Introduction

Like so many other approaches in combinatorial optimization, local search was first seriously investigated in the context of the traveling salesman problem. Lin [1965] calls a traveling salesman tour k-*optimal* when it cannot be improved by replacing a set of k of its edges by another set of k edges. It is not known whether, for any fixed value of $k \geq 2$, a k-optimal tour can be *generated* in polynomial time. However, it is trivial to observe that the k-optimality of a given tour through n cities can be *verified* in $O(n^k)$ time: there are $\binom{n}{k}$ ways to delete k edges; for each of these, there is a constant number of candidate improvements (where the constant depends on k); and each of these candidates can be evaluated in constant time. For example, if $k = 2$, two edges are replaced by two other edges, and only four cost coefficients have to be checked in order to compute the length of the new tour.

The above analysis implicitly assumes that the algorithm is to be executed on a traditional computer, which performs at most one computation at a time. Now suppose that we have a computer that can perform a number of operations in parallel. Such a computer has a greater processing power than a serial one. More specifically, assume we have a parallel random access machine (PRAM), a machine with an unbounded number of processors that operate in parallel and communicate with each other in constant time through a shared memory. In that case, the k-optimality of a tour through n cities can be verified by $O(n^k)$ processors in $O(\log n)$ time: each processor evaluates a single k-exchange in constant time, and the best of these is selected in logarithmic time. It is not hard to reduce the number of processors involved by a factor of $\log n$, as will be explained later in this paper. Hence, for the TSP, $O(n^k/\log n)$ processors do in time $O(\log n)$ what a single processor can do in time $O(n^k)$. We thus achieve a *perfect speedup*.

Report EUR-CS-90-06 (revision)
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Now suppose that each city has a single time window during which it must be visited, and again consider the case $k = 2$. If two edges are replaced by two other edges, then a certain segment of the tour will be traversed in the opposite direction. Therefore, in addition to the test for improvement, there has to be a test for feasibility with respect to the time windows. In a straightforward implementation, this requires an amount of work proportional to the length of the modified part of the tour. In general, evaluating the feasibility of a single k-exchange requires linear time on a sequential computer, or logarithmic time and a linear number of processors on a parallel computer. This leads to an algorithm for verifying k-optimality in $O(n^{k+1})$ time on a sequential machine and $O(\log n)$ time and $O(n^{k+1}/\log n)$ processors on a parallel machine, which is a perfect speedup again. In comparison to the case without time windows, the total effort has increased by a factor of n.

We will investigate more sophisticated implementations that avoid the additional factor of n. We concentrate on the verification of 2-optimality of a tour in the presence of time windows. In Sections 2 and 3 we review sequential and parallel local search for the unconstrained TSP; this material is relatively straightforward. Sections 4 and 5 discuss efficient implementations of sequential and parallel local search for the TSP with a single time window for each city. Finally, Sections 6 and 7 present our implementations for the TSP with multiple time windows.

2. Local search for the TSP

In the traveling salesman problem, one is given a complete undirected graph G with vertex set $\{1, \ldots, n\}$ and a travel time d_{ij} for each edge $\{i, j\}$, and one wishes to find a Hamiltonian cycle (i.e., a cycle passing through each vertex exactly once) of minimum total duration. We assume that the travel times satisfy the triangle inequality, i.e., $d_{ij} + d_{jk} \geq d_{ik}$ for each triple (i, j, k). The TSP is a well-known NP-hard problem, for which many optimization and approximation algorithms have been proposed; cf. Lawler, Lenstra, Rinnooy Kan & Shmoys [1985].

We consider the following local search algorithm for the TSP. Construct an initial Hamiltonian cycle by taking an arbitrary permutation of the vertices or by applying a specific heuristic method such as the *nearest neighbor* rule or the *double minimum spanning tree* algorithm. Then try to improve the tour by replacing a set of k of its edges by another set of k edges, and iterate until no further improvement is possible. Such replacements are called *k-exchanges*, and a tour that cannot be improved by a k-exchange is said to be *k-optimal*. Throughout the paper, we will consider the case $k = 2$ in detail. For $k > 2$, the analysis is conceptually similar but technically more involved.

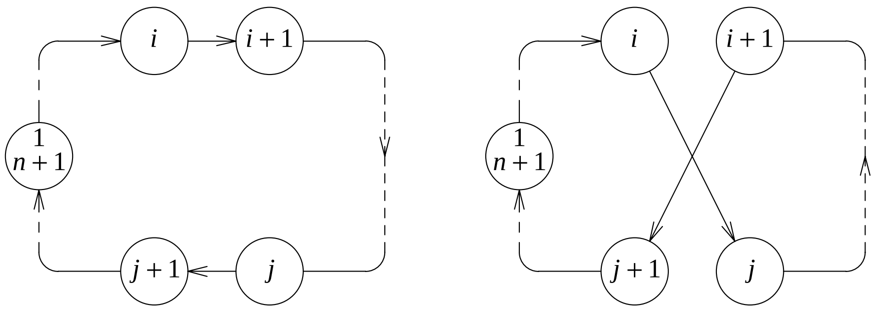

Figure 1. A 2-exchange.

For notational convenience, we consider the tour $(1, 2, \ldots, n, n+1)$, where the origin 1 and the destination $n+1$ denote the same vertex. A 2-exchange replaces two edges $\{i, i+1\}$ and $\{j, j+1\}$, with $j > i$, by the edges $\{i, j\}$ and $\{i+1, j+1\}$, thereby reversing the path from $i+1$ to j; see Figure 1. It is an open question if there exists a polynomial-time algorithm that obtains a 2-optimal tour by a sequence of 2-exchanges [Johnson, Papadimitriou & Yannakakis, 1988]. We therefore restrict ourselves to deciding whether a given tour is 2-optimal.

Because the travel times between the vertices do not depend on the direction, a 2-exchange results in a local improvement if and only if

$$d_{ij} + d_{i+1,j+1} < d_{i,i+1} + d_{j,j+1}.$$

Testing a single 2-exchange for improvement involves only a constant amount of information and hence requires constant time. It follows that verifying 2-optimality takes $O(n^2)$ time. No algorithm that proceeds by enumerating all possible improvements can run faster, as there are $\binom{n}{2}$ 2-exchanges.

3. Parallel local search for the TSP

Before discussing the verification of 2-optimality on the PRAM model, we will first consider an elementary problem and describe a basic technique in parallel computing for its solution. The algorithm consists of two phases. In some simple situations, as in this section, only the first phase is needed.

The problem is to find the *partial sums* of a given sequence of n numbers. For the sake of simplicity, let $n = 2^m$ and suppose that the n numbers are given by $a_n, a_{n+1}, \ldots, a_{2n-1}$. We wish to find the partial sums $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$. The following procedure is due to Dekel & Sahni [1983]:

```
for $l \leftarrow m-1$ downto 0 do
    par $[2^l \le j \le 2^{l+1}-1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
$b_1 \leftarrow a_1$;
for $l \leftarrow 1$ to $m$ do
    par $[2^l \le j \le 2^{l+1}-1]$ $b_j \leftarrow$ if $j$ odd then $b_{(j-1)/2}$ else $b_{j/2} - a_{j+1}$.
```

Here, a statement of the form '**par** $[\alpha \le j \le \omega]$ s_j' denotes that the statements s_j are executed in parallel for all values of j in the indicated range.

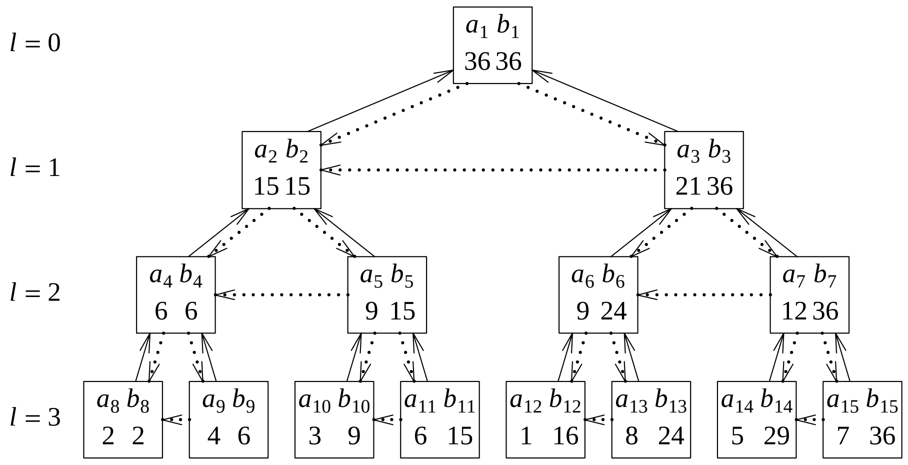

Figure 2. Partial sums: an instance with $n = 8$.

The computation is illustrated in Figure 2. In the first phase, represented by solid arrows, the sum of the a_j's is calculated. Note that the a-value corresponding to a non-leaf node is set equal to the sum of all a-values corresponding to the leaves descending from that node. In the second phase, represented by dotted arrows, each parent node sends a b-value (starting with $b_1 = a_1$) to its children: the right child receives the same value, the left one receives that value minus the a-value of the right child. The b-value of a certain node is therefore equal to the sum of all a-values of the nodes of the same generation, except those with a higher index. This implies, in particular, that at the end we have $b_{n+j} = a_n + \cdots + a_{n+j}$ for $j = 0, \ldots, n-1$.

The algorithm requires $O(\log n)$ time and n processors. This can be improved to $O(\log n)$ time and $O(n/\log n)$ processors by a simple device. First, the set of n numbers is partitioned into $n/\log n$ groups of size $\log n$ each, and $n/\log n$ processors determine the sum of each group in the traditional serial way in $\log n$ time. After this aggregation process, the above algorithm computes the partial sums over the groups; this requires $O(n/\log n)$ processors and $O(\log n)$ time. Finally, a disaggregation process is applied with the same processor and time requirements.

In the form given above, the algorithm does not work for operations such as maximization. The partial sums algorithm uses subtraction, which has no equivalent in the case of maximization. We therefore present a version of the partial sums algorithm which is not quite so elegant as the original one, but which has the desired property since it makes use of addition only. It also runs in $O(\log n)$ time using $O(n/\log n)$ processors:

```
for $l \leftarrow m-1$ downto 0 do
    par $[2^l \le j \le 2^{l+1}-1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
for $l \leftarrow 0$ to $m$ do
    par $[2^l \le j \le 2^{l+1}-1]$
        $b_j \leftarrow$ if $j = 2^l$ then $a_j$ else if $j$ odd then $b_{(j-1)/2}$ else $b_{(j-2)/2} + a_j$.
```

We now return to the verification of 2-optimality. The following procedure decides whether or not the tour $(1, 2, \ldots, n, n+1)$ is 2-optimal:

```
par $[1 \le i < j \le n]$ $\delta_{ij} \leftarrow d_{ij} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$;
$\delta_{\min} \leftarrow \min\{\delta_{ij} | 1 \le i < j \le n\}$;
if $\delta_{\min} \ge 0$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let $i^*$ and $j^*$ be such that $\delta_{i^* j^*} = \delta_{\min}$,
     $(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a shorter tour.
```

By adapting the first phase of the partial sums algorithm such that it computes the minimum of a set of numbers and also delivers an index for which the minimum is attained, the above procedure can be implemented to require $O(\log n)$ time and $O(n^2/\log n)$ processors. The total computational effort is $O(\log n \cdot n^2/\log n) = O(n^2)$, as it is in the serial case. This is called a *full processor utilization* or a *perfect speedup*.

Although the serial and parallel implementations seem similar, there is a basic distinction. When the tour under consideration is not 2-optimal, the serial algorithm will detect this after a number of steps that is somewhere in between 1 and $\binom{n}{2}$. In the parallel algorithm, confirmation and negation of 2-optimality always take the same amount of time.

4. Local search for the TSP with single time windows

In the TSP with time windows, each vertex i has a time window on the departure time, denoted by $[s_i, t_i]$. The time window is opened at time s_i and closed at time t_i. If the salesman arrives at i before s_i, he has to wait; if he arrives after t_i, he is late and his tour is infeasible. The salesman departs at the opening time of the time window associated with his starting vertex, and his objective is to be back as early as possible.

Due to the presence of time windows, there are feasible and infeasible tours, and this complexifies the problem. To start with, the problem of determining the existence of a feasible tour is *NP*-complete in the strong sense. This follows from the observation that the unconstrained TSP has a tour of duration no more than B if and only if there is a feasible tour for the constrained TSP in which each vertex has a time window $[0, B]$.

Second, when applying local search, we have to test all candidate improvements for feasibility. A k-exchange influences the arrival times at all vertices visited after the first change in the tour. This may lead to changes in the departure times and even to infeasibility. In a straightforward implementation, we need $O(n)$ time to handle a single k-exchange, which results in a time complexity of $O(n^{k+1})$ for the verification of k-optimality. We will show how to reduce this time bound by an order n, thereby obtaining the same time complexity as in the unconstrained case.

The basic idea is the use of a specific *search strategy* in combination with a set of *global variables* such that testing the feasibility of a single exchange and maintaining the set of global variables require no more than constant time. The discussion below is an adaptation of Savelsbergh [1986]. We consider the case $k = 2$ in detail. Related results can be found in Savelsbergh [1991].

As before, we consider the tour $(1, 2, \ldots, n, n+1)$. We assume that this tour is feasible. A 2-exchange involves the replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$. Such an exchange is both feasible and profitable if and only if the following three conditions are satisfied:

(1) the reversed path $(j, \ldots, i+1)$ is feasible, i.e., the new departure time at vertex k is not larger than t_k, for $k = i+1, \ldots, j$;

(2) the new departure time at vertex $j+1$ is smaller than it was before the exchange;

(3) a part of the gain at vertex $j+1$ can be carried through to the destination, i.e., the original departure time at vertex k is strictly larger than s_k, for $k = j+1, \ldots, n$.

Condition (3) needs further consideration. If it is violated, the exchange will not affect the duration of the tour. However, it will reduce the duration of the path from 1 to $k-1$, for the smallest k for which violation occurs. In the sequel, we will drop condition (3), for two reasons. First, introducing some slack may be beneficial for the rest of the procedure, even though the slack cannot be carried through to the end of the tour. In addition, taking condition (3) into account would make the presentation needlessly complicated. In this setting, a tour is 2-optimal if and only if there does not exist a feasible 2-exchange that reduces the duration of the path from 1 to k for any vertex k. This is a broader notion of 2-optimality, which implies the original one.

We propose a *search strategy* that examines the 2-exchanges in lexicographic order. We choose i successively equal to $1, 2, \ldots, n-2$; this will be referred to as the outer loop. For a fixed value of i, we choose j successively equal to $i+2, i+3, \ldots, n$; this will be called the inner loop. In the inner loop, the previously reversed path $(j-1, \ldots, i+1)$ is repeatedly expanded with the edge $\{j, j-1\}$; cf. Figure 3.

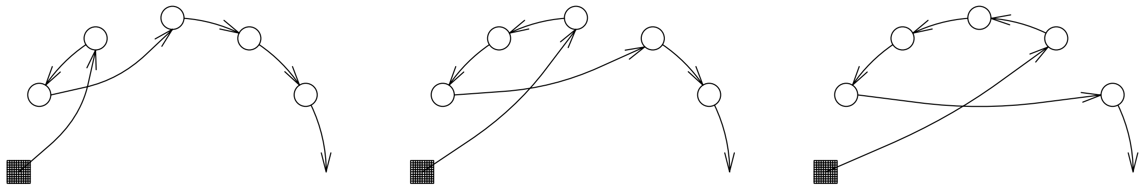

Figure 3. The search strategy for 2-exchanges.

In the following, we assume that i is fixed and consider the inner loop. The departure time at vertex k in the tour $(1, 2, \ldots, n, n+1)$ will be denoted by D_k, for $k = 1, \ldots, n+1$. The waiting and departure times at vertex k after reversal of the path $(i+1, \ldots, j)$ will be denoted by W_k^j and D_k^j, respectively, for $k > i$.

We define three *global variables*, which will be associated with the reversed path $(j-1, \ldots, i+1)$, and which will be maintained throughout the inner loop. First, T is equal to the total travel time along this path:

$$T = \sum_{k=i+1}^{j-2} d_{k,k+1}.$$

Second, W is equal to the total waiting time along the path after departing from vertex $j-1$:

$$W = \sum_{k=i+1}^{j-2} W_k^{j-1}.$$

Third, S is equal to the maximum forward shift in time of the departure time at vertex $j-1$ that would cause no time window violation along the path:

$$S = \min_{i+1 \le k \le j-1} t_k - (D_{j-1}^{j-1} + \sum_{l=k}^{j-2} d_{l,l+1}).$$

Note that in the definition of S we implicitly assume that the current reversed path is feasible; also note that this definition is independent of any waiting time along the current path.

Expanding the reversed path $(j-1, \ldots, i+1)$ with the edge $\{j, j-1\}$ may change the arrival time at vertex $j-1$ and thereby all departure times along the path $(j-1, \ldots, i+1)$. We define a *local variable* Δ to denote the difference between the new arrival time and the old departure time at vertex $j-1$:

$$\Delta = D_j^j + d_{j,j-1} - D_{j-1}^{j-1}.$$

Δ can be computed in constant time, using $D^j_j = \max\{s_j, D_i + d_{ij}\}$ and $D^{j-1}_{j-1} = \max\{s_{j-1}, D_i + d_{i,j-1}\}$.

In order to prove that we can verify 2-optimality of the tour $(1, 2, \ldots, n, n+1)$ in $O(n^2)$ time, we have to establish two facts: it is possible to update the values of the global variables in constant time, and the new values allow us to handle a single 2-exchange in constant time.

As to updating the global variables, we note that the definition of Δ covers two cases. In the case that $\Delta < 0$, the triangle inequality implies that the old arrival at $j-1$ cannot have been later than the new arrival; hence, the old arrival and departure times did not coincide, so that the old departure occurred at the opening of the time window. But then we have that $-\Delta = W^j_{j-1}$, the new waiting time at $j-1$. In the case that $\Delta \geq 0$, we obviously have $\Delta = D^j_{j-1} - D^{j-1}_{j-1}$, the forward shift of the departure time at $j-1$. We conclude that the new values of the global variables are obtained by

$$T \leftarrow T + d_{j-1,j},$$

$$W \leftarrow \max\{W - \Delta, 0\},$$

$$S \leftarrow \min\{t_j - D^j_j, S - \Delta\}.$$

These updates require constant time.

As to handling a single 2-exchange, the conditions (1), requiring feasibility, and (2), stipulating profitability at vertex $j+1$, can be written as

(1) $D^j_k \leq t_k$ for $k = i+1, \ldots, j$,

(2) $D^j_{j+1} < D_{j+1}$.

The inequalities (1) are obviously equivalent to $S \geq 0$; see Savelsbergh [1986] for a formal proof. For inequality (2), we observe that the new departure time at $j+1$ satisfies

$$D^j_{j+1} = \max\{s_{j+1}, D^j_j + T + W + d_{i+1,j+1}\}.$$

We conclude that conditions (1) and (2) can be tested in constant time.

5. Parallel local search for the TSP with single time windows

We will now present a parallel algorithm for verifying 2-optimality of a time-constrained TSP tour. It requires $O(\log n)$ time and $O(n^2/\log n)$ processors, and thereby has the same resource requirements as in the unconstrained case. An earlier, more cumbersome, variant of the algorithm was presented in a previous paper [Kindervater, Lenstra & Savelsbergh, 1989].

Again, we consider the tour $(1, 2, \ldots, n, n+1)$, which is assumed to be feasible. We start by considering all partial paths along the tour. This enables us to construct the tours that can be obtained by a 2-exchange. Our algorithm has three phases.

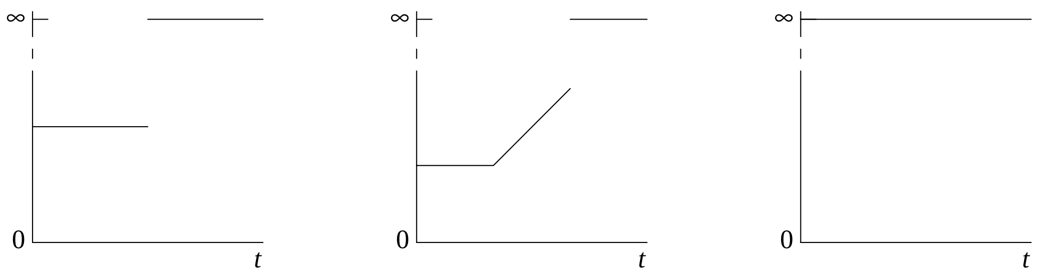

Figure 4. The three possible shapes of the function E_{ij}.

(1) For each pair of vertices $\{i, j\}$ with $i < j$, we define $E_{ij}(t)$ as the earliest possible arrival time at vertex j when traveling along the tour from i to j after arriving at vertex i at time t, and $E_{ji}(t)$ as the earliest possible arrival time at vertex i when traveling from j to i in the reverse direction along the tour after arriving at

vertex j at time t. Note that $E_{1,n+1}(s_1)$ is the arrival time at vertex 1 of the initial tour. We have for all i ($1 \le i \le n$) that

$$E_{i,i\pm 1}(t) = \begin{cases} \max\{s_i, t\} + d_{i,i\pm 1} & \text{for } t \le t_i, \\ \infty & \text{for } t > t_i. \end{cases}$$

The other functions E_{ij} can be obtained by composition. By considering all possibilities, one can show that each of these functions has one of the three shapes shown in Figure 4. Composing functions is an associative operation. Hence, we can use the partial sums algorithm from Section 3 for obtaining all functions E_{ij} in parallel. Since a composition of two functions of the type described here can be derived in constant time, we can in fact determine all functions E_{ij} in $O(\log n)$ time with $O(n^2/\log n)$ processors.

(2) Given all these functions, we compute the earliest arrival time $A_{ij}(k)$ at a few specific vertices k, including the origin, after the replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$:

```
par [$1 \le i < j \le n$] $A_{ij}(j) \leftarrow$ if $E_{1i}(s_1) \le t_i$ then $\max\{s_i, E_{1i}(s_1)\} + d_{ij}$ else $\infty$;
par [$1 \le i < j \le n$] $A_{ij}(i+1) \leftarrow E_{j,i+1}(A_{ij}(j))$;
par [$1 \le i < j \le n$] $A_{ij}(j+1) \leftarrow$ if $A_{ij}(i+1) \le t_{i+1}$ then $\max\{s_{i+1}, A_{ij}(i+1)\} + d_{i+1,j+1}$ else $\infty$;
par [$1 \le i < j \le n$] $A_{ij}(n+1) \leftarrow E_{j+1,n+1}(A_{ij}(j+1))$.
```

For this phase we need $O(1)$ time and $O(n^2)$ processors, or $O(\log n)$ time and $O(n^2/\log n)$ processors.

(3) We now decide whether or not the given tour is 2-optimal in the same way as in the case without time windows:

```
$A_{\min} \leftarrow \min\{A_{ij}(n+1) | 1 \le i < j \le n\}$;
if $E_{1,n+1}(s_1) \le A_{\min}$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let $i*$ and $j*$ be such that $A_{i*j*} = A_{\min}$,
     $(1, \ldots, i*, j*, j*-1, \ldots, i*+1, j*+1, \ldots, n+1)$ is a better feasible tour.
```

For this last phase, the same time and processor bounds as before suffice. So, we end up with an algorithm that runs in $O(\log n)$ time using $O(n^2/\log n)$ processors, which is the same as in the case without time windows.

For each fixed $k > 2$, we can derive a logarithmic-time algorithm along similar lines. One has to take into account that, given k edges, several k-exchanges are possible. Further, the influence of a k-exchange on a tour is more complex. However, it is not hard to see that the running time remains $O(\log n)$ using $O(n^k/\log n)$ processors, which is optimal with respect to the number $\Theta(n^k)$ of k-exchanges.

6. Local search for the TSP with multiple time windows

In Section 4, we have shown that k-exchange algorithms can be adapted to handle a single time window at each vertex without increasing the time complexity. A next natural step is to investigate whether they can also be adapted to handle multiple time windows at each vertex.

Suppose that each vertex has l time windows and must be visited in any one of these. It takes $O(\log l)$ time to determine whether the arrival time at a vertex falls within one of its time windows. The straightforward approach for the verification of 2-optimality therefore requires $O(n^2(n \log l))$ time. We will present an implementation with a time complexity of $O(n(ln \log ln))$, which is better for all realistic values of l.

To simplify the presentation, we will restrict ourselves to the case where each vertex i has two disjoint time windows, denoted by $[s_i^1, t_i^1]$ and $[s_i^2, t_i^2]$.

Let us briefly review the variables introduced in the single time window case. Considering the reversed path $(j-1, \ldots, i+1)$, we have that

T = total travel time along the path,

W = total waiting time along the path after departing from vertex $j-1$,

$S =$ maximum forward shift in time of the departure time at vertex $j - 1$
that would cause no time window violation along the path,

$\Delta =$ difference between the new arrival time and the old departure time at vertex $j - 1$
after expansion of the path with the edge $\{j, j - 1\}$.

In each iteration the global variables were updated using the following formulas:

$T \leftarrow T + d_{j-1,j}$,

$W \leftarrow \max\{W - \Delta, 0\}$,

$S \leftarrow \min\{t_j - D_j^j, S - \Delta\}$.

In the case of multiple time windows, the same set of variables will be used for the verification of feasibility and profitability. Although the use of these variables in handling a single 2-exchange remains unchanged, the rules for updating their values have to be reconsidered, because it is no longer possible to give a closed form expression for each of them. The update formula for T is obviously still valid. The other two are more complicated, as will be explained below.

First, we consider the update of the maximum forward shift. Clearly, infeasibility occurs when a departure time is later than the closing of the last time window. However, there is one other situation that has to be taken into account. It occurs when $D_j^j \le t_j^1$ and $t_j^1 - D_j^j \le S - \Delta < s_j^2 - D_j^j$. Considering the closing of the last window at vertex j would result in the update $S \leftarrow \min\{t_j^2 - D_j^j, S - \Delta\} = S - \Delta$, whereas S should be equal to $t_j^1 - D_j^j$. This deficiency can be circumvented by the following updated updating rule:

$$S \leftarrow \begin{cases} t_j^1 - D_j^j & \text{if } 0 \le t_j^1 - D_j^j \le S - \Delta < s_j^2 - D_j^j, \\ \min\{t_j^2 - D_j^j, S - \Delta\} & \text{otherwise.} \end{cases}$$

Second, we consider the update of the waiting time. An implicit but important characteristic of the single-window case is the fact that shifting the departure time at $j - 1$ forward in time never leads to an increase of the total waiting time on the path $(j - 1, \ldots, i + 1)$. In the multiple-window case this is no longer true. Waiting time might occur anywhere along the reversed path. This global nature of the waiting time is precisely the reason why we are not able to obtain the same time complexity as in the single-window case.

For a fixed value of i, the lexicographic search strategy enables us to maintain a set of triples that can be used to calculate any waiting time along the reversed path in $O(\log n)$ time. Considering the reversed path $(j - 1, \ldots, i + 1)$, this set, denoted by $\{(L_1, U_1, W_1), \ldots, (L_m, U_m, W_m)\}$, will have two properties:

(i) $(L_1, U_1], \ldots, (L_m, U_m]$ form a set of pairwise disjoint half open intervals, and $W_k \ge U_k$ for all k $(1 \le k \le m)$;

(ii) if the arrival time at vertex $j - 1$, denoted by A_{j-1}, satisfies the condition $L_k < A_{j-1} \le U_k$ for some k, then the total waiting time on the reversed path $(j - 1, \ldots, i + 1)$ is equal to $W_k - A_{j-1}$.

When the path $(j - 1, \ldots, i + 1)$ is extended with the edge $\{j, j - 1\}$, we transform the set of triples such that it is defined relative to the departure time at vertex j, by subtracting the travel time $d_{j,j-1}$ from all L_k and U_k. Next, we add the triples $(-\infty, s_j^1, s_j^1)$ and (t_j^1, s_j^2, s_j^2), which handle the waiting time at vertex j only.

There are five basic cases that have to be considered when a triple $(L_{\text{new}}, U_{\text{new}}, W_{\text{new}})$ is added to the current set of triples. Composite cases can all be handled as a sequence of basic ones. A more elaborate discussion of these intricate updates is given by Savelsbergh [1988]. Let A_j denote the arrival time at vertex j.

Case 1. $\forall_k\ (L_{\text{new}}, U_{\text{new}}] \cap (L_k, U_k] = \emptyset$. The simplest case. The new triple is simply added.

Case 2. $\exists_k\ (L_{\text{new}}, U_{\text{new}}] \cap (L_k, U_k] = (L_{\text{new}}, U_{\text{new}}]$. The waiting time incurred by the new triple is completely dominated by the waiting time incurred by the triple (L_k, U_k, W_k). The set of triples is therefore not changed.

Case 3. $\exists_k\ (L_{\text{new}}, U_{\text{new}}] \cap (L_k, U_k] = (L_k, U_k]$. Here, the situation is opposite to the previous case. The waiting time incurred by the new triple completely dominates the waiting time incurred by the triple (L_k, U_k, W_k). The triple (L_k, U_k, W_k) is therefore replaced by $(L_{\text{new}}, U_{\text{new}}, W_{\text{new}})$.

Case 4. $\exists_k\ (L_{\text{new}}, U_{\text{new}}] \cap (L_k, U_k] = (L_k, U_{\text{new}}]$. Here, the situation is a bit more complicated. At first glance, there is only partial dominance. In fact, a kind of chaining occurs. The waiting time incurred when the arrival time at vertex j falls inside the new interval is determined by $W_k - A_j$. The triple (L_k, U_k, W_k) is therefore replaced by $(L_{\text{new}}, U_k, W_k)$.

Case 5. $\exists_k\ (L_{\text{new}}, U_{\text{new}}] \cap (L_k, U_k] = (L_{\text{new}}, U_k]$. Here, there really is partial dominance. When $L_k < A_j \le L_{\text{new}}$, it will still induce a waiting time equal to $W_k - A_j$, but when $L_{\text{new}} < A_j \le U_{\text{new}}$, it will induce a waiting time equal to $W_{\text{new}} - A_j$ instead of $W_k - A_j$. The triple (L_k, U_k, W_k) is therefore replaced by $(L_k, L_{\text{new}}, W_k)$, and a new triple $(L_{\text{new}}, U_{\text{new}}, W_{\text{new}})$ is added.

As to the implementation of such an iteration, we do not actually transform the existing set of triples by subtracting $d_{j,j-1}$. It is more efficient to compute the new triple relative to vertex i by adding T to its three elements. In each iteration, this avoids an $O(m)$ amount of work.

To analyze the complexity of the 2-exchange procedure for the TSP with multiple time windows, let us drop the assumption that there are at most two time windows at each vertex. We assume instead that there are at most l time windows at each vertex, for a fixed l. Now, when the path $(j-1, \ldots, i+1)$ is expanded with the edge $\{j-1, j\}$, there are at most $2l$ intervals that have to be compared with the current set of intervals. The worst that can happen is that each interval leads to the creation of a new interval (Case 1 or Case 5), and the cardinality of the current set of intervals increases by exactly $2l$. In the worst case, we end up with $O(ln)$ intervals. With the appropriate data structures, such as trees, it is possible to perform all necessary operations on the set of intervals in $O(ln \log ln)$ time. This leads to an overall worst case time complexity for testing 2-optimality of $O(n(ln \log ln))$.

7. Parallel local search for the TSP with multiple time windows

Finally, we will give a parallel algorithm for the verification of 2-optimality in the presence of multiple time windows per vertex. As in the previous section, the algorithm will be presented for the case where each vertex has at most two time windows. At the end we will discuss the general case.

A straightforward approach would be a direct modification of the algorithm presented in Section 5, in the sense that we make use of adapted functions E_{ij}. These functions, however, are not that simple any more and the total computational effort would outgrow the one in the sequential case. Fortunately, it turns out that we do not have to determine all functions explicitly.

What we need is an algorithm for the following problem. Given are n vertices, numbered $1, \ldots, n$. Consider the tour $(1, 2, \ldots, n, n+1)$, where the vertices 1 and $n+1$ are the same. Let each vertex i have two time windows, and let $E_{i,i+1}$ deliver the arrival time at vertex $i+1$ as a function of the arrival time at vertex i $(1 \le i \le n)$. What is the arrival time at vertex i when leaving vertex 1 at a given time t_i, for $2 \le i \le n+1$?

The algorithm consists of two phases. The first phase is the same as the first phase of the partial sums algorithm from Section 3. Again, for the sake of simplicity, let $n = 2^m$.

```
**for** $l \leftarrow 1$ **to** $m$ **do**
    **par** $[0 \le j \le 2^{m-l} - 1]$ $E_{1+j2^l,1+(j+1)2^l} \leftarrow E_{1+(2j+1)2^{l-1},1+2(j+1)2^{l-1}} \circ E_{1+2j2^{l-1},1+(2j+1)2^{l-1}}$.
```

Since each vertex has two time windows the functions $E_{i,i+1}$ are piecewise linear with four breakpoints. The number of breakpoints in each of the functions that are obtained by composition may be as large as the sum of the number of breakpoints of the functions from which they are obtained. We start with n functions, with four breakpoints each. In the first iteration, we obtain $n/2$ functions with at most eight breakpoints, in the second iteration $n/4$ functions with at most sixteen breakpoints, and so on. Hence, in each iteration we have to consider $O(n)$ breakpoints in total. Forming a composition of two functions with k breakpoints each can be done in $O(\log k)$ time with $O(k)$ processors using binary search. Since we have to consider $O(n)$ breakpoints at each stage, this phase requires $O(\log^2 n)$ time with $O(n)$ processors.

We are now ready to compute the arrival time at vertex i when leaving vertex 1 at time t_i. This can be done for all i $(2 \le i \le n+1)$ in parallel. Below, v_i denotes an intermediate vertex on the path from vertex 1 to vertex i, and a_i the corresponding arrival time at v_i. Note that we only use those functions E_{ij} that are determined in the previous phase.

```
**par** $[2 \le i \le n+1]$ $a_i \leftarrow t_i$, $v_i \leftarrow 1$;
**for** $l \leftarrow m$ **downto** $0$ **do**
    **par** $[2 \le i \le n+1]$ **if** $v_i + 2^l \le i$ **then** $a_i \leftarrow E_{v_i, v_i+2^l}(a_i)$, $v_i \leftarrow v_i + 2^l$.
```

As in the previous phase, the time needed is $O(\log^2 n)$ with $O(n)$ processors.

So we end up with an algorithm that runs in $O(\log^2 n)$ time with $O(n)$ processors. Since the amount of work involved in the algorithm is $\Theta(n \log^2 n)$, we cannot reduce the number of processors by a significant factor without increasing the computation time.

We return to the verification of 2-optimality of the tour $(1, 2, \ldots, n, n+1)$. Let $A_{ij}(k)$ denote the arrival time at vertex k with respect to 2-exchange $\{i, j\}$. The steps of the verification algorithm are the following:

(1) Compute the arrival times $A_{ij}(i)$. This is achieved by a single invocation of the above algorithm.

(2) Compute the arrival times $A_{ij}(j)$. Here, we go directly from vertex i to vertex j.

(3) Compute the arrival times $A_{ij}(i+1)$. In this step we consider paths along the tour in the reversed direction with different starting times. For all j, we must apply the above routine with the arrival times obtained in the previous step.

(4) Compute the arrival times $A_{ij}(j+1)$. As in step (2), we travel over a single edge.

(5) Compute the arrival times at the origin $A_{ij}(n+1)$. We can obtain these times by applying the above algorithm n times in parallel and modifying the second phase such that it computes the arrival times at vertex $n+1$ given the starting times at the other vertices.

(6) Determine whether the given tour is 2-optimal in the same way as in Section 5.

Steps 3 and 5 dominate the time and processor requirements. Since in each of these steps we apply the basic routine n times in parallel, we have that the total time for verifying 2-optimality with two time windows per vertex is $O(\log^2 n)$ using $O(n^2)$ processors, which gives a total computational effort that is slightly worse than in the sequential case.

The extension to l time windows per vertex has only minor consequences. An operation in the binary trees part of the algorithm now requires $O(\log ln)$ time and $O(ln)$ processors. Further, the time needed for considering a single edge will increase to $O(\log l)$. Hence verifying 2-optimality can be done in $O(\log n \log ln)$ time with $O(ln^2)$ processors.

8. Conclusions

We have described various adaptations of the well-known 2-exchange local search algorithm for the TSP that are able to handle time windows efficiently, on sequential and parallel architectures. The presented techniques can also be used to implement general k-exchange local search algorithms.

In two of the situations considered (no time windows and single time windows), we have achieved a perfect speedup. That is, the total computational effort of a k-exchange algorithm remains the same when we move from a sequential to a parallel computer. In the multiple time window case, the work done by the parallel algorithm exceeds the one in the sequential case by a factor of $O(\log n)$.

An interesting open problem in this context is the following. Given a tour for the TSP and two time windows per vertex, its duration can obviously be computed in $O(n)$ time on a sequential machine and in $O(\log^2 n)$ time on a PRAM with $O(n)$ processors using the techniques described above. Does there exist a parallel algorithm that achieves a perfect speedup?

References

E. Dekel, S. Sahni (1983). Binary trees and parallel scheduling algorithms. *IEEE Trans. Comput. C-32*, 307-315.

D.S. Johnson, C.H. Papadimitriou, M. Yannakakis (1988). How easy is local search? *J. Comput. Syst. Sci. 37*, 79-100.

G.A.P. Kindervater, J.K. Lenstra, M.W.P. Savelsbergh (1989). Parallel local search for the time-constrained traveling salesman problem. J.K. Lenstra, H.C. Tijms, A. Volgenant (eds.) *Twenty-five Years of Operations Research in the Netherlands: Papers Dedicated to Gijs de Leve*, CWI Tract 70, Centre for Mathematics and Computer Science (CWI), Amsterdam, 61-75.

E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (eds.) (1985). *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*, Wiley, Chichester.

S. Lin (1965). Computer solutions of the traveling salesman problem. *Bell System Tech. J. 44*, 2245-2269.

M.W.P. Savelsbergh (1986). Local search for routing problems with time windows. *Ann. Oper. Res. 4*, 285-305.

M.W.P. Savelsbergh (1988). *Computer Aided Routing*, Ph.D. thesis, CWI, Amsterdam.

M.W.P. Savelsbergh (1991). *The Vehicle Routing Problem with Time Windows: Minimizing Route Duration*, COSOR-memorandum 91-03, Eindhoven University of Technology.