Local Search in Physical Distribution Management

G.A.P. Kindervater
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam
The Netherlands

M.W.P. Savelsbergh
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands

1. Introduction

Physical distribution management presents a variety of decision making problems at the three levels of strategic, tactical and operational planning. Decisions relating to the location of facilities (plants, warehouses or depots) may be viewed as strategic, while problems of fleet size and fleet mix determination can be termed tactical. On the operational level, two problems prevail: the routing of capacitated vehicles through a collection of customers to pickup or deliver goods, and the scheduling of vehicles to meet time or precedence constraints imposed upon their routes.

The importance of effective and efficient distribution management is evident from its associated costs. Physical distribution management at the operational level, which is considered in this paper, is responsible for an important fraction of the total distribution costs. Small relative savings in these expenses could already account for substantial savings in absolute terms. The significance of detecting these potential savings has become increasingly apparent due to the escalation of the costs involved, such as capital and fuel costs and driver salaries.

Not surprisingly, there is a growing demand for planning systems that produce economical routes. Although cost optimization is often the primary objective for purchasing computerized systems for physical distribution management, there are other benefits that should not be underestimated. The introduction of such systems enables companies to maintain a higher level of service towards their customers, it makes them less dependent of their planners, it supplies better management information facilities, and it makes the conduct of work faster and simpler.

Besides its obvious practical importance, physical distribution management also provides some fascinating basic models, such as the traveling salesman problem, the vehicle routing problem, and the pickup and delivery problem. Consequently, researchers from the fields of operations research, mathematics, and computer science, have spent man-centuries in trying to develop solution approaches for these problems. For an extensive survey of models and solution methods in vehicle routing and scheduling see Bodin, Golden, Assad & Ball [1983].

Report EUR-CS-92-05
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

In the last decade, enormous theoretical as well as practical advances have been made. Certainly one of the most important advances is the incorporation of 'real world' characteristics, such as time windows and precedence relations, into the basic models.

Some of the resulting vehicle routing and scheduling models will be discussed in this paper. The first is the *vehicle routing problem with time windows* (VRPTW) [Desrochers, Lenstra, Savelsbergh & Soumis 1988], which is defined as follows. A number of vehicles, each with a given capacity, is located at a single depot and must serve a number of geographically dispersed customers. Each customer has a given demand and must be served within a specified time window. The objective is to minimize the total cost of travel.

The special case in which the vehicle capacities are infinite is called the *multiple traveling salesman problem with time windows* (m-TSPTW). It arises in school bus routing problems. The problem here is to determine routes that start at a single depot and cover a set of trips, each of which starts within a time window. There are no capacity constraints, since each trip satisfies those by definition and vehicles moving between trips are empty.

The second model is the *pickup and delivery problem with time windows* (PDPTW) [Dumas, Desrosiers & Soumis 1991]. Again, there is a single depot, a number of vehicles with given capacities, and a number of customers with given demands. Each customer must be served to pick up goods at his origin during a specified time window, and to deliver the items at his destination during another specified time window. The objective is to minimize total travel cost.

The special case in which all customer demands are equal is called the *dial-a-ride problem* (DARP). It arises in transportation systems for the handicapped and the elderly. In these situations, the temporal constraints imposed by the customers strongly restrict the total vehicle load at any point in time, and the capacity constraints are of secondary importance. The cost of a route is a combination of travel time and customer dissatisfaction.

An important consideration in the formulation and solution of vehicle routing and scheduling problems is the required computational effort associated with various solution techniques. Virtually all vehicle routing and scheduling problems belong to the class of *NP*-hard problems. This indicates that it is difficult to solve even small instances of a problem to optimality with a reasonable computational effort. As a consequence, when we have to solve real-life problems, we should not insist on finding an optimal solution, but instead on finding an acceptable solution within an acceptable amount of computation time. To accomplish this we have to resort to approximation algorithms.

Approximation algorithms for vehicle routing and scheduling problems usually have two phases: a *construction phase*, in which an initial feasible solution is constructed, and an *improvement phase*, in which an attempt is made to improve that initial solution by repeatedly searching a specified neighborhood for a better one. This paper focuses on the techniques developed for the improvement phase.

To the best of our knowledge, most iterative improvement methods that have been applied to the vehicle routing and scheduling problems are extensions of the well-known k-exchange algorithms, which have been studied extensively in the context of the traveling salesman problem, see for instance Bentley & Johnson [1993]. Extensions are needed to handle the side constraints imposed by the vehicle routing and scheduling problem under consideration.

Actually, the main thrust of the research in this area has not been on how to handle side constraints, which is trivial, but on how to handle side constraints efficiently, which is non-trivial.

The paper is organized as follows. In Section 2, the heart of the paper, we will present a survey of the various methods that have been proposed to implement k-exchange methods for constrained variants of the TSP efficiently. In Section 3, we will see how these methods can be used in a situation with multiple routes.

2. The traveling salesman problem with side constraints

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 consider two local search strategies for the TSP. In both cases, the neighborhoods considered are tours that can be obtained from an initial Hamiltonian cycle by replacing a set of k of its edges by another set of k

edges. Such replacements are called *k-exchanges*, and a tour that cannot be improved by a *k*-exchange is said to be *k-optimal*. For two reasons, we only consider *k*-exchanges for $k \leq 3$. First, *k*-exchanges for $k > 3$ are seldomly used in iterative improvement methods for vehicle routing and scheduling problems. Second, *k*-exchanges for $k \leq 3$ suffice to illustrate the techniques we want to discuss. In the end, it should be clear how the presented techniques can be extended to the general case.

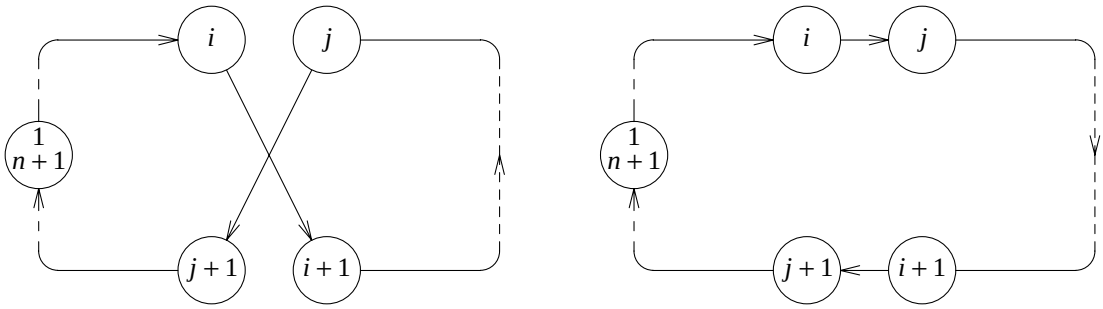


Figure 1. A 2-exchange.

(1) Try to improve the tour by replacing 2 of its edges by 2 other edges, i.e., a 2-exchange, and iterate until no further improvement is possible. An example of a 2-exchange is given in Figure 1.

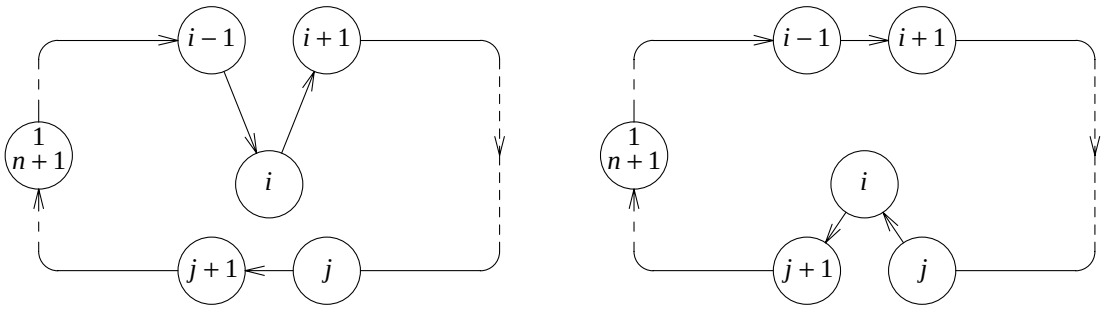


Figure 2. A forward Or-exchange.

(2) Try to improve the tour by relocating a chain of l consecutive vertices and iterate until no further improvement is possible. This type modification of the tour involves the replacement of a set of three edges, with a restriction on the choice of edges. Hence, the neighborhood is a subset of the 3-exchange neighborhood. Since the procedures involved are conceptually the same for all values of l, we will restrict ourselves to $l = 1$. This type of exchanges was first considered by Or [1976], and are, therefore, called *Or-exchanges*. We speak of a *forward* Or-exchange if the vertex is moved to a place further on the tour (cf. Figure 2), and of a *backward* Or-exchange otherwise (cf. Figure 3).

The complicating side constraints that appear in the VRPTW and PDPTW and that we want to incorporate are:

- precedence relations between vertices;
- collections or deliveries at vertices;
- time windows at vertices.

We will consider each of these side constraints separately, but it should be clear that most of the techniques presented can easily be used to produce a variant that handles a combination of them. We start by introducing the three variants of the TSP for which we will study the exchange procedures in more detail.

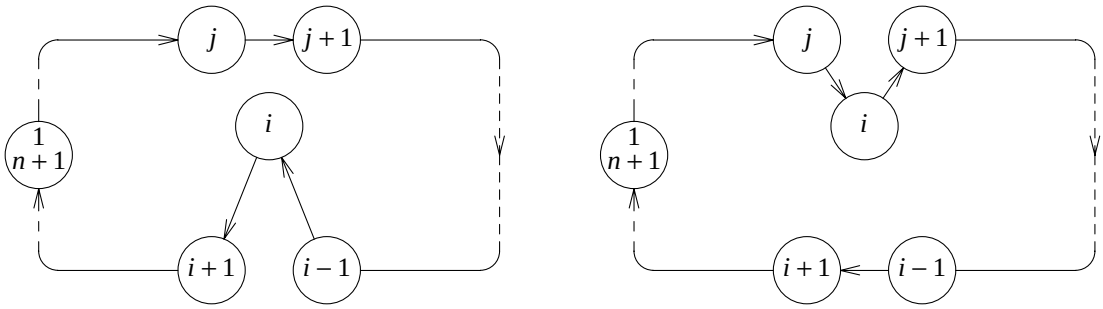


Figure 3. A backward Or-exchange.

In the *TSP with precedence constraints*, we are given, in addition to the travel times d_{ij} between each pair of vertices (i, j), precedence constraints specifying that some pairs of vertices have to be visited in a prescribed order. The *single-vehicle dial-a-ride problem*, where a single vehicle has to pick up and deliver n customers, is an example of the TSP with precedence constraints. Given a precedence related pair of vertices $u \rightarrow v$, we will often refer to the origin u and the destination v, the meaning of which is obvious.

In the *TSP with collections and deliveries*, we are given, in addition to the travel times between vertices, for each vertex i an associated load q_i that can be either positive or negative depending on whether the load has to be collected or delivered. The salesman uses a vehicle with fixed capacity Q.

In the *TSP with time windows*, we are given, in addition to the travel times, for each vertex i a time window on the departure time, denoted by $[e_i, l_i]$, where e_i specifies the earliest service time and l_i the latest service time. Arriving earlier than e_i introduces a waiting time at vertex i; arriving after l_i leads to infeasibility. We will use the following notation: A_i will denote the arrival time at vertex i, D_i will denote the departure time at vertex i, and W_i will denote the waiting time at vertex i.

For notational convenience, we will assume that the current tour is given by the sequence $(1, 2, \ldots, i, \ldots, n, n+1)$, where the origin 1 and the destination $n+1$ denote the same vertex, and i represents the i-th vertex. We also assume that a 2-exchange replaces two edges $\{i, i+1\}$ and $\{j, j+1\}$, with $j > i$, by the edges $\{i, j\}$ and $\{i+1, j+1\}$, and that an Or-exchange involves the substitution of edges $\{i-1, i\}$, $\{i, i+1\}$ and $\{j, j+1\}$ by $\{i-1, i+1\}$, $\{j, i\}$ and $\{i, j+1\}$.

The main difficulty with the use of exchange procedures in the TSP with side constraints is testing the feasibility of an exchange, as opposed to the TSP where one only has to test whether the exchange is profitable and one does not have to bother about feasibility. A 2-exchange, for instance, will reverse the path $(i+1, \ldots, j)$, which means that one has to check the feasibility of at least all the vertices on the new path with respect to those constraints. In a straightforward implementation this requires $O(n^2)$ time in the TSP with precedence constraints, and $O(n)$ time in the TSP with collections and deliveries and the TSP with time windows.

By applying preprocessing techniques [Psaraftis 1983, Solomon, Baker & Shaffer 1988], tailored updating schemes [Solomon, Baker & Shaffer 1988], and lexicographic search strategies [Savelsbergh 1986, 1990, 1992] several researchers have been able to incorporate the various side constraints with an acceptable or even without an increase in computation times.

Before focusing on improvement methods of a given tour, it is worthwhile to mention the influence of the side constraints on the construction of an initial feasible tour. In the TSP with precedence relations, an initial tour can be obtained in polynomial time by visiting the vertices in topological order. In the other two cases, the problem of deciding whether a feasible tour exists is NP-complete [Savelsbergh 1990].

2.1. Preprocessing

Psaraftis [1983] was the first to study k-exchange procedures for a constrained variant of the TSP. He studied 2-exchanges and Or-exchanges in the context of the single vehicle dial-a-ride problem. The dial-a-ride problem is a TSP with precedence constraints in which each vertex is related to precisely one other vertex.

First, we examine the 2-exchanges. A straightforward test for feasibility in this case requires $O(n^2)$ time. This can be seen as follows. The only way that a 2-exchange can be infeasible is if there is at least one precedence related pair of vertices on the segment of the tour that is reversed. The simplest way to find out whether such a pair of vertices exists is to examine all pairs of vertices in the segment. This requires $O(n^2)$ time, and would lead to an overall complexity for verification of 2-optimality of $O(n^4)$.

Psaraftis shows how this can be reduced to $O(n^2)$ by performing a *screening* procedure in the beginning of the algorithm which determines the feasibility of every possible 2-exchange. Information from the screening procedure is stored in a feasibility matrix to be examined during the execution of the actual algorithm.

The screening procedure is based on the following observation: if $firstdes_i$ denotes the first destination of a precedence relation for which both origin and destination lie beyond i, then the exchange of $\{i, i+1\}$ and $\{j, j+1\}$ with $\{i, j\}$ and $\{i+1, j+1\}$ is feasible if and only if $j < firstdes_i$. The screening procedure first computes the values $firstdes_i$ and then constructs the feasibility matrix $feas$, i.e., $feas_{ij} = 1$ if $j < firstdes_i$ and $feas_{ij} = 0$ otherwise. Psaraftis shows that the values $firstdes_i$ can be computed in $O(n^2)$ time, thus proving that verification of 2-optimality can be done in $O(n^2)$ time.

Next, we examine the Or-exchanges. An advantageous feature of Or-exchanges is that they are *direction preserving*, i.e., the segments determined by the deletion of $\{i-1, i\}$, $\{i, i+1\}$, and $\{j, j+1\}$ are traversed in the same direction in the final tour. Therefore, if these segment are feasible in the original tour, they will also be feasible in the final tour. There is only one situation which leads to infeasibility: a precedence related pair of vertices with origin i and destination in the segment $(i+1, \ldots, j)$ for a forward Or-exchange and a precedence related pair of vertices with origin in the segment $(j+1, \ldots, i-1)$ and destination i for a backward Or-exchange, since the order, in which the vertex i and the segment are traversed in the final tour, is reversed.

Similar screening procedures can be developed for both forward and backward Or-exchanges. For forward Or-exchanges, we need the following observation: if $firstdes_i$ denotes the first destination of a precedence relation with origin i, then the exchange of $\{i-1, i\}$, $\{i, i+1\}$ and $\{j, j+1\}$ with $\{i-1, i+1\}$, $\{j, i\}$ and $\{i, j+1\}$ is feasible if and only if $j < firstdes_i$. An analogous observation can be made for backward-Or-exchanges.

The values $firstdes_i$ can be computed in $O(n^2)$ time, thus proving an overall complexity of $O(n^2)$ for verification of Or-optimality.

Solomon, Baker & Schaffer [1988] have adapted the above presented preprocessing scheme for the TSP with time windows. The idea is to identify precedence relations between pairs of vertices based on their time windows. If $e_i + d_{ij} > l_j$, then vertex j has to precede vertex i. Note that the use of this type of preprocessing does not eliminate the need for further checking of feasibility; it may be used as a filter to reduce the number of complete feasibility checks required.

2.2. Direction preserving exchanges

Solomon, Baker & Schaffer [1988] have carried out an extensive computational study on the efficient implementation of k-exchange procedures for the TSP with time windows. Their implementation incorporated the preprocessing scheme discussed above, as well as tailored updating mechanisms for direction preserving exchanges. These updating mechanisms are based on the observation that if the direction in which we traverse a path is unchanged, we can check the feasibility of each vertex on the path by simply looking at the change in arrival time.

Consider a path $(u, u+1, \ldots, v)$ with associated departure times and suppose that the departure time at the first vertex of the path is decreased. This defines a *push backward*

$$B_u = D_u - D_u^{\text{new}},$$

where D_u and D_u^{new} define the current and the new departure time at vertex u. The push backward at the next

vertex on the path can be computed by

$$B_{u+1} = \min\{B_u, D_{u+1} - e_{u+1}\}.$$

Obviously, all vertices on the path remain feasible and the departure times D_k need to be adjusted sequentially for $k = u, \ldots, v$ as long as $B_k > 0$.

Similarly, consider a path $(u, u+1, \ldots, v)$ with associated departure times and suppose that the departure time at the first vertex of the path is increased. This defines a *push forward*

$$F_u = D_u^{\text{new}} - D_u.$$

The push forward at the next vertex on the path can be computed as follows

$$F_{u+1} = \max\{F_u - W_{u+1}, 0\},$$

where W_{u+1} denotes the waiting time at vertex $u+1$. Obviously, if $F_u > 0$ some vertices on the path could become infeasible. The vertices on the path have to be checked sequentially. At a vertex k ($u \le k \le v$), it may happen that $D_k + F_k > l_k$ in which case the path is no longer feasible, or $F_k = 0$ which indicates that the path from vertex k to vertex v has not been changed.

Observe that in the worst case testing the feasibility of an exchange takes $O(n)$ time. However, in practice, the use of a push backward and a push forward leads to a substantial reduction in the number of vertices being examined for time feasibility. Note that these techniques are only useful in direction preserving exchanges.

2.3. Lexicographic search

Savelsbergh [1986, 1990, 1992] introduced an approach that can be used to incorporate all three side constraints in exchange procedures without increasing the time complexity of verification of local optimality.

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 global variables requires no more than constant time. Because the search strategy is of crucial importance, we present it first.

Lexicographic search for 2-exchanges. We choose the edges $\{i, i+1\}$ in the order in which they appear in the current route starting with $i = 1$ up to $i = n-2$; this will be referred to as the outer loop. After fixing an edge $\{i, i+1\}$, we choose the edge $\{j, j+1\}$ successively equal to $\{i+2, i+3\}, \{i+3, i+4\}, \ldots, \{n, n+1\}$ (cf. Figure 4); this will be referred to as the inner loop.

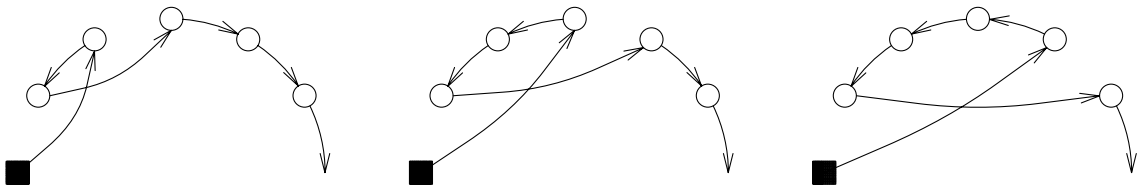


Figure 4. The search strategy for 2-exchanges.

Now consider all possible exchanges for a fixed edge $\{i, i+1\}$. The inspection of the 2-exchanges in the order given above implies that in the inner loop the previously reversed path $(i+1, \ldots, j-1)$, corresponding to the substitution of $\{i, i+1\}$ and $\{j-1, j\}$ with $\{i, j-1\}$ and $\{i+1, j\}$, is expanded by the edge $\{j-1, j\}$.

Lexicographic search for forward Or-exchanges. We choose vertex i in the order of the current route starting with i equal to 2. After fixing i, we choose the edge $\{j, j+1\}$ to be $\{i+1, i+2\}, \{i+2, i+3\}, \ldots, \{n, n+1\}$ consecutively. That is, the edge $\{j, j+1\}$ 'walks forward' through the route. Note that in the inner loop in each newly examined exchange the path $(i+1, \ldots, j-1)$ of the previously considered exchange is expanded with the edge $\{j-1, j\}$.

Lexicographic search for backward Or-exchanges. We choose vertex i in the order of the current route starting with i equal to 2. After fixing i, we choose the edge $\{j, j+1\}$ to be $\{i-2, i-1\}, \{i-3, i-2\}, \ldots, \{1, 2\}$ in that order. That is, the edge $\{j, j+1\}$ 'walks backward' through the route. Note that in the inner loop in each newly examined exchange the path $(j+2, \ldots, i-1)$ of the previously considered exchange is expanded with the edge $\{j+1, j+2\}$.

Now that we have presented the search strategy, let us return to the feasibility question. In order to test the feasibility of a single 2-exchange, we have to check all the vertices on the reversed path $(i+1, \ldots, j)$ and on the path $(j+1, \ldots, n+1)$, and in order to test the feasibility of a single forward (or backward) Or-exchange, we have to check besides vertex i the vertices on the paths $(i+1, \ldots, j)$ and $(j+1, \ldots, n+1)$ (or $(j+1, \ldots, i-1)$ and $(i+1, \ldots, n+1)$). In a straightforward implementation this takes $O(n)$ time for each single exchange. We will present an implementation that requires only constant time per exchange.

The idea is to define an appropriate set of global variables, which will of course depend on the constrained variant of the TSP we are considering, in such a way that

- the set of global variables makes it possible to test the feasibility of an exchange in constant time, i.e., the feasibility of all the vertices on the paths in question can be checked in constant time, and
- the lexicographic search strategy makes it possible to maintain the correct values for the set of global variables in constant time, i.e., the update of the global variables when we go from one exchange to the next one can be done in constant time.

To see how these ideas work out in actual implementations, we show the pseudo-code of a general framework for a 2-exchange procedure.

```
procedure TwoExchange
(* input: a route given as (1, ..., n+1) *)
(* output: a route that is 2-optimal *)
REPEAT:
  for i ← 1 to n − 2 do                                   (* outer loop: fix edge {i, i+1} *)
    InitGlobal (i, G)                                     (* initialize global variables *)
    for j ← i + 2 to n do                                 (* inner loop: fix edge {j, j+1} *)
      if ProfitableExchange (i, j, G) and                 (* test whether the exchange is profitable *)
        FeasibleExchange (i, j, G)                        (* test whether the exchange is feasible *)
      then
        PerformExchange (i, j)                            (* replace edges {i, i+1} and {j, j+1} by edges {i, j} and {i+1, j+1} *)
        goto REPEAT
      UpdateGlobal (i, j, G)                              (* update global variables for the next iteration *)
END:
```

Although the above pseudo-code looks rather simple, defining a set of global variables in such a way that, in combination with the lexicographic search strategy, the functions InitGlobal(), ProfitableExchange (), FeasibleExchange(), and UpdateGlobal() do what they are supposed to do and take only constant time, is often not so obvious.

Reexamining a 2-exchange, a forward Or-exchange and a backward Or-exchange, we see that the algorithms have to be able to handle reversing a path, relocating a path backward and relocating a path forward. As these three types of changes comprise all the possibilities that can occur in a k-exchange (for arbitrary k), the techniques can be used to implement k-exchange algorithms for the TSP with side constraints for arbitrary k without increasing the time complexity beyond $O(n^k)$.

Precedence relations
As a first illustration of the proposed technique we show how precedence relations can be handled. Obviously, we cannot improve the time complexity of $O(n^2)$ of Psaraftis' implementation for the verification of 2-optimality. In fact, we will just present an alternative and simpler implementation of his idea. Attach a label to each vertex u with information on the first vertex on the tour for which a precedence relation $u \rightarrow v$ exists: $first_u = \min\{v | u \rightarrow v\}$.

2-Exchanges. Recall that the exchange of $\{i, i+1\}$ and $\{j, j+1\}$ with $\{i, j\}$ and $\{i+1, j+1\}$ is feasible if and only if $j < firstdes_i$, where $firstdes_i$ denotes the first destination of a precedence relation for which both origin and destination lie beyond i. This is equivalent to stating that a 2-exchange is feasible if and only if there is no precedence related pair of vertices on the path that is reversed. Hence, at any stage information concerning this path is sufficient. The lexicographic search strategy makes it possible to gather the required knowledge using a single global variable. We introduce the variable F to denote the first destination of a precedence relation for which the corresponding origin is on the path that is to be reversed. Feasibility of a 2-exchange is now established by verifying that the vertex denoted by the variable F is not on the reversed path.

All we have to do now is to show that we can ensure that the global variable F contains the right information when it is examined. The lexicographic search strategy provides a simple way to accomplish this. In the outer loop, whenever we expand the path $(1, \ldots, i-1)$ with the edge $\{i-1, i\}$, we set F equal to $first_{i+1}$. In the inner loop, whenever we expand the path $(i+1, \ldots, j-1)$ with the edge $\{j-1, j\}$, we set F equal to the minimum of its current value and $first_j$.

Or-exchanges. A forward Or-exchange is feasible if and only if there is no pair of precedence-related vertices with the first being i and the other on the path $(i+1, \ldots, j)$. Whenever we try to expand the path $(i+1, \ldots, j-1)$ with the edge $\{j-1, j\}$ and $i \rightarrow j$, i.e., vertex i is a predecessor of vertex j, the expansion will only result in infeasible exchanges. Similarly, a backward Or-exchange is feasible if and only if there is no pair of precedence-related vertices with the first on the path $(j+1, \ldots, i-1)$ and the other being i. Whenever we try to expand the path $(j+2, \ldots, i-1)$ with the edge $\{j+1, j+2\}$ and $j+1 \rightarrow i$, i.e., vertex $j+1$ is a predecessor of vertex i, the expansion will only result in infeasible exchanges.

Before discussing collections and deliveries and time windows, we take another close look at a k-exchange and the lexicographic search strategy. A k-exchange is the substitution of k links of a tour with k other links. The first step is the deletion of k links, i.e., the tour is broken up into k paths. The second step is the addition of k other links, i.e., the k paths are concatenated in a different order to form a new tour.

More specifically, a 2-exchange deletes the links $\{i, i+1\}$ and $\{j, j+1\}$, with $j > i$, to form the paths $(1, \ldots, i)$, $(i+1, \ldots, j)$, and $(j+1, \ldots, n+1)$, and then adds the links $\{i, j\}$ and $\{i+1, j+1\}$ to obtain the new tour $(1, \ldots, i, j, \ldots, i+1, j+1, \ldots, n+1)$, a forward Or-exchange deletes the links $\{i-1, i\}$, $\{i, i+1\}$ and $\{j, j+1\}$ to form the paths $(1, \ldots, i-1)$, (i), $(i+1, \ldots, j)$, and $(j+1, \ldots, n+1)$, and then adds the links $\{i-1, i+1\}$, $\{j, i\}$ and $\{i, j+1\}$ to obtain the new tour $(1, \ldots, i-1, i+1, \ldots, j, i, j+1, \ldots, n+1)$, and a backward Or-exchange deletes the links $\{i-1, i\}$, $\{i, i+1\}$ and $\{j, j+1\}$ to form the paths $(1, \ldots, j)$, $(j+1, \ldots, i-1)$, (i), and $(i+1, \ldots, n+1)$, and then adds the links $\{i-1, i+1\}$, $\{j, i\}$ and $\{i, j+1\}$ to obtain the new tour $(1, \ldots, j, i, j+1, \ldots, i-1, i+1, \ldots, n+1)$.

The key feature of the lexicographic search strategy is that in consecutive iterations the k paths that result after the deletion of the k links differ by at most a single vertex. The proposed implementation scheme for k-exchange methods associates a set of global variables with each of these paths containing information on its feasibility and its profitability. The global variables are chosen such that initializing the global variables for a single vertex path, and computing the values of the global variables for a concatenated path, takes constant time.

Collections and deliveries

The following three quantities turn out to be sufficient for the analysis of feasibility for the TSP with collections and deliveries.

- The *maximum load* $L^{\max}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

  $L^{\max}_{(u_1,\ldots,u_k)} = \max_{1\le i\le k} \Sigma_{1\le j\le i} q_{u_j}$.

- The *minimum load* $L^{\min}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

  $L^{\min}_{(u_1,\ldots,u_k)} = \min_{1\le i\le k} \Sigma_{1\le j\le i} q_{u_j}$.

- The *final load* $L^{\text{final}}_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, assuming the vehicle is empty when it arrives at vertex u_1, i.e.,

  $L^{\text{final}}_{(u_1,\ldots,u_k)} = \Sigma_{1\le i\le k} q_{u_i}$.

Initializing these quantities for a single vertex path (u) is trivial: $L^{\max}_{(u)} = L^{\min}_{(u)} = L^{\text{final}}_{(u)} = q_u$. The next proposition shows that if we concatenate two (vertex-disjoint) paths, we can compute the quantities for the resulting path from the quantities of its constituent paths in constant time.

Proposition. If two (vertex-disjoint) feasible paths $(u_1,\ldots,u_k)$ and $(v_1,\ldots,v_l)$, with associated maximal loads $L^{\max}_{(u_1,\ldots,u_k)}$ and $L^{\max}_{(v_1,\ldots,v_l)}$, minimal loads $L^{\min}_{(u_1,\ldots,u_k)}$ and $L^{\min}_{(v_1,\ldots,v_l)}$, and final loads $L^{\text{final}}_{(u_1,\ldots,u_k)}$ and $L^{\text{final}}_{(v_1,\ldots,v_l)}$, are concatenated, the same values for the resulting path $(u_1,\ldots,u_k,v_1,\ldots,v_l)$ are given by:

$L^{\max}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \max\{L^{\max}_{(u_1,\ldots,u_k)}, L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\max}_{(v_1,\ldots,v_l)}\}$,
$L^{\min}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \min\{L^{\min}_{(u_1,\ldots,u_k)}, L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\min}_{(v_1,\ldots,v_l)}\}$, and
$L^{\text{final}}_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = L^{\text{final}}_{(u_1,\ldots,u_k)} + L^{\text{final}}_{(v_1,\ldots,v_l)}$.

Proof: Trivial. □

From the discussion on k-exchanges and the lexicographic search strategy, it should be clear that the above proposition shows that checking the feasibility of an exchange, i.e., $L^{\max}_{(1,\ldots,n+1)} \le Q$ and $L^{\min}_{(1,\ldots,n+1)} \ge 0$, as well as updating between consecutive iterations can be done in constant time.

Time windows

Under the assumption that on its way a vehicle always departs at a vertex as early as possible, which is the best choice from a feasibility point of view, a path can be completely specified by giving the sequence in which the vertices are visited and the departure time at the first vertex of the path.

The following quantities turn out to be extremely useful in the analysis of feasibility and profitability of k-exchanges in the TSP with time windows.

- The *total travel time* $T_{(u_1,\ldots,u_k)}$ on the path $(u_1,\ldots,u_k)$, i.e.,

  $T_{(u_1,\ldots,u_k)} = \Sigma_{1\le i<k} d_{u_i u_{i+1}}$.

- The *earliest departure time* $E_{(u_1,\ldots,u_k)}$ at vertex u_k of the path $(u_1,\ldots,u_k)$, assuming vertex u_1 is left at the opening of its time window, i.e.,

  $E_{(u_1,\ldots,u_k)} = \max_{1\le i\le k}\{e_i + T_{(u_i,\ldots,u_k)}\}$.

- The *latest arrival time* $L_{(u_1,\ldots,u_k)}$ at vertex u_1 of the path $(u_1,\ldots,u_k)$, such that the path remains feasible, i.e.,

  $L_{(u_1,\ldots,u_k)} = \min_{1\le i\le k}\{l_i - T_{(u_1,\ldots,u_i)}\}$.

Other interesting quantities can be obtained using the above values. So is, for example, the waiting time on the path $(u_1, \ldots, u_k)$ equal to $E_{(u_1,\ldots,u_k)} - e_{u_1} - T_{(u_1,\ldots,u_k)}$.

The following proposition shows that if we concatenate two paths $(u_1, \ldots, u_k)$ and $(v_1, \ldots, v_l)$, we can compute the same quantities for the resulting path $(u_1, \ldots, u_k, v_1, \ldots, v_l)$ from the quantities of its constituent paths in constant time.

Proposition. If two (vertex-disjoint) feasible paths $(u_1, \ldots, u_k)$ and $(v_1, \ldots, v_l)$, with associated total travel times $T_{(u_1,\ldots,u_k)}$ and $T_{(v_1,\ldots,v_l)}$, earliest departure times $E_{(u_1,\ldots,u_k)}$ and $E_{(v_1,\ldots,v_l)}$, and latest arrival times $L_{(u_1,\ldots,u_k)}$ and $L_{(v_1,\ldots,v_l)}$, are concatenated, the resulting path is feasible if and only if $E_{(u_1,\ldots,u_k)} + d_{u_k,v_1} \leq L_{(v_1,\ldots,v_l)}$ and the same values for the resulting path are given by:

$$T_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = T_{(u_1,\ldots,u_k)} + d_{u_k,v_1} + T_{(v_1,\ldots,v_l)},$$
$$E_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \max\{E_{(u_1,\ldots,u_k)} + d_{u_k,v_1} + T_{(v_1,\ldots,v_l)}, E_{(v_1,\ldots,v_l)}\}, \text{ and}$$
$$L_{(u_1,\ldots,u_k,v_1,\ldots,v_l)} = \min\{L_{(u_1,\ldots,u_k)}, L_{(v_1,\ldots,v_l)} - T_{(u_1,\ldots,u_k)} - d_{u_k,v_1}\}.$$

Proof: Trivial. □

From the discussion on k-exchanges and the lexicographic search strategy, it should be clear that this proposition shows that checking the feasibility of a k-exchange as well as updating between consecutive iterations can be done in constant time.

The presence of time windows also allows for the specification of a variety of objective functions, such as minimize the total travel time, i.e., $T_{(1,\ldots,n+1)}$, minimize the completion time, i.e., $E_{(1,\ldots,n+1)}$, and minimize the route duration, i.e., $\max\{E_{(1,\ldots,n+1)} - L_{(1,\ldots,n+1)}, T_{(1,\ldots,n+1)}\}$.

2.4. Parallel implementations

Nowadays, many computers are able to perform a number of operations in parallel. Such computers have a greater processing power than a serial one, thus making it possible to obtain substantial speedups. In this section, we will discuss the verification of local optimality on a parallel random access machine (PRAM), a machine model in which an unbounded number of processors operate in parallel and communicate with each other in constant time through a shared memory. The shared memory allows simultaneous reads from the same location but disallows simultaneous writes into the same location. Although the PRAM is hardly a realistic computer model, the resulting algorithms can be adequately used for implementation on any 'real-world' machine and the overhead introduced is only minimal (see for example Alt, Hagerup, Mehlhorn & Preparata [1987] and Karlin & Upfal [1988]).

Before addressing the issue of local optimality, 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, 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 $[2^l \leq j \leq 2^{l+1} - 1]$ $a_j \leftarrow a_{2j} + a_{2j+1}$;
$b_1 \leftarrow a_1$;
for $l \leftarrow 1$ **to** m **do**
    par $[2^l \leq j \leq 2^{l+1} - 1]$ $b_j \leftarrow$ **if** j odd **then** $b_{(j-1)/2}$ **else** $b_{j/2} - a_{j+1}$.

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

The computation is illustrated in Figure 5. 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

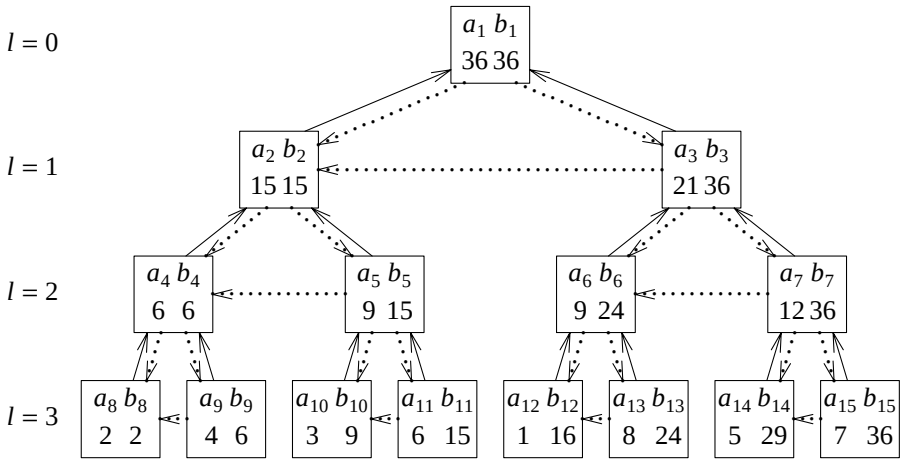


Figure 5. Partial sums: an instance with $n = 8$.

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. The total computational effort is $O(\log n \cdot n/\log n) = O(n)$, as it is in the serial case. This is called a *full processor utilization* or a *perfect speedup*.

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 local optimality. The following procedure decides whether or not the tour $(1, 2, \ldots, n, n+1)$ is 2-optimal:

par $[1 \le i < j \le n]$ $\delta_{ij} \leftarrow d_{ij} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$; (* compute the effect of all possible 2-exchanges *)
$\delta_{\min} \leftarrow \min\{\delta_{ij} | 1 \le i < j \le n\}$; (* search for the best 2-exchange *)
if $\delta_{\min} \ge 0$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^*, with $i^* < j^*$, 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.

For the verification of Or-optimality of the tour $(1, 2, \ldots, n, n+1)$, we obtain almost the same algorithm:

par $[1 < i < j \leq n \,\&\, 1 \leq j < i-1 < n]$ $\delta_{ij} \leftarrow \min\{d_{i-1,i+1} + d_{j,i} + d_{i,j+1} - d_{i-1,i} - d_{i,i+1} - d_{j,j+1}\}$;
$\delta_{\min} \leftarrow \min\{\delta_{ij} | 1 < i < j \leq n \text{ or } 1 \leq j < i-1 < n\}$;
if $\delta_{\min} \geq 0$
then $(1, 2, \ldots, n, n+1)$ is an Or-optimal tour
else let i^* and j^*, with $1 < i^* < j^* \leq n$ or $1 \leq j^* < i^* - 1 < n$, be such that $\delta_{i^*j^*} = \delta_{\min}$,
if $i^* < j^*$
then $(1, \ldots, i^*-1, i^*+1, \ldots, j^*, i^*, j^*+1, \ldots, n+1)$ is a shorter tour
else $(1, \ldots, j^*, i^*, j^*+1, \ldots, i^*-1, i^*+1, \ldots, n+1)$ 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 procedures 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. Hence, we have obtained 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 or Or-optimality always take the same amount of time.

Precedence relations

In Section 2.3 we have seen that the introduction of precedence relations into the basic model did not influence the time complexity in the sequential case. The idea used there cannot be applied in the parallel case since it is inherently sequential. It turns out, however, that a preprocessing phase as done by Psaraftis [1983] (see also Section 2.1) has an appropriate parallel equivalent. The preprocessing enables us to decide on local optimality of a given tour in the same way as before, without an increase in the time and processor requirements.

We consider the tour $(1, 2, \ldots, n, n+1)$, which is assumed to be feasible. Further let the set of precedence relations be given by $\{i_k \rightarrow j_k | 1 \leq k \leq m\}$, where $i_k \rightarrow j_k$ denotes that vertex i_k must be visited before vertex j_k. For technical reasons, it is convenient to assume that the set of relations contains the relations $i \rightarrow n+1$ for each vertex i $(1 \leq i \leq n+1)$.

2-optimality. We start by considering all partial paths along the tour, and then construct the tours that can be obtained by a 2-exchange.

(1) For each path $(i, i+1, \ldots, j-1, j)$, we define $first_{(i,i+1,\ldots,j-1,j)}$ as the smallest numbered vertex v for which there exists a precedence relation $k \rightarrow v$, where k is a vertex on the path. This path from vertex i to vertex j can be reversed and still satisfies the precedence constraints if and only if $first_{(i,i+1,\ldots,j-1,j)} > j$.

The computation of the values $first_{(i,i+1,\ldots,j-1,j)}$ is done in two steps. First we look at paths of one vertex and then we consider longer ones. This leads to the following computations:

par $[1 \leq i \leq n+1]$ $first_{(i)} \leftarrow \min\{v | i \rightarrow v, 1 \leq v \leq n+1\}$;
par $[1 \leq i \leq n+1]$ **par** $[i \leq j \leq n+1]$ $first_{(i,i+1,\ldots,j-1,j)} \leftarrow \min\{first_{(k)} | i \leq k \leq j\}$.

The first step requires $O(\log n)$ time with $O(n^2/\log n)$ processors, using the first phase of the partial sums algorithm, with addition replaced by taking the minimum, for all vertices in parallel. The second step also takes $O(\log n)$ time and $O(n^2/\log n)$ processors by applying both phases of the partial sums algorithm, with addition replaced by taking the minimum, again for all vertices in parallel.

(2) Since a tour that arises from a feasible tour after performing a 2-exchange is feasible if and only if the path that is reversed remains feasible, we can verify 2-optimality of the given tour by:

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

For this phase, we also need $O(\log n)$ time and $O(n^2/\log n)$ processors. Hence, for the complete process of verification of 2-optimality of a given tour the presence of precedence relations does not affect the time and processor requirements, and the total computational effort is the same as in the sequential case.

Or-optimality. The algorithm for the verification of Or-optimality proceeds along the same lines as above. Again, we have two phases.

(1) For each vertex i $(1 \le i \le n+1)$, we define $first_{(i)}$ as the smallest numbered vertex v for which there exists a precedence relation $i \to v$, and we define $last_{(i)}$ as the highest numbered vertex v for which there exists a precedence relation $v \to i$.

par $[1 \le i \le n+1]$ $first_{(i)} \leftarrow \min\{v | i \to v, 1 \le v \le n+1\}$;
par $[1 \le i \le n+1]$ $last_{(i)} \leftarrow \max\{v | v \to i, 1 \le v \le n+1\}$.

(2) Verification of optimality with respect to Or-exchanges is established by:

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

As in the case of 2-optimality, we can perform the computation in $O(\log n)$ time with $O(n^2/\log n)$ processors, thus achieving a perfect speedup.

Collections and deliveries

We will now present a parallel algorithm for the verification of local optimality for the traveling salesman problem with collections and deliveries. We assume that customer i has a (nonzero) demand q_i, and interpret a positive value as a pick-up of goods and a negative one as a delivery. Further, let the maximum capacity of the vehicle be Q and let the given tour $(1, 2, \ldots, n, n+1)$ be feasible. We consider the case of 2-optimality.

In the sequential algorithm, we introduced the quantities $L_p^{\max}$, $L_p^{\min}$ and L_p^{final} as respectively the maximum load, the minimum load and the final load on the path p assuming the vehicle is empty when it arrives at the first vertex on the path. We will first compute these values for all partial paths along the given tour and then for the tours that can be obtained by a 2-exchange.

(1) We start with the paths along the given tour in both directions.

par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\text{final}}_{(i,i+1,\ldots,j-1,j)} \leftarrow \text{sum}\{q_k | i \le k \le j\}$;
par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\text{final}}_{(j,j-1,\ldots,i+1,i)} \leftarrow \text{sum}\{q_k | j \ge k \ge i\}$;

par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\max}_{(i,i+1,\ldots,j-1,j)} \leftarrow \max\{L^{\text{final}}_{(i,i+1,\ldots,k-1,k)} | i \le k \le j\}$;
par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\max}_{(j,j-1,\ldots,i+1,i)} \leftarrow \max\{L^{\text{final}}_{(j,j-1,\ldots,k+1,k)} | j \ge k \ge i\}$;

par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\min}_{(i,i+1,\ldots,j-1,j)} \leftarrow \min\{L^{\text{final}}_{(i,i+1,\ldots,k-1,k)} | i \le k \le j\}$;
par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $L^{\min}_{(j,j-1,\ldots,i+1,i)} \leftarrow \min\{L^{\text{final}}_{(j,j-1,\ldots,k+1,k)} | j \ge k \ge i\}$.

Using the partial sums algorithm, we can compute all values in $O(\log n)$ time with $O(n^2/\log n)$ processors.

(2) The next step is to construct the tours that can be obtained from a 2-exchange and to compute the maximum and minimum load on these tours, using the proposition of the previous section.

par $[1 \le i < j \le n]$ $L^{\text{final}}_{(1,\ldots,i,j,\ldots,i+1)} \leftarrow L^{\text{final}}_{(1,\ldots,i)} + L^{\text{final}}_{(j,\ldots,i+1)}$;
par $[1 \le i < j \le n]$ $L^{\max}_{(1,\ldots,i,j,\ldots,i+1)} \leftarrow \max\{L^{\max}_{(1,\ldots,i)}, L^{\text{final}}_{(1,\ldots,i)} + L^{\max}_{(j,\ldots,i+1)}\}$;
par $[1 \le i < j \le n]$ $L^{\min}_{(1,\ldots,i,j,\ldots,i+1)} \leftarrow \min\{L^{\min}_{(1,\ldots,i)}, L^{\text{final}}_{(1,\ldots,i)} + L^{\min}_{(j,\ldots,i+1)}\}$,

and

par $[1 \le i < j \le n]$ $L^{\text{final}}_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)} \leftarrow L^{\text{final}}_{(1,\ldots,i,j,\ldots,i+1)} + L^{\text{final}}_{(j+1,\ldots,n+1)}$;
par $[1 \le i < j \le n]$ $L^{\max}_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)} \leftarrow \max\{L^{\max}_{(1,\ldots,i,j,\ldots,i+1)}, L^{\text{final}}_{(1,\ldots,i,j,\ldots,i+1)} + L^{\max}_{(j+1,\ldots,n+1)}\}$;
par $[1 \le i < j \le n]$ $L^{\min}_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)} \leftarrow \min\{L^{\min}_{(1,\ldots,i,j,\ldots,i+1)}, L^{\text{final}}_{(1,\ldots,i,j,\ldots,i+1)} + L^{\min}_{(j+1,\ldots,n+1)}\}$.

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

(3) We decide whether or not a tour is 2-optimal by:

par $[1 \le i < j \le n]$ $\delta_{ij} \leftarrow d_{ij} + d_{i+1,j+1} - d_{i,i+1} - d_{j,j+1}$;
$\delta_{\min} \leftarrow \min\{\delta_{ij} | L^{\max}_{(1,\ldots,i,j+1,\ldots,i+1,j,\ldots,n+1)} \le Q, L^{\min}_{(1,\ldots,i,j+1,\ldots,i+1,j,\ldots,n+1)} \ge 0, 1 \le i < j \le n\}$;
if $\delta_{\min} \ge 0$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^*, with $i^* < j^*$ be such that $\delta_{i^*j^*} = \delta_{\min}$ and the corresponding 2-exchange is feasible, $(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ is a better feasible tour.

As a result, we obtain an algorithm that runs in $O(\log n)$ time with $O(n^2/\log n)$ processors.

The verification of Or-optimality can be done along similar lines. The main difference is that we do not have to consider reversed paths, and that we consider different intermediate points on the tours. It will be clear that the parallel algorithm for verification of Or-optimality has the same time and processor requirements as for verification of 2-optimality.

Time windows

Finally, we will 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. We do not deal with the verification of Or-optimality, because it can be done along the same lines with very much the same adaptations as in the previous case.

Again, we consider the tour $(1, 2, \ldots, n, n+1)$, which is assumed to be feasible, and start by considering all partial paths along the tour. This enables us to construct the tours that can be obtained by a 2-exchange. For each path $(u_1, \ldots, u_k)$, we define $A_{(u_1,\ldots,u_k)}(t)$ as the earliest possible arrival time at vertex u_k when traveling along the path from vertex u_1 to vertex u_k after arriving at vertex u_1 at time t. Note that $A_{(1,\ldots,n+1)}(e_1)$ is the arrival time at vertex $n+1$ of the given tour.

Our algorithm has three phases.

(1) First, we compute the functions A for paths consisting of only one edge.

$$\textbf{par } [1 \le i \ne j \le n+1]\ A_{(i,j)}(t) \leftarrow \begin{cases} \max\{e_i, t\} + d_{ij} & \text{for } t \le l_i, \\ \infty & \text{for } t > l_i. \end{cases}$$

Next, we compute the functions A for all paths along the tour in both directions by composition.

par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $A_{(i,i+1,\ldots,j-1,j)}(t) \leftarrow (A_{(j-1,j)} \circ \cdots \circ A_{(i,i+1)})(t)$;
par $[1 \le i \le n+1]$ **par** $[i \le j \le n+1]$ $A_{(j,j-1,\ldots,i+1,i)}(t) \leftarrow (A_{(i+1,i)} \circ \cdots \circ A_{(j,j-1)})(t)$.

By considering all possibilities, one can show that each of these functions has one of the three shapes shown in Figure 6. Composing functions is an associative operation. Hence, we can use the partial sums algorithm for obtaining the functions A 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 A in $O(\log n)$ time with $O(n^2/\log n)$

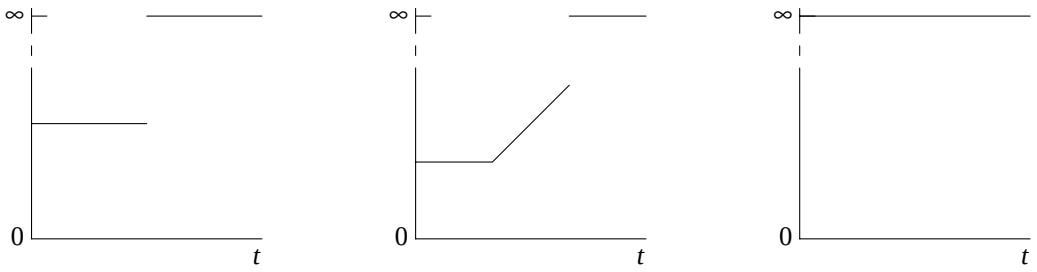


Figure 6. The three possible shapes of the functions A.

processors.

(2) Given the functions A for all paths along the tour, we can compute them for the tours that are obtained after the replacement of the edges $\{i, i+1\}$ and $\{j, j+1\}$ by the edges $\{i, j\}$ and $\{i+1, j+1\}$:

par $[1 \le i < j \le n]$ $A_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)}(t) \leftarrow (A_{(j+1,\ldots,n+1)} \circ A_{(i+1,j+1)} \circ A_{(j,\ldots,i+1)} \circ A_{(1,\ldots,i)})(t)$.

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

$A_{\min} \leftarrow \min\{A_{(1,\ldots,i,j,\ldots,i+1,j+1,\ldots,n+1)}(e_1) | 1 \le i < j \le n\}$;
if $A_{(1,\ldots,n+1)}(e_1) \le A_{\min}$
then $(1, 2, \ldots, n, n+1)$ is a 2-optimal tour
else let i^* and j^*, with $i^* < j^*$, be such that $A_{(1,\ldots,i^*,j^*,\ldots,i^*+1,j^*+1,\ldots,n+1} = A_{\min}$,
$(1, \ldots, i^*, j^*, j^*-1, \ldots, i^*+1, j^*+1, \ldots, n+1)$ 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 all the other cases.

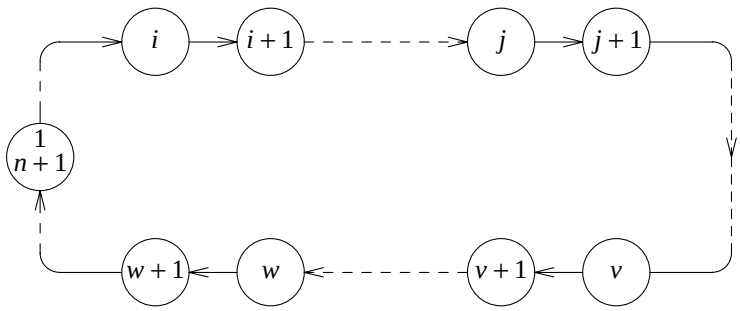
We will end this section by considering the verification of k-optimality for fixed $k > 2$. In all cases, we can derive a logarithmic-time algorithm along similar lines. One has to take into account that, given k edges, several k-exchanges are possible. Further, the influence of a k-exchange on a tour is more complex. However, 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.

2.5 Variable-depth exchanges

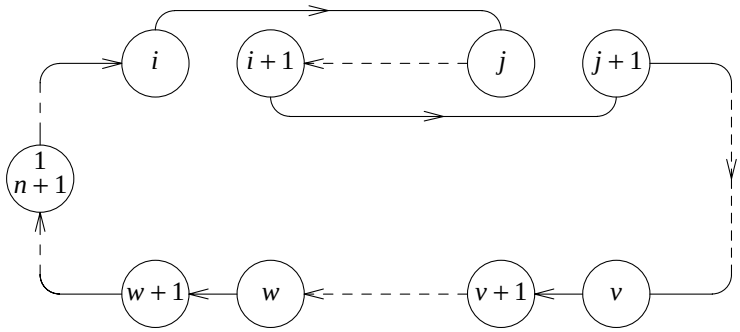
The neighborhoods considered so far are tours that can be obtained from an intial tour by replacing a set of edges of fixed cardinality by another set of edges. Not surprisingly, the quality of an exchange may improve considerably, when the neighborhood is enlarged by increasing the cardinality of the set of edges to be replaced. Unfortunately, the computational effort required to search the neighborhood becomes much greater as well. Lin & Kernighan [1973] developed an effective and efficient variable-depth exchange procedure for the unconstrained TSP by dynamically determining the cardinality of the set of edges to be replaced, thus finding a good balance between the computational effort and the quality of the solution.

The algorithm by Van der Bruggen, Lenstra & Schuur [1990] is based on the same principles and is able to handle the side constraints efficiently by using the lexicographical search strategy. It works as follows (cf. Figure 7).

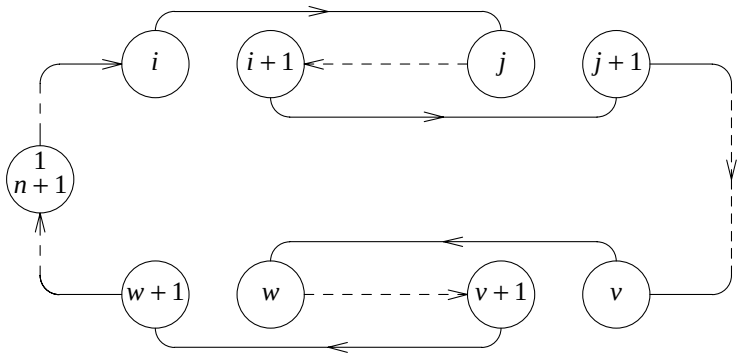
Apply the lexicographic search strategy until a feasible and profitable 2-exchange is found. Now, instead of actually performing the exchange, continue the search for a 2-exchange on the path that has not been affected by the exchange. Repeat this as long as the combined exchanges are profitable. In this way, in a single step, a series of exchanges is examined from which the best is selected. From the previous section it must be clear that this procedure can be efficiently implemented, both sequentially as in parallel.



(a) The initial tour.



(b) The tour after the first step.



(c) The tour after the second step.

Figure 7. A variable depth exchange.

There are many variants of this procedure. Any type of exchange, e.g, Or-exchange or 3-exchange, can be used as a basic improvement step in the procedure. Also, the best possible exchange can be chosen at each step, instead of the first feasible and profitable one encountered.

3. Edge-exchanges for the vehicle routing problem

After analyzing iterative improvement methods for the TSP, we now turn to the VRP. Two types of decisions have to be made to obtain a solution to a VRP: *assignment* decisions to determine which vehicle will serve which customers and *routing* decisions to determine in which order the customers assigned to a vehicle will be visited. The local search methods in the previous section try to improve the routing decisions. Another possibility is to improve the assignment decisions. Improving assignment decisions may even be more effective when we consider the fact that in most VRPs the number of customers per vehicle is fairly small, i.e., the resulting TSPs are fairly simple. It is uncommon to find VRPs where the number of customers per vehicle exceeds 30 customers. More often we are in a situation where the number of customers per vehicle is much less; in the order of 5 to 15. In that case, it is unrealistic to hope for major improvements when routing decisions are revised. However, the total number of customers is usually large. Therefore, there is much more potential for improvement when assignment decisions are revised. In view of the above, it is strange that there is very little known about effective and efficient local search methods that revise assignment decisions.

Savelsbergh [1992] describes three k-exchange neighborhoods for the VRP that relocate vertices between two routes. The neighborhoods are chosen such that testing optimality over the neighborhood requires $O(n^2)$ time. For presentational convenience, we will first describe relocations of single vertices. Possible extensions will be given later on. Further, we will use the notation pre_i and suc_i to denote the predecessor and successor of vertex i, and, in the figures, split the depot.

Relocation. The edges $\{pre_i, i\}$, $\{i, suc_i\}$ and $\{j, suc_j\}$ are replaced by $\{pre_i, suc_i\}$, $\{j, i\}$ and $\{i, suc_j\}$, i.e., vertex i from the origin route is placed in the destination route. A relocation is pictured in Figure 8.

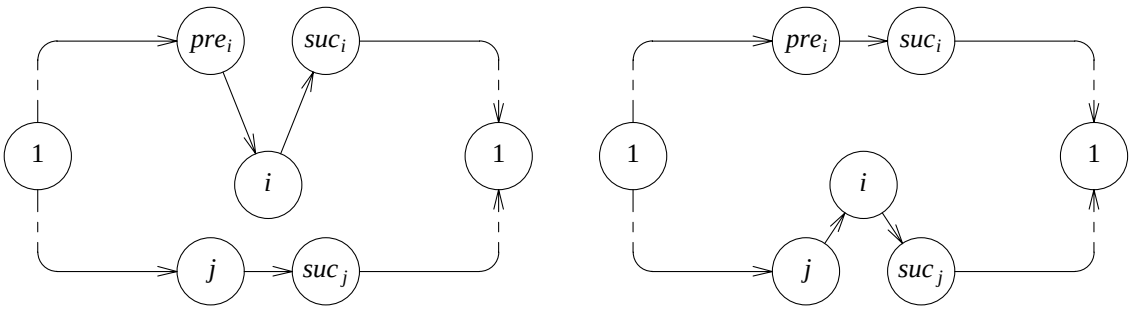


Figure 8. A relocation.

Exchange. The edges $\{pre_i, i\}$, $\{i, suc_i\}$, $\{pre_j, j\}$ and $\{j, suc_j\}$ are replaced by $\{pre_i, j\}$, $\{j, suc_i\}$, $\{pre_j, i\}$ and $\{i, suc_j\}$, i.e., two vertices from different routes are simultaneously placed into the other routes. An exchange is pictured in Figure 9.

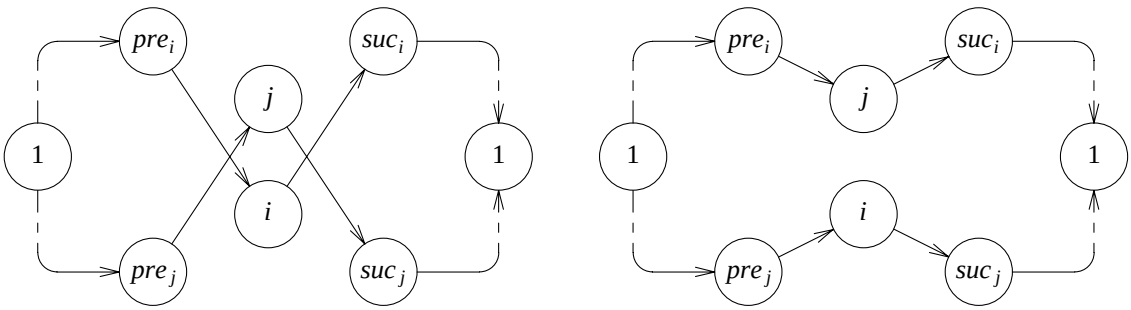


Figure 9. An exchange.

Crossover. The edges $\{i, suc_i\}$ and $\{j, suc_j\}$ are replaced by $\{i, suc_j\}$ and $\{j, suc_i\}$, i.e., the crossing links of two routes are removed. A crossover is pictured in Figure 10. As a special case, it can combine two routes into one. Note that if a crossover is actually performed, the last part of either route will become the last part of the other.

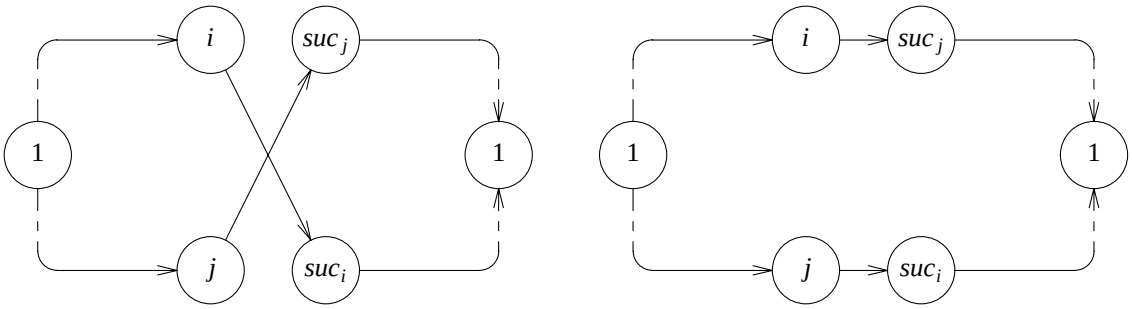


Figure 10. A crossover.

The extension of the lexicographic search strategy to these three neighborhoods is straightforward. We choose *i* in reverse order of the first route. After *i* has been fixed, we choose *j* in reverse order of the second route. To test the feasibility and profitability of an exchange, we proceed as before, the routes are split into paths and the concatenation propositions are applied to obtain the necessary information.

The above described iterative improvement methods can easily be extended to larger neighborhoods by the introduction of paths instead of vertices. The paths have to be checked but that involves only local information. Figure 11 illustrates some possible extensions.

References

H. Alt, T. Hagerup, K. Mehlhorn, F.P. Preparata (1987). Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM J. Comput. 16*, 808-835.

J.L. Bentley, D.S. Johnson (1993). A case study for the traveling salesman problem. E.H.L. Aarts, J.K. Lenstra (eds.). *Local Search in Combinatorial Optimization*, Wiley, Chichester, to appear.

L. Bodin, B. Golden, A. Assad, M. Ball (1983). Routing and scheduling of vehicles and crews - The state of the art. *Comput. Oper. Res. 10*, 63-211.

L.J.J. van der Bruggen, J.K. Lenstra, P.C. Schuur (1990). *A Variable Depth Approach for the Single-Vehicle Pickup and Delivery Problem with Time Windows*, COSOR Memorandum 90-48, Eindhoven University of Technology.

M. Desrochers, J.K. Lenstra, M.W.P. Savelsbergh, F. Soumis (1988). Vehicle routing with time windows: optimization and approximation. B.L. Golden, A.A. Assad (eds.). *Vehicle Routing: Methods and Studies*, North Holland, Amsterdam, 65-84.

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

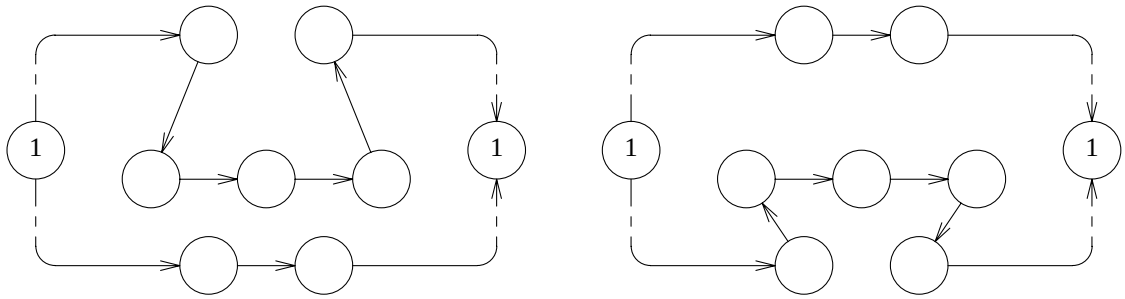
Y. Dumas, J. Desrosiers, F. Soumis (1991). The pickup and delivery problem with time windows. *European J. Oper. Res. 54*, 7-22.

A.R. Karlin, E. Upfal (1988). Parallel hashing - an efficient implementation of shared memory. *J. Assoc. Comput. Mach. 35*, 876-892.

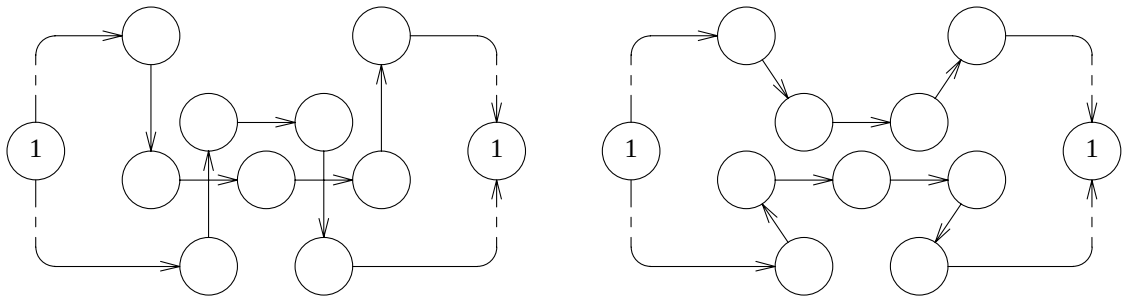
S. Lin, B.W. Kernighan (1973). An effective heuristic algorithm for the traveling salesman problem. *Oper. Res. 21*, 498-516.

I. Or (1976). *Traveling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Blood Banking*, Ph.D. Thesis, Dept. of Industrial Engineering and Management Sciences, Northwestern University.

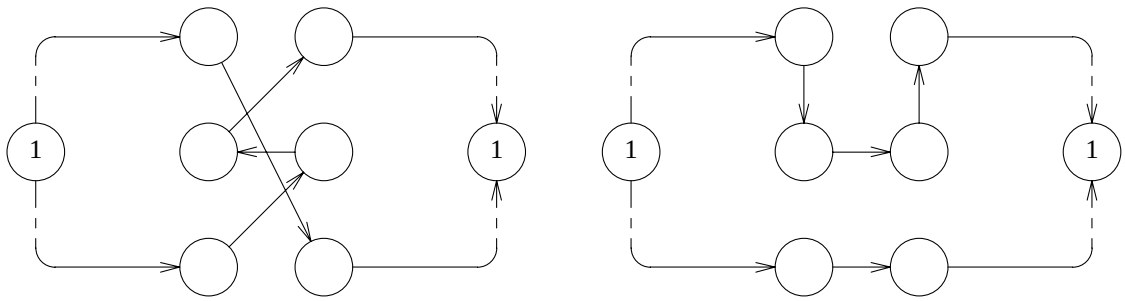
H.N. Psaraftis (1983). *k*-Interchange procedures for local search in a precedence-constrained routing



(a) Relocation of a path.



(b) Exchange of two paths.



(c) A crossover plus 2-exchange.

Figure 11. Possible extensions of the neighborhoods.

problem. *European J. Oper. Res. 13*, 391-402.

R.A. RUSSELL (1977). An effective heuristic for the *m*-tour traveling salesman problem with some side constraints. *Oper. Res. 25*, 517-524.

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

M.W.P. SAVELSBERGH (1990). Efficient implementation of local search methods for the vehicle routing problems with side constraints. *European J. of Oper. Res. 47*, 75-85.

M.W.P. SAVELSBERGH (1992). The vehicle routing problem with time windows: minimizing route duration. *ORSA J. on Computing 4*, 146-154.

M.M. SOLOMON, E.K. BAKER, J.R. SCHAFFER (1988). Vehicle routing and scheduling problems with time window constraints: efficient implementations of solution improvement procedures. B.L. GOLDEN, A.A. ASSAD (eds.). *Vehicle Routing: Methods and Studies*, North-Holland, Amsterdam, 85-105.