Pergamon

Computers Ops Res. Vol. 25, No. 11, pp. 969–979, 1998

Printed in Great Britain
PII: S0305-0548(97)00093-2
0305-0548/98 $19.00 + 0.00

A LOCAL SEARCH TEMPLATE

R. J. M. Vaessens,[1]‡ E. H. L. Aarts[1,2]†§ and J. K. Lenstra[1,3]¶

[1]Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[2]Philips Research Laboratories, P.O. Box 80000, 5600 JA Eindhoven, The Netherlands
[3]CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

(Received May 1996; in revised form October 1997)

Scope and Purpose—Many problems in combinatorial optimization are NP-hard which implies that it is generally believed that no algorithms exist that solve each instance of such a problem to optimality using a running time that can be bounded by a polynomial in the instance size. As a consequence, much effort has been devoted to the design and analysis of algorithms that can find high quality approximative solutions in reasonable, i.e. polynomial, running times. Many of these algorithms apply some kind of neighborhood search and over the years a great variety of such local search algorithms have been proposed, applying different kinds of search strategies often inspired by optimization processes observed in nature. The purpose of this paper is to capture within a single template the common features of the best-known local search algorithms. The template is also used for the purpose of indicating directions for novel algorithmic ideas in the area of local search.

Abstract—A template is presented that captures a vast majority of the local search algorithms proposed in the literature, including iterative improvement, simulated annealing, threshold accepting, tabu search and genetic algorithms. The template leads to a classification of existing local search algorithms and offers the possibility to fit in new types of local search approaches.

Key words: Local search, iterative improvement, simulated annealing, threshold accepting, tabu search, genetic algorithms

1. INTRODUCTION

Local search is a generally applicable approach that can be used to find approximate solutions to hard optimization problems. The basic idea is to start from an initial solution and to search for successive improvements by examining neighboring solutions. Local search approaches date back to the late 1950s, when Bock [3] and Croes [4] developed their link exchange procedures for the traveling salesman problem. Ever since, a large variety of local search algorithms has been proposed, each aiming at different remedies to the risk of getting stuck in poor local optima.

At present, there is a proliferation of local search algorithms or, as they are often called, *metaheuristics* [15]. Since all of these methods apply some kind of neighborhood search technique, we will simply call them local search algorithms, following the early textbook treatment by Papadimitriou and Steiglitz [16], the survey by Yannakakis [21] and the book edited by Aarts and Lenstra [2].

In all their different guises, local search algorithms seem to be based on a few basic ideas only. We present a local search template that has been designed to capture most of the variants

†To whom all correspondence should be addressed.
‡Rob Vaessens received his M.Sc. from the Eindhoven University of Technology and his Ph.D. from the Eindhoven University of Technology. His technical interests are Job Scheduling: Complexity and Local Search. He is currently employed as revenue management analyst for the Royal Dutch Airlines KLM in Amstelveen, The Netherlands.
§Emile H. L. Aarts is a Senior Scientist at the Philips Research Laboratories in Eindhoven, The Netherlands. He also holds an appointment as a Professor of Computer Science at the Eindhoven University of Technology. He holds an M.Sc. degree in physics from the University of Nijmegen, The Netherlands, and a Ph.D. degree from the University of Groningen, The Netherlands. His research field is combinatorial optimization in planning and design.
¶Jan Karel Lenstra is Professor of Optimization at Eindhoven University of Technology. His research interests are in combinatorial Optimization, in particular routing, scheduling, complexity, and approximation algorithms. He is Past Chairman of the Mathematical Programming Society and Editor-in-Chief of Mathematics of Operations Research.

proposed in the literature. The aim of the template is to provide a classification of the various existing local search algorithms. Furthermore, it should also be sufficiently general to capture new approaches to local search and thereby to suggest novel variants.

The organization of the paper is as follows. Section 2 introduces some basic definitions. Section 3 discusses some complexity issues. Section 4 considers deterministic iterative improvement algorithms. Section 5 presents our local search template and Section 6 shows for a number of well-known local search algorithms how they fit into this template. Section 7 mentions some lesser known or new algorithms that fit into our template. Section 8, finally, presents some conclusions and suggestions for future research.

2. PRELIMINARIES

An *optimization problem* is either a maximization or a minimization problem specified by a class of problem instances. Without loss of generality we restrict ourselves to minimization problems. An *instance* is defined by the implicit specification of a pair $(\mathcal{S}, f)$, where the *solution space* $\mathcal{S}$ is the set of all (feasible) solutions and the *cost function* f is a mapping $f:\mathcal{S}\rightarrow\mathrm{R}$. The optimal cost f_{opt} of an instance is defined by $f_{\mathrm{opt}}=\min\{f(s)|s\in\mathcal{S}\}$ and the set of optimal solutions is denoted by $\mathcal{S}_{\mathrm{opt}}=\{s\in\mathcal{S}|f(s)=f_{\mathrm{opt}}\}$. The objective is to find some solution $s_{\mathrm{opt}}\in\mathcal{S}_{\mathrm{opt}}$.

A *neighborhood function* $\mathcal{N}$ is a mapping $\mathcal{N}:\mathcal{S}\rightarrow\mathcal{P}(\mathcal{S})$, which specifies for each $s\in\mathcal{S}$ a subset $\mathcal{N}(s)$ of $\mathcal{S}$ of neighbors of s. A solution $s\in\mathcal{S}$ is called a *local minimum with respect to* $\mathcal{N}$ if $f(s)\leq f(t)$ for all $t\in\mathcal{N}(s)$. Furthermore, to distinguish between local minima and elements of $\mathcal{S}_{\mathrm{opt}}$, we call the latter ones *global* minima. A neighborhood function $\mathcal{N}$ is called *exact* if every local minimum with respect to $\mathcal{N}$ is also a global minimum.

In some applications of local search algorithms to optimization problems the search is not done directly in the solution space $\mathcal{S}$, but in a search space, in which solutions are represented in one way or another.

For problems with very few feasible solutions or for which it is not even known whether feasible solutions exist, the search space may also contain infeasible solutions in addition to feasible ones. In this case, the cost function f should be extended in such a way that the a solution of minimum cost corresponds to a feasible solution. The same is true if one wants to include partial solutions in the search space.

Another possibility is that some solutions are represented by more than one element in the search space. This occurs very often in applications of genetic algorithms, which are discussed in Section 6.3. Here, solutions are represented by strings over a finite alphabet, and several such strings may represent the same solution.

However, since each problem described by a non-trivial representation can be seen as an optimization problem of its own, we consider in the remainder of this paper only problems of the form introduced at the beginning of this section.

3. THE COMPLEXITY OF LOCAL SEARCH

Complexity analyses have reveiled a marked difference between the empirical and theoretical performance of local search.

3.1. Empirical results

Over the years the empirical performance of local search has been extensively studied for a large variety of problems. A general conclusion is that local search provides a robust approach to obtain good solutions to problems of a realistic size within low order polynomial running times [2, 15]. This conclusion has been reached in studies for problems of a theoretical as well as practical origin. For instance, for the traveling salesman and job shop scheduling problems, local search algorithms have been shown to be the best approximation algorithms from an empirical point of view and it appears that for large problem instances, the difference with other existing algorithms becomes even more pronounced [10, 19]. For many practical problems, for instance in the areas of VLSI design, distribution planning and production control, local search leads to high-quality solutions in a reasonable amount of time. In addition, its ease of use and

flexibility make it a valuable industrial problem solving tool, especially when compared to more classical constructive heuristics that are often tailored to specific problems and may require major modifications when minor changes in the problem setting occur.

3.2. Theoretical results

Studies of the theoretical performance have exhibited the limitations of local search, at least from a worst-case point of view. The literature presents a number of bad examples, for which the following results hold [17, 20].

- Minimum exact neighborhoods may be of exponential size.
- It may take an exponential number of steps to find a local optimum.
- Final solutions may deviate arbitrarily far from the optimum in cost.

Johnson *et al.* [11] addressed the question how easy it is to find a local optimum. They introduced several notions. A local search problem L is given by a set of instances, each of which is defined by a finite set of solutions, a cost function and a neighborhood function. This problem belongs to the complexity class $\mathscr{PLS}$ of "polynomial-time local search" problems, if polynomial-time algorithms exist, only depending on L, for producing an arbitrary solution, for computing the cost of a given solution and for determining for a given solution a neighbor that has lower cost (or reporting that no such neighbor exists). The problem now is to find a local minimum for any given instance. Informally speaking, $\mathscr{PLS}$ defines the class of local search problems for which local optimality can be verified in polynomial time. The class $\mathscr{PLS}$ is situated between the search problem variants of $\mathscr{P}$ and $\mathscr{NP}$. However, it has been shown that a problem in $\mathscr{PLS}$ cannot be NP-hard, unless $\mathscr{NP} = \text{co-}\mathscr{NP}$ [11].

Furthermore, the concept of a PLS-reduction has been introduced, which shows some resemblance to the classical concept of polynomial-time reductions [6]. A problem in $\mathscr{PLS}$ is PLS-complete if any problem in $\mathscr{PLS}$ is PLS-reducible to it. The PLS-complete problems are the hardest ones in $\mathscr{PLS}$ and if one of them can be shown to be solvable in polynomial time, then all the others can.

Since its introduction, the class $\mathscr{PLS}$ has received considerable attention and many local search problems have been proven PLS-complete. It is even conjectured that PLS-completeness is the normal behavior of local search variants of NP-hard problems and that more than half of the problems mentioned in Garey and Johnson's [6] NP-completeness catalogue have PLS-complete variants.

4. DETERMINISTIC ITERATIVE IMPROVEMENT

The basic local search algorithm is the so-called deterministic iterative improvement algorithm. We assume that an instance of an optimization problem and a neighborhood function are given. The deterministic iterative improvement algorithm starts with an initial solution and then continually searches its neighborhood for a solution of better quality. If such a neighbor is found, it replaces the current solution. The algorithm terminates as soon as the current solution has no neighbors of better quality, at which point a local optimum is found.

The pseudo-Pascal procedure given in Fig. 1 represents the basic part of the deterministic iterative improvement algorithm. Here, the procedure `GENERATE NEIGHBOR` deterministically generates a solution t from the neighborhood $\mathscr{N}(s)$ of the current solution s, such that every $t \in \mathscr{N}(s)$ is generated only once as a neighbor of s. The procedure `DETERMINISTIC ITERATIVE IMPROVEMENT` returns a solution s that is locally optimal with respect to the neighborhood function $\mathscr{N}$.

The deterministic iterative improvement algorithm terminates at the first local optimum that is found and, in general, the quality of such a local optimum may be arbitrarily bad. To improve the quality one can consider applying the following ideas.

- Generating several or all neighbors of the current solution instead of just one neighbor in each iteration. If all neighbors are generated and a best one is accepted, one obtains a *steepest descent or best improvement* algorithm.
- Repeatedly starting iterative improvement runs from different (e.g. randomly chosen) initial solutions. This is often referred to as *multistart iterative improvement* [2].

```
procedure DETERMINISTIC ITERATIVE IMPROVEMENT (s ∈ S);
  { input:  s ∈ S, N
    output: s ∈ S, s locally optimal with respect to N }
  begin
    repeat
      GENERATE NEIGHBOR (s, t, N);
      if c(t) < c(s) then s := t
    until ∀t ∈ N(s) : f(t) ≥ f(s)
  end
```

Fig. 1. The procedure DETERMINISTIC ITERATIVE IMPROVEMENT.

- Using more intricate functions to determine a new solution from the current solution and its neighbor, for instance, accepting solutions of quality worse than that of the current solution. The well-known *simulated annealing, threshold accepting* and *tabu search* algorithms fall into this category.
- Replacing the single current solution by a population of current solutions. This is the basic idea of *genetic* algorithms.
- Alternating between two or more neighborhood functions. Such a *multilevel* algorithm has been proposed by Martin *et al.* [14].

Based on one or more of the above ideas, a considerable number of algorithms has been proposed in the literature. In Section 5 we present a generic local search template that captures most of these ideas.

5. A LOCAL SEARCH TEMPLATE

Our local search template generalizes the iterative improvement algorithm of Section 4 in the following ways.

(1) The search may proceed at several *levels*, each with its own specifications. Here, the notion of levels primarily refers to the subsequent use of different search strategies and neighborhoods.

(2) The single current solution is replaced by a *population* of current solutions.

(3) The neighborhood function associated with a single solution is replaced by a neighborhood function associated with a *cluster* of solutions.

More formally, each level l has a *population size* p_l and a cluster size c_l, with p_l, $c_l \in \mathbb{N}$. A *population* at level l is a p_l-tuple $P \in \mathcal{S}^{p_l}$ of solutions that represents the current state of the search at level l. We will talk about *point-based* local search if, at the first level, $p_1 = 1$ and about *population-based* local search otherwise. A *cluster* at level l is a c_l-tuple $C \in \mathcal{S}^{c_l}$ of solutions, such that with each cluster a neighborhood is associated. That is, there is a *hyper-neighborhood function* $\mathcal{N}_l : \mathcal{S}^{c_l} \to \mathcal{P}(\mathcal{S})$ which, for each cluster C, defines a set $\mathcal{N}_l(C)$ of neighboring solutions. In case $c_l = 1$, this function reduces to the standard neighborhood function of Section 2.

The local search template consists of two components. It first calls the procedure INITIALIZE, which generates an initial population $P \in \mathcal{S}^{p_1}$; this procedure usually depends on the problem type under consideration. Then, the recursive procedure LOCAL SEARCH is called with level 1 and population P as its parameters. LOCAL SEARCH works as follows.

At level l, the procedure takes a population $P \in \mathcal{S}^{p_l}$ as input and uses the hyper-neighborhood function $\mathcal{N}_l$ to produce a new population $P \in \mathcal{S}^{p_l}$ as output. This is done in two nested loops, i.e. an outer loop of *generations* and an inner loop of *iterations*.

The generation loop creates a number of generations of populations until a stopping condition is satisfied. In each generation, the procedure GENERATE CLUSTERS assembles from the current population P a finite multiset $\mathscr{C}$ of clusters $C \in \mathcal{S}^{c_l}$. Hence, each of the c_l components $C_1, \ldots, C_{c_l}$ of C is a solution from P.

For each cluster $C \in \mathscr{C}$, the iteration loop applies a number of iterations until a stopping criterion is satisfied. Each iteration starts with a call of the procedure GENERATE NEIGHBORS, which selects a finite multiset $Q \in (\mathcal{N}_l(C))^{p_{l+1}}$. Hence, each of the p_{l+1} components of Q is a

```
program LOCAL SEARCH TEMPLATE;
  begin
    INITIALIZE (P);
    LOCAL SEARCH (1, P)
  end;
```

Fig. 2. The LOCAL SEARCH TEMPLATE.

hyper-neighbor of C; note that $Q \in \mathcal{S}^{p_{l+1}}$. The procedure LOCAL SEARCH is then called recursively, with level $l+1$ and population Q as its parameters. The result is a modified population $Q \in \mathcal{S}^{p_{l+1}}$. After this, the procedure REDUCE NEIGHBORS reduces the union of the original cluster C and the new population Q into a new cluster $C \in \mathcal{S}^{c_l}$, which then serves as input for the next iteration.

If, at level l, the iteration loop has terminated for all $C \in \mathcal{C}$, the procedure CREATE collects the solutions found in (usually) the final iteration for each $C \in \mathcal{C}$ into a single set $\hat{P} \subseteq \mathcal{S}$. The procedure REDUCE POPULATION finally merges P and $\hat{P}$ into a new current population $P \in \mathcal{S}^{p_l}$.

Figure 2 shows the LOCAL SEARCH TEMPLATE and Fig. 3 shows the procedure LOCAL SEARCH, both in pseudo-Pascal. We now give a short description of the procedures and functions used in the procedure LOCAL SEARCH.

- The Boolean function CONTINUE POPULATION GENERATION has the current level l as input. Based on additional information from previous generations, it returns the value TRUE as long as new generations of populations have to be generated and FALSE otherwise.
- The procedure GENERATE CLUSTERS has a level l and a population P as input and a finite multiset $\mathcal{C} \subseteq \mathcal{S}^{c_l}$ as output. It clusters P into a collection $\mathcal{C}$ of c_l-tuples $C \in P^{c_l}$, either deterministically or probabilistically. In this way, the hyper-neighborhood function $\mathcal{N}_l$ can be applied indirectly to the given population P.
- The Boolean function CONTINUE ITERATION has a level l as input. Based on additional information from previous iterations it returns the value TRUE as long as iterations have to go on in the iteration loop and FALSE otherwise.
- The procedure GENERATE NEIGHBORS has a level l, a multiset C of size c_l and the hyper-neighborhood function $\mathcal{N}_l$ as input and a multiset Q of size p_{l+1} as output. It generates a

```
procedure LOCAL SEARCH (l: integer; P ∈ S^{p_l} );
  { input:  l ∈ ℕ, P ∈ S^{p_l}
    output: P ∈ S^{p_l}          }
  begin
    while CONTINUE POPULATION GENERATION (l) do
      begin
        GENERATE CLUSTERS (l, P, C);
        for all C ∈ C do
          begin
            while CONTINUE ITERATION (l) do
              begin
                GENERATE NEIGHBORS (l, C, N_l, Q);
                LOCAL SEARCH (l + 1, Q);
                REDUCE NEIGHBORS (l, C, Q)
              end
          end;
        CREATE (l, P̂);
        REDUCE POPULATION (l, P, P̂)
      end
  end;
```

Fig. 3. The procedure LOCAL SEARCH

multiset Q of neighbors from C using $\mathcal{N}_l$. The basic part of the procedure prescribes how a neighbor of C is to be determined, i.e. randomly or deterministically, and how many neighbors are to be determined.

• The procedure REDUCE NEIGHBORS has a level l, a c_l-tuple C and a p_{l+1}-tuple Q as input, and a modified version of C as output. It determines how to merge the old cluster C and the collection Q of (modified) neighbors into a new cluster C.

• The procedure CREATE has a level l as input and a population $\hat{P}$ as output. It puts a population $\hat{P}$ together from solutions found in (usually) the final iteration for each $C \in \mathscr{C}$.

• The procedure REDUCE POPULATION merges P and $\hat{P}$ into a new population P.

To make the recursive procedure finite, we need to define a bottom level l^*. At this level l^*, the Boolean function CONTINUE POPULATION GENERATION assumes the value FALSE. The levels $l < l^*$ are called *active* levels. Obviously, for the description of a local search algorithm only a specification of the active levels is needed. We know of no algorithms that use more than two active levels.

Section 6 shows how most local search algorithms proposed in the literature fit into our template.

6. INSTANTIATIONS OF THE LOCAL SEARCH TEMPLATE

The local search template captures most types of local search algorithms proposed in the literature. This is shown by the specification of the bottom level l^* and, for each active level, by an instantiation of the procedures GENERATE CLUSTERS, REDUCE NEIGHBORS, CREATE and REDUCE POPULATION. The other procedures are usually less characteristic of an algorithm; they are instantiated only if they constitute a relevant part of the algorithm.

In handling the various local search algorithms we distinguish between *point-based* and *population-based* local search and between local search with exactly one and more than one active level.

6.1. Single-level point-based local search

Among point-based local search algorithms with one active level, first the classes of *threshold* and *tabu search* algorithms are discussed. Next, *variable-depth search* is discussed.

6.1.1. Threshold algorithms and tabu search. Both threshold and tabu search algorithms are characterized by the fact that only one generation is created. Hence, they are determined by the iteration loop of the procedure LOCAL SEARCH. Both algorithms can be instantiated as follows.

• CONTINUE POPULATION GENERATION returns the value TRUE for the first generation and FALSE for each subsequent generation. In this way only one generation is created.

• GENERATE CLUSTERS sets $\mathscr{C} = C$ equal to P, hence the multi-set $\mathscr{C}$ contains only one cluster. Furthermore, both C and P are 1-tuples, that is, each of them contains one solution only.

• CREATE sets $\hat{P}$ equal to the current cluster C.

• REDUCE POPULATION sets the new population P equal to $\hat{P}$.

We now consider threshold and tabu search algorithms separately.

Threshold algorithms constitute a class of algorithms that contains iterative improvement, simulated annealing [1, 12] and threshold accepting [5]. They are characterized by the following instantiations.

• GENERATE NEIGHBORS generates only one neighbor in Q using the neighborhood function $\mathcal{N}_1$. In most cases a neighbor is generated randomly; sometimes this is done deterministically.

• REDUCE NEIGHBORS determines whether the solution Q_1, which is the unique component of Q, satisfies $f(Q_1) - f(C_1) < t$ for a certain threshold value $t \geq 0$, where C_1 denotes the first (and only) component of C. If this is the case, Q_1 replaces the current solution C_1; otherwise, C_1 remains unchanged. Depending on the nature of the thresholds, one distinguishes between several types of threshold algorithms. Iterative improvement and threshold accepting both use deterministic thresholds, in contrast to simulated annealing, which uses probabilistic thresholds. The class of iterative improvement algorithms includes the deterministic iterative improvement algorithm introduced in Section 4 as a special case.

• CONTINUE ITERATION returns the value TRUE as long as the best solution found so far is not of a prescribed quality, or as long as a prescribed number of iterations has not yet been reached. Otherwise it returns the value FALSE.

Tabu search [7] combines the deterministic iterative improvement algorithm with a possibility to accept cost increasing solutions. In this way the search is directed away from local minima, such that other parts of the search space can be explored. This is done by keeping up a finite list of solutions that are not acceptable in the next few iterations. This list is called the *tabu list*. However, a solution on the tabu list may be accepted if its quality is in some sense good enough, in which case it is said to attain a certain *aspiration level*. Tabu search algorithms are characterized by the following instantiations.

• GENERATE NEIGHBORS selects deterministically all neighbors of the current solution C_1 with respect to $\mathcal{N}_1$ by inspecting these in a prespecified order.

• REDUCE NEIGHBORS determines among all solutions in Q that are not on the tabu list, and all solutions in Q that are on the tabu list but attain a certain aspiration level, a solution $Q_j \neq C_1$ of minimum cost. C_1 is then replaced by Q_j.

• CONTINUE ITERATION returns the value TRUE as long as the best solution found so far is not of a prescribed quality, or as long as a prescribed number of iterations has not yet been reached. Furthermore, when the tabu list contains all neighbors of C_1 and none of these attains the aspiration level, the function CONTINUE ITERATION returns the value FALSE.

When the tabu list contains all neighbors of a current cluster C_1 and none of these attains the aspiration level, it is impossible to determine a neighbor of C_1. Some variants of tabu search solve this problem by letting the function CONTINUE ITERATION return the value FALSE. Other variants modify the tabu list in such a way that neighbors of the current cluster are removed from the tabu list. In this way neighbors of C_1 become available again.

In other variants, the procedure GENERATE NEIGHBORS selects only one neighbor per iteration and the function REDUCE NEIGHBORS accepts this neighbor when it is not on the tabu list or attains the aspiration level, and rejects it otherwise. But in that case the tabu list has to be significantly larger, so as to avoid that the procedure accepts a solution with a cost larger than the current solution too often, but this in turn would require unacceptably large amounts of memory space and computation time.

6.1.2. Variable-depth search algorithms. In contrast to the above algorithms, a variable-depth search algorithm creates several generations. In each generation a finite sequence of iterations is generated, in each of which a neighbor of the previous solution is computed. In principle, each neighbor chosen is a minimum cost neighbor of the previous solution. However, in this approach the risk of cycling is large. To avoid cycling, a sort of tabu list is introduced, which prevents the search from generating a solution that has occurred in the sequence before. Before starting the first iteration in a generation the tabu list is emptied and the solution contained in the single cluster is chosen from the solutions that occurred in the previous generation.

There are two main variants to choose the solution that a new generation is started with. In the first variant a solution with smallest cost is chosen among those generated in the previous generation, but it is not allowed to choose the solution that this previous solution was started with. In the second variant the first solution is chosen among those generated in the previous generation that has smaller cost than the solution that the iteration loop was started with, provided that such a solution has been found. Otherwise, an arbitrary solution obtained in the previous generation is chosen.

The instantiations for variable-depth search are as follows.

• GENERATE CLUSTERS sets $\mathscr{C} = (C)$ equal to (P). Both C and P contain one solution only.

• GENERATE NEIGHBORS selects all neighbors of the current solution deterministically by inspecting these in a prespecified order.

• REDUCE NEIGHBORS determines among all solutions in Q not on the tabu list a solution $Q_j \neq C_1$ of minimum cost. C_1 is then replaced by Q_j.

• CREATE sets $\hat{P}$ equal to the current $C = (C_1)$, where C_1 is a solution found in the last iteration loop that is different from the solution with which the iteration loop started. We mention the following possibilities for choosing C_1, each of which also leads to a different choice for CONTINUE ITERATION.

(1) In the first variant a solution is chosen that has smallest cost among those obtained in the last iteration loop. In this case `CONTINUE ITERATION` returns the value true as long as the number of iterations has not yet reached a specified upper bound.

(2) In the second variant the first solution is chosen with smaller cost than the solution that the iteration loop was started with, provided that such a solution has been found. Otherwise, an arbitrary solution obtained in the last iteration loop is chosen. As soon as a solution is found that has smaller cost than the solution that the iteration loop was started with, the loop is terminated by letting `CONTINUE ITERATION` return the value `FALSE`.

- `REDUCE POPULATION` simply selects the best of the two solutions in P and $\hat{P}$. Ties are broken arbitrarily.
- In some variants `CONTINUE POPULATION GENERATION` returns the value `TRUE` as long as the sequence of the costs of solutions in P for the subsequent generations is strictly decreasing. Other variants use different rules to stop the generation of new generations.

6.2. Multi-level point-based local search

We now discuss point-based local search algorithms with more than one active level.

Compared to single-level local search, the outstanding strength of multi-level local search is the combined use of several neighborhoods. More specifically, the use of multiple levels enables the search process the escape from a local minimum with respect to a given neighborhood by continuing from a neighbor in a different neigbourhood. Algorithms of this type are usually composed from single-level point-based local search algorithms. For this reason we do not detail the corresponding procedures and functions here.

Nevertheless, since algorithms of this kind seem to give good results, we briefly discuss one example due to Martin *et al.* [14]. Their algorithm for the symmetric traveling salesman problem uses, in our terminology, two active levels.

At level 1 they use simulated annealing. Their neighborhood is a subset of the 4-exchange neighborhood. After selecting a single neighbor at level 1, at level 2 they determine a local minimum with respect to a special 3-exchange neighborhood, using any single-level point-based local search algorithm that is able to do so. Then this local minimum is compared with the current solution at level 1 and is accepted using simulated annealing. The authors attribute the power of their algorithm to the fact that, after making a single 4-exchange and then applying 3-exchanges until a local optimum is reached, typically many links in the tour have been changed.

Johnson and McGeoch [10] further refined the algorithm of Martin *et al.* [14] to include the Lin–Kernighan variable-depth search neighborhood [13] at level 2. The resulting *iterated Lin–Kernighan* algorithm is probably the most effective existing approximation algorithm for the symmetric traveling salesman problem. It performs extremely well for instances with hundreds of thousands of cities.

We conclude that algorithms of this type, which use more levels in the local search template, seem to be powerful and deserve wider attention.

6.3. Single-level population-based local search

We now discuss a class of single-level population-based local search algorithms, called *genetic* algorithms. These were first introduced by Holland [9] and have been well described in a textbook by Goldberg [8].

In each generation, first some clusters C of the current population P are created. To each C, the hyper-neighborhood function $\mathcal{N}_1$ is applied to produce a set of new solutions. From these new solutions and the solutions of the current population, the low cost solutions are selected to form a new population, which then starts up a next generation. The generation loop terminates as soon as some stopping criterion, which is usually chosen heuristically, is satisfied. The instantiations for the class of genetic algorithms are as follows.

- `GENERATE CLUSTERS` generates from the population P of size p_1 a multiset $\mathscr{C}$ of clusters C of size c_1. In most cases the clusters are formed heuristically and in such a way that solutions with lower cost are contained in a cluster with higher probability. Note that a solution in P can occur in more than one cluster and even several times in the same cluster.
- `REDUCE NEIGHBORS` takes from the current cluster C and from Q the c_1 best solutions to form a new cluster C.

• CREATE sets $\hat{P}$ equal to the union of all current clusters $C \in \mathscr{C}$.

• REDUCE POPULATION merges P and $\hat{P}$ into a new population P. In most variants, this is done by choosing from P and $\hat{P}$ exactly p_1 elements, with a preference for low-cost solutions.

The remaining procedures can be chosen as follows.

• GENERATE NEIGHBORS selects randomly a number of neighbors of the current cluster C using the hyper-neighborhood function $\mathscr{N}_1$. In many implementations, this number of neighbors also equals c_1.

• CONTINUE ITERATION usually returns the value TRUE for the first iteration and FALSE otherwise. In this way, only one set of neighbors is generated for each chosen cluster, after which the iteration loop is left. In this case, the function REDUCE NEIGHBORS can be skipped, since there is no reason to create a new current cluster C when there is one iteration only.

• CONTINUE POPULATION GENERATION gives the value TRUE for instance as long as a certain upper bound on the number of generations has not been exceeded, or as long as the population contains different solutions.

6.4. Multi-level population-based local search

Few examples of population-based local search algorithms with more than one active level are known. Here we discuss the so-called *genetic local search* approach [18], which is a variant of the class of genetic algorithms. The only difference is that there is now a second active level, in which a point-based hyper-neighborhood function $\mathscr{N}_2 : \mathscr{S} \rightarrow \mathscr{P}(\mathscr{S})$ is used.

After the computation of a tuple Q of neighbors at the first level, a local minimum is computed for each solution Q_j at the second level, using the neighborhood function $\mathscr{N}_2$. After that, back at level 1 the function REDUCE NEIGHBORS is applied to the current Q, which now contains local minima with respect to the neighborhood function $\mathscr{N}_2$. The instantiations for the second level are as follows.

• CONTINUE POPULATION GENERATION gives the value TRUE for the first generation and FALSE for the subsequent generations. In this way, only one generation is created.

• GENERATE CLUSTERS creates for each solution $P_i \in P$ a cluster $C = (P_i)$.

• CREATE sets $\hat{P}$ equal to the set that, for each cluster, contains a local minimum obtained in the iteration loop for the corresponding cluster.

• REDUCE POPULATION sets the new population P equal to $\hat{P}$.

CONTINUE ITERATION, REDUCE NEIGHBORS and GENERATE NEIGHBORS are the same as the ones specified for the deterministic iterative improvement algorithm.

7. OPEN SPOTS IN THE LOCAL SEARCH TEMPLATE

When looking at the template one can try to find types of local search algorithms that have not been proposed before, or that have no widespread application. In this section we mention some of these algorithms, and show how they fit into our template. We first deal with single-level point-based algorithms and then with single-level population-based algorithms. We do not consider multi-level algorithms, since these are composed of single-level ones. However, multi-level algorithms are important, as they may have an impressive performance.

7.1. Single-level point-based local search

In almost all known single-level point-based algorithms also the number of clusters equals one. In this case it often happens that several neighbors of a current solution are promising while only one neighbor is allowed to be chosen. One would like to postpone the decision of choosing the neighbor for a while until after a few iterations it becomes clear which of the neighbors is most promising. This idea fits in our template in the following way.

At the beginning of a generation several clusters are generated, each containing a copy of the solution in the current one-element population P. Then for each cluster an iteration loop is started. When this loop terminates depends on how reduce neighbors is chosen. Reasonable points for terminating the iteration loop are when a local optimum has been found or when a given number of iterations has been executed. When for each cluster its corresponding iteration loop is terminated, a solution must be selected to become the new solution in P. It is reasonable

to select the best solution that resulted from the various iteration loops in the last generation. Next, a new generation is started up.

7.2. Single-level population-based local search

Below we give some ideas for single-level population-based local search algorithms. A distinction is made between algorithms with cluster size 1, cluster size 2 and cluster size larger than 2.

When the cluster size is 1, only ordinary neighborhoods can be used. Since we have a population-based algorithm, it is reasonable to let the number of clusters be larger than 1. The idea of the following algorithm is that several parallel runs of a point-based algorithm are interrupted now and then and that the runs with the worse results are stopped definitely. The remaining runs are continued in such a way that one run may proceed in several directions. It is therefore necessary that the point-based algorithm uses some type of randomization. This idea fits into our template in the following way.

At the beginning of a generation several clusters are generated, each containing a copy from one of the solutions in the current population P. Then for each cluster an iteration loop is started, using a point-based algorithm. When this loop terminates after the execution of a given number of iterations, a population $\hat{P}$ is formed from the final or from intermediate solutions obtained in the previous iterations. The populations P and $\hat{P}$ are then merged into a new population P, using a selection criterion like those used in genetic algorithms.

All present genetic algorithms use hyper-neighborhoods that are based on clusters of size 2. Furthermore, in a given generation, for each cluster there is exactly one iteration in which the hyper-neighborhood is applied. One can think of algorithms that use more than one iteration for the same cluster in a given generation. The execution of iterations in one generation can be stopped after a certain number of iterations. One can also use another stop criterion. Therefore, we define a cluster to be locally optimal with respect to a hyper-neighborhood $\mathcal{N}$ if it has no neighbor in $\mathcal{N}$ that has lower cost than the solutions of this cluster. Now, one can stop the execution of iterations as soon as the current cluster is locally optimal. This idea fits into our template in an obvious way.

Until now, no genetic algorithms have been proposed that use hyper-neighborhoods that are based on clusters of size larger than 2. For many standard hyper-neighborhoods based on cluster size 2, it is not difficult to generalize them to hyper-neighborhoods based on cluster size 3 or on even larger cluster sizes. Whether these generalizations will give computational results of the same quality as those for hyper-neighborhoods based on cluster size 2, is not clear at present. Here again, this idea obviously fits in our template.

8. CONCLUSION

We have proposed a unifying template for the many different types of local search algorithms in combinatorial optimization. The main component is a recursive procedure `LOCAL SEARCH`. On the basis of the depth of the recursion (one or more levels) and the size of the population (one or more representatives), local search algorithms have been classified and several variants proposed in the literature have been shown to fit into the classification. We hope that our template and the corresponding classification may stimulate the development of new types of local search algorithms. In this respect, multi-level local search seems to deserve special attention, as existing algorithms of that type have led to impressive computational results.

REFERENCES

1. Aarts, E. H. L. and Korst, J. H. M. (1989) *Simulated Annealing and Boltzmann Machines*. Wiley, Chichester.
2. Aarts, E. H. L. and Lenstra, J. K. (eds.) (1997) *Local Search in Combinatorial Optimization*. Wiley, Chichester.
3. Bock, F. (1958) An algorithm for solving "traveling-salesman" and related network optimization problems: abstract. *Bulletin Fourteenth National Meeting of the Operations Research Society of America*. p. 897.
4. Croes, G. A., A method for solving traveling salesman problems. *Operations Research*, 1958, **6,** 791–812.
5. Dueck, G. and Scheuer, T., Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 1990, **90,** 161–175.
6. Garey, M. R. and Johnson, D. S. (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA.
7. Glover, F., Tabu search: Part I. *ORSA Journal on Computing*, 1989, **1,** 190–206.

8. Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA.
9. Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.
10. Johnson, D. S. and McGeoch, L. A. (1997) The traveling salesman problem: a case study. In *Local Search in Combinatorial Optimization*, eds. E. H. L. Aarts and J. K. Lenstra. Wiley, Chichester, pp. 215–310.
11. Johnson, D. S., Papadimitriou, C. H. and Yannakakis, M., How easy is local search? Journal of Computer and System Science, 1988, **37,** 79–100.
12. Kirkpatrick, S., Gelatt, C. D., Jr. and Vecchi., M. P., Optimization by simulated annealing. *Science*, 1983, **220,** 671–680.
13. Lin, S. and Kernighan, B. W., An effective heuristic for the traveling-salesman problem. *Operations Research*, 1973, **21,** 498–516.
14. Martin, O., Otto, S. W. and Felten, E. W., Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 1991, **5,** 299–326.
15. Osman, I. H., and Kelly, J. P. (eds.) (1996) *Metaheuristics: Theory and Applications*. Kluwer, Boston, MA.
16. Papadimitriou, C. H. and Steiglitz, K. (1982) *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, New York.
17. Tovey, C. A. (1997) Local improvement on discrete structures. In *Local Search in Combinatorial Optimization*, eds. E. H. L. Aarts and J. K. Lenstra. Wiley, Chichester, pp. 57–89.
18. Ulder, N. L. J., Aarts, E. H. L., Bandelt, H.-J., van Laarhoven, P. J. M. and Pesch, E. (1991) Genetic local search algorithms for the traveling salesman problem. In *Parallel Problem Solving from Nature. Lecture Notes in Computer Science*, Vol. 496, eds. H.-P. Schwefel and R. Männer. Springer, Berlin, pp. 109–116.
19. Vaessens, R. J. M., Aarts, E. H. L. and Lenstra, J. K., Job shop scheduling by local search. *INFORMS Journal on Computing*, 1996, **8,** 302–317.
20. Yannakakis, M. (1997), Computational complexity. In *Local Search in Combinatorial Optimization*, eds. E. H. L. Aarts and J. K. Lenstra. Wiley, Chichester, pp. 19–55.
21. Yannakakis, M. (1990), The analysis of local search problems and their heuristics. In *STACS 90: 7th Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science*, Vol. 415, eds. C. Choffrut and T. Lengauer. Springer, Berlin, pp. 298–311.