HAS-SOP:
HYBRID ANT SYSTEM
FOR THE
SEQUENTIAL ORDERING PROBLEM

Luca Maria Gambardella and Marco Dorigo

Technical Report IDSIA 11-97

Luca Maria Gambardella
IDSIA, Corso Elvezia 36, 6900 Lugano, Switzerland
Tel +41 91 9119838
Fax +41 91 9119839
email: luca@idsia.ch.
http://www.idsia.ch/~luca

Marco Dorigo
IRIDIA, Université Libre de Bruxelles, CP 194/6, Avenue Franklin Roosevelt 50
1050 Bruxelles, Belgium
mdorigo@ulb.ac.be
http://iridia.ulb.ac.be/dorigo/dorigo.html

Abstract

We present HAS-SOP, a new approach to solving sequential ordering problems. HAS-SOP combines the ant colony algorithm, a population-based metaheuristic, with a new local optimizer, an extension of a TSP heuristic which directly handles multiple constraints without increasing computational complexity. We compare different implementations of HAS-SOP and present a new data structure that improves system performance. Experimental results on a set of twenty-three test problems taken from the TSPLIB show that HAS-SOP outperforms existing methods both in terms of solution quality and computation time. Moreover, HAS-SOP improves most of the best known results for the considered problems.

Key words: ant colony optimization, metaheuristic, sequential ordering problem.

1. Introduction

There are many NP-hard combinatorial optimization problems for which it is impractical to find an optimal solution. Among them is the sequential ordering problem (SOP). For such problems the only reasonable enterprise is to look for heuristic algorithms which quickly produce good, although not necessarily optimal, solutions. These algorithms often use some problem specific knowledge to either build or improve solutions. Recently, many researchers have focused their attention on a new class of algorithms called metaheuristics. Metaheuristics are rather general algorithmic frameworks which can be applied to several different optimization problems with few modifications. Examples of metaheuristics are simulated annealing, evolutionary computation and tabu search. Metaheuristics are often inspired by natural processes. In fact, the above cited metaheuristics were inspired respectively by the physical annealing process, the Darwinian evolutionary process and the clever management of memory structures. One of the most recent nature-inspired metaheuristics is ant colony optimization (Dorigo, Maniezzo and Colorni, 1991; 1996; Dorigo, 1992). There, the inspiring natural process is the foraging behavior of ants. Ant colony system (ACS), a particular instance of ant colony optimization, has recently been shown (Dorigo and Gambardella, 1997) to be competitive with other metaheuristics on the symmetric and asymmetric traveling salesman problems. Although this is an interesting and promising result, it remains clear that ant colony optimization, as well as other metaheuristics, in many cases cannot compete with specialized local search methods. A current trend (Johnson and McGeoch, 1997) is therefore to associate with the metaheuristic a local optimizer. This is a very interesting marriage since local optimizers often suffer from an "initialization" problem. That is, the performance of a local optimizer is often a function of the type of initial solution to which it is applied. For example, multistart, that is, the application of local search to different initial solutions, has been found to be a poor choice, since the local search procedure spends most of its time improving the initial low quality solution (Aarts and Lenstra, 1997). It therefore becomes interesting to find good metaheuristic-local optimizer couplings, where a coupling is good if the metaheuristic generates initial solutions that can be carried to very good local optima by the local optimizer.

In previous work (Dorigo and Gambardella, 1997) we have shown that by coupling ACS with an extended version of the 3-opt local search procedure it is possible to obtain high-quality solutions for both symmetric and asymmetric TSPs. Also, we have recently applied HAS-QAP, an ant colony algorithm coupled with a simple form of local search, to the quadratic assignment problem (QAP). HAS-QAP produced better solutions (Gambardella, Taillard and Dorigo, 1997) than the best known algorithms on structured, real-world problem instances (comparison includes reactive tabu search (Battiti and Tecchioli, 1994), robust tabu search (Taillard, 1991), simulated annealing (Connolly, 1990), and genetic hybrid search (Fleurent and Ferland, 1994)).

In this paper we attack the sequential ordering problem (SOP) by an ant colony algorithm coupled with a modified version of the 3-opt search procedure. The resulting hybrid ant system for the SOP (HAS-SOP) outperforms all known heuristic approaches to the SOP. Also, we have been able to improve many of the best results published in TSPLIB[1], one of the most important databases of difficult TSP-related optimization problems available on the Internet.

---

[1] TSPLIB: http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html (maintained by G. Reinelt).

1.1 The sequential ordering problem

The sequential ordering problem with precedence constraints (SOP) was first formulated by Escudero (1988) to design heuristics for a production planning system. It consists of finding a minimum weight Hamiltonian path on a directed graph with weights on the arcs and the nodes, subject to precedence constraints among nodes.

Consider a complete graph G=(V,A) with node set V and arc set A, where nodes correspond to jobs 0, ..., *i*, ..., *n* (n+1 = |V|). To each arc (*i*,*j*) is associated a cost $t_{ij} \in \Re$ with $t_{ij} \geq 0$. this cost represent the required waiting time between the end of job *i* and the beginning of job *j*. To each node *i* is associated a cost $p_i \in \Re$, $p_i \geq 0$, which represents the processing time of job *i*. The set of nodes V includes a starting node (node 0) and a final node (node *n*) connected with all the other nodes. The costs between node 0 and the other nodes is equal to the setup time of node *i*, $t_{0i}=p_i \; \forall i$, and $t_{in}=0 \; \forall i$.

Precedence constraints are given by an additional acyclic digraph P=(V,R) that is defined on the same node set V. An arc $(i,j) \in R$ if job *i* has to precede job *j* in any feasible solution. R has the transitive property (that is, if $(i,j) \in R$ and $(j,k) \in R$ then $(i,k) \in R$). Since a sequence always starts at node 0 and ends at node *n*, $(0,i) \in R \; \forall i \in V \backslash \{0\}$, and $(i,n) \in R \; \forall i \in V \backslash \{n\}$. In the following we will indicate with *predecessor[i]* and *successor[i]* the sets of nodes that have to precede/succeed node *i* in any feasible solution.

Given the above definitions, the SOP problem can be stated as the problem of finding a job sequence subject to the precedence constraints which minimizes the total makespan. This is therefore equivalent to the problem of finding a feasible Hamiltonian path with minimal cost in G under precedence constraints given by P.

SOP can also be formulated as a general case of the asymmetric traveling salesman problem (ATSP) by giving only the weights on the edges (in SOP a solution connects the first and the last city by a path that visits all nodes once, as opposed to the ATSP in which a solution is a closed tour which visits all nodes once). This formulation is equivalent to the previous: it suffices to remove weights from nodes and to redefine the weight c_{ij} of arc (*i*,*j*) by adding to each t_{ij} the weight p_j of node *j*. In this representation c_{ij} is an arc weight (where c_{ij} may be different from c_{ji}) which can either represent the cost of arc (*i*, *j*) when $c_{ij} \geq 0$, or an ordering constraint when $c_{ij}=-1$ ($c_{ij}=-1$ means that element *j* must precede, not necessarily immediately, element *i*). In this paper we will use this last formulation.

1.2 Heuristic methods for the SOP

The SOP models real-world problems like production planning (Escudero, 1988), single-vehicle routing problems with pick-up and delivery constraints (Pulleyblank and Timlin, 1991; Savelsbergh, 1990), and transportation problems in flexible manufacturing systems (Ascheuer, 1996).

The SOP can be seen as a general case of both the asymmetric TSP and the pick-up and delivery problem. It differs from ATSP because the first and the last cities are fixed, and in the additional set of precedence constraints on the order in which cities must be visited. It differs from the pick-up and delivery problem because this is usually based on symmetric TSPs, and because the pick-up and delivery problem includes a set of constraints between cities with a unique predecessor defined for each city, in contrast to SOP where multiple precedences can be defined.

Approaches based on the ATSP. SOP problems were initially solved as constrained versions of the asymmetric TSP. The main effort has been in extending the mathematical definition of ATSP problems by introducing new equations to model constraints. The first mathematical model for SOP problems was introduced in (Ascheuer et al., 1993) where a cutting-plane approach was proposed to compute lower bounds on the optimal solution. In (Escudero et al., 1994) a Lagrangian relax-and-cut method was described and new valid cuts

to obtain strong lower bounds were defined. More recently, based on the polyhedral investigation carried out on ATSP problems with precedence constraints by (Balas, Fischetti and Pulleyblank, 1995), Ascheuer (1996) has proposed a new class of valid inequalities and has described a branch-and-cut algorithm for a broad class of SOP instances. His approach also investigates the possibility to compute and improve sub-optimal feasible problem solutions starting from the upper bound computed by the polyhedral investigation. The upper bound is the initial solution of an heuristic phase based on well-known ATSP heuristics that are iteratively applied in order to improve feasible solutions. These heuristics do not handle constraints directly: resulting unfeasible solutions are simply rejected. With this approach Ascheuer was able to compute new upper bounds for the SOP problems in TSPLIB, although a genetic algorithm called Maximum Partial Order/Arbitrary Insertion (MPO/AI) recently proposed by Chen and Smith (1996) seems to work better on the same class of problems. MPO/AI always works in the space of feasible solutions by introducing a sophisticated crossover operator that preserves the common schema of two parents by identifying their maximum partial order through matrix operations. The new solution is completed using a constructive heuristic.

Approaches based on the pick-up and delivery problem. Heuristic approaches to pick-up and delivery problems are based on particular extensions of TSP heuristics able to handle precedence constraints while improving feasible solutions without any increase in computation times. Psaraftis (1983) has introduced a preprocessing technique to ensure feasibility checking in constant time by starting the algorithm with a screening procedure which, at an initial cost of $O(n^2)$, produces a feasibility matrix that contains information about feasible edge exchanges. Subsequently (Solomon, 1987) proposed a search procedure based on a tailored updating mechanism, while (Savelsbergh, 1990; Van der Bruggen, Lenstra and Schuur, 1993; Kindervater and Savelsberg, 1997) presented a lexicographic search strategy, a variation of traditional edge-exchange TSP heuristics, that reduces the number of visited nodes without loosing any feasible exchange. In order to ensure constraints checking in constant time, the lexicographic search strategy has been combined with a labeling procedure where nodes in the sequence are labeled with information related to their unique predecessor/successor, and a set of global variables are updated to keep this information valid. Savelsbergh (1990) presented a lexicographic search based on 2-opt and 3-opt strategies that exchanges a fixed number of edges, while Van der Bruggen et al. (1993) proposed a variable-depth search based on the Lin and Kernighan (1973) approach. Unfortunately, this labeling procedure is not applicable in case of multiple precedence constraints because it is based on the presence in the sequence of nodes with a unique predecessor/successor. On the other hand, the lexicographic search strategy itself is independent of the number of precedence constraints and can therefore be used to solve SOP problems in which multiple precedence constraints are allowed.

The approach to SOP problems presented in this paper is the first in the literature that uses an extension of a TSP heuristic to handle multiple constraints directly without any increase in computational time. Our approach combines a constructive phase based on the ACS algorithm (Dorigo and Gambardella, 1997) with the lexicographic search heuristic and with a new labeling procedure able to handle multiple precedence constraints. In addition, we test and compare different methods to select nodes during the search and different stopping criteria. In particular we test two different selection heuristics: one based on the *don't look bit* data structure introduced by Bentley (1992), the other based on a new data structure called *don't push stack* introduced by the authors.

2. Ant colony optimization

The ant colony optimization metaheuristic is a population-based approach in which a set of agents (artificial ants) build solutions by a probabilistic nearest neighbor algorithm which uses a special distance measure. Once they have built a solution, artificial ants use the quality of the generated solutions to update the distance information. Let us consider, for presentation purposes, the Euclidean TSP. Each edge of the graph has two associated measures: the closeness η_{ij}, and the pheromone trail τ_{ij}. Closeness, defined as the inverse of the edge length, is a static value, that is, it never changes for a given problem instance, while pheromone trail is a dynamic value changed at runtime by ants. Each ant has a starting city and its goal is to build a solution, that is, a complete tour. A tour is built city by city (cities are the vertices of the graph): when ant k is in city i it chooses to move to city j using a probabilistic rule which favors cities that are close and connected by edges with a high pheromone trail value. Cities are always chosen among those not yet visited in order to enforce the construction of feasible solutions. Once all ants have built a complete tour, pheromone trail is updated on visited edges. The guiding principle is that of increasing pheromone trail on those edges visited by those ants that found a short tour. Pheromone trail also evaporates so that memory of the past is gradually lost (this prevents bad initial choices will from having a lasting effect on the search process). The above informally described algorithm (see also Figure 1) can be implemented in many different ways and details about specific implementation choices for the TSP can be found in (Dorigo, Maniezzo and Colorni, 1991; 1996; Dorigo, 1992; Dorigo and Gambardella, 1997).

It is clear that the ant colony optimization metaheuristic can easily be adapted to the SOP: it is sufficient to have ants build a path from source to destination while respecting the ordering constraints (this can easily be achieved by having ants choose not yet visited cities which do not violate any ordering precedence).

```
1) Initialize the pheromone trail
2) For I^max iterations repeat:
   2a) For each ant k = 1, ..., m, build a new solution exploiting
       the pheromone trail
   2b) Update the pheromone trail
```

Figure 1. A generic ant algorithm

The most important component of ant colony optimization is the management of pheromone trails which are used, in conjunction with the objective function, for constructing new solutions. The information contained in the pheromone trails and the use of this information are the key elements of an ant system. Informally, pheromone levels give a measure of how desirable it is to insert a given element in a solution. Pheromone trails are used for exploration and exploitation. Exploration concerns the probabilistic choice of the components used to construct a solution: a higher probability is given to elements with a strong pheromone trail. Exploitation chooses the component that maximizes a blend of pheromone trail values and partial objective function evaluations.

3. The hybrid ant system

Hybrid ant system for the SOP (HAS-SOP) is an ant colony algorithm as described above plus local search. HAS-SOP builds a set of solutions by the ant colony algorithm and then the local search procedure takes them to a local optimum. These locally optimal solutions are used to calculate the amount of change in pheromone trails. Pheromone trails are then updated and HAS-SOP is reiterated. This continues until a termination condition is met. The termination condition is met when one of the following conditions becomes true: a fixed

number of solutions has been generated, a fixed CPU time has elapsed, or no improvement has been achieved during the last NO_IMP iterations.

In what follows we will give a detailed description of both the ant algorithm and the local search procedure.

3.1 The ant algorithm

The ant algorithm implements the constructive phase of HAS-SOP, whose goal is to build a feasible solution to the SOP problem. It generates feasible solutions with a computational cost of order $O(n^2)$.

Informally, it works as follows. Each ant iteratively starts from node 0 and adds new nodes until all nodes have been visited and node *n* is reached. When in node *i*, an ant chooses probabilistically the next node *j* from the set $F(i)$ of feasible nodes. $F(i)$ contains all the nodes *j* still to be visited and such that all nodes that have to precede *j*, according to precedence constraints, have already been inserted in the sequence.

The system chooses with probability q_0 the node with the best $\tau_{ij} \cdot \eta_{ij}$, $j \in F(i)$ (deterministic rule), while with probability $(1-q_0)$ the node is chosen according to a probabilistic rule which favors nodes connected by edges with higher values of $\tau_{ij} \cdot \eta_{ij}$, $j \in F(i)$.

$$j = \begin{cases} \max_{j \in F(i)} \left(\tau_{ij} \cdot \eta_{ij} \right) & \text{with probability } q_0 \\ p_{ij}{}_{j \in F(i)} = \dfrac{\tau_{ij} \cdot \eta_{ij}}{\sum_{l \in F(i)} \tau_{il} \cdot \eta_{il}} & \text{with probability } (1 - q_0) \end{cases} \quad (1)$$

The value q_0 is given by

$$q_0 = 1 - \frac{s}{n}$$

Q_0 is based on a parameter *s* that is the number of nodes we would like to choose using the probabilistic rule in formula (1). *S* allows the system to define q_0 independently of the problem size such that the expected number of nodes selected with the probabilistic rule is *s* (*s* is fixed to be 10 in our experiments).

Once each ant has built a feasible solution, this is carried to a local optimum by an application of the local search routine described in Section 4. Locally optimal solutions are used to update pheromone trails on arcs, according to a pheromone trail update rule. In HAS-SOP only the best ant, that is the ant which built the shortest tour, is allowed to deposit pheromone trail. The rationale is that in this way a "preferred route" is memorized in the pheromone trail matrix and future ants will use this information to generate new solutions in a neighborhood of this preferred route. The formula used is:

$$\tau_{ij} = (1-a) \cdot \tau_{ij} + a / L_{best} \quad (2)$$

where L_{best} is the length of the path built by the best ant, that is, the length of the shortest path generated since the beginning of the computation.

Pheromone trail is also updated during solution building. In this case, however, it is removed from visited edges. In other words, each ant when moving from city *i* to city *j* applies a pheromone updating rule which causes the amount of pheromone trail on edge (*i*,*j*) to decrease. The rule is:

$$\tau_{ij} = (1-a) \cdot \tau_{ij} + a \cdot \tau_0 \quad (3)$$

where τ_0 is the initial value of trails. We found that a good value for this parameter is

$$\tau_0 = (FirstSolution \cdot n)^{-1}$$

The rationale for using formula (3) is that ants eat away pheromone trail while they build solutions so that a certain variety in generated solutions is assured (if pheromone trail was not consumed by ants they would tend to generate very similar tours).

Below we give a commented Pascal-like description of the algorithm; the local optimization routine is described in the next section.

The HAS-SOP algorithm

1. */* Initialization phase */*
   For each pair (r,s) $\tau(r,s) := \tau_0$ **End-for**
2. */* First step */*
   For k:=1 **to** m **do**
   Let r_k be the city where agent k is located
   $r_k \leftarrow 0$ */* All ants start from city 0 */*
   End-for
3. */* This is the step in which agents build their Hamiltonian paths. The path of agent k is stored in $Path_k$. */*
   For k:=1 **to** m **do**
   For i:=1 **to** n-1 **do**
   Starting from r_k compute the set $F(r_k)$ of feasible cities
   / $F(r_k)$ contains all the nodes j still to be visited and such that all nodes that have to precede j have already been inserted in the sequence */*
   Choose the next city s_k according to formula (1)
   $Path_k(i) \leftarrow (r_k, s_k)$
   $\tau(r_k,s_k) \leftarrow (1-\alpha)\cdot\tau(r_k,s_k)+\alpha\cdot\tau_0$ */* This is formula (**3**) */*
   $r_k \leftarrow s_k$ */* New city for agent k */*
   End-for
   End-for
4. */* In this step the local optimizer is applied to the solutions built by each ant */*
   For k:=1 **to** m **do**
   $Optmized\_Path_k \leftarrow local\_opt\_routine(Path_k)$
   End-for
5. */* In this step pheromone trails are updated using formula (**2**) */*
   For k:=1 **to** m **do**
   Compute L_k */*L_k is the length of the path $Optmized\_Path_k$ */*
   End-for
   Let L_{best} be the shortest L_k from beginning and $Optmized\_Path_{best}$ the corresponding path
   For each edge $(r,s) \in Optmized\_Path_{best}$
   Update pheromone trail applying formula (**2**)
   End-for
6. **If** (End_condition = True)
   then Print L_{best} and $Optmized\_Path_{best}$
   else goto Step 2
   end-if

4. Local search for SOP problems

4.1 Edge-exchange heuristics

In recent years much research went into defining ad-hoc TSP heuristics, see (Johnson and McGeoch, 1997) for an overview. They can be classified as tour-constructive heuristics and tour-improvement heuristics (the latter are also called local optimization heuristics). Tour-constructive heuristics, see Bentley (1992) for an overview, usually start selecting a random city from the set of cities and then incrementally build a feasible TSP solution by adding new cities chosen according to some heuristic rule. For example, the nearest neighbor heuristic builds a tour by adding the closest city in terms of distance to the last city inserted in the path. On the other hand, tour improvement heuristics start from a given tour and attempt to reduce its length by exchanging edges chosen according to some heuristic rule until a local optimum is found (i.e., until no further improvement is possible using the heuristic rule). It

has been experimentally shown (Reinelt, 1994) that, in general, tour improvement heuristics produce better quality results than tour-constructive heuristics. Still, a tour constructive heuristic is necessary at least to build the initial solution for the tour improvement heuristic.

Starting from an initial solution, an edge-exchange procedure generates a new solution by replacing *k* edges with another set of *k* edges. This operation is usually called a *k-exchange* and is iteratively executed until no additional improving *k-exchange* is possible. When this is the case the final solution is said to be *k-optimal*; the verification of *k-optimality* requires $O(n^k)$ time. For a *k-exchange* procedure to be efficient it is necessary that the improving criterion for new solutions can be computed in constant time.

It has been shown that increasing *k* produces solutions of increasing quality but the computational effort to completely test the *k-exchange* set for a given solution usually restricts our attention to *k-exchange* with $k \leq 3$. The most widely used edge-exchange procedures set *k* to 2 or 3 (2-opt and 3-opt edge-exchange procedures (Lin, 1965)), or to a variable value (Lin and Kernighan, 1973) in which case a variable-depth edge-exchange search is performed.

In this section we first make some considerations about edge-exchange techniques for TSP/ATSP problems. Then, we concentrate our attention on *path-preserving-edge-exchanges* for ATSPs, that is, edge-exchanges that do not invert the order in which paths are visited. Next, we discuss *lexicographic-path-preserving-edge-exchange*, a path-preserving-edge-exchange procedure which searches only in the space of feasible exchanges. We then add to the lexicographic-path-preserving-edge-exchange a labeling procedure whose function is to check feasibility in constant time. Finally, we present different possible strategies to select nodes during the search, as well as different search stopping criteria.

4.2 Path-preserving edge-exchange heuristics

A *k-exchange* deletes *k* edges from the initial solution creating *k* disjointed paths that are reconnected with *k* new edges. In some situations this operation requires to invert the order in which nodes are visited inside one of the paths (*path-inverting-edge-exchange*), while in other situations this inversion is not required (*path-preserving-edge-exchange*).

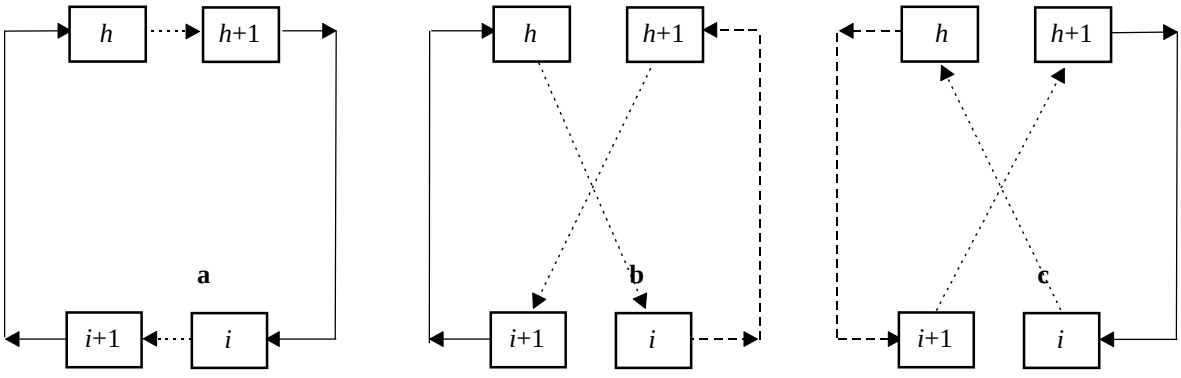

Figure 2. A 2-exchange always inverts a path

Consider a *2-exchange* (Figure 2) where two edges to be removed $\{h,h+1\}$ and $\{i,i+1\}$ have been selected. In this situation there are only two ways to perform the exchange: in the first case (Figure 2b) edges $\{h,i\}$ and $\{h+1,i+1\}$ are inserted and the traveling direction for path $(i, ..., h+1)$ is inverted; in the second case (Figure 2c) edges $\{i,h\}$ and $\{i+1,h+1\}$ are inserted inverting the traveling direction for path $(h, ..., i+1)$.

In case of a *3-exchange*, however there are several possibilities to build a new solution when edges $\{h,h+1\}$, $\{i,i+1\}$, and $\{j,j+1\}$ are selected to be removed (Figure 3). In Figure 3 a path preserving *3-exchange* (Figure 3b) and a path inverting *3-exchange* (Figure 3c) are shown.

It is then clear that any *2-exchange* procedure determines the inversion of one of the involved paths, while for k=3 this inversion is caused only by particular choices of the inserted edges.

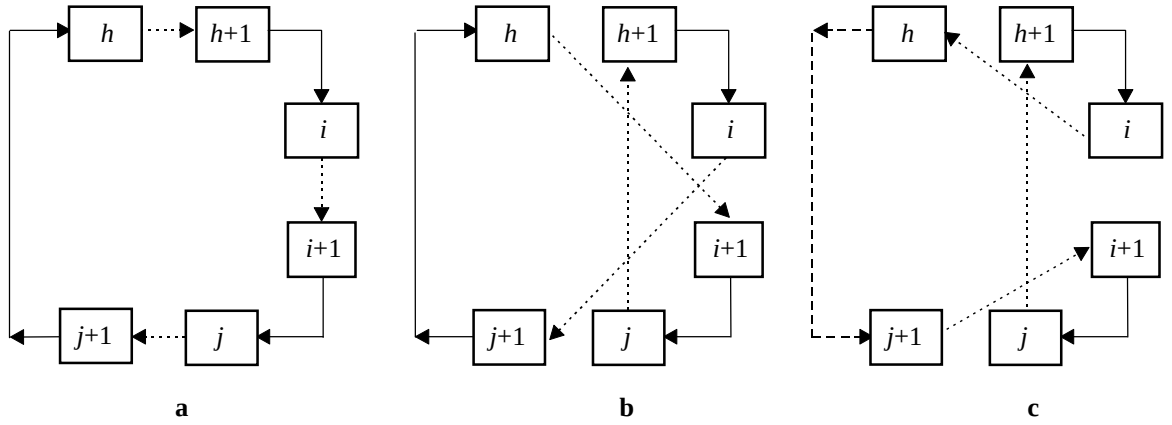

Figure 3. A 3-exchange without (b) and with (c) path inversion

In case of TSP problems, where costs $\eta_{ij}=\eta_{ji}$ $\forall(i,j)$, inverting a path does not modify its length. Therefore, the quality of the new solution only depends on the length of the inserted and deleted edges. On the other hand, for ATSP problems, where $\eta_{ij} \neq \eta_{ji}$ for at least one (i,j), inverting a path can modify the length of the path itself, and therefore the length of the new solution does not depend only on the inserted and deleted edges. This situation contrasts with the requirement that the improving criterion be checkable in constant time. Therefore, the only suitable *edge-exchange* procedures for SOP problems, which are a constrained version of ATSP problems, are *path-preserving-edge-exchange* heuristics. In the following, we concentrate on *path-preserving-k-exchange*, *pp-k-exchange* for short, with k=3, that is, the smallest k that allows a path preserving edge exchange.

4.3 Handling precedence constraints

Starting from a feasible SOP sequence H, a *pp-3-exchange* tries to reduce the length of H by replacing edges $\{h,h+1\},\{i,i+1\}$ and $\{j,j+1\}$ with edges $\{h,i+1\}$, $\{i,j+1\}$and $\{j,h+1\}$ (see Figure 4a).

The result of a *pp-3-exchange* is a new sequence H_1 (Figure 4b) where, while walking from node 0 to node n, the order we visit *path_left*=(h+1, ..., i) and *path_right*=(i+1, ..., j) is inverted.

In this situation, the new sequence H_1 is feasible only if in the initial solution H there are no precedence constraints between a generic node $l \in$*path_left* and a generic node $r \in$*path_right*.

Given two generic paths *path_left* and *path_right*, to test the feasibility of the *pp-3-exchange* requires a computational effort of order $O(n^2)$ (precedence constraints must be checked between each couple of nodes in the two paths).

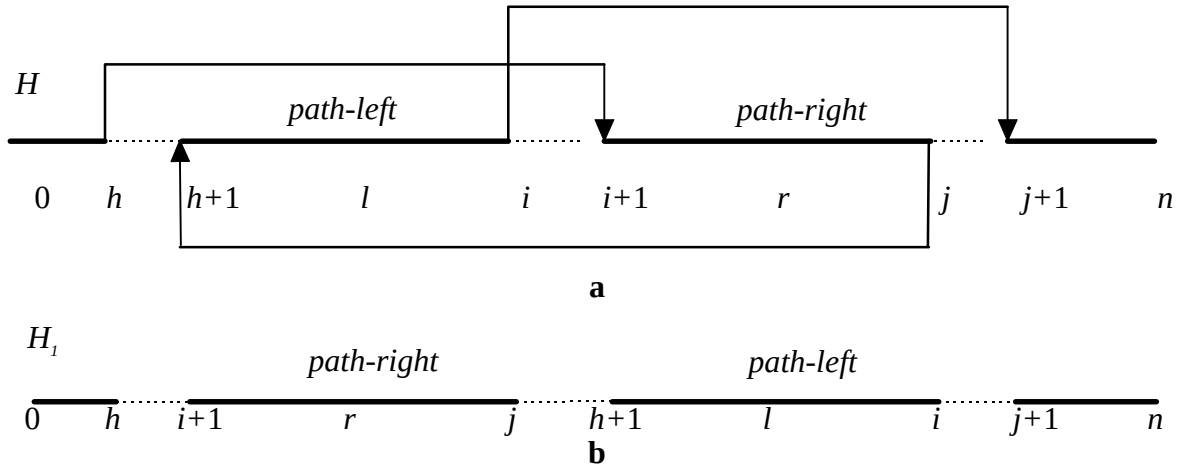

Figure 4. A path-preserving-3-exchange.

How to limit the computational effort needed for checking solution feasibility in case of precedence constraints has been studied for dial-a-ride problems by Savelsberg (1990), who introduced a particular exploration strategy called *lexicographic search strategy* that allows for generating and exploring only feasible exchanges. Savelsberg presents a combination of the mentioned lexicographic search strategy with a *labeling procedure* where a set of global variables is updated so that precedence constraints checking can be performed in constant time.

The lexicographic search strategy was introduced to solve dial-a-ride problems where only one precedence constraint for each node is allowed. Nevertheless, it is independent of the number of constraints. We have applied a version of Savelsberg's lexicographic search strategy restricted to the case k=3, *lpp-3-exchange,* to SOP problems with multiple constraints for each node.

Savelsberg's labeling procedure however was designed to handle unique precedence constraints under particular search conditions and cannot be extended to SOP problems. Before explaining our new labeling procedure for the SOP, we present the *lpp-3-exchange.*

4.4 Lexicographic search strategy in case of precedence constraints

The *lpp-3-exchange* procedure identifies two paths, *path_left* and *path_right*, which once inverted give raise to a new feasible solution. These two paths are initially composed of one single node and are incrementally expanded adding one node at each step. This feature makes it possible to test feasibility easily because precedence conditions must be checked only for the new added node.

We distinguish between forward and backward *lpp-3-exchange.* We talk of a forward *lpp-3-exchange* (*f-lpp-3-exchange*) when in the initial sequence $\pi(h)<\pi(i)<\pi(j)$ ($\pi(x)$ indicates the position of node x in the sequence), and of a backward *lpp-3-exchange* (*b-lpp-3-exchange*) when $\pi(j)<\pi(i)<\pi(h)$.

An *lpp-3-exchange* starts by setting node h=0 and then moves h through the sequence until node n–2 is reached. Each time node h is selected the search performs a forward and a backward lexicographic search for the same h.

The *f-lpp-3-exchange* procedure starts by setting the value of h; then sets the value of i, which identifies the rightmost node of *path_left* and performs a loop on the value of j, which identifies the rightmost node of *path_right* (Figures 5b and 5d): *path_right*=(i+1, ..., j) is iteratively expanded by the new edge {j,j+1}. Once all available nodes have been added to *path_right*, *path_left* is expanded by adding the new edge {i,i+1} (Figure 5c), and then

path_right is searched again. *Path_left* and *path_right* are initially composed of only one element $i=h+1$ and $j=i+1$ (Figure 5a).

In case of a *b-lpp-3-exchange* (Figure 6) *path_left* is identified by (j+1, ..., i) and *path_right* by (i+1, ..., h). After fixing h, we set $i=h-1$ and $j=i-1$ (Figure 6a) and we expand backward *path_left* moving j until the beginning of the sequence, iteratively setting j to the values $i-2$, $i-3$, ..., 0 (Figure 6b). Then, *path_right* is iteratively expanded in backward direction with the new edge $\{i,i+1\}$, and the loop on *path_left* is repeated.

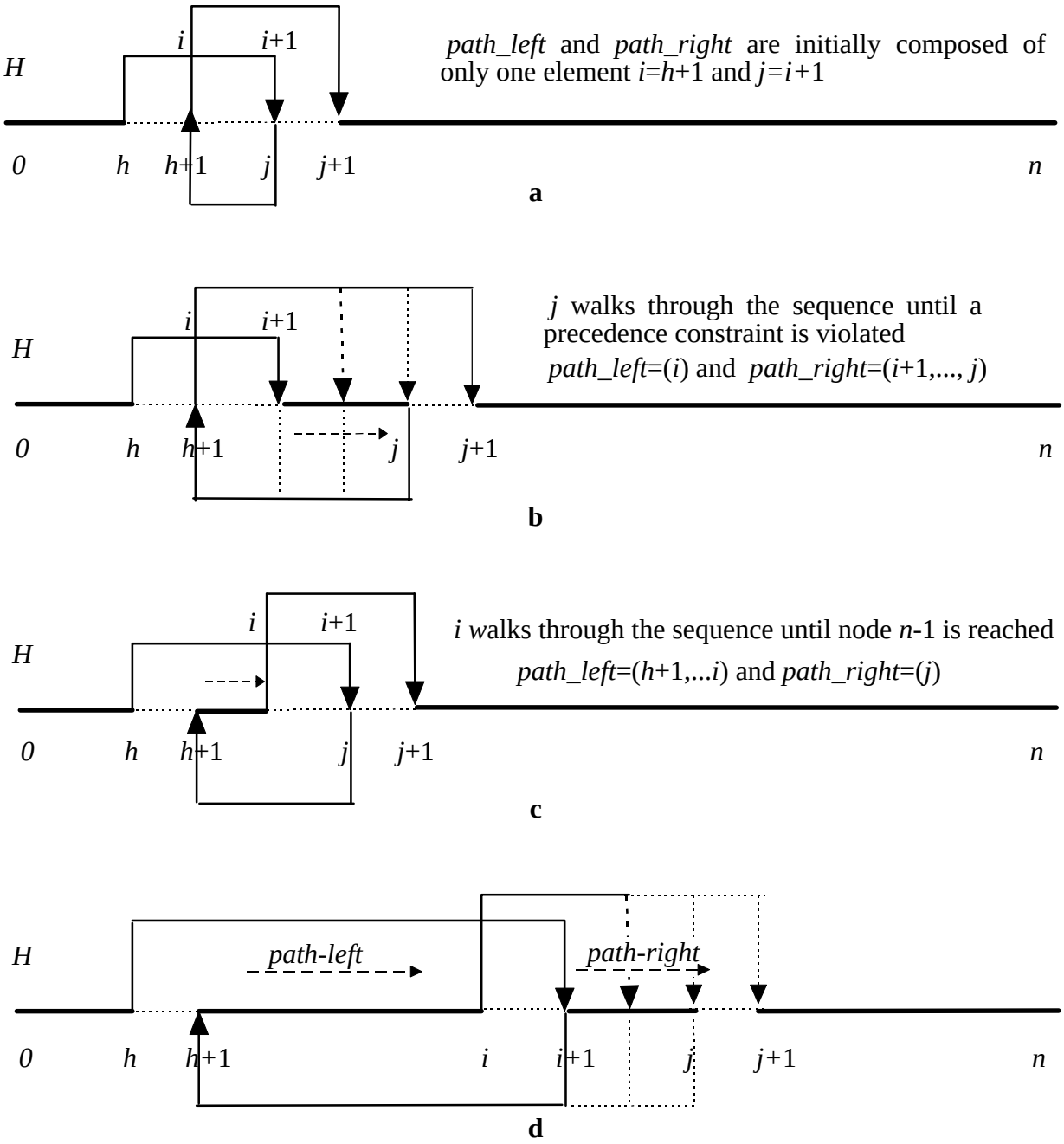

Figure 5. Lexicographic forward path-preserving-3-exchange

The complete *3-exchange lexicographic search* visits all possible nodes in the sequence (just like any other *3-exchange* procedure) but the method for defining *path_left* and *path_right* is useful to easily solve the feasibility checking problem: the search is restricted to feasible exchanges only, since it can be stopped as soon as an unfeasible exchange is found. Consider for example a *f-lpp-3-exchange*: once *path_left*=(h+1, ..., i) has been fixed, we set

path_right to $j=i+1$. In this situation it is possible to check exchange feasibility by testing if there is a precedence relation between node j and nodes in *path_left*. Before expanding *path_right* with the new edge $\{j,j+1\}$ we check if the resulting paths are still feasible by testing again precedence relations between the new node $j+1$ and nodes in *path_left*. In case the test is not feasible we stop the search. In fact, any further expansion of $j+1$ in ($j+2$, $j+3$, ..., n) will always generate an unfeasible exchange because it still violates at least the precedence constraint between $j+1$ and *path_left.*

Notice that expanding *path_left* with edge $\{i,i+1\}$ does not induce any precedence constraint violations because the order of nodes inside *path_left* in not modified and the search for a profitable *f-lpp-3-exchange* always starts by setting *path_right* equal to element $j=i+1$.

Without considering any additional labeling procedure, the feasibility test in this situation has a computational cost of O(n): each time a new j is selected we test if there is a precedence relation between j and the nodes in *path_left*. In case of SOP problems this test should check if $c_{jl} \neq -1 \ \forall l \in$ *path_left* (recall that for SOP problems $c_{jl}=-1$ if l has to precede j, and in the final solution H_1 the order in which we visit *path_left* and *path_right* is inverted and therefore l will follow j, see Figure 4b).

Similar considerations should be made in case of *b-lpp-3-exchange* where the feasibility test checks if $c_{r,j+1} \neq -1 \ \forall r \in$ *path_right.*

The previous *complete lexicographic search procedure* requires to check all predecessors/successors of node j. This procedure increases the computational effort to check *3-optimality* from O(n^3) to O(n^4).

In order to keep the cost at O(n^3) we introduce the *SOP labeling procedure* able to handle multiple constraints.

4.5 The SOP labeling procedure

The *SOP labeling procedure* is used to mark nodes in the sequence with a label that allows for feasibility checking for each selected j in constant time. The basic idea is to associate to each node a label that indicates, given *path_left* and *path_right*, whether or not it is feasible to expand *path_right* with the following node $j+1$.

We have implemented and tested different *SOP labeling procedures* that set and update nodes in different phases of the search. In the following, we will present a combination of the best performing *SOP labeling procedure* with the *lexicographic search strategy*, with different selection criteria for node h and with different search stopping criteria.

Our *SOP labeling procedure* is based on a set of global variables that are updated during the *lexicographic search* procedure. As in the previous subsection, we will distinguish between forward and backward search.

First we introduce a global variable *count_h* that is set to 0 at the beginning of the search and which is increased by 1 each time a new node h is selected. Second, we associate a global variable *f-mark*(v) to each node $v \in H$, in case of *f-lpp-3-exchange*, and a global variable *b-mark*(v), in case of *b-lpp-3-exchange*. This global variable is initially set to 0 for each v.

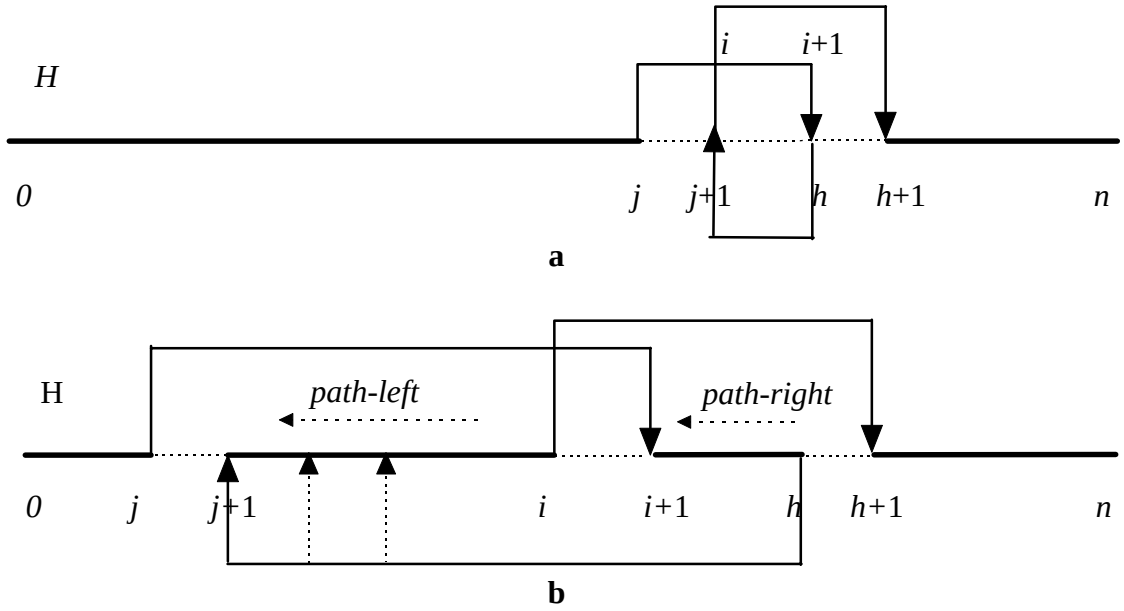

Figure 6. Lexicographic backward path-preserving-3-exchange

An *f-lpp-3-exchange* starts by fixing *h*, *i*=*h*+1, and *path_left*=(*i*). At this point, for all nodes *s* ∈*successor[i]* we set *f_mark*(*s*)=*count_h*. We repeat this operation each time *path_left* is expanded with a new node *i*. Therefore the labeling procedure marks with the value *count_h* all the nodes in the sequence that should follow one of the nodes belonging to *path_left*. When *path_right* is expanded moving *j* in (*i*+2, ..., *n*) if *f_mark*(*j*) =*count_h* we stop the search because the label indicates that *j* should follow a node in *path_left*. At this point, if no other search termination condition is met, the procedure restarts expanding again *path_left*. In this situation all the previous defined labels remain valid and the search continues by labeling all the successors of the new *i* .

On the other hand, when we move *h* forward into the sequence we invalidate all previously set labels by setting *count_h:=count_h*+1.

The same type of reasoning is valid in case of *b-lpp-3-exchange.* Each time node *i* is selected we identify a new *path_right*=(*i*+1, ..., *h*) and for all nodes *s*∈*predecessor[i*+1*]* we set *b_mark*(*s*)=*count_h.*

When expanding *path_left* by iteratively adding a new edge *{j,j+1},* the expansion is not accepted if *b_mark*(*j*+1)=*count_h.*

4.6 Heuristics for the selection of node *h* and search stopping criteria

The presented *sequential* search procedure for SOP problems is a general description of how the *lexicographic search* works in combination with the *SOP labeling procedure*.

Although the SOP labeling procedure reduces the complexity of the lexicographic search to O(n^3), this is still too expensive from a practical point of view; in fact, the exploration of all the feasible exchanges is still required. There are different ways to reduce this effort: for example, one can introduce some heuristic criteria to reduce the number of visited nodes, or one can stop the search and execute the exchange as soon as some improving condition is met.

Heuristic selection of node *h*. In order to reduce the number of explored nodes, Savelsbergh (1990) and Van der Bruggen et al. (1993) propose to use a particular type of *k-exchange* called *OR-exchange* (Or, 1976) that limits the choice of *i* among the three closest node of *h*. In practice, *i* is selected among (*h*+1, *h*+2, *h*+3) in case of a forward exchange, and among (*h*-1, *h*-2, *h*-3) in case of a backward exchange.

Differently, we decrease the number of visited nodes introducing two heuristics that influence how node *h* is chosen: one is based on the *don't look bit* data structure introduced

by Bentley (Bentley, 1992), while the other is based on a new data structure called *don't push stack* introduced by the authors.

The *don't look bit* is a data structure in which a bit is associated with each node of the sequence. At the beginning of the search all bits are turned off. The bit associated with node *h* is turned on when a search for an improving move starting from *h* fails. The bit associated with node *h* is turned off again when an improving exchange involving *h* is executed.

The use of *don't look bits* favors the exploration of nodes that have been involved in a profitable exchange. The search procedure always visits all nodes in the sequence starting from the first node 0 to the last node *n* but only nodes with the *don't look bit* turned off are taken into consideration as candidates for node *h*. Using this approach the search procedure tries to optimize the sequence always starting from the leftmost node not yet visited or such that the last time it was involved in an exchange this was a profitable one.

The *don't push stack* is a data structure based on a *stack*, which contains the set of nodes *h* to be selected, associated with a particular push operation.

At the beginning of the search the *stack* is initialized with all the nodes (that is, it contains *n*+1 elements). During the search, node *h* is popped off the stack and feasible 3-exchanges starting form *h* are investigated. In case a profitable exchange is executed the six nodes involved in this exchange (that is, *j*+1, *i*+1,*h*+1, *j*, *i*, *h*) are pushed onto the stack (if they do not already belong to it). Using this heuristic, once a profitable exchange is executed starting from node *h*, the top node in the *don't push stack* remains node *h*. In addition, we limit the maximum size of the stack to *n*+1 elements.

The use of the *don't push stack* gives the following benefits. First, the search is focused on the neighborhood of the most recent exchange: this has been experimentally shown to result in a better performance than that obtained using the don't look bit (see section 5.1). Second, the selection of node *h* is not constrained to be a sequential walk through the sequence H. This is an important feature given the fact that the *SOP labeling procedure* is designed to work with random choices of *h*. In fact, it does not require, as is the case of Savelsberg's labeling procedure (Savelsbergh, 1990), to keep valid labeling information walking through the sequence from one *h* to the next: a new labeling is started as soon as a new *h* is chosen; this allows for selecting *h* in any sequence position without introducing additional computational costs.

Stopping criteria. The number of visited nodes can be decreased by stopping the search once an improving exchange is found. In our experiments we have tested three different stopping conditions: (*ExchangeFirst=h,i,j*) stops the search as soon as the first feasible exchange is found in the *h*, *i* or *j* loops respectively. We have also tested the standard exploration strategy where the most profitable exchange is selected among all the possible exchanges but this method is not presented here because the obtained results are much worse given the same amount of computation time. In the following we present our SOP-3-exchange procedure with all the possible options.

The SOP-3-exchange procedure

```
/* input:    a sequence given as (0.......n)
             a SOP problem G
             a SelectionCriterium for h in
             (sequential,dont_look_bit,dont_push_stack)
             a WalkingCriterium for i in (3-exchange,OR-exchange)
             ExchangeFirst in (h,i,j)
 output:     a sequence that is 3-optimal */
  REPEAT
   while there is an available h /* h LOOP */
      /*  selects node h according to SelectionCriterium.
          In case SelectionCriterium=sequential h is the next node in the sequence.
          In case SelectionCriterium=dont_look_bit h is the first node in the sequence with dont_look_bit[h]=off.
          In case SelectionCriterium=dont_push_stack h is popped from the stack */
```

```
h ← SelectAccordingCriterium(SelectionCriterium,G);
direction ← forward; /* the search starts in forward direction */
gain ← 0
i ← h+1
j ← h+2
while search is made in forward and backward directions
       according to WalkingCriterium /* i LOOP */
    /* when i has reached the end of the sequence during a forward search we start a new search in backward
       direction starting from the same h. In case WalkingCriterium=OR-exchange the direction is inverted
       after three selections of i */
    if direction=forward and EndOfSequence(i,WalkingCriterium,G)
       direction ← backward
       i ← h-1
       j ← h-2
    end-if
    feasible ← true
    /* In case direction=forward we update labeling information for successors[i]
       in case direction=backward we update labeling information for predecessors[i+1] */
    UpdateGlobalVariables(h,i,direction,G)
    while feasible /*j LOOP */
          /* Using labeling information we test if the 3-exchange involving h,i,j is feasible */
          if feasible←FeasibleExchange(h,i,j,direction,G)
          /* we check if the new 3-exchange is better then the previous one and, in that case, we save it */
             gain ← ComputeBestExchange(h,i,j,direction,G,gain)
          end-if
          if gain>0 and (ExchangeFirst=j)
             break
          /* j is moved through the sequence according to direction.
             In case direction=forward j←j+1, in case direction=backward j←j-1 */
          end-if
          j ← JWalkThroughTheSequence(h,i,j,direction,G)
    end-while
    if gain>0 and (ExchangeFirst=(i or j))
       break
    /* i is moved through the sequence according to direction.
       In case direction=forward i←i+1, in case direction=backward i←i-1 */
        i ← I2WalkThroughTheSequence(h,i,direction,G)
    end-if
end-while
 if gain>0 and (ExchangeFirst=(i or j or h))
    /* the best exchange found until now is executed and the search starts again */
    PerformExchange(h,i,j,direction,G)
 end-if
 goto REPEAT
end-while
```

5. Computational results

We tested our algorithms on a set of problems in TSPLIB (available at http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html).

SOP problems in TSPLIB can be classified as follows: a set of problems (rbgxxxa) are real-life problems derived from a stacker crane application (Ascheuer, 1996). These problems were originally defined as ATSPs with time windows: to obtain SOP instances time window precedences are relaxed to generate SOP precedences. Prob100 (Ascheuer, 1996) is a randomly generated problem, and problems (ftxx.x, and kroxxxp.x) have been generated (Ascheuer, 1996) starting from ATSP instances in TSPLIB by adding a number <=*k* of random precedence constraints, where *k=(n*/4, *n*/2, 2, 2**n)* correspond to the problem extension (.1, .2, .3, .4). ESC63 and ESC78 are taken from (Escudero, 1988). Experiments have been run on a SUN Ultra1 SPARC Station (167Mhz). The code was written in C++.

5.1 Selection criteria for node *i* and search stopping criteria

In this section we test different selection criteria for node *i* and different search stopping criteria. We ran five experiments for each problem, setting the computational time to 100 sec for the ft53.x, ft70.x and ESCxx problems, to 300 sec for the kro124p.x problems, and to 600 sec for the other problems.

The tested stopping criteria are: *ExchangeFirst=j,i,h*. The tested selection criteria are *sequential, don't look bit,* and *don't push stack,* coupled with either the *3-exchange* or the *OR-exchange* walking criterium.

For each considered test problem (the problems are reported in Tables 2 and 3), we ranked results computed by the different combinations of selection and stopping criteria according to the average results obtained. In Table 1 the methods are ranked by the median of each method over the set of test problems.

Results indicate that the *don't push stack* is the best selection criterium, followed by the *don't look bit* and finally by the sequential selection criterium. Figure 7 compares the three selection criteria for the same values of ExchangeFirst and WalkingCriterium. Again, it is clear that *don't push stack* performs better than the other two criteria.

Figure 7. Comparisons of selection criteria on median ranks. Each comparison involves the same value of ExchangeFirst and WalkingCriterium.

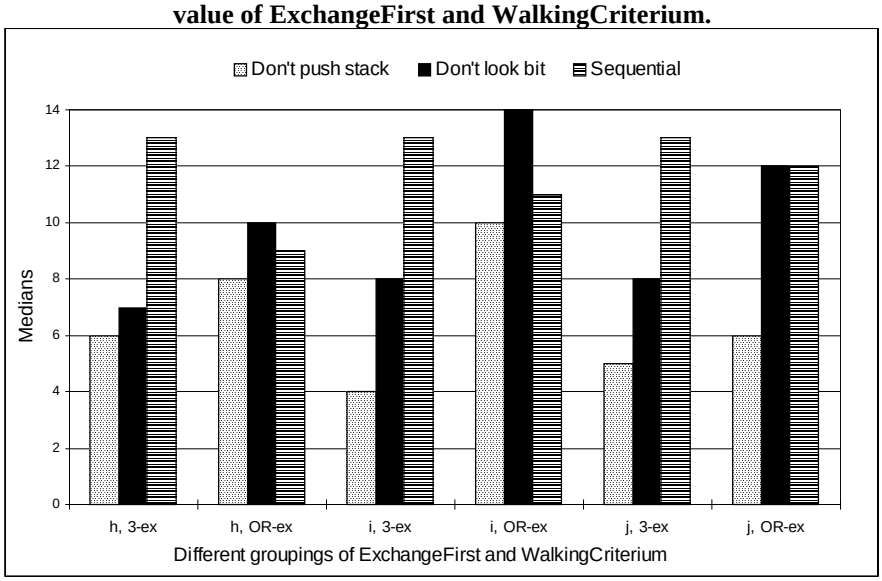

Table 1. Ranking of median rank on 23 SOP test problems for different combinations of selection and stopping criteria

SelectionCriterium	ExchangeFirst	WalkingCriterium	Median
don't push stack	*i*	3-exchange	4
don't push stack	*j*	3-exchange	5
don't push stack	*h*	3-exchange	6
don't push stack	*j*	OR_exchange	6
don't look bit	*h*	3-exchange	7
don't push stack	*h*	OR_exchange	8
don't look bit	*j*	3-exchange	8
don't look bit	*i*	3-exchange	8
sequential	*h*	OR_exchange	9
don't push stack	*i*	OR_exchange	10
don't look bit	*h*	OR_exchange	10
sequential	*i*	OR_exchange	11
don't look bit	*j*	OR_exchange	12
sequential	*j*	OR_exchange	12
sequential	*h*	3-exchange	13
sequential	*j*	3-exchange	13
sequential	*i*	3-exchange	13
don't look bit	*i*	OR_exchange	14

5.2 Computational results and comparisons with other methods

In this section, we present computational results obtained by the best HAS-SOP procedure, that is HAS-SOP with *don't push stack*, *ExchangeFirst=i*, and *WalkingCriterium = 3-exchange*, and compare them with results available in literature.

Our method was able to reach the best known available upper bound (or best known solution) for all the problems (see Table 2). Moreover, HAS-SOP led to improvements in many of these upper bounds (see boldface values in Table 2).

Experiments presented in Table 2 were run for different amounts of CPU time, as a function of their dimension: if $n < 100$ HAS-SOP was run for T=300 sec, if $100 <= n < 300$ then T=600 sec, and for bigger problems T=3600 sec. For the very difficult prob.100.sop we set T=3600.

Each experiment was run 10 times and in Table 2 we report:

- *n*: the size of the problem in term of number of nodes.
- |R|: the number of constraints.
- The bounds reported in TSPLIB.
- *Best Result*: the best result obtained over 10 experiments.
- *Avg Result*: average of the best results obtained in each experiment.
- *Std.Dev*: standard deviation of the best results obtained in each experiment.
- *Avg Time*: average time (in sec) needed by HAS-SOP to reach the best result in each experiment.

In Table 3 we compare HAS-SOP with Maximum Partial Order/Arbitrary Insertion (MPO/AI, Chen and Smith, 1996). The comparison was run on those problems for which data about MPO/AI performance was available. For most of these problems HAS-SOP outperformed MPO/AI both in terms of solution quality (best and average) and computational time required.

In conclusion, in Table 4 we report the new upper bounds obtained by HAS-SOP. These have been obtained by HAS-SOP with *don't push stack*, *ExchangeFirst=i*, and *WalkingCriterium = 3-exchange*, except for those in boldface for which a post-optimization phase was run. The post-optimization consisted of running HAS-SOP again, starting from the best found solution but using one of the variants presented in Table 1 (the post-optimization was in fact run for all problems, but only in three cases it was able to improve the best solution found in the first optimization phase).

Norbert Ascheuer (1997) has run his branch&cut program starting from our best found solutions in Table 4. He could not improve any of our solutions within 24-CPU hours on a SUN SPARC Station 4 (110Mhz) but was able to prove optimality for rbg378a and to compute the new lower bounds reported in Table 4.

Table 2. Results obtained by HAS-SOP on a set of 23 test problems. For 14 problems out of 23 HAS-SOP found a result which improved the TSPLIB upper bound (these results are in boldface). For six problems HAS-SOP found the already known optimal solution, and for three problems it found the same value as the TSPLIB upper bound.

PROB	n	|R|	TSPLIB Bounds	Best Result	Avg Result	Std.Dev.	Avg Time (sec)
ESC63.sop	65	95	62	62	62.0	0	0.1
ESC78.sop	80	77	18230	18230	18230.0	0	6.9
ft53.1.sop	54	12	[7438,7570]	**7531**	7531.0	0	9.9
ft53.2.sop	54	25	[7630,8335]	**8026**	8026.0	0	18.4
ft53.3.sop	54	48	[9473,10935]	**10262**	10262.0	0	2.9
ft53.4.sop	54	63	14425	14425	14425.0	0	0.4
ft70.1.sop	71	17	39313	39313	39313.0	0	29.8
ft70.2.sop	71	35	[39739,40422]	**40419**	40433.5	24.6	114.1
ft70.3.sop	71	68	[41305,42535]	42535	42535.0	0	64.4
ft70.4.sop	71	86	[52269,53562]	**53530**	53566.5	7.6	38.2
kro124p.1.sop	101	25	[37722,40186]	**39420**	39420.0	0	115.2
kro124p.2.sop	101	49	[38534,41677]	**41336**	41336.0	0	119.3
kro124p.3.sop	101	97	[40967,50876]	**49499**	49648.8	249.7	262.8
kro124p.4.sop	101	131	[64858,76103]	76103	76103.0	0	57.4
prob.100.sop	100	41	[1024,1385]	**1226**	1302.4	39.4	1918.7
rbg109a.sop	111	622	1038	1038	1038.0	0	14.6
rbg150a.sop	152	952	[1748,1750]	1750	1750.0	0	159.1
rbg174a.sop	176	1113	2033	2033	2034.7	1.4	99.3
rbg253a.sop	255	1721	[2928,2987]	**2950**	2950.0	0	81.5
rbg323a.sop	325	2412	[3136,3157]	**3141**	3146.0	1.4	1685.5
rbg341a.sop	343	2542	[2543,2597]	**2580**	2591.9	11.8	2149.6
rbg358a.sop	360	3239	[2518,2599]	**2555**	2561.2	5.2	2169.3
rbg378a.sop	380	3069	[2761,2833]	**2817**	2834.3	10.7	2640.3

Table 3. Comparison with MPO/AI. For almost all problems HAS-SOP outperformed MPO/AI both in terms of quality of the best and average solution found, and in terms of computational time.

PROB	TSPLIB Bounds	MPO/AI Best	MPO/AI Avg	MPO/AI Time (sec)	HAS-SOP Best	HAS-SOP Avg	HAS-SOP Time (sec)
ft70.1.sop	39313	39545	39615	120	39313	39313.0	29.8
ft70.2.sop	[39739,40422]	40422	40435	120	40419	40433.5	114.1
ft70.3.sop	[41305,42535]	42535	42558	120	42535	42535.0	64.4
ft70.4.sop	[52269,53562]	53562	53583	120	53530	53566.5	38.2
kro124p.1.sop	[37722,40186]	40186	40996	240	39420	39420.0	115.2
kro124p.2.sop	[38534,41677]	41667	42576	240	41336	41336.0	119.3
kro124p.3.sop	[40967,50876]	50876	51085	240	49499	49648.8	262.8
kro124p.4.sop	[64858,76103]	76103	76103	240	76103	76103.0	57.4
rbg323a.sop	[3136,3157]	3157	3161	2760	3141	3146.0	1685.5
rbg341a.sop	[2543,2597]	2597	2603	3840	2580	2591.9	2149.6
rbg358a.sop	[2518,2599]	2599	2636	6120	2555	2561.2	2169.3
rbg378a.sop	[2761,2833]	2833	2843	8820	2817	2834.3	2640.3

Table 4. New bounds for SOP problems. New upper bounds were obtained by HAS-SOP, while new lower bounds were obtained by a branch&cut program starting from HAS-SOP solutions. The last column gives the best result found by HAS-SOP.

PROB	TSPLIB Bounds	NEW Lower Bounds	NEW Upper Bounds	HAS-SOP All Best
ESC63.sop	62			62
ESC78.sop	18230			18230
ft53.1.sop	[7438,7570]		7531	7531
ft53.2.sop	[7630,8335]		8026	8026
ft53.3.sop	[9473,10935]		10262	10262
ft53.4.sop	14425			14425
ft70.1.sop	39313			39313
ft70.2.sop	[39739,40422]	39803	40419	40419
ft70.3.sop	[41305,42535]	41305		42535
ft70.4.sop	[52269,53562]	53072	53530	53530
kro124p.1.sop	[37722,40186]	37761	39420	39420
kro124p.2.sop	[38534,41677]	38719	41336	41336
kro124p.3.sop	[40967,50876]	41578	49499	49499
kro124p.4.sop	[64858,76103]			76103
prob.100.sop	[1024,1385]	1027	1190	**1190**
rbg109a.sop	1038			1038
rbg150a.sop	[1748,1750]			1750
rbg174a.sop	2033			2033
rbg253a.sop	[2928,2987]	2940	2950	2950
rbg323a.sop	[3136,3157]	3137	3141	3141
rbg341a.sop	[2543,2597]	2543	2574	**2574**
rbg358a.sop	[2518,2599]	2529	2545	**2545**
rbg378a.sop	[2761,2833]	2817	2817	2817

6. Conclusions

Ant colony optimization[2] is a recent approach to distributed combinatorial optimization. Up to now, ant colony algorithms have been successfully applied to the solution of symmetric and asymmetric traveling salesman problems (Dorigo, Maniezzo and Colorni, 1991; 1996; Dorigo and Gambardella, 1997), quadratic assignment problems (Gambardella, Taillard and Dorigo, 1997), graph coloring problems (Costa and Hertz, 1997), and telecommunications routing problems (Di Caro and Dorigo, 1997; 1998). In this paper we have presented an application of an hybrid algorithm, HAS-SOP, which combines an ant colony algorithm with local search to solve the sequential ordering problem. HAS-SOP has been evaluated experimentally on a set of test problems available on TSPLIB, and it has been compared to MPO/AI (Chen and Smith, 1996), one of the best-performing heuristics for the sequential ordering problem. Results show that, in addition to outperforming MPO/AI both in terms of quality of the results and CPU time required to find them, HAS-SOP was also able to improve most of the upper bounds for the above-mentioned set of test problems.

Acknowledgments

We wish to thank Norbert Ascheuer for checking the optimality of our results with his branch&cut program.

[2] Up-to-date information on applications of ant colony optimization can be found at http://iridia.ulb.ac.be/dorigo/ACO/ACO.html, maintained by Marco Dorigo.

References

Aarts E. H. and J. K. Lenstra, 1997, Introduction, in *Local Search in Combinatorial Optimization*, Aarts E. H. and J. K. Lenstra (Eds.). Chichester: John Wiley & Sons, 1–17.

Ascheuer N., 1996. Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Systems. *ZIB Technical Report TR 96-3*.

Ascheuer N., 1997, Personal communication.

Ascheuer N., Escudero L. F., Grotschel M. and Stoer M., 1993, A Cutting Plane Approach to the Sequential Ordering Problem (with Applications to Job Scheduling in Manufactoring), *SIAM Journal on Optimization* 3,25–42.

Balas E., Fischetti M., Pulleyblank W.R., 1995, The Precedence-constrained Asymmetric Traveling Salesman Polytope, *Mathematical Programming* 65, 241–265.

Battiti R. and Tecchiolli G., 1994, The reactive tabu search, *ORSA Journal on Computing*, 6, 126-140.

Bentley J.L., 1992, Fast algorithms for geometric traveling salesman problem, *ORSA Journal on Computing*, vol. 4, pp. 387–411.

Chen S. and Smith S., 1996, S.F. Commonality and Genetic Algorithms. *Carnegie Mellon University, The Robotic Institute, Technical Report CMU-RI-TR-96-27.*

Connolly D. T., 1990, An improved annealing scheme for the QAP, *European Journal of Operational Research*,46, 93-100.

Costa D. and A. Hertz, 1997, Ants Can Colour Graphs, *Journal of the Operational Research Society*, 48, 295–305.

Di Caro G. and M. Dorigo, 1997. AntNet: A Mobile Agents Approach to Adaptive Routing, *Tech. Rep. IRIDIA/97-12*, Université Libre de Bruxelles, Belgium.

Di Caro G. and M. Dorigo, 1998. Mobile Agents for Adaptive Routing, *Proceedings of the 31st Hawaii International Conference on System Sciences (HICSS-31)*, Big Island of Hawaii, January 6-9, 1998, in press.

Dorigo M., 1992, "*Ottimizzazione, Apprendimento Automatico, ed Algoritmi Basati su Metafora Naturale* (Optimization, Learning and Natural Algorithms)," Doctoral dissertation, Dipartimento Elettronica e Informazione, Politecnico di Milano, Italy (in Italian).

Dorigo M., V. Maniezzo and A. Colorni, 1991, The Ant System: An Autocatalytic Optimizing Process. *Tech. Rep. No. 91-016*, Politecnico di Milano, Italy.

Dorigo M., Maniezzo V. and Colorni A., 1996, The Ant System: Optimization by a colony of cooperating agents, *IEEE Transactions on Systems, Man, and Cybernetics–Part B* 26**:**(1), 29–41.

Dorigo M. and Gambardella L. M., 1997, Ant colony system: A cooperative Learning Approach to the Traveling Salesman Problem, *IEEE Transactions on Evolutionary Computation 1*,53–66.

Escudero L. F., 1988. An Inexact Algorithm for the Sequential Ordering Problem. *European Journal of Operational Research* **37**, 232–253.

Escudero L.F., Guignard M., Malik K, 1994. A Lagrangian Relax-and-cut Approach for the Sequential Ordering Problem with precedence Relationships, *Annals of Operations Research* 50, 219–237.

Fleurent C. and Ferland J., 1994, Genetic hybrids for the quadratic assignment problem, *DIMACS Series in Mathematical Theoretical Computer Science* 16, 190-206.

Gambardella L. M., È. D. Taillard and M. Dorigo, 1997, Ant Colonies for the QAP. *Tech. Rep. IDSIA/4-97*, IDSIA, Lugano, Switzerland.

Kindervater G.A.P. and Savelsbergh M. W. P., 1997, Vehicle Routing: Handling Edge Exchanges, In Local Search in Combinatorial Optimization, Aarts E. H. and J. K. Lenstra (Eds.). Chichester: John Wiley & Sons, 311–336.

Johnson D.S. and McGeoch L.A., 1997, The Traveling Salesman Problem; a Case Study, In Local Search in Combinatorial Optimization, Aarts E. H. and J. K. Lenstra (Eds.). Chichester: John Wiley & Sons, 215–310.

Lin S., 1965, Computer solutions of the traveling salesman problem, *Bell Systems Journal*, vol. 44, pp. 2245–2269.

Lin S. and Kernighan B.W., 1973, An effective heuristic algorithm for the traveling salesman problem, *Operations Research*, vol. 21, pp. 498–516.

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

Psaraftis H.N., 1983, K-interchange Procedures for local Search in a precedence-Constrained Routing Problem, *European Journal of Operational Research* **13**, 341–402.

Pulleyblank W. and M. Timlin, 1991, Precedence Constrained Routing and Helicopter Scheduling: Heuristic Design. *IBM Technical Report RC17154 (#76032)*.

Reinelt G., 1994, The traveling salesman: computational solutions for TSP applications. *Springer-Verlag*.

Savelsbergh M. W. P., 1990, An efficient implementation of local search algorithms for constrained routing problems. *European Journal of Operational Research* **47**, 75–85.

Solomon M.M., 1987, Algorithms for the Vehicle Routing and Scheduling Problems with Time Windows Constraints, *Journal of Operational Research* **35**, 254–265.

Taillard É. D., 1991, Robust taboo search for the quadratic assignment problem, *Parallel Computing*, 17, 443-455.

Van der Bruggen L.J.J., Lenstra L.K., Schuur P.C., 1993, Variable-depth Search for the Single-Vehicle Pickup and Delivery Problem with Time Windows, *Transportation Science* 27, 298–311.