AN OBJECT-ORIENTED METHODOLOGY FOR SOLVING ASSIGNMENT-TYPE PROBLEMS WITH NEIGHBORHOOD SEARCH TECHNIQUES

JACQUES A. FERLAND

University of Montreal, Montreal, Quebec, Canada

ALAIN HERTZ

Ecole Polytechniques Fédérale de Lausanne, Lausanne, Switzerland

ALAIN LAVOIE

University of Montreal, Montreal, Quebec, Canada

(Received March 1994; revision received July 1995; accepted August 1995)

Because of its specificity, it is usually difficult to reuse computer code developed for a given combinatorial problem to deal with another one. We use the Object Oriented Programming methodology to derive general purpose software including four different neighborhood search techniques (descent method, tabu search, exchange procedure, simulated annealing) to deal with any assignment-type problem with a bare minimum of coding effort. The structure of the software even allows a more advanced user to play around with several parameters of these techniques and to modify the techniques to create specific variants.

Because of its specificity, it is difficult to reuse computer code developed for a given combinatorial optimization problem to solve another one. In this paper, we introduce computer software that features several heuristic iterative optimization procedures to deal with any problem that can be formulated as an assignment-type problem (ATP). The object-oriented programming (OOP) method is a very efficient approach for developing software of this kind. It induces modularity and a rational order of the software structures, and it allows reusability.

The software presented in this paper includes two basic hierarchies. The first one establishes an interface with the data structures for the user's problem and the second hierarchy includes four different neighborhood search techniques: the descent method (DM), the tabu search (TS), the exchange procedure (EP), and simulated annealing (SA).

The objective of this paper is not to compare the relative efficiency of these techniques for dealing with a specific problem, but rather to introduce more general software allowing anyone to complete efficiently such a comparison for his specific assignment-type problem with a bare minimum of coding effort. For a more advanced user, the OOP structure of the hierarchy of techniques allows playing around with several parameters of these techniques and even modifying them to create specific variants with a bare minimum of coding effort.

We see this software as a first step in the line of development to facilitate the resolution of difficult combinatorial problems with a minimal investment in computer coding. Several potential extensions of the software are mentioned in the conclusion. The software is coded in Object-Oriented Turbo Pascal, and it runs on PC-compatible workstations.

In Section 1, assignment-type problems are characterized, and several examples of combinatorial problems of this type are given. The approach that characterizes neighborhood search techniques is described in Section 2. In this section, the four techniques (DM, TS, EP, and SA) included in the software are also summarized. In Section 3 we introduce a brief summary of the OOP method. The software, which is called NST-ATP, is described in Section 4 together with indications of how to use it to deal with three different ATPs: the graph coloring problem (GCP), the graph partitioning problem (GPP), and the capacitated vehicle routing problem (VRP). Section 5 concludes with indications for future developments of this kind.

1. PROBLEM FORMULATION

An assignment-type problem (ATP) can be summarized as follows:

Given n items and m resources, the problem is to determine an assignment of the items to the resources optimizing an objective function and satisfying K additional side constraints.

The associated mathematical model is as follows.

Problem ATP

Minimize $F(x)$

$$\text{subject to} \quad \sum_{j \in J_i} x_{ij} = 1 \quad 1 \leq i \leq n \tag{1}$$

Subject classifications: Computers/computer science, software: object-oriented methodology. Programming, integer, heuristic: neighborhood search techniques, assignment-type problems.
Area of review: COMPUTING.

0030-364X/96/4402-0347 $01.25

$$G_k(x) \leq 0 \quad 1 \leq k \leq K \tag{2}$$

$$x_{ij} = 0 \text{ or } 1 \quad 1 \leq i \leq n, j \in J_i\,, \tag{3}$$

where x_{ij} is a decision variable

$$x_{ij} = \begin{cases} 1 & \text{if item } i \text{ is assigned to resource } j \\ 0 & \text{otherwise} \end{cases}$$

and $J_i \subseteq \{1, 2, \ldots, m\}$ is the set of admissible resources for i, $1 \leq i \leq n$.

The assignment constraints (1) together with (3) indicate that each item i has to be assigned to exactly one resource j. The objective function F and the side constraint functions G_k are not restricted to having any specific property other than being calculable. (They can even be evaluated by solving a subproblem.)

Several well known problems are instances of **ATP** with specific side constraints (2). For example, the classic assignment problem (AP), the generalized assignment problem (GAP) (Ross and Soland 1975) and the timetabling problem (TP) (Ferland and Lavoie 1992) can be seen as special cases of ATP. Their objective function is

$$F(x) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}\,,$$

where c_{ij} is the cost of assigning item i to resource j, and $J_i = \{1, 2, \ldots, m\}$ for all i, $1 \leq i \leq n$. Their side constraints are specific:

1. AP (resource assignment constraints):

$$\sum_{i=1}^{n} x_{ij} = 1 \quad 1 \leq j \leq m\,,$$

where $m = n$.

2. GAP (resource capacity constraints):

$$\sum_{i=1}^{n} r_{ij} x_{ij} \leq b_j \quad 1 \leq j \leq m\,,$$

where r_{ij} is the amount of resource j used when item i is assigned to j, and b_j is the total amount of resource j available.

3. TP (linear side constraints):

$$\sum_{i=1}^{n} \sum_{j=1}^{m} r_{ijk} x_{ij} \leq b_k \quad 1 \leq k \leq K\,.$$

In this paper, we illustrate our approach using three combinatorial problems that can be formulated as **ATP**. In the Graph Coloring Problem (**GCP**) (Hertz and de Werra 1987), the objective is to color the vertices of a graph $G = (V, E)$ with m colors in such a way that no two adjacent vertices have the same color. Referring to the **ATP** formulation, the items are the vertices in V and the resources are the colors. Constraints (1), where $J_i = \{1, 2, \ldots, m\}$ and (3) indicate that each vertex has to receive exactly one color. The objective function can be taken equal to 0 (i.e., $F(x) \equiv 0$), and for each color $k \in \{1, 2, \ldots, m\}$ there is a side constraint

$$G_k(x) = \sum_{(i,j) \in E} x_{ik} x_{jk} \leq 0 \quad 1 \leq k \leq m\,.$$

The graph partitioning problem (**GPP**) (Johnson et al. 1989) is to find a partition of the set of vertices of a graph $G = (V, E)$ (with $|V|$ even) into two subsets V_1, V_2 such that $|V_1| = |V_2|$, which minimizes the sum of the cost of the edges having end-points in different sets. The items are the vertices, and the resources are the subsets V_1 and V_2. Constraints (1), where $J_i = \{1, 2\}$, and (3) indicate that each vertex is either in V_1 or V_2. The objective function is specified as:

$$F(x) = \sum_{(i,j) \in E} c_{ij}(x_{i1}x_{j2} + x_{i2}x_{j1})$$

where c_{ij} denotes the cost of the edge linking i with j.

There is a side constraint associated with each subset V_k, $1 \leq k \leq 2$:

$$G_k(x) = \sum_{i=1}^{n} x_{ik} - \frac{|V|}{2} \leq 0\,.$$

Finally, the capacitated Vehicle Routing Problem (VRP) (Bodin et al. 1983) is to specify the routes of m vehicles delivering (or picking up) goods to n cities represented by vertices in a graph $G = (V, E)$. The objective is to minimize the total length of the routes. The items are the cities, and the resources are the vehicles. Constraints (1), where $J_i = \{1, 2, \ldots, m\}$, and (3) indicate that each city has to be visited exactly once, by one of the vehicles. The objective function $F(x)$ equals the sum of the lengths of the routes, where the length of a route is evaluated by solving a traveling salesman problem through the cities served by the route. There is a capacity (side) constraint associated with each vehicle j:

$$G_j(x) = \sum_{i=1}^{n} a_i x_{ij} - c_j \leq 0 \quad 1 \leq j \leq m\,,$$

where c_j is the capacity of the jth vehicle ($1 \leq j \leq m$) and a_i is the demand of the ith city ($1 \leq i \leq n$).

Other ATP formulations exist for GCP, GPP, and VRP. One of these is given in Section 4 for GPP. In the next section, we review neighborhood search techniques. It will be seen that this type of procedure is very well suited to dealing with problems having assignment constraints (1).

2. NEIGHBORHOOD SEARCH TECHNIQUES

Among the optimization procedures, the iterative solution methods play an important role. The general step of an iterative procedure consists of generating from a current solution s a new solution s' and checking whether one should stop there or perform another step. Neighborhood search techniques (NST) are iterative procedures in which a neighborhood $N(s)$ is defined for each solution s and the next solution s' is selected among the solutions in $N(s)$.

Given a set S of solutions and their neighborhoods $N(s)$, a state-space graph $G = (S, A)$ is defined whose vertex set is S, and there is an arc $(s, s') \in A$ from a vertex s to a

vertex s' if $s' \in N(s)$. A neighborhood search method may be viewed as a walk in the state-space graph G; one step of such a procedure consists of moving from one vertex s to an adjacent vertex s'.

Given a real-valued objective function $f: S \to \mathbb{R}$, the aim of a neighborhood search technique is to find some solution s^* in S such that $f(s^*)$ is acceptable with respect to some criterion. In general, a criterion of acceptability for a solution s^* is $f(s^*) \leq f(s)$ for every s in S. If s^* can be reached, then the neighborhood search method is an exact minimization algorithm. However, in most contexts, there is no such guarantee, and therefore the method can be viewed as a heuristic procedure. The simplest neighborhood search method is the descent method:

1. Let $s \in S$ be an initial solution: continue = true;
2. While continue:
2.1. find a "best" s' in $N(s)$ (i.e., such that $f(s') \leq f(s'')$ for any s'' in $N(s)$)
2.2. if $f(s') \geq f(s)$), then continue = false
else $s = s'$
3. s is a local minimum of f.

Such a method clearly may stop at a local minimum which is not global for f. More elaborate neighborhood search methods have been designed to avoid being trapped in a local minimum. Among these are simulated annealing (SA), the tabu search (TS) and the exchange procedure (EP).

Usually, all neighborhood search methods share many common features when they are applied to the same type of problem. For example, most iterative procedures which have been designed for solving assignment-type problems use the same definition of the neighborhood $N(s)$: A solution s' belongs to $N(s)$ if it can be obtained from s by modifying the assignment of one item. Hence, they work on the same state space graph G. All these techniques differ, however, in the way they choose the next solution s' to move away from s. In other words, it can be noticed that the ingredients of the neighborhood search techniques may be hierarchized. Some of them are common to most such procedures while others are particular to the technique which is used. This hierarchy can be handled in an efficient way by using object-oriented programming, whose most important concepts are described in the next section. We then present an object-oriented program which has been developed for applying neighborhood search techniques to assignment-type problems.

The current version of the software includes four such techniques: the descent method (DM), the tabu search (TS), simulated annealing (SA), and the exchange procedure (EP). First we introduce their common features in our implementations, and then we summarize them briefly.

2.1. Common Features

We now summarize the common features of the four techniques when dealing with an assignment-type problem, reformulated as follows to simplify the notation.

Problem ATP

Minimize $F(x)$

$$\text{subject to} \quad G_k(x) \leq 0 \quad 1 \leq k \leq K \qquad (2)$$
$$x \in X,$$

where $X = \{x: \sum_{j \in J_i} x_{ij} = 1, 1 \leq i \leq n; x_{ij} = 0 \text{ or } 1, 1 \leq i \leq n, j \in J_i\}$.

For each $x \in X$, $Ir(x)$ denotes the *total violation* of (2); i.e.,

$$Ir(x) = \sum_{k=1}^{K} \text{Max}\{0, G_k(x)\}.$$

A penalty approach is used to deal with **ATP** in the software. Hence, instead of minimizing $F(x)$ on the feasible domain specified by (1)–(3), we minimize a *penalty function* $P(x) = \alpha F(x) + \beta Ir(x)$ where $\alpha, \beta \geq 0$ are parameters. This approach is common to several neighborhood search procedures because it might be easier to escape from local minimum for the original problem if it is allowed to move out of its domain. In our implementation, the values of α and β are fixed, but the software can easily be extended to allow for variable values to accommodate other variants (Gendreau, Hertz, and Laporte 1994), or to put emphasis on feasibility by using larger values for β.

The four techniques refer to a *current best solution* ISBEST as the best solution encountered so far during the resolution in the sense that among those solutions with the smallest Ir value, it is the one minimizing F. Hence, assuming that $x^1, x^2, \ldots, x^p$ have been generated by the procedure until now, then

$$\Gamma_p = \{x: x = \underset{1 \leq r \leq p}{\text{Argmin}} \{Ir(x)\} \quad \text{and}$$

$$\text{ISBEST} = \underset{x \in \Gamma_p}{\text{Argmin}} \{F(x)\}.$$

This is a useful piece of information because we are minimizing P instead of F. Given a solution x, then $x <$ ISBEST if $Ir(x) < Ir(\text{ISBEST})$ or $Ir(x) = Ir(\text{ISBEST})$ and $F(x) < F(\text{ISBEST})$.

Denote $R(x, i)$ the resource to which item i is assigned in x. As mentioned before, a solution $x' \in X$ belongs to the neighborhood $N(x)$ of $x \in X$ if it can be generated from x by modifying the assignment of exactly one item. Let us write $x' = x \oplus (i, j)$ with the meaning that x' is obtained from x by assigning item i to a new resource j (i.e., $R(x', i) = j$). Then

$$N(x) = \{x' \in X: \exists \ (i, j) \text{ with } x' = x \oplus (i, j), 1 \leq i \leq n, j \in J_i, j \neq R(x, i)\}.$$

Moreover, for a solution $x' = x \oplus (i, j)$ in $N(x)$, we have

$$x'_{pq} = x_{pq} \quad 1 \leq p \leq n, p \neq i, q \in J_p$$
$$x'_{iq} = x_{iq} \quad q \in J_i, q \neq j, R(x, i)$$
$$x'_{ij} = 1, x'_{iR(x,i)} = 0.$$

Note how straightforward it is to specify $N(x)$ on X. This is another advantage of using a penalty approach since it

would be harder (if even possible) to specify $N(x)$ while taking into account the side constraints $G_k(x) \leq 0$, $1 \leq k \leq K$.

An "aspiration test" is used in all techniques to stop the search in the neighborhood $N(x)$ of the current solution x whenever $x' \in N(x)$ is reached satisfying it. The tests (denoted ASP_DM, ASP_TS, ASP_EP, and ASP_SA in the descent method, the tabu search, the exchange procedure, and simulated annealing, respectively) differ from one technique to the other, but they share the common feature of stopping the search whenever $x <$ ISBEST. It is worthwhile to note the close relationship with the notion of aspiration in the context of tabu search.

Since the size of $N(x)$ increases rapidly with the number of items and resources, it might be appropriate to consider only a subset of promising elements to reduce $P(x)$. To specify such a subset, we introduce the notion of interesting items. An item i, $1 \leq i \leq n$ is *interesting* if modifying $x_{iR(x,i)}$ from 1 to 0 induces a decrease of $Ir(x)$ or a decrease of $F(x)$ whenever $lr(x) = 0$.

We define the *interesting neighborhood* INN(x) of x as the set of interesting items in $N(x)$; i.e.,

$$\text{INN}(x) = \{x' \in X: \exists\ (i, j) \text{ with } x' = x \oplus (i, j),$$
$$i \text{ interesting}, 1 \leq i \leq n, j \in J_i, j \neq R(x, i)\}.$$

This strategy not only increases the chance of reducing $P(x)$, but also improves convergence in general. Indeed, modifying the assignment of an interesting item seems the thing to do for moving out of a local minimum because modifying noninteresting items will frequently generate alternate local minima to the current one.

Finally, the stopping criteria are gathered together in the Boolean function **StopAlgorithm** taking value true whenever the technique has to be stopped. Four different stopping criteria are common to all techniques for inducing the value true to **StopAlgorithm**:

- the maximum number of iterations MA is reached;
- the maximum computation time MAXTIME is reached;
- the lower bound BI on the value of P is reached;
- the user interrupts the execution of the technique.

Other stopping criteria are specific to the techniques. To notify their satisfaction (inducing value true to **StopAlgorithm**), a Boolean variable QUIT is set equal to true.

2.2. The Techniques

Since the techniques included in the current version of the NST-ATP software are widely known, we restrict ourselves to summarizing them briefly. Appropriate references are mentioned to guide the reader needing more specific details.

2.2.1. Descent Method (DM)

At each iteration of the descent method, the interesting neighborhood INN(x) of the current solution x is scanned until an element x' is identified for which the aspiration test ASP_DM(x') is satisfied, and x' becomes the current solution. Note that ASP_DM(x') is verified if $P(x') < P(x)$ or $x' <$ ISBEST. If no such x' exists, then the additional stopping criterion for the descent method is satisfied, and QUIT is set equal to true. Ir(ISBEST) is the lowest level of violation reached and ISBEST is the best solution for this level. The procedure is summarized in Figure 1.

2.2.2. Tabu Search (TS)

(Glover 1986, Hansen 1986) We refer to a simplified form of tabu search to demonstrate its incorporation in our framework, based on a fundamental "short-term component". (Longer term components, which are critical to the most effective applications, are reviewed in Glover and Laguna (1993).) In the form we consider, TS is an iterative procedure where the next solution x' is an element of a subset $N^*(x)$ of the interesting neighborhood INN(x) of the current solution x such that

$$x' = \underset{z \in N^*(x)}{\text{Argmin}} \{P(z)\}.$$

The size of $N^*(x)$ is fixed throughout the resolution according to a parameter NV. If the aspiration test ASP_DM(x') for the descent method is satisfied, then x' corresponds to a descent move. But the tabu search technique allows nondescent moves (i.e., we may have $P(x') \geq P(x)$) to move out of a local minimum region. However, to avoid cycles, tabu lists are used to register the most recent moves, and the elements in $N^*(x)$ are selected to avoid these moves. Nevertheless, it might be interesting to use a tabu move if an aspiration test ASP_TS(x') indicates that it generates a significant improvement because $x' <$ ISBEST or $P(x') <$ PBEST (where PBEST denotes the best value of P encountered so far). Furthermore, the additional stopping criterion specific to the tabu search technique is verified (and, hence, QUIT = true) whenever a maximum number MI of successive iterations inducing no improvement has been completed.

In the short-term variant summarized in Figure 2, each element $z \in$ INN(x) is generated by choosing randomly an interesting item i and an element $j \in J_i$, $j \neq R(x, i)$, such that $z = x \oplus (i, j)$. Whenever ASP_TS(z) is verified, then $x' = z$ and the generation of $N^*(x)$ is stopped.

Three different tabu lists can be used: a list TI of items, a list TR of resources, and a list TIR of pairs (i, j), where i

1. Let x be an initial solution.
   Specify MA, BI, MAXTIME (StopAlgorithm parameters)
   ISBEST=x, ITER=0, QUIT=false
2. while not StopAlgorithm:
   - 2.1 Generate elements of INN(x) until x' is identified such that ASP_DM(x') is verified
   - 2.2 if no such x' exists, then QUIT=true
   - 2.3 if x'< ISBEST, then ISBEST=x'
   - 2.4 x=x'
   - 2.5 ITER=ITER+1
3. Ir(ISBEST) is the lowest level of violation reached and ISBEST is the best solution reached with initial solution and parameters specified in 1.

Figure 1. Descent method.

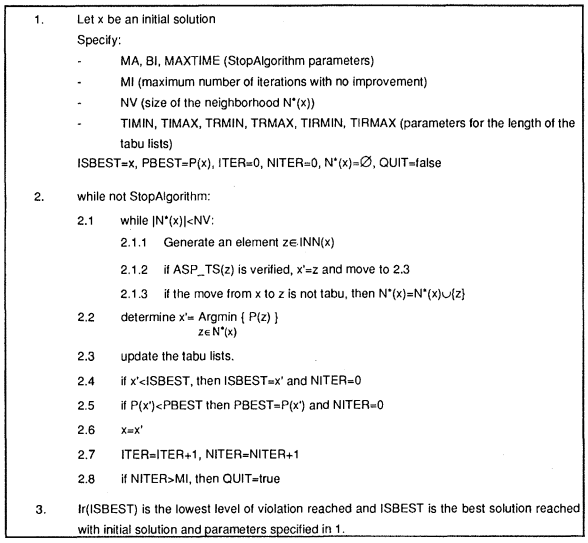

Figure 2. Short-term tabu search.

is an item and j is a resource. When the assignment of an item i is modified in a solution x, item i is introduced into TI, resource $R(x, i)$ is introduced into TR, the pair $(i, R(x, i))$ is introduced into TIR, and the oldest element of each list is removed. The lists TI and TR may be very restrictive, and usually we use only TIR. A move $x' = x \oplus (i, j)$ is not tabu, if $i \notin$ TI whenever TI is used, if $j \notin$ TR whenever TR is used, and if $(i, j) \notin$ TIR whenever TIR is used. The lengths of the tabu lists can be fixed or variable according to some probability distribution. In our implementation, we use TIR with variable length in the interval [TIRMIN, TIRMAX].

2.2.3. Exchange Procedure (EP)

(Ferland and Lavoie 1992) The exchange procedure is also an iterative technique in the spirit of the descent method and the tabu search technique. Indeed, like the tabu search technique, it allows nondescent moves to reach out of a local minimum region. But, these moves are monitored to limit the number of moves and the increase of P before reaching a point x' such that $P(x') < P(x)$ or $x' <$ ISBEST. Hence, the technique is similar to a descent method, but x' belongs to a different (and more complex) neighborhood than $N(x)$.

This neighborhood arises by executing selected moves in sequence, with the goal of generating a 'compound move' that satisfies the aspiration condition. Limits are set on the acceptable sequence depth and the unattractiveness tolerated in the associated compound move. Governed by these limits, the method performs a truncated depth-first tree search until its aspiration is met (or it terminates without finding an acceptable move). The special neighborhood generated by this truncated search is an instance of the 'restructured move' approach proposed as a longer term TS strategy in Glover (1990). An alternative variant has been effectively applied to scheduling problems by Nowicki and Smutnicki (1993, 1994). The exchange method is a 'short-term' variant that erases memory of previous solutions after executing the current compound move.

The procedure is summarized in Figure 3. To illustrate a major iteration of the procedure, assume that x^0 is the current solution. We are looking for a solution x where the aspiration test ASP_EP(x) is satisfied (i.e., $P(x) <$ PITER $= P(x^0)$ or $x <$ ISBEST). A series of forward moves is completed to generate a sequence $x^0, x^1, x^2, \ldots, x^{\text{LEV}}$ such that $x^{i+1} \in \text{INN}(x^i)$ until ASP_EP(x^{LEV}) is verified, or $P(x^{\text{LEV}}) -$ PITER $>$ MR, or LEV $=$ LMAX.

If ASP_EP(x^{LEV}) is verified (i.e., $P(x^{\text{LEV}}) <$ PITER or $x^{\text{LEV}} <$ ISBEST), then x^{LEV} is a better solution, and a new major iteration is initialized with $x^0 = x^{\text{LEV}}$.

If $P(x^{\text{LEV}}) -$ PITER $>$ MR or LEV $=$ LMAX, then the limit of tolerance MR on the increase of P or the limit LMAX on the number of forward moves is reached. Hence, a backward move is completed to analyze an alternate interesting neighbor of $x^{\text{LEV}-1}$ using another forward move. If all the interesting neighbors of $x^{\text{LEV}-1}$ have been visited, then another backward move is completed to reach $x^{\text{LEV}-2}$. Such backward moves are completed until all neighbors of x^0 have been visited, in which case the procedure stops (i.e., QUIT $=$ true).

To implement the procedure in Figure 3, we use $T(x)$ to denote the interesting neighbors of x already visited and T to denote the solutions already visited since starting from x^0. Furthermore PR(x) denotes the predecessor of x in the sequence on hand; i.e., PR(x^{i+1}) $= x^i$. Also, LMAX and the limit of tolerance MR can be specified according to the user's needs. Indeed, these parameters allow him to control the level of thoroughness of the search. For instance, larger values of LMAX and MR induce a more

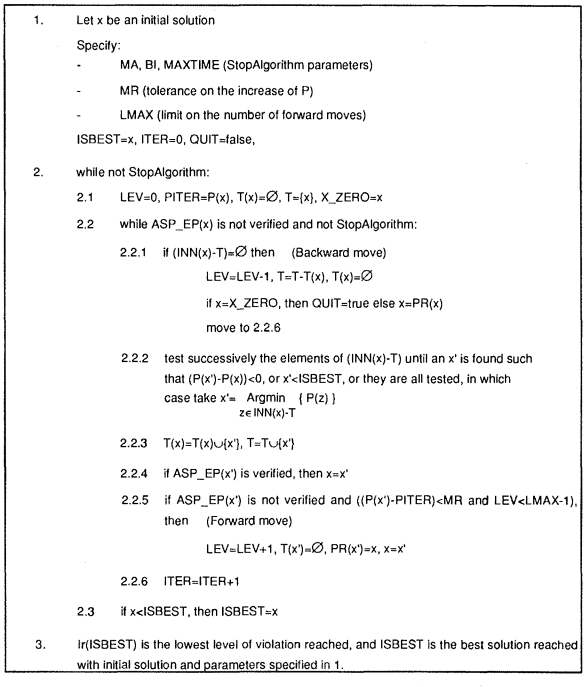

Figure 3. Exchange procedure.

1. let x be an initial solution
   specify:
   MA, BI, MAXTIME (StopAlgorithm parameters)
   FREEZELIMIT, SIZEFACTOR, CUTOFF, TEMPFACTOR, MINPERCENT, TP
   ISBEST=x, PBEST=P(x), FREEZECOUNT=0, ITER=0, QUIT=false
2. while not StopAlgorithm:
   2.1 CHANGES =0, TRIALS=0
   while TRIALS<SIZEFACTOR and CHANGES<CUTOFF:
   2.1.1 generate x'∈ INN(x)
   2.1.2 ΔP=P(x')-P(x)
   2.1.3 if ASP_SA(x') is verified, then
   CHANGES=CHANGES+1, x=x'
   if x<ISBEST then ISBEST=x
   if P(x)<PBEST then PBEST =P(x)
   2.1.4 if ΔP>0 THEN
   select a random number r∈[0,1]
   If r≤$e^{-\Delta P/TP}$ then CHANGES=CHANGES +1, x=x'
   2.1.5 TRIALS=TRIALS+1, ITER=ITER+1
   2.2 TP=TEMPFACTOR·TP
   if ISBEST or PBEST have changed during 2.1 then FREEZECOUNT=0
   if $\frac{\text{CHANGES}}{\text{TRIALS}}$ <MINPERCENT then FREEZECOUNT=FREEZECOUNT+1.
   2.3 if FREEZECOUNT=FREEZELIMIT, then QUIT=true
3. Ir(ISBEST) is the lowest level of violation reached, and ISBEST is the best solution reached with initial solution and parameters specified in 1.

Figure 4. Simulated annealing.

exhaustive search in the neighborhood of the current solutions to identify an improving solution.

2.2.4. Simulated Annealing (SA)

(Johnson et al. 1989, 1991) Simulated annealing is another iterative procedure allowing nondescent moves. This is a probabilistic technique where x' is chosen randomly in the interesting neighborhood of x. If the aspiration test ASP_SA(x') is verified, (i.e., $\Delta P = P(x') - P(x) \leq 0$ or $x' <$ ISBEST), then a nonascent move is completed. Otherwise, the move from x to x' is completed according to a probability decreasing with the value of $\Delta P = P(x') - P(x)$ and with the value of the parameter TP (called temperature) which decreases by a factor TEMPFACTOR (a parameter in (0, 1)) as the number of iterations completed increases. More specifically, the move is completed with probability $e^{-\Delta P/TP}$. The critical issues in this technique are the initialization of TP and how its value is decreased during the iterations. See Hajek (1988) and Greene and Supowit (1986) for discussion of these issues.

The simulated annealing technique included in the software is summarized in Figure 4. This variant is proposed by Johnson et al. (1991). FREEZELIMIT corresponds more or less, to the parameter MI in the tabu search technique. It represents the maximal number of successive temperatures where the ratio of moves accepted (CHANGES) to the total number of moves analyzed (TRIALS) is too small (i.e., smaller than a threshold value MINPERCENT). Furthermore, whenever ISBEST has changed, FREEZECOUNT is reduced to zero to signal this improvement.

SIZEFACTOR corresponds to the maximum number of iterations at the same temperature. But the temperature has to be modified whenever the number of CHANGES at this temperature reaches a threshold value CUTOFF, even if the number of iterations completed is smaller than SIZEFACTOR. This is mostly useful for higher temperatures where the probability of a change is larger.

3. OBJECT-ORIENTED PROGRAMMING (OOP)

In this section, we give a brief overview of object-oriented programming (OOP) (Meyer 1988, Borland 1988, Booch 1991) to introduce the basic concepts of this programming method. Our objective is to give a general understanding of the approach rather than getting into technicalities. While reading this section, keep in mind that our purpose is to develop general computer code for implementing neighborhood search techniques including several well known methods (descent method, tabu search, exchange procedure, simulated annealing) to allow testing easily the relative efficiency of these methods for dealing with any problem that can be formulated as an ATP. Furthermore, these comparisons are fair because all the methods are run in the same computer environment using the same data structures. Hence, none of them is favored over the others.

OOP is a method of programming which is highly structured. It induces modularity and a rational order of software structures which is very similar to that found in a taxonomy chart: More specific entities are specified by adding items to those inherited from more general entities. Reusability is another key feature of OOP. Computer code previously produced can be easily extended or slightly modified to deal with other applications.

The notion of *object* is central to OOP. It is the basic entity including both data and methods (procedures, functions, etc.) manipulating the data. At first sight, it may appear that another completely new *object* has to be specified whenever additional data or methods are required. But the *inheritance* feature of OOP allows this problem to be avoided. Indeed, a *new object* can be specified simply by adding the new data or methods to those inherited from the *ancestor object*. This approach induces a hierarchy between objects.

Furthermore, an *object* can be adapted specifically to several different contexts by creating *descendant objects* and defining or modifying some of the *virtual* procedures to make them specific for each of these contexts. This property of OOP is referred to as *polymorphism*.

These concepts are now used to develop general software for neighborhood search techniques (NST). Any ATP can be solved with these procedures. A first object hierarchy is used to specify a specific problem as an ATP and to establish an interface between the problems and the techniques. In a second object hierarchy, the optimization techniques are specified to deal with assignment type problems. The first hierarchy becomes a parameter for the second one.

4. OOP HIERARCHIES IN THE NST-ATP SOFTWARE

In this section, we first introduce the ATP hierarchy establishing the interface between the problem and the techniques. Once the main object has been described, we indicate how to complete polymorphism for the three problems introduced in Section 1: graph coloring problem (GCP), graph partitioning problem (GPP), and capacitated vehicle routing problem (VRP). Furthermore, we illustrate how to generate the interface associated with each specific problem. Then the second NST hierarchy concerning the techniques is introduced. Finally, pseudocode is given to illustrate how to use the software to solve a GCP with the TS technique.

4.1. ATP Hierarchy

Assume that a user has data structures to generate the information required to deal with a specific ATP. On the one hand, the ATP hierarchy allows obtaining the relevant information from the user data structures and transforming it into the specific ATP format. Then, this transformed information is used by the NST hierarchy including the techniques. On the other hand, the **ATP** hierarchy is also useful to transfer information from the NST hierarchy to the user data structures.

To be more specific, the main object ATP_INTERFACE of this hierarchy includes many functions and procedures, several of which are virtual (in the sense that the user can modify them to polymorph the main object into one more specific for his problem). They are listed in Appendix A, together with a brief description of each of them. Now, some of these virtual functions and procedures must be written by the user since they correspond to basic elements required to deal with the problem:

1. the meaning of $x_{ij} = 1$: since the techniques indicate that item i is assigned to resource j by fixing $x_{ij} = 1$, the user has to indicate how this information translates into its own data structures.
2. $F(x)$ and $G_k(x)$: given a current solution x, the interface obtains the values of $F(x)$ and $G_k(x)$ evaluated within the user data structures.

Whenever possible, it is also strongly recommended to rewrite some of the other virtual functions to evaluate $Ir(x)$ and $\Delta P = P(x') - P(x)$ to increase the performance of the techniques. Knowing $F(x)$ and $G_k(x)$, these values can be computed. But referring to the user data structure, the incidence of moving from x to x' over Ir and F (i.e., $\Delta Ir = Ir(x') - Ir(x)$ and $\Delta F = F(x') - F(x)$) might be evaluated more efficiently to reduce the computation time required.

The performance of the techniques can be increased further if we use more specifically the user data structures to specify J_i for all i, $1 \leq i \leq n$ and the set of interesting items associated with the current solution x. By default (i.e., using the functions in ATP_INTERFACE) these are specified using the values of $F(x)$ and $G_k(x)$.

The rest of the main object ATP_INTERFACE is not modified in general. These functions and procedures are used to save and to get information about the size of the problem, the parameters, the current solution and ISBEST. They also include basic reading and writing procedures.

4.2. Polymorphism of the ATP_INTERFACE

We now illustrate the polymorphism for the three examples GCP, GPP, and VRP.

4.2.1. GCP_INTERFACE

For the GCP, $x_{ij} = 1$ if and only if vertex i has color j. Referring to Section 1, $F(x) = 0$, and $G_k(x)$ is simply the number of edges with both end-points having the same color k.

$Ir(x)$ is equal to the number of edges with both endpoints having the same color. The value $\Delta Ir = Ir(x') - Ir(x)$ can be evaluated more efficiently than computing it directly using the functions $G_k(x)$, $1 \leq k \leq K$. Indeed, assuming that $x' = x \oplus (i, j)$, then ΔIr is equal to the number of vertices of color j adjacent to i minus the number of vertices of color $R(x, i)$ adjacent to i. Hence, this approach requires $O(n)$ operations to evaluate ΔIr rather than $O(n^2)$ when using the functions $G_k(x)$, $1 \leq k \leq K$.

The set of interesting items corresponds to the set of vertices having at least one adjacent vertex of the same color. Hence, this set is easy to update as follows. Assuming again that $x' = x \oplus (i, j)$, a vertex of color $R(x, i)$ adjacent to i remains in the set of interesting items if it has another adjacent vertex (different from i) of color $R(x, i)$. Also, any vertex of color j adjacent to i becomes interesting, if is not already. Finally, vertex i remains interesting if it has at least one adjacent vertex of color j.

To illustrate how to generate a GCP_INTERFACE, assume that the Boolean $n \times n$ square matrix A is the adjacency matrix associated with a graph $G = (V, E)$; i.e., for $1 \leq i, j \leq n$

$$a[i, j] = \begin{cases} \text{true} & \text{if vertex } i \text{ is adjacent to vertex } j \\ \text{false} & \text{otherwise.} \end{cases}$$

Furthermore, let color be an n-vector indicating the color of the vertices; i.e., for $1 \leq i \leq n$,

color[i] = color assigned to vertex i .

The interface can be specified by rewriting the type-1 procedures and functions described in Appendix A as follows.

```
Procedure GetCurrentAssignment (i: integer; var j: integer);
begin
  j: = color[i]
end;

Procedure SetCurrentAssignment (i: integer; j: integer);
begin
```

```
  color[i]: = j
end;

Function ComputeFx: real;
begin
  ComputeFx: = 0.0
end;

Function ComputeGk(k: integer): real;
begin
  ComputeGk: = 0.0;
  for i: = 1 to n − 1 do
    if color[i] = k then
      for j: = i + 1 to n do
        if (color[j] = k) and a[i, j] then ComputeGk: =
          ComputeGk + 1
end;
```

The efficiency of the techniques can be even improved by rewriting the type-2 function to evaluate ΔIr:

```
Function ComputeDeltaIr(i: integer; j: integer): real;
begin
  ComputeDeltaIr: = 0.0;
  for v: = 1 to n do
    if a[v, i] then
    begin
      if color[v] = color[i], then ComputeDeltaIr: =
        ComputeDeltaIr − 1;
      if color[v] = j, then ComputeDeltaIr: =
        ComputeDeltaIr + 1;
    end;
end;
```

4.2.2. GPP_INTERFACE

For the GPP, the two parts are the resources, and $x_{ij} = 1$ if and only if vertex i belongs to part j. Referring to the formulation of GPP in Section 1, $F(x)$ corresponds to the total cost of the edges with end-points in different parts, and $G_k(x)$ corresponds to the difference between the number of vertices in part $k (k = 1, 2)$ and $|V|/2$.

With this formulation, $Ir(x)$ corresponds to the number of vertices exceeding $|V|/2$ in one of the parts. Furthermore, whenever one part V_k ($k = 1$ or 2) has more vertices than the other, then the interesting items correspond to the vertices in this part. If both parts have the same number of vertices, then any vertex having at least one adjacent vertex in the other part is an interesting item. Hence, balancing the number of vertices between the parts has priority over reducing the total cost of the edges with end-points in both parts.

Now, consider the following alternate formulation:

$$F(x) = \sum_{(i,j)\in E} c_{ij}(x_{i1}x_{j2} + x_{i2}x_{j1}) + \rho \sum_{k=1}^{2} \left| \sum_{i=1}^{n} x_{ik} - \frac{|V|}{2} \right|$$

with no additional side constraints. Then each vertex belonging to the part with more elements is an interesting vertex. A vertex in the other part is interesting if the sum of the costs of edges linking it with vertices in the first part is larger than ρ.

According to the user's priority in dealing with the graph partitioning problem, this alternate formulation may be more appropriate. Nevertheless, the software can deal with either of the two formulations referring to the same user data structures.

Now we illustrate the GPP_INTERFACE associated with the formulation in Section 1. Assume that the Boolean $n \times n$ square matrix A is the adjacency matrix associated with a graph $G = (V, E)$; i.e.,

$$a[i, j] = \begin{cases} \text{true} & \text{if vertex } i \text{ is adjacent to vertex } j \\ \text{false} & \text{otherwise.} \end{cases}$$

For each edge $[i, j] \in E$,

Cost [i, j]=cost of the edge linking i and j .

Furthermore, for each vertex $i \in V$, denote

$$\text{WhichPart}\,[i] = \begin{cases} 1 & \text{if i}\in V_1 \\ 2 & \text{if i}\in V_2 \end{cases}.$$

The interface is specified by rewriting the type-1 procedures and functions described in Appendix A.

```
Procedure GetCurrentAssignment (i: integer; var j: integer);
begin
  j: = WhichPart[i]
end;

Procedure SetCurrentAssignment (i: integer; j: integer);
begin
  WhichPart[i]: = j
end;

Function ComputeFx: real;
begin
  ComputeFx: = 0.0;
  for i: = 1 to n − 1 do
    for j: = i + 1 to n do
      if (WhichPart[i] ≠ WhichPart[j]) and a[i, j], then
        ComputeFx: = ComputeFx + Cost[i, j];
end;

Function ComputeGk (k: integer): real;
begin
  ComputeGk: = 0.0;
  for i: = 1 to n do
    if WhichPart[i] = k, then ComputeGk: = Com-
      puteGk + 1;
  ComputeGk: = ComputeGk − (n div 2);
end;
```

To use the alternate formulation of the GPP just given, it is sufficient to redefine the functions ComputeFx and ComputeGk as follows:

```
Function ComputeFx: real;
begin
  ComputeFx: = 0.0; Counter: = 0;
```

```
  for i: = 1 to n do
  begin
    if WhichPart[i] = 1, then Counter: = Counter + 1;
    for j: = i + 1 n do
      if (WhichPart[i] ≠ WhichPart[j]) and a[i, j], then
        ComputeFx: = ComputeFx + Cost[i, j];
  end;
  ComputeFx: = ComputeFx + abs (Counter − n div 2)
    + abs(n − Counter − n div 2);
end;

Function ComputeGk(k: integer): real;
begin
  ComputeGk = 0.0
end;
```

4.2.3. VRP_INTERFACE

For the capacitated vehicle routing problem, $x_{ij} = 1$ if and only if city i is served by vehicle j. $G_j(x)$ corresponds to the difference between the amount of goods transported by vehicle j and its capacity c_j. Furthermore, $Ir(x)$ is equal to the sum of exceeding loads of vehicles whose load exceeds their capacity.

The objective function $F(x)$ for the VRP is an example of a calculable function which is not straightforward to evaluate. Indeed, to compute the exact value of $F(x)$, a traveling salesman problem has to be solved for each vehicle. Instead, to reduce computation time, we use an estimation of $F(x)$ generated as follows. Given an assignment x, assume that for each city i we know its successor and its predecessor in the route $R(x, i)$ serving it. Then $F(x)$ is estimated as the sum of the route lengths for visiting the cities in the order specified by the corresponding predecessor-successor structure. (Note that applying a 2-opt procedure or any other optimization technique to evaluate the length of each route can improve the value of $F(x)$ in some cases.) Furthermore, if $x' = x \oplus (i, j)$ (implying that city i is now served by vehicle j instead of $R(x, i)$), the successor-predecessor structure of routes j and $R(x, i)$ are updated as follows:

- for route j, a shortest detour insertion procedure (Bodin et al. 1983) is used to include city i;
- for route $R(x, i)$, the predecessor of i is linked directly to the successor of i instead of using i as an intermediate city.

Finally, the interesting items correspond to the cities served by vehicles whose load exceeds their capacity. Whenever the capacity constraint is satisfied for all vehicles, then all cities are interesting items.

To generate a VRP_INTERFACE, assume that for each vehicle j ($1 \leq j \leq m$),

Capacity $[j]$ = capacity of the j-th vehicle ,

and for each city i ($1 \leq i \leq n$),

WhichVehicle $[i]$ = index of the vehicle serving city i .

Furthermore, assume that the following functions and procedures are provided by the user:

- **Function** RouteLength (j: integer): real; to evaluate the route length of vehicle j according to the current predecessor-successor structure;
- **Function** VehicleLoad (j: integer): integer; to evaluate the sum of the city demands served by vehicle j.
- **Procedure** Remove (i: integer); to update the predecessor-successor structure of route WhichVehicle[i] by linking the predecessor of i to its successor.
- **Procedure** Insert (i: integer; j: integer); to update the predecessor-successor structure of route j by inserting i according to the shortest detour insertion policy.

The interface is specified by rewriting the type-1 procedures and functions described in Appendix A.

```
Procedure GetCurrentAssignment (i: integer; var j: integer);
begin
  j: = WhichVehicle[i]
end;

Procedure SetCurrentAssignment (i: integer; j: integer);
begin
  Remove (i);
  Insert (i, j);
end;

Function ComputeFx: real;
begin
  ComputeFx: = 0.0;
  j: = 1 to m do ComputeFx: = ComputeFx + Route-
    Length (j);
end;

Function ComputeGk(k: integer): real;
begin
  ComputeGk: = VehicleLoad (k) − Capacity[k];
end;
```

4.3. NST Hierarchy

The NST hierarchy contains four different objects NST_DM, NST_TS, NST_EP, and NST_SA inheriting from a main object NST_ALGORITHM. These objects are described in Appendix B. This hierarchy calls on the **ATP** hierarchy to interact with a specific problem.

The main object NST_ALGORITHM is used mainly to set the common parameters of the techniques, to control the time elapsed and the common stopping criteria. Each of the other four objects correspond to an implementation of a specific technique described in subsection 2.2.

Any of these techniques can be used as they are provided in their respective objects to solve any **ATP** for which an ATP_INTERFACE has been specified. Furthermore, these objects include several virtual functions and procedures allowing a user to generate a polymorphism corresponding to any variant that he would like to implement.

However, the level of technicality is too high and alternate polymorphisms are too numerous to pursue this issue any further in this paper. Any interested user can contact any of the authors to get a copy of the software to experiment with his own variants.

4.4. Pseudocode Example

To illustrate how the software can be used, consider the case of a GCP to be solved with the tabu search technique. Hence, given the GCP_INTERFACE in subsection 4.2.1 and the NST_INTERFACE, the pseudocode to deal with the problem reduces to the following:

```
Program example;
var
  interface:  GCP_INTERFACE;
  technique:  NST_TS
begin
  interface.init (n, m, K);
  technique.init (interface, MA, BI, MAXTIME, MI, NV,
                  TIMIN, TIMAX, TRMIN, TRMAX,
                  TIRMIN, TIRMAX);
  technique.ProcessAlgorithm;
  technique.done;
end.
```

5. EXTENSIONS

This work is a first step toward developing easy to use software to deal with classes of combinatorial problems. The current version dealing with assignment-type problems could be modified and extended to deal with more complex basic problems. For instance, instead of treating the resource assignment constraints of an assignment problem (AP) as side constraints, they could be included in the basic structure to generate the class of assignment problems with additional constraints (Mazzola and Neebe 1984). Then, it may be more efficient to use neighborhood search techniques specifically implemented for this class of problems rather than using the current software. But the neighborhood structure would be more complex since instead of simply modifying the resource of an item, we would have to exchange the resources of two items to maintain feasibility. Hence, the tradeoff between this increase of complexity to maintain feasibility and inefficiency of the techniques has to be analyzed to evaluate the gain (or loss) in the overall efficiency of the procedures.

Another extension would be to introduce a third hierarchy calling on NST_ALGORITHM to control the parameters of the techniques during the execution or to control how to use hybrids of the four techniques and hybrids of these with other approaches to solve ATPs. For instance, the variations of the parameters α and β could be controlled via Lagrangian relaxation. Another example would be to generate a hybrid where each individual of a population of solutions controlled by a genetic algorithm is improved with one or more of the four techniques. These are only a few possible extensions. We hope that this contribution will become the starting point for further research to provide general purpose software implementing heuristic techniques to solve combinatorial optimization problems.

APPENDIX A: ATP_INTERFACE

Here is a list of the functions and procedures of this interface together with an indication whether they are virtual or not, and a brief description of them. They are grouped in different categories according to the benefit of rewriting them to improve the performance.

Type-1 Procedures and functions that have to be written by the user

1.1 **Procedure** GetCurrentAssignment (*i*: integer; var *j*: integer); -virtual-
Procedure to obtain the current assignment j of an item i from the user data structures.

1.2 **Procedure** SetCurrent Assignment (*i*: integer; *j*: integer); -virtual-
Procedure to transfer the current assignment j of an item i to the user data structures.

1.3 **Function** Compute*Fx*: real; -virtual-
Function to obtain the current value of $F(x)$ from the user data structures.

1.4 **Function** Compute*Gk* (*k*: integer): real; -virtual-
Function to obtain the current value of $G_k(x)$ from the user data structures.

Type-2 Functions strongly recommended to be rewritten by the user

2.1 **Function** Compute *Irx*: real; -virtual-
Function to compute the value of $Ir(x)$ for the current solution.

2.2 **Function** ComputeDelta*Fx* (*i*: integer; *j*: integer): real; -virtual-
Function to compute the modification of F induced by the move from x to $x' = x \oplus (i, j)$.

2.3 **Function** ComputeDelta*Irk* (*k*: integer; *i*: integer; *j*: integer): real; -virtual-
Function to compute the modification of the violation of the k-th constraint induced by the move from x to $x' = x \oplus (i, j)$.

2.4 **Function** ComputeDelta*Ir* (*i*: integer; *j*: integer): real; -virtual-
Function to compute the modification of Ir induced by the move from x to $x' = x \oplus (i, j)$.

Type-3 Procedures and functions recommended to be rewritten by the user

3.1 **Function** Isbest solution (*Fx*: real; *Irx*: real): boolean -virtual-
Logical function to test if $x <$ ISBEST, where $Fx = F(x)$ and $Irx = Ir(x)$.

3.2 **Function** InterestingItem (*i*: integer): boolean -virtual-
Logical function to indicate if item *i* is interesting.

3.3 **Procedure** CandidateResourceList (*i*: integer; *Ji*: Resource_List); -virtual-
Procedure to determine the set J_i for an item *i*.

Type-4 Procedures that may be rewritten by the user

4.1 **Procedure** Initx; -virtual-
Procedure to determine the initial solution. By default, it is randomly generated.

4.2 **Procedure** GetCurrentSolution (var *x*: Solution); -virtual-
Procedure to obtain the current solution *x* from the user data structures.

4.3 **Procedure** SetCurrentSolution (*x*: Solution); -virtual-
Procedure to transfer the current solution to the user data structures.

4.4 **Procedure** InterestingItemList (var *INNx*: ResourceList); -virtual-
Procedure to specify the list $INN(x)$ of interesting items associated with *x*.

Type-5 Functions that should not be rewritten by the user

5.1 **Function** Compute*Px*: real; -virtual-
Function to compute the value $P(x) = \alpha F(x) + \beta Ir(x)$ of the current solution *x*.

5.2 **Function** ComputeDelta*Px* (*i*: integer; *j*: integer): real; -virtual-
Function to compute the modification of $P(x)$ induced by the move from x to $x' = x \oplus (i, j)$.

Type-6 Procedures and functions that cannot be modified

6.1 **Procedure** SaveBestSolution;
Procedure to save the ISBEST solution.

6.2 **Procedure** RestoreBestSolution;
Procedure to set the current solution equal to ISBEST.

6.3 **Procedure** SetAlphaBeta (alpha: real; beta: real);
Procedure to modify the values of α and β.

6.4 **Procedure** GetAlphaBeta (var alpha: real; var beta: real);
Procedure to obtain the current values of α and β.

6.5 **Function** GetNbItems: integer;
Function to obtain the number of items.

6.6 **Function** GetNbResources: integer;
Function to obtain the number of resources.

6.7 **Function** GetNbConstraints: integer;
Function to obtain the number of constraints.

6.8 **Function** GetCurrent*Fx*: real;
Function to obtain the current value of $F(x)$.

6.9 **Function** GetCurrent*Irx*: real;
Function to obtain the current value of $Ir(x)$.

6.10 **Function** GetCurrent*Px*: real;
Function to obtain the current value of $P(x)$.

6.11 **Function** GetBest*Fx*: real;
Function to obtain F(ISBEST).

6.12 **Function** GetBest*Irx*: real;
Function to obtain Ir(ISBEST).

6.13 **Function** GetBest*Px*: real;
Function to obtain P(ISBEST).

6.14 **Procedure** WriteSolution (Filename: string);
Procedure to save the current solution in the file FileName.

6.15 **Procedure** ReadSolution (Filename: string);
Procedure to read a solution in the file FileName and set the current solution to this seed solution.

APPENDIX B: NST HIERARCHY

In this section we list only the virtual functions and procedures of the objects in this hierarchy that could be modified by an advanced user. The remaining elements of these objects are not to be modified since they correspond to the basis of these techniques.

B1. NST_ALGORITHM

Procedure ProcessAlgorithm; -virtual-
Abstract algorithm.

Function StopAlgorithm: Boolean; -virtual-
Boolean function to verify if one of the stopping criteria is satisfied.

Function Asp: Boolean; -virtual-
Boolean function indicating if the aspiration test is verified (including only the test $x' <$ ISBEST at this level and to be completed later for each technique).

Procedure NextNeighbor (var *i*: integer; var *j*: integer); -virtual-
Procedure to generate the next element $x' = x \oplus (i, j)$ of INN(x).

B2. NST_DM

Procedure ProcessAlgorithm; -virtual-
Procedure to implement the Descent Method of Figure 1.

Function ASP: Boolean; -virtual-
Boolean function indicating if ASP_DM (x') is verified in Step 2.1 of Figure 1.

Procedure NextNeighbor (var i: integer; j: integer); -virtual-
Procedure to generate the next element $x' = x \oplus (i, j)$ of INN(x) in Step 2.1 of Figure 1.

Function StopAlgorithm: Boolean; -virtual-
Boolean function to verify if one of the common stopping criteria is satisfied or if QUIT = true in Step 2.2 of Figure 1.

B3. NST_TS

Procedure ProcessAlgorithm; -virtual-
Procedure to implement the tabu search technique of Figure 2.

Function ASP: Boolean; -virtual-
Boolean function indicating if ASP_TS(z) is verified in Step 2.1.2 of Figure 2.

Procedure NextNeighbor (var i: integer; var j: integer); -virtual-
Procedure to generate the next element $x' = x \oplus (i, j)$ of INN(x) in Steps 2.1 and 2.2 of Figure 2.

Function StopAlgorithm: Boolean; -virtual-
Boolean function to verify if one of the common stopping criteria is satisfied or if QUIT = true in Step 2.8 of Figure 2.

Function Is Tabu (i: integer; j: integer): boolean; -virtual-
Boolean function indicating if the move from x to $z = x \oplus (i, j)$ is tabu in Step 2.1.3 of Figure 2.

Procedure AddTabu (i: integer; j: integer); -virtual-
Procedure to update the tabu lists in step 2.3 of Figure 2.

B4. NST_EP

Procedure ProcessAlgorithm; -virtual-
Procedure to implement the exchange procedure of Figure 3.

Function ASP: Boolean; -virtual-
Boolean function indicating if ASP_EP (x') is verified in Steps 2.2, 2.2.4, and 2.2.5 of Figure 3.

Procedure NextNeighbor (var i: integer; var j: integer); -virtual-
Procedure to generate the next element $x' = x \oplus (i, j)$ of INN(x) in Step 2.2.2 of Figure 3.

Function StopAlgorithm: Boolean; -virtual-
Boolean function to verify if one of the common stopping criteria is satisfied or if QUIT = true in Step 2.2.1 of Figure 3.

Function IsForwardMove (i: integer; j: integer): Boolean; -virtual-
Boolean function indicating if a forward move from x to $x' = x \oplus (i, j)$ has to be completed in Step 2.2.5 of Figure 3.

B5. NST_SA

Procedure ProcessAlgorithm; -virtual-
Procedure to implement the simulated annealing technique of Figure 4.

Function ASP: Boolean; -virtual-
Boolean function indicating if ASP_SA (x') is verified in Step 2.1.3 of Figure 4.

Procedure NextNeighbor (var i: integer; var j: integer); -virtual
Procedure to generate the next element $x' = x \oplus (i, j)$ of INN(x) in Step 2.1.1 of Figure 4.

Function StopAlgorithm: Boolean; -virtual-
Boolean function to verify if one of the common stopping criteria is satisfied or if QUIT = true in Step 2.3 of Figure 4.

Procedure ModifyTemp; -virtual-
Procedure to modify the temperature in Step 2.2 of Figure 4.

ACKNOWLEDGMENT

This research was supported by an NSERC grant (A8312) from Canada, an FCAR grant (93ER1654) from Québec, an NSERC grant (BEFTR106) from Canada, and an FNRS fellowship (83NC-037029) from Switzerland.

REFERENCES

BODIN, L. D., B. L. GOLDEN, A. A. ASSAD AND M. O. BALL. 1983. Routing and Scheduling of Vehicles and Crews: The State of the Art. *Comput. Opns. Res.* **10,** 69–211.

BOOCH, G. 1991. Object Oriented Design With Applications. Benjamin-Cammings, Redwood City, Calif.

BORLAND INTERNATIONAL INC. 1988. Turbo Pascal Version 5.5, Object-Oriented Programming Guide, Scotts Valley, Calif.

FERLAND, J. A., AND A. LAVOIE. 1992. Exchanges Procedures for Timetabling Problems. *Discr. Appl. Math.* **35,** 237–253.

GAREY, M. R., D. S. JOHNSON AND L. STOCKMEYER. 1976. Some Simplified NP-Complete Graph Problems. *Theoret. Comput. Sci.* **1,** 237–267.

GENDREAU, M., A. HERTZ AND G. LAPORTE. 1994. A Tabu Search Heuristic for the Vehicle Routing Problem. *Mgmt. Sci.* **40,** 1276–1290.

GLOVER, F. 1986. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. and Opns. Res.* **13,** 533–549.

GLOVER, F. 1989. Tabu Search, Part I. *ORSA J. Comput.* **1,** 190–206.

GLOVER, F. 1990. Tabu Search, Part II. *ORSA J. Comput.* **2,** 4–32.

GLOVER, F., AND M. LAGUNA. 1993. Tabu Search. In *Modern Heuristic Techniques for Combinatorial Problems*. C. Reeves, (ed.). Blackwell Scientific Publishing, 70–150.

GREEN, J. M., AND K. J. SUPOWIT. 1986. Simulated Annealing Without Rejecting Moves. *IEEE Trans. Comp. Aid. Des.* **CAD-5,** 221–228.

HAJEK, B. 1988. Cooling Schedules for Optimal Annealing. *Math. O.R.* **13,** 311–329.

HANSEN, P. 1986. The Steepest Ascent, Mildest Descent Heuristic for Combinatorial Programming. Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy.

HERTZ, A., AND D. DE WERRA. 1987. Using Tabu Search Techniques for Graph Coloring. *Computing* **39,** 345–351.

JOHNSON, D. S., C. R. ARAGON, L. A. MCGEOCH AND C. SHEVON. 1989. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Opns. Res.* **37,** 865–892.

JOHNSON, D. S., C. R. ARAGON, L. A. MCGEOCH AND C. SHEVON. 1991. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Opns. Res.* **39,** 378–406.

KIRKPATRICK, S., C. D. GELATT AND M. P. VECCHI. 1983. Optimization by Simulated Annealing. *Science* **220,** 671–680.

MAZZOLA, J. B., AND A. W. NEEBE. 1986. Resource Constrained Assignment Scheduling. *Opns. Res.* **34,** 560–572.

MEYER, B. 1988. Object Oriented Software Construction. Prentice-Hall, Englewood Cliffs, N.J.

NOWICKI, E., AND C. SMUTNICKI. 1993. A Fast Tabu Search Algorithm for the Job Shop Problem. *ORSA J. Comput.* (to appear).

NOWICKI, E., AND C. SMUTNICKI. 1994. A Fast Tabu Search Algorithm for the Flow Shop Problem. to appear in *Eur. J. Opnl. Res.* (to appear).

ROSS, C. T., AND R. M. SOLAND. 1975. A Branch and Bound Algorithm for the Generalized Assignment Problem. *Math. Prog.* **8,** 91–103.