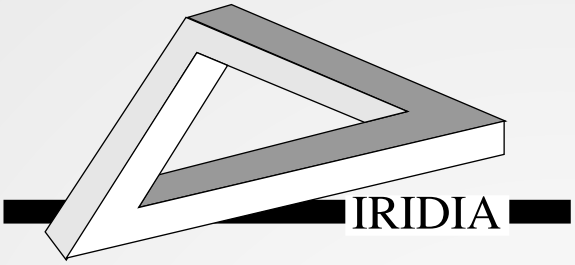
TECHNICAL REPORT UNIVERSITÉ LIBRE DE BRUXELLES



Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison

Christian BLUM
Andrea ROLI

Technical Report No.

TR/IRIDIA/2001-13
October 2001

Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle

50 av. F. Roosevelt • CP 194/6 • B-1050 Bruxelles • Belgium
Phone: +32-2-650.27.29 • Fax: +32-2-650.27.15

Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison

Christian Blum
Université Libre de Bruxelles, IRIDIA
Avenue Franklin Roosevelt 50, CP 194/6, 1050 Brussels, Belgium
cblum@ulb.ac.be

Andrea Roli
DEIS - Università degli Studi di Bologna
Viale Risorgimento, 2 - Bologna (Italia)
aroli@deis.unibo.it

October 23, 2001

Abstract

In this report we overview metaheuristics to solve Combinatorial Optimization (CO) problems. We survey the most important classifications of metaheuristics, each of them being the result of a specific viewpoint. There are differences in how these algorithms try to tackle a problem, but there are also a lot of similarities in the strategies and concepts used by the different metaheuristics. The two most important concepts used in metaheuristics are, in our opinion, the concepts of "intensification" and "diversification", which are in some way contrary and complementary to each other. In this work we compare metaheuristics in the way intensification and diversification are used. We conclude outlining some strengths and weaknesses of the different approaches thus leading to the development of hybrid algorithms combining concepts originating from different metaheuristics.

Contents

1 Introduction **3**

2 Metaheuristics **7**

2.1 Classifications of metaheuristics 7

2.2 Trajectory Methods 8

2.2.1 Basic Local Search: Iterative Improvement 9

2.2.2 Simulated Annealing 9

2.2.3 Tabu Search 11

2.2.4 Explorative Local Search methods 13

2.2.4.1 GRASP 13

2.2.4.2 Variable Neighborhood Search 14

2.2.4.3 Guided Local Search 17

2.2.4.4 Iterated Local Search 19

2.3 Population-based methods 22

2.3.1 Evolutionary Computation (EC) 22

2.3.1.1 Scatter Search and Path Relinking 24

2.3.1.2 Evolutionary Algorithms based on probabilistic models . . . 26

2.3.2 Ant Colony Optimization (ACO) 27

3 Conceptual Comparison **31**

3.1 The basic use of intensification and diversification 31

3.2 More sophisticated intensification and diversification 34

3.3 Hybridization of metaheuristics 35

3.4 Conclusions 36

List of Algorithms

1 Iterative Improvement 9
2 Simulated Annealing (SA) 9
3 Simple Tabu Search (TS) 11
4 Tabu Search (TS) 12
5 Greedy Randomized Adaptive Search Procedure (GRASP) 13
6 Greedy randomized solution construction 13
7 Variable Neighborhood Search (VNS) 15
8 Variable Neighborhood Descent (VND) 16
9 Variable Neighborhood Decomposition Search (VNDS) 16
10 Skewed Variable Neighborhood Search (SVNS) 17
11 Guided Local Search (GLS) 19
12 Iterated Local Search (ILS) 20
13 Evolutionary Computation (EC) 23
14 Scatter Search and Path Relinking 25
15 Estimation of Distribution Algorithms (EDAs) 26
16 Ant System (AS) 27
17 Ant Colony Optimization (ACO) 29

Chapter 1

Introduction

Many optimization problems of both practical and theoretical importance concern themselves with the choice of a "best" configuration of a set of parameters to achieve some goals. They seem to divide naturally into two categories: those were the solutions are encoded with *continuous* variables, and those were solutions are encoded with *discrete* variables. Among the latter ones we find a class of problems called combinatorial optimization problems. According to [60], in combinatorial optimization problems, we are looking for an object from a finite – or possibly countably infinite – set. This object is typically an integer, a subset, a permutation, or a graph structure.

Definition 1 *A* **Combinatorial Optimization** *problem $\mathcal{P} = (\mathcal{S}, f)$ can be defined by:*

- *a set of variables $X = \{x_1, \ldots, x_n\}$;*
- *variable domains $D_1, \ldots, D_n$;*
- *constraints among variables;*
- *an objective function to be minimized $f : D_1 \times \ldots \times D_n \rightarrow \mathbb{R}^+$;*

The set of all possible feasible assignments is

$$\mathcal{S} = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in D_i, i = 1, \ldots, n \ \wedge \ s \text{ satisfies all the constraints}\}$$

$\mathcal{S}$ is usually called search (or solution) space, as each element of the set can be seen as a candidate solution.

To solve a combinatorial optimization problem one has to find a solution $s \in \mathcal{S}$ with minimal[1] *objective function value, that is, $f(s^*) \leq f(s)\ \forall s \in S$. s^* is called a globally optimal solution of $(\mathcal{S}, f)$ and the set $\mathcal{S}^* \subseteq \mathcal{S}$ is called the set of globally optimal solutions.*

Examples for combinatorial optimization problems are the Traveling Salesman problem (TSP), the Quadratic Assignment problem (QAP), Constraint Satisfaction problems (CSPs) and

[1] As maximizing an objective function f is the same as minimizing $-f$, we will deal without loss of generality with minimization problems in this work

Scheduling problems. Due to the practical importance of combinatorial optimization (CO) problems, many algorithms for their solution have been developed. These algorithms can be classified as either *complete* or *approximate* algorithms. Complete algorithms are guaranteed to find for every finite size instance of a CO problem an optimal solution in bounded time (see [60, 57]). Yet, for many CO problems that are $\mathcal{NP}$-hard [30] complete methods need exponential computation time in the worst-case and even for small problem instances these algorithms might take an amount of execution time much too high for practical purposes. Therefore, the use of approximate methods to solve CO problems has been getting more and more attention in the last 30 years. In approximate methods we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time.

Among the basic approximate methods we usually distinguish between *constructive* methods and *local search* methods. Constructive algorithms generate solutions from scratch by adding – to an initially empty partial solution – components until a solution is complete. They are typically the fastest approximate methods, yet they often return solutions of inferior quality when compared to local search algorithms. Local search algorithms start from some initial solution and iteratively try to replace the current solution by a better solution in an appropriately defined neighborhood of the current solution, where the neighborhood is formally defined as follows:

Definition 2 *A* **neighborhood structure** *is a function $\mathcal{N} : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ that assigns to every $s \in \mathcal{S}$ a set of neighbors $\mathcal{N}(s) \subseteq \mathcal{S}$. $\mathcal{N}(s)$ is also called the neighborhood of s.*

With the introduction of a neighborhood structure we can also define the concept of *locally* minimal solutions.

Definition 3 *A* **locally minimal solution (or local minimum)** *with respect to a neighborhood structure $\mathcal{N}$ is a solution $\hat{s}$ such that $\forall\ s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$. We call $\hat{s}$ a strict locally minimal solution if $f(\hat{s}) < f(s)\ \forall\ s \in \mathcal{N}(\hat{s})$*

In the last 10 to 15 years a new kind of approximate algorithm has emerged which basically tries to combine basic heuristic methods in higher level frameworks aimed at exploring a search space. These kind of methods are nowadays commonly called *metaheuristics*. The term *metaheuristic* derives from the composition of two Greek words. *Heuristic* derives from the verb *heuriskein* ($\epsilon\upsilon\rho\iota\sigma\kappa\epsilon\iota\nu$) which means "to find", while the suffix "meta" means "beyond, in an upper level".

This class of algorithms includes but is not restricted to[2] Ant Colony Optimization (ACO), Evolutionary Computation (EC) with its most famous representative the Genetic Algorithm (GA), Iterated Local Search (ILS), Simulated Annealing (SA), and Tabu Search (TS). Up to now there is no commonly accepted definition for the term metaheuristic. It is just in the last few years that some researchers in the field tried to propose a definition. In the following we quote some of them:

"A metaheuristic is formally defined as an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the

[2]in alphabetical order

search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions." This citation is taken from a metaheuristics bibliography by Osman and Laporte [59].

"A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high-quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method." This citation is taken from "Meta-heuristics: Advances and Trends in Local Search Paradigms for Optimization", by Voss, Martello, Osman and Roucairol [73].

"Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descent by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form." This citation is taken from the PhD thesis of Stützle [67].

Summarizing, we outline fundamental properties which characterize metaheuristics:

- Metaheuristics are strategies that "guide" the search process.
- The goal is to efficiently explore the search space in order to find (sub)optimal solutions.
- Techniques which constitute metaheuristics algorithms range from simple local search procedures to complex learning processes.
- Metaheuristic algorithms are approximate and non-deterministic.
- They incorporate mechanisms to avoid getting trapped in confined areas of the search space.
- The basic concepts of metaheuristics permit an abstract level description.
- Metaheuristics are not problem-specific.
- Metaheuristics make use of domain-specific knowledge and/or search experience (memory) to bias the search.

In short we could say: Metaheuristics are high level concepts for exploring search spaces by using different strategies. These strategies should be chosen in such a way that a dynamic balance is given between the exploitation of the accumulated search experience (which is commonly called *intensification*) and the exploration of the search space (which is commonly called *diversification*). This balance is necessary on one side to quickly identify regions in the search space with high quality solutions and on the other side not to waste too much time

in regions of the search space which are either already explored or don't provide high quality solutions.

The structure of the strategies is highly dependent on the philosophy of the metaheuristic itself. To compare the strategies used in different metaheuristics is one of the goals in the last chapter of this work. There are several different philosophies apparent in the existing metaheuristics. Some of them can be seen as "intelligent" extensions of local search algorithms. The goal of this kind of metaheuristic is to escape from local minima to proceed in the exploration of the search space and to move on to find other hopefully better local optima. This is for example true for Tabu Search, Iterated Local Search, Variable Neighborhood Search, GRASP and Simulated Annealing. These metaheuristics (also called trajectory methods) work on one or several neighborhood structure(s) imposed on the members (the solutions) of the search space.

We can find a different philosophy in algorithms like Ant Colony Optimization and Evolutionary Computation. They incorporate a learning component in the sense that they implicitly or explicitly try to learn correlations between decision variables to identify high quality areas in the search space. This kind of metaheuristic performs in a sense a biased sampling of the search space. For instance, in Evolutionary Computation this is achieved by a recombination of solutions and in Ant Colony Optimization this is achieved by sampling the search space in every iteration according to a probability distribution.

There are several approaches to classify metaheuristics according to their properties. In the next chapter we will briefly summarize some criteria for classifications and we will give a description of the basic metaheuristics nowadays. In the last chapter we will compare the different metaheuristics on the way they implement the concepts of intensification and diversification of the search process. We point out similarities and differences in terms of intensification and diversification.

Chapter 2

Metaheuristics

In this chapter we describe the state-of-the-art of research on metaheuristics by presenting the most important metaheuristic algorithms. First we will survey some criteria which are used to classify metaheuristics, then we will give a detailed description of the algorithms in their fundamental structures.

2.1 Classifications of metaheuristics

There are different ways to classify and describe metaheuristic algorithms. Depending on the characteristics selected to differentiate between them, several classifications are possible each of them being the result of a specific viewpoint. In this section we briefly summarize the most important ways of classifying metaheuristics.

Nature-inspired vs. non-nature inspired. Perhaps, the most intuitive way of classifying metaheuristics is based on the origins of the algorithm. There are nature-inspired algorithms, like Genetic Algorithms and Ant Algorithms, and non nature-inspired ones like Tabu Search and Iterated Local Search. In our opinion this classification is not very meaningful for the following two reasons. First, many recent hybrid algorithms do not fit either class (or, in a sense, they fit both at the same time). Second, it is sometimes difficult to clearly tell the genesis of an algorithm.

Population-based vs. single point search. Another characteristic which can be used for the classification of metaheuristics is the way of performing the search: Does the algorithm work on a population or on a single solution at any time? Algorithms working on single solutions are called *trajectory methods* and encompass local search-based metaheuristics, like Tabu Search, Iterated Local Search and Variable Neighborhood Search. They all share the property of describing a trajectory in the search space during the search process. Population-based metaheuristics on the contrary perform search processes which describe the evolution of a set of points in the search space.

Dynamic vs. static objective function. Metaheuristics can also be classified according to the way they make use of the objective function. While some algorithms keep the objective function given in the problem representation "as it is", some others like Guided Local Search (GLS) modify it during the search. The idea behind this approach is to escape from local

optima by modifying the search landscape. Accordingly, during the search the objective function is altered by trying to incorporate information collected during the search process.

One vs. various neighborhood structures. Most metaheuristic algorithms work on one single neighborhood structure. In other words, the fitness landscape which is searched doesn't change in the course of the algorithm. Other metaheuristics like Variable Neighborhood Search (VNS) use a set of neighborhood structures which gives the possibility to diversify the search and tackle the problem jumping between different fitness landscapes.

Memory usage vs. memory-less methods. A very important feature to classify metaheuristics is the use they make of the search history, that is whether they use memory or not. Memory-less algorithms perform a Markov process, as the information they need is only the current state of the search process. There are several different ways of making use of memory. Usually we differentiate between short term and long term memory structures. The first usually keeps track of recently performed moves, visited solutions or, in general, decisions taken. The second is usually an accumulation of synthetic parameters and indexes about the search. The use of memory is nowadays recognized as one of the fundamental elements of a powerful metaheuristic.

In the following we describe the most important metaheuristics following the single point vs. population-based search classification, which divides metaheuristics into trajectory methods and population-based methods. This choice is motivated by the fact that this categorization permits a clearer description of the algorithms. Moreover, a current trend is the hybridization of methods in the direction of the integration of single point search algorithms in population-based ones.

2.2 Trajectory Methods

In this section we describe metaheuristics called *trajectory methods*, as their search process is characterized by a trajectory in the search space. In the following we will refer to a minimization problem $\mathcal{P}$ with an objective function f, a set of feasible solutions $\mathcal{S} = \{s_1, s_2, \ldots\}$ and a neighborhood structure $\mathcal{N}$.

The search process of trajectory methods can be seen as the evolution in (discrete) time of a discrete dynamical system [6, 15]. The algorithm starts from an initial state (the initial solution) and describes a trajectory in the state space. The system dynamics depends on the strategy used: simple algorithms generate a trajectory composed of two parts: a *transient* phase followed by an *attractor* (a fixed point, a cycle or a complex attractor). Algorithms with advanced strategies generate more complicated trajectories which can not be subdivided in those two phases. The characteristics of the trajectory outline the behavior of the algorithm and its effectiveness with respect to the instance it is attacking. It is worth underlining that the dynamics is the result of the combination of algorithm, problem representation and instance.

We will first describe the basic local search algorithms and then we will survey more complex strategies, ending with algorithms which define general strategies and can include other trajectory methods as components.

2.2.1 Basic Local Search: Iterative Improvement

The basic local search is usually called *iterative improvement*, since each move is only performed if the solution it produces is better than the current solution. The algorithm stops as soon as it finds a local minimum. The high level algorithm is sketched in Algorithm 1.

Algorithm 1 Iterative Improvement

```
s ← GenerateInitialSolution()
repeat
  s ← Improve(s,N(s))
until no improvement is possible
```

The function $Improve(s,\mathcal{N}(s))$ can be either a *first improvement*, or a *best improvement* function. The former scans the neighborhood $\mathcal{N}(s)$ and chooses the first solution which improves the objective function, the latter exhaustively explores the neighborhood and returns the solution with the lowest objective function value. Both methods stop at local minima, therefore their performance strongly depends on the definition of S,f and $\mathcal{N}$.

The performance of iterative improvement procedures on CO problems is usually quite unsatisfactory, thus several techniques have been developed to prevent algorithms from getting trapped in local optima or to escape from them. In the following we describe the most important and successful ones.

2.2.2 Simulated Annealing

Simulated Annealing (SA) is commonly said to be the oldest among the metaheuristics and surely one of the first algorithms which had an explicit strategy to avoid local optima. The origins of the algorithm are in statistical mechanics (Metropolis algorithm) and it was first presented as a search algorithm for CO problems in [45] and [11]. The fundamental idea is to allow moves toward solutions with higher objective function values (uphill moves) in order to escape from local minima. The probability of doing such a move is decreased during the search. The high level algorithm is described in Algorithm 2.

Algorithm 2 Simulated Annealing (SA)

```
s ← GenerateInitialSolution()
T ← T_0
while termination conditions not met do
  Pick s' ∈ N(s) at random
  if f(s') < f(s) then
    s ← s' {s' replaces s}
  else
    Accept s' as new solution with probability p(T, s', s)
  end if
  Update(T)
end while
```

The algorithm starts by generating an initial solution (either randomly or heuristically con-

structed) and by initializing the so-called temperature parameter T. Then it repeats the search process until the termination condition is reached. Several termination criteria are possible: a maximum CPU time, a maximum number of iterations, a solution s with $f(s)$ less than a predefined threshold value is found, or the maximum number of iterations without improvements is reached. The instructions in the inner cycle are very simple: a solution $s' \in \mathcal{N}(s)$ is randomly sampled and it is accepted as new current solution depending on $f(s)$, $f(s')$ and T. s' replaces s if $f(s') < f(s)$ or, in case $f(s') \geq f(s)$, with a probability which is a function of T and $f(s') - f(s)$. The probability is generally computed following the Boltzmann distribution $exp(-\frac{f(s')-f(s)}{T})$.

The temperature T is decreased[1] during the search process, thus at the beginning of the search the probability of accepting uphill moves is high and it gradually decreases, converging to a simple iterative improvement algorithm. This process is analogous to the annealing process of metals and glass, which assume a low energy configuration when cooled with an appropriate cooling schedule. Regarding the search process, this means that the algorithm is the result of two combined strategies: random walk and iterative improvement. In the first phase of the search, the bias toward improvements is low and it permits the exploration of the search space; this erratic component is slowly decreased thus leading the search to converge to a (local) minimum. The probability of accepting uphill moves is controlled by two factors: the difference of the objective functions and the temperature. On the one hand, at fixed temperature, the higher the difference $f(s') - f(s)$, the lower the probability to accept a move from s to s'. On the other hand, the higher T, the higher the probability of uphill moves.

The choice of an appropriate cooling schedule is crucial for the performance of the algorithm. The cooling schedule defines the value of T at each iteration k, $T_{k+1} = Q(T_k, k)$, where $Q(T_k, k)$ is a function of the temperature at the previous step and of the iteration number. Theoretical results on non-homogeneous Markov chains [1] state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global minimum for $k \to \infty$. More precisely:

$$\exists \Gamma \in I\!R \quad \text{s.t.} \quad \lim_{k \to \infty} Prob[\text{global minimum found after } k \text{ steps}] = 1$$
$$\text{iff} \quad \sum_{k=1}^{\infty} exp(\frac{\Gamma}{T_k}) = \infty$$

A particular cooling schedule which fulfills the hypothesis for the convergence is the one that follows a logarithmic law: $T_{k+1} = \frac{\Gamma}{\log(k+k_0)}$. Unfortunately, cooling schedules which guarantee the convergence to a global optimum are not feasible in practice because they take infinite time. Therefore, faster cooling schedules are adopted in applications. One of the most used follows a geometric law: $T_{k+1} = \alpha T_k$, where $\alpha \in]0, 1[$, which corresponds to an exponential decay of the temperature.

The cooling rule can vary during the search, with the aim of tuning the balance between diversification and intensification. For example, at the beginning of the search, T can be constant or linearly decreasing, in order to sample the search space; then, T can follow a rule like the geometric one, to converge to a local minimum at the end of the search. *Non-monotonic* schedules are also used [58, 47], which alternate phases of cooling and reheating,

[1] T is not necessarily decreased in a monotonic fashion. Elaborate cooling schemes also incorporate an occasional increase of the temperature.

thus providing an oscillating strategy to control diversification and intensification.

The cooling schedule and the initial temperature should be adapted to the particular problem instance, since the cost of escaping from local minima depends on the structure of the search landscape. A simple way of empirically determining the starting temperature T_0 is to initially sample the search space with a random walk to roughly evaluate the average and the variance of objective function values. But also more elaborate schemata can be implemented [44].

SA has been applied to several CO problems, from the TSP to layout design and protein structure problems [2, 44, 26]. It is nowadays used as a component in metaheuristics, rather than applied as stand-alone search algorithm.

The dynamic process described by SA is a *Markov chain* [23], as it follows a trajectory in the state space whereby the successor state is chosen depending only on the incumbent one. Therefore this process is memory-less.

2.2.3 Tabu Search

Tabu Search (TS) is among the most cited and used metaheuristics for CO problems. TS basic ideas were first introduced in [31] and independently sketched in [38]. A description of the method and its concepts can be found in [33]. TS explicitly uses the history of the search, both to avoid local minima and to implement an explorative strategy. We will first describe a simple version of TS, to introduce the basic concepts. Then, we will explain a more applicable algorithm and finally we will discuss some improvements.

Algorithm 3 Simple Tabu Search (TS)

```
s ← GenerateInitialSolution()
TabuList ← ∅
while termination conditions not met do
  s ← Best_Improvement(s,N(s) \ TabuList)
  Update(TabuList)
end while
```

The simple TS algorithm (see Algorithm 3) applies a best improvement local search as basic ingredient and uses a *short term memory* to escape from local minima and to avoid cycles. The short term memory is realized as a *tabu list* which keeps track of the most recently visited solutions and forbids moves toward them. The neighborhood of the current solution is thus restricted to the solutions which do not belong to the tabu list. In the following we will refer to this set as *allowed set*. The new solution is then added to the list and one of the solutions in the list is discarded (usually in a FIFO order). Due to this dynamic restriction of allowed solutions in a neighborhood, TS can be considered as a dynamic neighborhood search technique [67]. The termination condition can be a maximum CPU time, a maximum number of iterations, the generation of a solution s with $f(s)$ less than a predefined threshold value, or a maximum number of iterations without improvements. The algorithm can also terminate if the allowed set is empty, that is, if all the solutions in $\mathcal{N}(s)$ are forbidden by the tabu list[2].

[2]Other escaping strategies are possible, to avoid to stop when the allowed set is empty. For instance, one can choose the least recently visited solution, even if it is tabu.

The use of a tabu list prevents from returning to recently visited solutions, therefore it prevents from indefinite cycling[3] and forces the search to accept even uphill moves. The length l of the tabu list (*tabu tenure*) controls the memory of the search process. With small tabu tenure the search will concentrate on small areas of the search space. On the opposite, a large tabu tenure forces the search process to explore larger regions, because it forbids revisiting a higher number of solutions. The tabu tenure can be varied during the search, leading to more robust algorithms. An example can be found in [70], where the tabu tenure is periodically reinitialized at random in the interval $[l_{min}, l_{max}]$. A more advanced use of dynamic tabu tenure is presented in [8, 7], where the tabu tenure is increased if there is evidence for repetitions of solutions (thus a higher diversification is needed), while it is decreased if there are no improvements (thus intensification should be boosted).

The implementation of short term memory as a list of visited solution is not practical, because managing a list of solutions is highly inefficient. Therefore, instead of the solutions themselves, solution *attributes* are stored. Attributes are usually components of solutions, like moves and differences between two solutions. Since more than one attribute can be considered, a tabu list is introduced for each of them. The set of attributes and related tabu lists define the *tabu conditions* which are used to filter the neighborhood of a solution and generate the allowed set.

Storing moves instead of complete solutions is much more efficient, but it introduces a loss of information, as forbidding a move means assigning the tabu status to probably more than one solution. Thus, it is possible that unvisited solutions of good quality are excluded from the allowed set. To overcome this problem, *aspiration criteria* are defined which allow to include a solution in the allowed set even if it is forbidden by tabu conditions. Aspiration criteria define the aspiration conditions that are used to construct the allowed set. The most commonly used aspiration criterion selects solutions which are better than the current one. The complete algorithm, as described above, is reported in Algorithm 4.

Algorithm 4 Tabu Search (TS)

$s \leftarrow$ GenerateInitialSolution()
InitializeTabuLists($TL_1, \ldots, TL_r$)
$k \leftarrow 0$
while termination conditions not met **do**
  $AllowedSet(s,k) \leftarrow \{z \in \mathcal{N}(s) \mid$ no tabu condition is violated or at least one aspiration condition is satisfied$\}$
  $s \leftarrow$ BestImprovement(s,$AllowedSet(s,k)$)
  UpdateTabuListsAndAspirationConditions()
  $k \leftarrow k+1$
end while

Tabu lists are only one of the possible ways of taking advantage of the history of the search and they realize a short term memory. Information collected during the overall search process can be very useful, especially for a strategic guidance of the algorithm. This kind of long term memory is usually added to TS by referring to four principles: *recency*, *frequency*, *quality* and *influence*. Recency-based memory records for each solution (or attribute) the most recent

[3]Cycles of high order are possible, since the tabu list has a finite length l which is smaller than the cardinality of the search space.

iteration it was involved in. Orthogonally, frequency-based memory keeps track of how many times each solution (attribute) has been visited. This information identifies the regions (or the subsets) of the solution space where the search was confined, or where it stayed for a high number of iterations. This kind of information about the past are usually exploited to diversify the search. The third principle, quality, is a guidance to learn and extract information from the search history in order to identify good solution components. This information can be usefully integrated in the initial solution construction. Other metaheuristics (e.g., Ant Colony Optimization) explicitly use this principle to learn good building blocks for high quality solutions. Finally, influence is a property regarding choices made during the search and can be used to indicate which choices have shown to be the most critical.

TS search has been applied to many CO problems; the most successful applications are the Robust Tabu Search to the Quadratic Assignment Problem [70], the Reactive Tabu Search to the MAXSAT problem [7], and others to routing, scheduling and assignment problems [13].

2.2.4 Explorative Local Search methods

In this subsection we present more recent trajectory methods. These are the Greedy Randomized Adaptive Search Procedure (GRASP), Variable Neighborhood Search (VNS), Guided Local Search (GLS) and Iterated Local Search (ILS).

2.2.4.1 GRASP

The Greedy Randomized Adaptive Search Procedure (GRASP) [24, 63] is a simple algorithm which combines constructive heuristics and local search. Its structure is sketched in Algorithm 5.

Algorithm 5 Greedy Randomized Adaptive Search Procedure (GRASP)

```
while termination conditions not met do
  s ← ConstructGreedyRandomizedSolution(){see algorithm 6}
  ApplyLocalSearch(s)
  MemorizeBestFoundSolution()
end while
```

GRASP is an iterative procedure, composed of two phases: solution construction and solution improvement. The best found solution is returned at the end of the iterations, when the termination condition is reached (any of the usual ones).

Algorithm 6 Greedy randomized solution construction

```
s ← ∅{s donotes a partial solution in this case}
while solution not complete do
  RCL ← GenerateRestrictedCandidateList()
  x ← SelectElementAtRandom(RCL)
  s ← s ∪ {x}
  UpdateGreedyFunction(s){update the heuristic values (see text below)}
end while
```

The solution construction algorithm (see Algorithm 6) is characterized by two main ingredients: a dynamic constructive heuristic and randomization. Assuming that a solution s is consisting of a set of elements $\{x_1, \ldots, x_n\}$, the solution is constructed one element at a time by randomly picking it from a candidate list. The elements are ranked by means of a heuristic criterion which gives them a score as a function of the (myopic) benefit if inserted in the solution. The candidate list is composed of the first cl elements. The heuristic values are updated at each step, thus the score of elements changes during the construction phase, as the choice of an element might decrease/increase the desirability of another not yet inserted element. This constructive heuristic is called *dynamic*, in contrast to the *static* one which assigns a score to elements only before starting the construction. For instance, a static heuristic for TSP is based on the arc cost: the lower the cost, the higher its desirability. An example of a dynamic heuristic is cheapest insertion, where the desirability of an element is evaluated depending on the current partial solution.

The length cl of the candidate list controls the heuristic bias. In the extreme case of $cl = 1$ the best element would be added, thus the construction would be fully greedy. On the opposite, for $cl = n$ the construction would be completely random (as the choice of an element from the candidate list is done at randomly). Therefore, cl is a critical parameter which influences the sampling of the search space.

The second phase of the algorithm is a local search process. This can be implemented with a basic local search algorithm like iterative improvement, or with more advanced techniques, like SA and TS.

GRASP can be effective if two conditions hold:

- the constructive heuristic samples the most promising regions in the search space.
- the solutions constructed by the constructive heuristic belong to basins of attraction of different locally minimal solutions.

The first condition can be met by the choice of a good constructive heuristic and an appropriate length of the candidate list, whereas the second condition can be met by choosing the constructive heuristic and the local search in a way such that they fit well.

As can be noted by observing its basic structure, GRASP does not use the history of the search, since the only memory it needs is to store the current best solution. This is one of the reasons why GRASP is often outperformed by other metaheuristics. However, due to its simplicity, it is generally very fast and it is able to produce quite good solutions in a very short amount of computation time. Among the applications of GRASP we mention the Set Covering Problem, the MAXSAT problem and some industrial problems [25].

2.2.4.2 Variable Neighborhood Search

Variable Neighborhood Search (VNS, [39, 40]) is a metaheuristic which explicitly applies a strategy based on dynamically changing neighborhood structures. The algorithm is very general and many degrees of freedom exist for designing variants and particular instantiations.

At the initialization step, a set of neighborhood structures has to be defined. These neighborhoods can be arbitrarily chosen, but often a sequence $|\mathcal{N}_1| < |\mathcal{N}_2| < \ldots < |\mathcal{N}_{k_{max}}|$ of

Algorithm 7 Variable Neighborhood Search (VNS)

```
Select a set of neighborhood structures N_k, k = 1,...,k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do
    s' ← PickAtRandom(N_k(s)){Shaking phase}
    s'' ← LocalSearch(s')
    if f(s'') < f(s) then
      s ← s''
      k ← 1
    else
      k ← k + 1
    end if
  end while
end while
```

neighborhoods with increasing cardinality is defined[4]. Then an initial solution is generated, the neighborhood index is initialized and the algorithm iterates until a stopping condition is met (Algorithm 7). VNS' main cycle is composed of three phases: *shaking*, *local search* and *move*. In the *shaking* phase a solution s' in the k-th neighborhood of the current solution s is randomly selected. Then, s' becomes the local search starting point. The local search can use any neighborhood structure and it is not restricted to the set defined for VNS. At the end of the local search process (terminated as soon as a predefined termination condition is verified) the new solution s'' is compared with s and, if it is better, it replaces s and the algorithm starts again with $k = 1$. Otherwise, k is incremented and a new shaking phase starts using a different neighborhood.

The objective of the shaking phase is to "kick" the solution in such a way as to provide a good starting point for the local search. The starting point should belong to the basin of attraction of a different local optimum than the current one, but should not be "too far" from s, otherwise the algorithm would degenerate into a simple multi-start. Moreover, choosing s' in the neighborhood of the current best solution is likely to produce a solution which maintains some good features of the current one.

The process of changing neighborhoods in case of no improvements corresponds to a diversification of the search. In particular the choice of neighborhoods of increasing cardinality yields a progressive diversification. The effectiveness of this dynamic neighborhood strategy can be explained by the fact that a "bad" place on the search landscape given by one neighborhood could be a "good" place on the search landscape given by another neighborhood. Moreover, a solution which is locally optimal with respect to a neighborhood is probably not locally optimal with respect to another neighborhood. This fact is being exploited by a local search called Variable Neighborhood Descent (VND), sketched in Algorithm 8. In VND a *best improvement* local search (see section 2.2.1) is applied, and, in case a local minimum is found, the search proceeds with another neighborhood structure.

[4]In principle they could be one included in the other, $\mathcal{N}_1 \subset \mathcal{N}_2 \subset \ldots \subset \mathcal{N}_{k_{max}}$. Nevertheless, such a sequence might produce an inefficient search, because a large number of solutions could be revisited.

Algorithm 8 Variable Neighborhood Descent (VND)

```
Select a set of neighborhood structures N_k, k = 1,...,k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do
    s' ← BestImprovement(N_k(s))
    if f(s') < f(s) then
      s ← s'
    else
      k ← k + 1
    end if
  end while
end while
```

As can be observed from the description given above, the choice of the neighborhood structures is the critical point of VNS and VND. The neighborhoods chosen should show and represent "different" properties and characteristics of the search space, that is, the neighborhood structures should give different *abstractions* of the search space. A variant of VNS is obtained by selecting the neighborhoods in such a way as to produce a problem decomposition. The algorithm is called Variable Neighborhood Decomposition Search (VNDS) and it is described in Algorithm 9. VNDS follows the usual VNS scheme, but the neighborhood structures and the local search are defined on sub-problems. For each solution, all attributes (usually variables) are kept fixed except for k of them. For each k a neighborhood structure $\mathcal{N}_k$ is defined. Local search only regards changes on the variables belonging to the sub-problem it is applied to.

Algorithm 9 Variable Neighborhood Decomposition Search (VNDS)

```
Select a set of neighborhood structures N_k, k = 1,...,k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do
    s' ← PickAtRandom(N_k(s)) {s and s' differs in a set of k attributes}
    s'' ← LocalSearch(s',Attributes) {only moves involving the k attributes are allowed}
    if f(s'') < f(s) then
      s ← s''
      k ← 1
    else
      k ← k + 1
    end if
  end while
end while
```

The decision to perform or not to perform a move can be varied as well. The acceptance criterion based on improvements is strongly steepest descent-oriented and it might not be suited to effectively exploring the search space. For example, when local optima are clustered,

VNS can quickly find the best optimum in a cluster, but it has no guidance to leave that cluster and find another one. Skewed VNS (SVNS) extends VNS by providing a more flexible acceptance criterion which takes also into account the distance from the current solution[5]. The algorithm is sketched in Algorithm 10. The new acceptance criterion is the following: besides always accepting improvements, worse solutions can be accepted if they are distant from the current one less than a value $\alpha\rho(s, s'')$. $\rho(s, s'')$ measures the distance between s and s'' and α is a parameter which quantifies the importance of the distance between the two solutions in the acceptance criterion.

Algorithm 10 Skewed Variable Neighborhood Search (SVNS)

```
Select a set of neighborhood structures N_k, k = 1,...,k_max
s ← GenerateInitialSolution()
while termination conditions not met do
  k ← 1
  while k < k_max do
    s' ← PickAtRandom(N_k(s))
    s'' ← LocalSearch(s')
    if f(s'') < f(s) then
      s ← s''
    end if
    if f(s'') − αρ(s, s'') < f(s) then
      s ← s''
      k ← 1
    else
      k ← k + 1
    end if
  end while
end while
```

Concluding, VNS and its variants have been successfully applied in several CO problems: like MAXSAT problems and also on graph problems [40].

2.2.4.3 Guided Local Search

Tabu Search and Variable Neighborhood Search deal explicitly with dynamic neighborhoods with the aim of efficiently and effectively exploring the search space. A different approach for guiding the search is to dynamically change the objective function. Among the most general methods which use this approach is Guided Local Search (GLS, [75, 50]).

The basic GLS principle is to help the search moving out gradually from local optima by changing the search landscape. In GLS the set of solutions and the neighborhood structure are kept fixed, while the objective function f is dynamically changed with the aim of making the current local optimum "less desirable". A pictorial description of this idea is given in Figure 2.1.

The mechanism used by GLS is based on *solution features*, which are any kind of properties

[5]Some distance measure between solutions has thus to be formally defined.

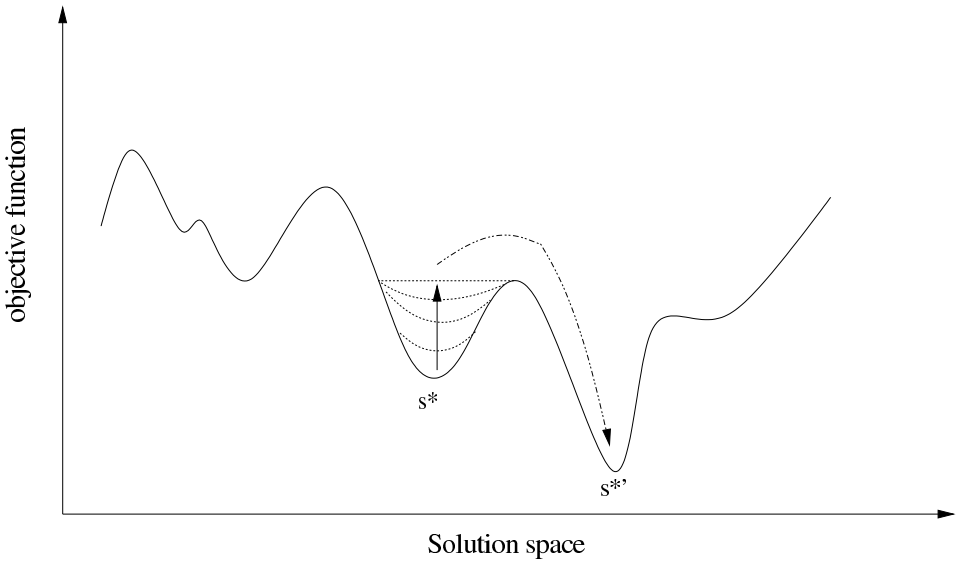


Figure 2.1: Basic GLS idea: moving from a local minimum by increasing its objective function value.

or characteristics which can be used to discriminate between solutions. For example, solution features in the TSP could be arcs between pairs of cities, while in the MAXSAT problem they could be the number of unsatisfied clauses. An indicator function $I_i(s)$ indicates whether the feature i is present in solution s:

$$I_i(s) = \begin{cases} 1 & : \text{ if feature } i \text{ is present in solution } s \\ 0 & : \text{ otherwise} \end{cases}$$

The objective function f is modified to yield a new objective function f' by adding a term which depends on the m features:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i \cdot I_i(s),$$

where p_i are called *penalty parameters* and λ is called the *regularization parameter*. The penalty parameters weight the importance of the features: the higher p_i, the higher the importance of feature i, thus the higher the cost of having that feature in the solution. The regularization parameter balances the relevance of features with respect to the original objective function.

The algorithm (see Algorithm 11) behaves as follows: it starts from an initial solution and it applies a local search method until a local minimum is reached. Then the array $\mathbf{p} = (p_1, \ldots, p_m)$ of penalties is updated by incrementing some of the penalties and the local search is started again. The penalized features are those which have the maximum *utility*:

$$Util(s,i) = I_i(s) \cdot \frac{c_i}{1+p_i}$$

where c_i are *costs* assigned to every feature i giving a heuristic evaluation of the relative importance of features with respect to others. The higher the cost, the higher the utility of features. Nevertheless, the cost is scaled by the penalty parameter to prevent the algorithm from being totally biased toward the cost and to make it sensitive to the search history.

Algorithm 11 Guided Local Search (GLS)

```
s ← GenerateInitialSolution()
while termination conditions not met do
  s ← LocalSearch(s, f')
  for all feature i with maximum utility Util(s, i) do
    p_i ← p_i + 1
  end for
  Update(f',p){where p is the penalty vector}
end while
```

The penalties update procedure can be modified by adding a multiplicative rule to the simple incrementing rule (that is applied every iteration). The multiplicative rule has the form: $p_i \leftarrow p_i \cdot \alpha$, where $\alpha \in]0,1[$. This rule is applied with a lower frequency than the incrementing one (for example every few hundreds of iterations) with the aim to decay the weights of penalized features so as to prevent the landscape from becoming too rugged. It is important to note that the penalties update rules are often very sensitive to the problem instance.

GLS has been successfully applied to TSP, QAP, weighted MAXSAT, Scheduling, and radio link frequency assignment problems [74].

2.2.4.4 Iterated Local Search

We conclude this presentation of explorative strategies with Iterated Local Search (ILS), the most general schema among the explorative strategies. On the one hand, its generality makes it a framework for other metaheuristics (like VNS); on the other hand, other metaheuristics can be easily included in it as sub-components. ILS is a simple but powerful metaheuristic algorithm [67, 66, 46, 48]. It applies local search to an initial solution until it finds a local optimum; then it perturbs the solution and it restarts local search. The importance of the perturbation is obvious: a very small change in the solution could not make it possible to escape from the basin of attraction of the local optimum just found, and on the other side, a too big modification gets close to be a simple random restart.

A local search is effective if it is able to find good local optima, that is, if it can find the basin of attraction of fixed points corresponding to those optima. When the search space is wide and/or when the basins of attraction of good local optima are small[6] a simple multi-start algorithm is almost useless. An effective search could be designed as a trajectory only in the set of local optima S^*, instead of in the set S of all the states. Unfortunately, the only way of introducing a neighborhood structure in S^* would be constructing (or at least, sampling) the basin of attraction of local optima involved in the search. As this is not feasible in practice, a trajectory along local optima $s_1^*, s_2^*, \ldots, s_t^*$ can be performed without explicitly introducing a neighborhood structure, by applying the following schema:

[6]The basin of attraction size of a point s (in a finite space), is defined as the fraction of initial states of trajectories which converge to point s.

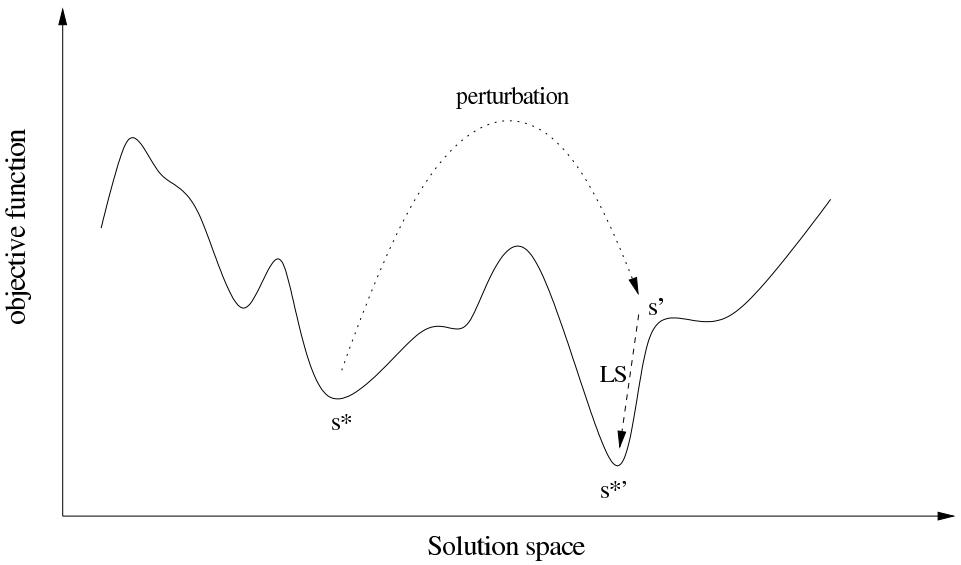


Figure 2.2: A desirable ILS step: the local optimum s^* is perturbed, then LS is applied and a new local minimum is found.

1. Execute local search (LS) from an initial state s until a local optimum s^* is found.
2. *Perturb* s^* and obtain s'.
3. Execute LS from s' until it finds a local optimum $s^{*'}$.
4. On the basis of an *acceptance criterion* decide whether to set $s^* \leftarrow s^{*'}$.
5. Goto step 2.

The task of the *perturbation* on s^* is to produce a starting point for local search such that it ends in a local optimum different from s^*, but *closer* than a local optimum reachable with a random restart. The *acceptance criterion* acts as a counterbalance, as it filters and gives feedbacks to the perturbation action, depending on the characteristics of the new local optimum. The high level algorithm for ILS is reported in Algorithm 12, as it appears in [46] and Figure 2.2 shows a possible (lucky) ILS step.

Algorithm 12 Iterated Local Search (ILS)

$s_0 \leftarrow$ GenerateInitialSolution()
$s^* \leftarrow$ LocalSearch(s_0)
while termination conditions not met **do**
  $s' \leftarrow$ Perturbation($s^*, history$)
  $s^{*'} \leftarrow$ LocalSearch(s')
  $s^* \leftarrow$ ApplyAcceptanceCriterion($s^*, s^{*'}, history$)
end while

The design of ILS algorithms has several degrees of freedom in the choice of the initial solution, perturbation and acceptance criteria. A key role is played by the *history* of the search which can be exploited both as short and long term memory.

The construction of initial solutions should be fast while computationally not expensive, and initial solutions should be a good starting point for local search. The fastest way of producing an initial solution is to generate it at random; however, this is the easiest way for problems where every possible assignment is a feasible solution, whilst in other cases the construction of a feasible solution requires also constraint checking. Constructive methods, guided by heuristics, can also be adopted. It is worth underlining that an initial solution is considered a good starting point depending on the particular LS applied and on the problem structure, thus the algorithm designer's goal is to find a trade-off between speed and quality of solutions.

The perturbation is usually non-deterministic, to avoid cycling. Its most important characteristic is the *strength*, roughly defined as the amount of changes made on the current solution. The strength can be either fixed or variable. In the first case, the distance between s^* and s' is kept constant, independently of the problem size. However, a dynamic strength is in general more effective, since it has been experimentally found that, in most of the problems, the bigger the problem size, the larger should be the strength. More sophisticated schemata are possible; for example, the strength can be adaptive: it increases when more diversification is needed and it decreases when intensification seems preferable. VNS and its variants belong to this category. An orthogonal choice with respect to strength is the kind of perturbation. It can be just random, or produced by an algorithm (e.g., a LS different from the one used in the main algorithm).

The third important component is the acceptance criterion. There are two extreme cases: accept the new local optimum only in case of improvement or always accept the new state. In between, there are several possibilities. For example, it is possible to adopt a kind of annealing schedule: accept all the improving new local optima and accept also the non-improving ones with a probability which is a function of the temperature T and the difference of objective function values. In formulas:

$$Prob[Accept(s^*, s', history)] = \begin{cases} 1 & : \text{ if} f(s^{*'}) < f(s^*) \\ exp(-\frac{f(s^{*'}) - f(s^*)}{T} & : \text{ otherwise} \end{cases}$$

The cooling schedule can be either monotonic (non-increasing function of time) or non-monotonic (adapted to tune the balance between diversification and intensification). The non-monotonic schedule is particularly effective if it exploits the history of the search, in a way similar to the Reactive Tabu Search [70] mentioned at the end of the section about Tabu Search. When intensification seems no longer effective, a diversification phase is needed and the temperature is increased.

Due to its flexibility and high level of abstraction, ILS can be considered the basic framework for most metaheuristics. Its applications range from TSP and MAXSAT problem to scheduling problems (for a summary of applications see [46]).

2.3 Population-based methods

In population-based methods we are dealing in every iteration of the algorithm with a set – a population – of solutions[7] rather than with a single solution to be manipulated. As they are dealing with a population of solutions, population-based algorithms provide a natural, intrinsic way for the exploration of the search space. Yet, the final performance depends strongly on the way the population is manipulated. The most studied population-based methods in combinatorial optimization are Evolutionary Computation (EC) and Ant Colony Optimization (ACO). In EC algorithms a population of individuals is modified by recombination and mutation operators, and in ACO a colony of artificial ants is used to construct solutions guided by the pheromone trails and heuristic information.

2.3.1 Evolutionary Computation (EC)

Evolutionary Computation (EC) algorithms are inspired by nature's capability to evolve living beings well adapted to their environment. EC algorithms can shortly be characterized as computational models of evolutionary processes. In every iteration a number of operators is applied to the individuals of the current population to generate the individuals of the population of the next generation (iteration). Usually we have operators to recombine two or more individuals to produce new individuals called *recombination* or *crossover* operators, and we have operators which cause a self-adaptation of individuals called *mutation* operators. The thriving force in evolutionary algorithms is the *selection* of individuals based on their *fitness* (this can be the value of an objective function or the result of a simulation experiment, or some other kind of quality measure). Individuals with a higher fitness have a higher probability to be chosen as members of the next iterations population (or as parents for the generation of new individuals). This corresponds to the principle of *survival of the fittest* in natural evolution. It is the capability of nature to adapt itself to a changing environment, which gave the inspiration for EC algorithms.

There has been a variety of slightly differing EC algorithms proposed over the years. Basically they are falling into three different categories which have been developed independently from each other. These are Evolutionary Programming (EP) developed by Fogel et al. in 1966 [28] [29], Evolutionary Strategies (ES) proposed by Rechenberg in 1973 [64] and Genetic Algorithms initiated by Holland in 1975 [43] (see [35] and [51] for further literature). EP arose from the desire to generate machine intelligence. While EP originally was proposed to operate on discrete representations of finite state machines, most of the present variants are used for continuous optimization problems. The latter also holds for most present variants of ES, whereas most applications of GAs are to solve combinatorial optimization problems. Over the years there have been quite a few overviews and surveys about EC methods. Among those are the ones by Bäck [3], by Fogel [27], by Spears et al. [65] and by Michalewicz et al. [49]. Calégary et al. [10] tried to give a taxonomy of EC algorithms.

In the following we concentrate on a "combinatorial optimization"-oriented introduction to EC algorithms. For doing this we follow an overview work by Hertz et al. [42], which gives, in our opinion, a good overview of the different components of EC algorithms and of the possibilities to define them. Algorithm 13 shows the basic structure of every EC algorithm.

[7] In general, especially in EC algorithms, we talk about a population of individuals rather than solutions

Algorithm 13 Evolutionary Computation (EC)

```
P ← GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met do
  P' ← Recombine(P)
  P'' ← Mutate(P')
  Evaluate(P'')
  P ← Select(P'' ∪ P)
end while
```

In this algorithm, P denotes the population of individuals. A population of offspring is generated by *recombination* and *mutation* operators and the individuals for the next population are *selected* from the union of the old population and the offspring population. The main features of an EC algorithm are:

Description of the individuals: EC algorithms handle populations of individuals. These individuals are not necessarily solutions of the considered problem. They may be partial solutions, or sets of solutions, or any object which can be transformed into one or more solutions in a structured way. Most commonly used in combinatorial optimization is the representation of solutions as bit-strings or as permutations of n integer numbers. Also possible are tree-structures or more complex structures for more complex optimization problems.

Evolution process: In each iteration it has to be decided which individuals will enter the population of the next iteration. This is done by a selection scheme. The strategy of only choosing among the offspring as individuals for the next population is called *generational replacement*. If it is possible to transfer individuals of the current population into the next population, then we are dealing with a so-called *steady state* evolution process.
Most EC algorithms work with populations of fixed size keeping at least the best individual always in the current population. It is also possible to have a variable population size. In case of a continuously shrinking population size, the situation where only one individual is left in the population (or no crossover partners can be found for any member of the population) can be one of the stopping criteria of the algorithm.

Neighborhood structure: A neighborhood function $\mathcal{N}_{\mathcal{EC}} : \mathcal{I} \to 2^{\mathcal{I}}$ on the set of individuals $\mathcal{I}$ assigns to every individual $i \in \mathcal{I}$ a set of individuals $\mathcal{N}_{\mathcal{EC}}(i) \subseteq \mathcal{I}$ which are permitted to act as recombination partners for i to create offspring. If an individual can be recombined with any other individual (as for example in the simple GA) we talk about an *unstructured* population, otherwise we talk about a *structured* population. An example for an EC algorithm using structured populations is the Parallel Genetic Algorithm proposed by Mühlenbein [54].

Information sources: The most common form of information sources to create offspring (i.e., new individuals) is a couple of parents (two-parent crossover). But there are also recombination operators recombining more than two individuals to create a new individual (multi-parent crossover), see [21]. More recent developments even use population statistics for generating the individuals of the next population. Examples are the recombination operators called Gene Pool Recombination [56] and Bit-Simulated Crossover [69] which are using a distribution over the search space given by the current population to generate the next population.

Infeasibility: An important characteristic of an EC algorithm is the way it is dealing with infeasible individuals. When recombining individuals the case might occur that the resulting individual is not feasible. There are basically three different ways to handle such a situation. The most simple thing to do is to *reject* infeasible individuals. On the other hand, for many problems it might be very difficult to find feasible individuals (for the timetabling problem for example it is in general very difficult to find a feasible solution). Therefore, the strategy of *penalizing* infeasible individuals in the function which measures the quality of an individual is sometimes more appropriate (or even unavoidable). The third possibility consists in trying to *repair* an infeasible solution (see [22] for an example).

Intensification strategy: In many applications it proved to be quite beneficial to use *improvement algorithms* to improve the fitness of individuals. EC algorithms using a local search algorithm on every individual of a population are often called Memetic Algorithms [52, 53]. While the use of a population ensures an exploration of the search space, the use of local search techniques helps to quickly identify "good" areas in the search space.
Another intensification strategy is the use of recombination operators which explicitly try to combine "good" parts of individuals (rather than, for example, a simple one-point crossover for bit-strings). This also concentrates the search performed by the EC algorithm to areas of individuals with certain "good" properties. Techniques of this kind are sometimes called *linkage learning* or *building block learning* (see [36, 72, 76, 41] as examples).

Diversification strategy: One of the major difficulties of EC algorithms (especially when applying local search) is the premature convergence toward sub-optimal solutions. The most common mechanism to diversify the search process is the use of a mutation operator. The simple form of a mutation operator just performs a small random perturbation of an individual, introducing a kind of *noise*. A more guided way of applying diversification is to systematically introduce new individuals coming from areas in the search space not explored so far. This can be achieved by constructing individuals according to some history memory keeping track for example of how often components of individuals appeared in individuals of past generations (see concepts of *global frequency* [33]).

This concludes the list of EC algorithm features. This list is of course not exhaustive and other ways of describing EC algorithms might be perfectly valid as well. In the next two sections we are going to introduce two other populations-based methods which are sometimes also regarded as being EC algorithms.

2.3.1.1 Scatter Search and Path Relinking

Scatter Search and its generalized form called Path Relinking [32, 34] differ from EC algorithms mainly by providing unifying principles for joining (or recombining) solutions based on generalized path constructions in Euclidean or neighborhood spaces. They also incorporate a few ideas coming from Tabu Search methods, as, for example, the use of adaptive memory and associated memory-exploiting mechanisms. The template for Scatter Search (resp. Path Relinking) is shown in Algorithm 14.

Scatter Search (resp. Path Relinking) is a search strategy that generates a set of solutions from a chosen set of *reference solutions* corresponding to feasible solutions to the problem under consideration. This is done by making combinations of subsets of the current set of reference solutions. The resulting solutions are called *trial solutions*. These trial solutions

Algorithm 14 Scatter Search and Path Relinking

Initial Phase

SeedGeneration()
repeat
  DiversificationGenerator()
  Improvement()
  ReferenceSetUpdate()
until the reference set is of cardinality n

Scatter Search/Path Relinking Phase:

repeat
  SubsetGeneration()
  SolutionCombination()
  Improvement()
  ReferenceSetUpdate()
until termination criteria met

may be infeasible solutions and are therefore usually modified by means of a repair procedure that transforms them into feasible solutions. An improvement algorithm is then applied in order to try to improve the set of trial solutions (usually this improvement procedure would be a local search). These improved solutions form the set of *dispersed solutions*. The new set of reference solutions that will be used in the next iteration is selected from the current set of reference solutions and the newly created set of dispersed solutions. This is captured in the procedures shown in algorithm 14. The components of the pseudo-code are explained in the following:

SeedGeneration(): One or more seed solutions, which are arbitrary trial solutions, are created and used to initiate the remainder of the method.

DiversificationGenerator(): A procedure to generate a collection of diverse trial solutions, using an arbitrary trial solution (or seed solution) as an input.

Improvement(): A method – usually a local search – to transform a trial solution into one or more enhanced trial solutions. Neither the input nor the output solutions are required to be feasible, though the output solutions will more usually be expected to be so. It might be necessary to apply repair methods to infeasible solutions.

ReferenceSetUpdate(): The reference set update procedure is responsible for building and maintaining a reference set consisting of a number of "best" solutions found in the course of the algorithm. The attribute "best" covers features such as quality of solutions and diversity of solutions (the solutions in the reference set should be of good quality and they should be diverse).

SubsetGeneration(): This method operates on the reference set, to produce a subset of its solutions as a basis for creating combined solutions.

SolutionCombination(): A procedure to transform a given subset of solutions produced by the

subset generation method into one or more combined solutions. In Scatter Search, which was introduced for solutions encoded as points in the Euclidean space, new solutions are created by building linear combinations of reference solutions using both positive and negative weights. This means that trial solutions can be both, inside and outside the convex region spanned by the reference solutions. In Path Relinking the concept of combining solutions by making linear combinations of reference points is generalized to neighborhood spaces. Linear combinations of points in the Euclidean space can be re-interpreted as paths between and beyond solutions in a neighborhood space. To generate the desired paths, it is only necessary to select moves that fulfill the following: upon starting from an *initiating solution*, the moves must progressively introduce attributes contributed by a *guiding solution*. Multi-parent path generation possibilities emerge in Path Relinking by considering the combined attributes provided by a set of guiding solutions, where these attributes are weighted to determine which moves are given higher priority.

2.3.1.2 Evolutionary Algorithms based on probabilistic models

In the last decade more and more researchers tried to overcome the drawbacks of usual recombination operators of EC algorithms which are likely to break good building blocks. So, a number of algorithms – sometimes called Estimation of Distribution Algorithms (EDA) [55] – have been developed (see algorithm 15). These algorithms, which have a theoretical foundation in probability theory, are also based on populations that evolve as the search progresses. EDAs use probabilistic modeling of promising solutions to estimate a distribution over the search space which is then used to generate the next generation by sampling the search space according to the estimated distribution. After every iteration the distribution is re-estimated. For a survey of EDAs see [62].

Algorithm 15 Estimation of Distribution Algorithms (EDAs)

$P \leftarrow$ InitializePopulation()
while termination criteria not met **do**
  $P_{sel} \leftarrow$ Select(P)
  $\rho(\mathbf{x}) = \rho(\mathbf{x}|P_{sel}) \leftarrow$ EstimateProbabilityDistribution()
  $P \leftarrow$ SampleProbabilityDistribution()
end while

One of the first EDAs proposed for combinatorial optimization is called Population-based Incremental Learning (PBIL) [4, 5]. The objective of this method is to create a real valued probability vector (each position corresponds to a decision variable) which, when sampled, generates high quality solutions with high probability. Initially, the values of the probability vector are initialized to 0.5. The goal of shifting the values of this probability vector in order to generate high quality solutions is accomplished as follows: a number of solution vectors are generated according to the probability vector. Then the probability vector is pushed toward the generated solution vector(s) with highest quality. The distance that the probability vector is pushed depends on the learning rate parameter. After that, the cycle is repeated. The probability vector can be viewed as a prototype vector for generating high quality solution vectors with respect to the available knowledge about the search space. The drawback of this method is the fact that it does not automatically provide a way to deal with

constrained problems. In contrast to PBIL which is estimating a distribution of promising solutions assuming that the decision variables are independent, various other approaches try to estimate distributions taking into account dependencies between decision variables. An example for EDAs regarding pairwise dependencies between decision variables is MIMIC [12] and an example for multivariate dependencies is the Bayesian Optimization Algorithm (BOA) [61].

2.3.2 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is a metaheuristic approach proposed by Dorigo [16] and later by Dorigo and colleagues [20, 18]. In the course of this section we keep close to the description of ACO given in [17].
The inspiring source of ACO is the foraging behavior of real ants. This behavior – described by Deneubourg et al. in [14] – enables them to find shortest paths between food sources and their nest. While walking from food sources to the nest and vice versa, ants deposit a substance called *pheromone* on the ground. When they decide about a direction to go they choose, in probability, paths marked by strong pheromone concentrations. This basic behavior is the basis for a cooperative interaction which leads to the emergence of shortest paths.
In ACO algorithms, an artificial ant incrementally constructs a solution by adding opportunely defined solution components to a partial solution under consideration[8]. For doing that, artificial ants perform randomized walks on a completely connected graph $\mathcal{G} = (\mathcal{C}, \mathcal{L})$ whose vertices are the solution components $\mathcal{C}$ and the set $\mathcal{L}$ are the connections. This graph is commonly called *construction graph*. The problem constraints Ω are build into the ants' constructive procedure in a way such that in every step of the construction process only feasible solution components are permitted to be added to the current partial solution. In most applications, ants are implemented to build feasible solutions, but sometimes it is unavoidable to work on unfeasible solutions. Components $c_i \in \mathcal{C}$ and connections $l_{ij} \in \mathcal{L}$ can have associated a *pheromone trail* τ (τ_i if associated to components, τ_{ij} if associated to connections), and a *heuristic value* η (η_i and η_{ij} respectively) representing *a priori* or run time information about the problem instance. These values are used by the ants' heuristic rule to make probabilistic decisions on how to move on the construction graph. The probabilities involved in moving on the construction graph are commonly called *transition probabilities*.

Algorithm 16 Ant System (AS)

```
InitializePheromoneValues()
while termination conditions not met do
  for all ants a ∈ A do
    s_a ← ConstructSolution(τ,η)
  end for
  ApplyOnlineDelayedPheromoneUpdate()
end while
```

The first ACO algorithm proposed is called Ant System (AS) [20]. The pseudo-code for this algorithm is shown in algorithm 16. In this algorithm $\mathcal{A}$ denotes the set of ants and s_a denotes

[8]Therefore, the ACO metaheuristic can be applied to any combinatorial optimization problem for which a constructive heuristic can be defined

the solution constructed by ant $a \in \mathcal{A}$. After initialization of the pheromone values, in every step of the algorithm every ant constructs a solution. These solutions are then used to update the pheromone values. The components of this algorithm are to be explained in more detail in the following:

InitializePheromoneValues(): At the beginning of the algorithm the pheromone values (τ_i and/or τ_{ij}) are initialized to the same small numerical value $ph > 0$.

ConstructSolution(τ,η): In the construction phase an ant incrementally constructs a solution by step-by-step adding solution components to the partial solution constructed so far. The probabilistic choice of the next solution component to be added is done by using the transition probabilities, which in AS are determined by the following *state transition rule*:

$$p(c_r|s_a[c_l]) = \begin{cases} \frac{[\eta_r]^\alpha [\tau_r]^\beta}{\sum_{c_u \in J(s_a[c_l])} [\eta_u]^\alpha [\tau_u]^\beta} & \text{if} \quad c_r \in J(s_a[c_l]) \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

In this formula α and β are parameters to adjust the relative importance of heuristic information and pheromone values and $J(s_a[c_l])$ denotes the set of solution components which are allowed to be added to the partial solution $s_a[c_l]$ with c_l as the last component added (note that for the sake of simplicity in the above formula we are only dealing with pheromone on solution components).

ApplyOnlineDelayedPheromoneUpdate(): Once all ants have constructed a solution, the *online delayed pheromone update rule* is applied:

$$\tau_j \leftarrow (1 - \rho) \cdot \tau_j + \sum_{a \in \mathcal{A}} \Delta\tau_j^{s_a} \tag{2.2}$$

where

$$\Delta\tau_j^{s_a} = \begin{cases} \frac{Q}{f(s_a)} & \text{if } c_j \text{ was used in solution } s_a \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where $f(s_a)$ is the quality of solution s_a, $0 < \rho < 1$ is a pheromone evaporation rate and Q is a parameter usually set to 1. This pheromone update rule leads to an increase of pheromone on solution components which have been found in better quality solutions than other solution components (where the pheromone value will decrease).

In the following we are dealing with the more general ACO metaheuristic, which is based on the same basic principles as AS. The ACO metaheuristic framework shown in algorithm 17 covers all the improvements and extensions of AS which have been developed over the years. It is consisting of three parts gathered in the ScheduleActivities construct. The ScheduleActivities construct does not specify how these three activities are scheduled and synchronized. This is up to the algorithm designer.

ManageAntsActivities(): An ant builds constructively a solution to the problem by moving through nodes of the construction graph $\mathcal{G}$. Ants move by applying a stochastic local decision policy that makes use of the pheromone values and the heuristic values on components and/or connections of the construction graph (see the state transition rule 2.1 of AS as an example). While moving the ant is keeping in memory the partial solution it has built in terms of the path it was walking on the construction graph. When adding a component c_j to the current

Algorithm 17 Ant Colony Optimization (ACO)

```
while termination conditions not met do
  ScheduleActivities
    ManageAntsActivity()
    EvaporatePheromone()
    DaemonActions() {optional}
  end ScheduleActivities
end while
```

partial solution, it can update the pheromone trail(s) τ_i and/or τ_{ji} (in case the ant was walking on connection l_{ji} in order to reach component c_i). This kind of pheromone update is called *online step-by-step pheromone update*. Once an ant has build a solution, it can (by using the memory of the walked path) retrace the same path backward and update the pheromone trails of the used components and/or connections according to the quality of the solution it has build. This is called *online delayed pheromone update*.

EvaporatePheromone(): Pheromone evaporation is the process by means of which the pheromone trail intensity on the components decreases over time. From a practical point of view, pheromone evaporation is needed to avoid a too rapid convergence of the algorithm toward a sub-optimal region. It implements a useful form of *forgetting*, favoring the exploration of new areas in the search space.

DaemonActions(): Daemon actions can be used to implement centralized actions which cannot be performed by single ants. Examples are the use of a local optimization procedure applied to the solutions build by the ants, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from an non-local perspective. As a practical example, the daemon can observe the path found by each ant in the colony and choose to deposit extra pheromone on the components used by the ant that built the best solution. Pheromone updates performed by the daemon are called *offline pheromone updates*.

Within the ACO metaheuristic framework as shortly described above the currently best working versions in practice are Ant Colony System (ACS) [19] and $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) [68]. In the following we are going to outline the peculiarities of these algorithms shortly:

Ant Colony System (ACS): The ACS algorithm has been introduced to improve on the performance of Ant System (AS) [20]. ACS is based on AS but presents some important differences. First, the daemon updates pheromone trails offline: At the end of an iteration of the algorithm – once all the ants have built a solution – pheromone is added to the arcs used by the ant that found the best solution from the start of the algorithm. Second, ants use a different decision rule to decide to which component to move next in the construction graph. The rule is called *pseudo-random-proportional* rule. With this rule, some moves are chosen deterministically (in a greedy manner), others are chosen probabilistically with the usual decision rule. Third, in ACS, ants perform only online step-by-step pheromone updates. These updates are performed to favor the emergence of other solutions than the best so far.

$\mathcal{MAX}$-$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS): $\mathcal{MM}$AS is again an extension of AS. First, the pheromone trails are only updated offline by the daemon (the arcs that were used by the

iteration best ant or the best ant since the start of the algorithm receive additional pheromone). Second, the pheromone trail values are restricted to an interval $[\tau_{min}, \tau_{max}]$ and the pheromone trails are initialized to their maximum value τ_{max}. Putting explicit limits on pheromone trails prevents the probability for any solution to be constructed to drop below a certain value greater than 0. This means that the chance of finding a global optimum never vanishes during the course of the algorithm.

Recently researchers have been dealing with finding similarities between ACO algorithms and algorithm from EC or other Probabilistic Learning algorithms. An important step into this direction was the development of the Hyper-cube Framework for Ant Colony Optimization (HC-ACO) proposed by Blum et al. [9]. In this framework the introduction of a kind of normalized online delayed pheromone update rule enables to draw explicit connections to algorithms from EC like a simple GA using a recombination operator called Gene Pool Recombination [56].

Chapter 3

Conceptual Comparison

As we mentioned earlier in this work, there are several characteristics which can be used to compare or distinguish the different metaheuristic approaches. We could for example compare nature-inspired algorithms against non nature-inspired ones, or population-based ones against trajectory methods. The disadvantage of such a kind of comparison is that it probably would not discover qualitative differences. To our opinion, the most important ingredients of metaheuristic approaches are intensification and diversification strategies. Depending on the paradigm behind a particular metaheuristic, the way of implementing intensification and diversification is realized in different ways. In this chapter we try to give an overview over the way intensification and diversification are implemented by the different metaheuristics. This can be seen as a first step toward exploring the reasons for the success or the failure of metaheuristic approaches to different optimization problems. Furthermore, we want to point out some strengths and weaknesses of the different metaheuristic approaches which could lead to the goal-directed development of hybrid algorithms combining concepts originating from different metaheuristics.

3.1 The basic use of intensification and diversification

Every metaheuristic is designed with the aim of effectively and efficiently exploring a search space. The search performed by a metaheuristic should be "clever" enough to both intensively explore areas of the search space with high quality solutions and to move to unexplored areas of the search space when necessary. The first aim has a kind of local character (exploration in a local sense) which is called *intensification* of the search process. The second aim has a kind of global character (exploration in a global sense) which is achieved by an appropriate *diversification* of the search process. The basic versions of all the metaheuristics have built-in mechanisms to deal with intensification and diversification of the search process.

The notion of "area" (or "region") of the search space and of "locality" can only be expressed in a fuzzy way, as they always depend on the characteristics of the search space as well as on the definition of metrics on the search space (distances between solutions). In the following we will refer to the intuitive notion of "area" with the meaning of subset (connected or not) of the search space and of "locality" as an attribute of pairs of points with *small* distance in the search space.

The intensification and diversification mechanisms occurring in metaheuristics can be divided in intrinsic (basic) ones and strategic ones. The intrinsic intensification (diversification) mechanisms are given by the basic behavior of the algorithm. On the other side, strategic intensification (diversification) mechanisms are composed of techniques and strategies the algorithm designer adds to the basic procedure in order to improve global performance. These strategies are applicable to almost all metaheuristics and some of them, originally developed for a specific algorithm, can be useful also for the others. It is worth observing that basic intensification and diversification in general act simultaneously, whilst the strategic counterparts usually alternate. Moreover, some metaheuristics have a static balance of intensification and diversification, whilst others dynamically change it. The following considerations are referring to the basic variants of the metaheuristics, as they were described in their first appearance in the literature. A synthesis of the description given in the following is presented in Table 3.1.

Simulated Annealing. The basic intensification strategy of Simulated Annealing is the local search which is intrinsic to the system. At the beginning of the search process the temperature parameter T is set to a high value, such that the local search which is performed by the system is not very goal oriented. This results in a high diversification and a low intensification. As the system proceeds, T is decreased and the balance between intensification and diversification changes. The importance of intensification is gradually increased and at the same time the importance of diversification is gradually decreased until the system works nearly like a strictly descending local search. The parameter T defines a changing balance between intensification and diversification converging to a state where we only have intensification and no diversification. This eventually leads to a convergence[1] of the system.

Tabu Search. The basic intensification strategy of simple Tabu Search is like in Simulated Annealing the local search as the basic search strategy of the system. A further intensification action is achieved by the aspiration condition.

The goal of diversification is reached by the use of one or more tabu lists which prevent the system from returning (or staying) in areas of the search space recently explored. The balance between intensification and diversification is determined by the size of the tabu list. Smaller tabu lists result in higher intensification and lower diversification. On the contrary, large tabu lists result in lower intensification and higher diversification. In contrast to Simulated Annealing, in Tabu Search the balance between intensification and diversification is static. Tabu Search is not designed to be a converging system, although it might get trapped in local minima if the size of the tabu list is not big enough[2].

Evolutionary Computation. The intensification strategy of a basic evolutionary algorithm is given by the selection operator. The selection operator concentrates the search process in some areas of the search space. This process is reinforcing from iteration to iteration and leads eventually to the convergence of the system.

There are several diversification mechanisms inherent to an EC algorithm. The search process performed by EC algorithms keeps a natural diversity by working on populations rather than on a single individual. Further diversification is reached by applying recombination and

[1] In this section we use the term convergence in the sense of getting stuck in the basin of attraction of a local minimum

[2] Recently a result about the convergence to a gobal minimum based on a enumeration scheme was published [37], but the considered version of TS is unfortunately not applicable in practice, because it needs exponential storage space

modification operators. Both, recombination of individuals and modification of individuals, lead the search process potentially to areas of the search space not "covered" by the current population. As in Simulated Annealing, during the evolution of the search process, the balance between intensification and diversification changes from high diversification and low intensification to low diversification and high intensification as the diversity in the population is decreasing.

Ant Colony Optimization. The basic intensification mechanism of Ant Colony Optimization is given by the pheromone updating rules which reinforce the selection of "proven to be good" solution components during the solution construction steps performed by the ants in consecutive iterations. The application of pheromone updating rules leads eventually to the convergence of the system. The structure of the pheromone updating rules determines the rate of change in the balance between intensification and diversification.

Diversification is reached in Ant Colony Optimization by the probabilistic nature of the solution construction mechanism. At the beginning of a "run" of the algorithm, usually all the pheromone values are set to the same small and positive constant. This corresponds to a maximum amount of diversification. By applying pheromone update the search process is continuously intensified and diversification decreases.

Explorative Local Search Methods

- **Iterated Local Search.** The most important intrinsic intensification mechanism in ILS algorithms is local search. Diversification is achieved by the function which modifies the current local optimum (also called a kick-move). The balance between intensification and exploration is determined by the structure of the acceptance criterion. Intensification is achieved, for example, by accepting only local optima which are better than the current one. Higher diversification can be achieved by accepting also local optima which are worse than the current one. Diversification in the extreme case may be achieved by accepting every new local optimum.

- **Variable Neighborhood Search.** VNS is one of the metaheuristic methods fitting into the ILS framework. It provides an oscillating balance between intensification and diversification. This is achieved by performing kick-moves choosing solutions from a set of neighborhoods of increasing distance from the current solution (see section about VNS). If the last local optimum was not accepted, a neighborhood with higher distance is choosen (increasing diversification). On the contrary, if the last local optimum was accepted, in the next iteration again the neighborhood with the lowest distance is choosen (intensification).

- **GRASP.** In a simple GRASP algorithm we have basically two "ingredients" to achieve intensification of the search process. The first one is the local search used in the improvement phase. The second one is the length of the restricted candidate list in the construction phase. This parameter basically determines the balance between intensification and diversification. If this parameter is set to 1, the construction of solutions is done in the "greedy" manner and intensification is high. On the contrary, if this parameter permits any possible solution component to be chosen in the next step of the construction mechanism, diversification is high, because then the starting solutions for local search are basically randomly chosen solutions.

Table 3.1: Comparisons of basic and strategic intensification and diversification components between metaheuristics

Metaheuristic	Basic Intensification	Basic Diversification
SA	basic local search	uphill moves (temperature T)
TS	basic local search	tabu list (short term memory)
EC	selection	modification operators (e.g., mutation)
ACO	pheromone updating rules	probabilistic construction rule
ELS	basic local search	acceptance criterion and perturbation

- **Guided Local Search.** In GLS the basic intensification mechanism is local search. Diversification is achieved by augmenting the objective function of the problem with a term penalizing features of solutions visited in the recent past. The *regularization parameter* λ used to weight the influence of the penalty term determines the balance between intensification and diversification. If λ is large then the search process will mainly concentrate on removing penalized solution features from the current solution. This leads to a high diversification. On the contrary, if λ is small then the system will have difficulties to escape local minima.

In our opinion the right balance between intensification and diversification is needed to obtain an effective metaheuristic. Moreover we believe that this balance must not be fixed or only changing into one direction (i.e., continuously decreasing diversification and increasing intensification as in basic EC algorithms). We believe that the balance between intensification and diversification should be oscillating while moving from a state of high diversification and low intensification to a state of low diversification and high intensification. As it is clear from the description above, except for VNS (which provides an oscillating balance) the balance of intensification and diversification provided by the basic versions of the metaheuristics is either static or starts with high diversification and low intensification converging to low diversification and high intensification. In many applications additional mechanisms for intensification and diversification have been introduced to extend the basic methods.

3.2 More sophisticated intensification and diversification

The simplest way of achieving an oscillating balance is restarting a metaheuristic depending on the search history. The concept of *global frequency* is well known from Tabu Search applications. In this concept, the number of occurrences of solution components is counted during the run of the algorithm. These numbers, called the global frequency numbers, are then used for changing the heuristic method, for example to generate a starting population of individuals for an EC method. Similarly, the concept of *global desirability* (which keeps for every solution component the objective function value of the best solution it had been a member of) can be used to restart algorithms depending on search history.

More sophisticated ways of achieving this balance for some of the metaheuristics include:

Simulated Annealing: An increase in diversification and simultaneous decrease in intensification can be achieved by "re-heating" the system and then cooling it down again (which

corresponds to increasing parameter T and decreasing it again according to some scheme). Such a cooling scheme is called *non-monotonic* cooling scheme (e.g., see [47] or [58]).

Tabu Search: In Tabu Search there are several ways of achieving an oscillating balance. One of them is used in the well known Reactive Tabu Search by Battiti et al. [8]. It consists in dynamically changing the length of the tabu list(s) used. To intensify the search the length has to be reduced, and to diversify the search the length has to be increased. The length of the tabu list is varied according to the properties of the trajectory in the search space.

Evolutionary Computation: In evolutionary algorithms it is not very usual to interrupt the dynamics of the process (transition from high diversification and low intensification to low diversification and high intensification). Some applications though change the selection operator when the system is nearly converged to diversify the search process. This corresponds to a kind of soft restart of the algorithm.

Ant Colony Optimization: For ACO algorithms the same considerations hold as for EC algorithms. The convergence process is usually not interrupted. The best known exception is Ant Colony System (ACS). This ACO algorithm uses an additional diversification mechanism during the solution construction phase. While an ant is walking on the construction graph to construct a solution it reduces the pheromone values on the nodes/arcs of the construction graph that it visits. This has the effect that for the other ants the probability of taking the same path is reduced. This additional pheromone update mechanism is called *step-by-step online pheromone update rule*. This leads to an oscillating balance of intensification, performed by the *online delayed pheromone update rules* and diversification performed by the *online pheromone update rule*. Another diversification mechanism to obtain an oscillating balance is smoothing or resetting pheromone values [68].

3.3 Hybridization of metaheuristics

The comparative analysis of the metaheuristics based on basic/strategic components helps in outlining similarities and differences. Moreover, it gives insight into the behavior of the algorithms and might lead to understand what components are the most (least) effective for a given kind of problem. This in turn can lead to the design of hybrid metaheuristics (see [71] for a taxonomy of hybrid metaheuristics).

The power of population-based methods is certainly to be found in the concept of recombining solutions to obtain new ones. In EC algorithms and Scatter Search we have explicit recombinations implemented by one or more recombination operators. In ACO and EDAs we have implicit recombination, because it is a distribution over the search space (in ACO given by the pheromone values) generated by the last and/or earlier populations which is then used to generate new solutions. This permits to make guided steps in the search space which are usually "larger" than the steps done in trajectory methods. In other words, the solution resulting from a recombination in population-based methods is usually more "different" from the parents than, say, a predecessor solution to a successor solution (obtained by applying a move) in Tabu Search. We also have "big" steps in trajectory methods like Iterated Local Search and Variable Neighborhood Search, but in these methods the steps are usually not guided (these steps are rather called "kick move" or "perturbation" indicating the lack of guidance). It is interesting to note, that in all population-based methods we have

mechanisms where the good solutions found during the search are influencing the search process in the hope to find better solutions in-between those solutions and current solutions. In Path Relinking this idea is implemented in the most explicit way. We have guiding solutions (which are the good solutions found) and we have initiating solutions. Then new solutions are produced by applying moves to decrease the distance of the resulting solution and the guiding solution. In EC algorithms this is often realized by keeping the best (or a number of best) solution(s) found since the beginning of the respective run of the algorithm in the population. This is called a *steady state* evolution process. In Scatter Search we have a steady state process by definition. In some ACO implementations (see for example [68]) a pheromone updating schedule is applied such that in a situation where the algorithm is nearly converged to a solution only the best found solution since the start of the algorithm is used for updating the pheromone trails. This corresponds to "changing direction" and directing the search process toward a very good solution in the hope to find better ones on the way.

The strength of trajectory methods is rather to find in the way they explore a promising region in the search space. As local search is the intrinsic intensification method of all trajectory methods, a promising area in the search space is searched in a more structured way than in population-based methods. In this way the danger of being close to good solutions but "missing" them is not as high as in population-based methods.
In summary we could say that population-based methods are better in identifying promising areas in the search space whereas trajectory methods are better in exploring promising areas in the search space. This is one of the reasons why metaheuristic hybrids are the most successful metaheuristics applications in practice and that's certainly the way to go in the future.

3.4 Conclusions

In this work we presented and compared metaheuristic methods. In section 2 we have outlined the basic metaheuristics as they are described in the literature. In the last section we then tried to do a conceptual comparison of the different metaheuristics on the way they implement the two main concepts in the search process performed by metaheuristics: Intensification and diversification of the search process.
In summary, we can say that although the metaheuristics are different in the sense that some of them are population-based (EC, ACO), and others are trajectory methods (SA, TS, ILS, VNS, GRASP), and although they are based on different philosophies, the mechanisms to efficiently explore a search space are all based on intensification and diversification. Nevertheless, it is possible to identify "sub-tasks" in the search process where some metaheuristics perform better than others. This has to be examined more closely in the future in order to be able to produce hybrid metaheuristics performing considerably better than their "pure" parents. In fact we can find this phenomenon in many facets of life, not just in the world of algorithms. Mixing and hybridizing is often better than purity.

Acknowledgments

We thank Marco Dorigo for his support, suggestions and useful discussions. This work was supported by the "Metaheuristics Network", a Research Training Network funded by the Improving Human Potential program of the CEC, contract HPRN-CT-1999-00106. Andrea Roli acknowledges support from the CEC through a "Marie Curie Training Site" fellowship, contract HPMT-CT-2000-00032. The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication.

Bibliography

[1] E.H.L. Aarts, J.H.M. Korst, and P.J.M. van Laarhoven. Simulated annealing. In Emile H. L. Aarts and Jan Karel Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 91–120. Wiley-Interscience, Chichester, England, 1997.

[2] E.H.L. Aarts and J.K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience, 1997.

[3] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[4] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical Report No. CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA, 1994.

[5] S. Baluja and R. Caruana. Removing the Genetics from the Standard Genetic Algorithm. In A. Prieditis and S. Russel, editors, *The International Conference on Machine Learning 1995*, pages 38–46, San Mateo, California, 1995. Morgan Kaufmann Publishers.

[6] Y. Bar–Yam. *Dynamics of Complex Systems*. Studies in nonlinearity. Addison–Wesley, 1997.

[7] R. Battiti and M. Protasi. Reactive Search, a history-base heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 1997.

[8] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[9] C. Blum, A. Roli, and M. Dorigo. HC–ACO: The hyper-cube framework for Ant Colony Optimization. In *Proceedings of MIC'2001 – Meta–heuristics International Conference*, volume 2, pages 399–403, Porto, Portugal, 2001. Also available as technical report TR/IRIDIA/2001-16, IRIDIA, Université Libre de Bruxelles.

[10] P. Calegari, G. Coray, A. Hertz, D. Kobler, and P. Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. *Journal of Heuristics*, 5:145–158, 1999.

[11] V. Cerny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[12] J.S. de Bonet, C.L. Isbell Jr., and P. Viola. MIMIC: Finding optima by estimating probability densities. In M.C. Mozer, M.I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9, page 424. The MIT Press, 1997.

[13] M. Dell'Amico, A. Lodi, and F. Maffioli. Solution of the Cumulative Assignment Problem with a well–structured Tabu Search method. *Journal of Heuristics*, 5:123–143, 1999.

[14] J.-L. Deneubourg, S. Aron, S. Goss, and J.-M. Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behaviour*, 3:159–168, 1990.

[15] R.L. Devaney. *An introduction to chaotic dynamical systems*. Addison–Wesley, second edition, 1989.

[16] M. Dorigo. *Optimization, Learning and Natural Algorithms* (in Italian). PhD thesis, DEI, Politecnico di Milano,Italy, 1992. pp. 140.

[17] M. Dorigo and G. Di Caro. The Ant Colony Optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, 1999. Also available as Technical Report IRIDIA/99-1, Université Libre de Bruxelles, Belgium.

[18] M. Dorigo, G. Di Caro, and L. M. Gambardella. Ant algorithms for discrete optimization. *Art. Life*, 5(2):137–172, 1999.

[19] M. Dorigo and L.M. Gambardella. Ant colony system: A cooperative learning approach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

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

[21] A.E. Eiben, P.-E. Raué, and Z. Ruttkay. Genetic algorithms with multi-parent recombination. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, volume 866 of *Lecture Notes in Computer Science*, pages 78–87, Berlin, 1994. Springer.

[22] A.E. Eiben and Z. Ruttkay. Constraint satisfaction problems. In T. Bäck, D. Fogel, and M. Michalewicz, editors, *Handbook of Evolutionary Computation*. IOP Publishing Ltd. and Oxford University Press, 1997.

[23] W. Feller. *An Introduction to Probability Theory and its Applications*. John Whiley, 1968.

[24] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[25] P. Festa and M.G.C. Resende. GRASP: an annotated bibliography. In P. Hansen and C.C. Ribeiro, editors, *Essays and Surveys on Metaheuristics*. Kluwer Academic Publishers, 2001.

[26] M. Fleischer. Simulated Annealing: past, present and future. In C. Alexopoulos, K. Kang, W.R. Lilegdon, and G. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 155–161, 1995.

[27] D.B. Fogel. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, 5(1):3–14, January 1994.

[28] L.J. Fogel. Toward inductive inference automata. In *Proceedings of the International Federation for Information Processing Congress*, pages 395–399, Munich, 1962.

[29] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, 1966.

[30] M.R. Garey and D.S. Johnson. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[31] F. Glover. Future paths for integer programming and links to artificial intelligence. *Comp. Oper. Res.*, 13:533–549, 1986.

[32] F. Glover. Scatter search and path relinking. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, Advanced topics in computer science series. McGraw-Hill, 1999.

[33] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[34] F. Glover, M. Laguna, and R. Marti. Fundamentals of scatter search and path relinking. *Control and Cybernetics*, 29(3):653–684, 2000.

[35] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, Reading, MA, 1989.

[36] D.E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, La Jolla, CA, 1991. Morgan Kaufmann.

[37] S. Hanafi. On the convergence of tabu search. *Journal of Heuristics*, 7:47–58, 2000.

[38] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[39] P. Hansen and N. Mladenović. An introduction to variable neighborhood search. In Stefan Voss, Silvano Martello, Ibrahim Osman, and Catherine Roucairol, editors, *Metaheuristics: advances and trends in local search paradigms for optimization*, chapter 30, pages 433–458. Kluwer Academic Publishers, 1999.

[40] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467, 2001.

[41] G. Harik. Linkage learning via probabilistic modeling in the ECGA. Technical Report No. 99010, IlliGAL, University of Illinois, 1999.

[42] A. Hertz and D. Kobler. A framework for the description of evolutionary algorithms. *European Journal of Operational Research*, 126:1–12, 2000.

[43] J.H. Holland. *Adaption in natural and artificial systems*. The University of Michigan Press, Ann Harbor, MI, 1975.

[44] L. Ingber. Adaptive simulated annealing (ASA): Lessons learned. *Control and Cybernetics – Special Issue on Simulated Annealing Applied to Combinatorial Optimization*, 25(1):33–54, 1996.

[45] S. Kirkpartick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science, 13 May 1983*, 220(4598):671–680, 1983.

[46] H. Ramalhino Lourenco, O. Martin, and T. Stützle. Iterated Local Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*. To appear. Also available at http://www.intellektik.informatik.tu-darmstadt.de/~tom/pub.html.

[47] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34(1):111–124, 1986.

[48] O. Martin, S.W. Otto, and E.W. Felten. Large-step markov chains for the traveling salesman problem. *Complex Systems*, 5(3):299–326, 1991.

[49] Z. Michalewicz and M. Michalewicz. Evolutionary computation techniques and their applications. In *Proceedings of the IEEE International Conference on Intelligent Processing Systems*, pages 14–24, Beijing, China, 1997. Institute of Electrical & Electronics Engineers, Incorporated.

[50] P. Mills and E. Tsang. Guided Local Search for solving SAT and weighted MAX-SAT Problems. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000*, pages 89–106. IOS Press, 2000.

[51] M. Mitchell. *An introduction to genetic algorithms*. MIT press, Cambridge, MA, 1998.

[52] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Toward memetic algorithms. Tech. Rep. Caltech Concurrent Computation Program 826, California Institute of Technology,Pasadena, California, USA, 1989.

[53] P. Moscato. Memetic algorithms: A short introduction. In F. Glover D. Corne and M. Dorigo, editors, *New Ideas in Optimization*. McGraw-Hill, 1999.

[54] H. Mühlenbein. Evolution in time and space – the parallel genetic algorithm. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann, San Mateo, USA, 1991.

[55] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1411 of *Lecture Notes in Computer Science*, pages 178–187, Berlin, 1996. Springer.

[56] H. Mühlenbein and H.-M. Voigt. Gene Pool Recombination in Genetic Algorithms. In I.H. Osman and J.P. Kelly, editors, *Proc. of the Metaheuristics Conference*, Norwell, USA, 1995. Kluwer Academic Publishers.

[57] G.L. Nemhauser and A.L. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, New York, 1988.

[58] I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[59] I.H. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:513–623, 1996.

[60] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications, Inc., New York, 1982.

[61] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, and R.E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532, Orlando, FL, 13-17 1999. Morgan Kaufmann Publishers, San Fransisco, CA.

[62] M. Pelikan, D.E. Goldberg, and F. Lobo. A survey of optimization by building and using probabilistic models. Technical Report No. 99018, IlliGAL, University of Illinois, 1999.

[63] L.S. Pitsoulis and M.G.C. Resende. Greedy Randomized Adaptive Search procedures. Technical report, AT&T Labs Research, 2001.

[64] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, 1973.

[65] W.M. Spears, K.A. De Jong, T. Bäck, D.B. Fogel, and H. de Garis. An overview of evolutionary computation. In Pavel B. Brazdil, editor, *Proceedings of the European Conference on Machine Learning (ECML-93)*, volume 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[66] T. Stützle. Iterated local search for the quadratic assignment problem. Technical report aida-99-03, FG Intellektik, TU Darmstadt, 1999.

[67] T. Stützle. *Local Search Algorithms for Combinatorial Problems - Analysis, Algorithms and New Applications*. DISKI - Dissertationen zur Künstliken Intelligenz. infix, 1999.

[68] T. Stützle and H. H. Hoos. $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System. *Future Generation Computer Systems*, 16(8):889–914, 2000.

[69] G. Syswerda. Simulated Crossover in Genetic Algorithms. In L.D. Whitley, editor, *Proc. of the second workshop on Foundations of Genetic Algorithms*, pages 239–255, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[70] E. Taillard. Robust Taboo Search for the Quadratic Assignment Problem. *Parallel Computing*, 17:443–455, 1991.

[71] E-G. Talbi. A taxonomy of hybrid meta-heuristics. Technical Report AS-183, LIFL, Université de Lille 1, 1998.

[72] C.H.M. van Kemenade. Explicit filtering of building blocks for genetic algorithms. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Proceedings of the 4th Conference on Parallel Problem Solving from Nature – PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 494–503, Berlin, 1996. Springer.

[73] S. Voss, S. Martello, I.H. Osman, and C. Roucairol, editors. *Meta-Heuristics - Advances and Trends in Local Search Paradigms for Optimization.* Kluwer Academic Publishers, 1999.

[74] C. Voudouris. *Guided Local Search for Combinatorial Optimization Problems.* PhD thesis, Department of Computer Science, University of Essex, 1997. pp. 166.

[75] C. Voudouris and E. Tsang. Guided Local Search. Technical Report CSM–247, Department of Computer Science, University of Essex, 1995.

[76] R.A. Watson, G.S. Hornby, and J.B. Pollack. Modeling building-block interdependency. In John R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 1998. Stanford University Bookstore.