

COMBINATORIAL OPTIMIZATION AND METAHEURISTICS

S. CONSOLI

SUPERVISOR: PROFESSOR K. DARBY-DOWMAN

ABSTRACT: Today, combinatorial optimization is one of the youngest and most active areas of discrete mathematics. It is a branch of optimization in applied mathematics and computer science, related to operational research, algorithm theory and computational complexity theory. It sits at the intersection of several fields, including artificial intelligence, mathematics and software engineering. Its increasing interest arises for the fact that a large number of scientific and industrial problems can be formulated as abstract combinatorial optimization problems, through graphs and/or (integer) linear programs. Some of these problems have polynomial-time (“efficient”) algorithms, while most of them are NP-hard, i.e. it is not proved that they can be solved in polynomial-time. Mainly, it means that it is not possible to guarantee that an exact solution to the problem can be found and one has to settle for an approximate solution with known performance guarantees. Indeed, the goal of approximate methods is to find “quickly” (reasonable run-times), with “high” probability, provable “good” solutions (low error from the real optimal solution). In the last 20 years, a new kind of algorithm commonly called metaheuristics have emerged in this class, which basically try to combine heuristics in high level frameworks aimed at efficiently and effectively exploring the search space. This report briefly outlines the components, concepts, advantages and disadvantages of different metaheuristic approaches from a conceptual point of view, in order to analyze their similarities and differences. The two very significant forces of intensification and diversification, that mainly determine the behavior of a metaheuristic, will be pointed out. The report concludes by exploring the importance of hybridization and integration methods.

1. INTRODUCTION

Combinatorial optimization (CO) is the general name given to the problem of finding the best solution out of a very large, but finite, number of possible solutions. It lies at the interface of Discrete Mathematics, Computer Science and Operational Research. Many real-life decision problems can be formulated as combinatorial optimization problems and consequently there is a large and growing interest in both theoretical and practical aspects of the subject. In practice, combinatorial optimization problems are often large-scale and difficult to solve. Thus, much attention has been given to studying computational complexity and algorithm design with a view to developing efficient

solution procedures. A combinatorial optimization problem $P = (S, f)$ may be specified as follows:

- A set of variables $X = \{x_1, \dots, x_n\}$,
- Variable domains D_1, \dots, D_n ,
- Constraints among variables,
- An objective function f to minimize (maximize), where $f : D_1 \times \dots \times D_n \rightarrow \mathfrak{R}^+$.

The set of all possible feasible solutions is

$$S = \{s = \{(x_1, v_1), \dots, (x_n, v_n)\} / v_i \in D_i \text{ and } s \text{ satisfies the constraints}\}.$$

S is usually called a *search (or solution) space*, as each element of the set can be seen as a candidate solution. To solve a combinatorial optimization problem means to find a solution $s^* \in S$ with minimum (maximum) objective function value; that is, $f(s^*) \leq f(s)$, $\forall s \in S$. s^* is called a “globally optimal solution” of (S, f) and let the set $S^* \subseteq S$ be the “set of globally optimal solutions”.

Examples of such problems are Network Flow Problems (e.g. Shortest Path Problem, Minimum Spanning Tree Problem, Maximum/Minimum Cost Flow Problem), Matching Problems (e.g. Cardinality Matching Problem, Job Assignment Problem, Maximum/Minimum Weight Matching Problem), Matroids (e.g. Maximization/Minimization Problem For Independent Systems, Matroid Intersection Problem, Matroid Partitioning Problem), Set Covering Problem, Colouring Problems (e.g. Vertex-Colouring Problem, Edge-Colouring Problem), 3-Occurrence Max-Sat Problem, Knapsack Problems, Bin-Packing Problem, Network Design Problems (e.g. Survivable Network Design Problem, Steiner Tree Problem), and Traveling Salesman Problem.

By considering the Knapsack Problem, it is easy to understand the difficulty of finding an optimized solution. Suppose a hitchhiker has to fill up his knapsack by selecting from among various possible objects those that will give him maximum comfort: these and many other example of Knapsack Problems can be mathematically formulated by numbering the objects from 1 to n , introducing a vector of binary variables x_j ($j=1, \dots, n$) having the following meaning:

$$x_j = \begin{cases} 1 & \text{if object } j \text{ selected} \\ 0 & \text{otherwise.} \end{cases}$$

Then, if p_j is a measure of the comfort given by object j , w_j its size and c the size of the knapsack, the problem will be to select, from among all binary vectors x satisfying the constraint

$$\sum_{j=1}^n w_j x_j \leq c,$$

the one which maximizes the objective function f

$$\max f = \max \sum_{j=1}^n p_j x_j.$$

There are many applications of the knapsack model. For example, suppose an investment of up to c dollars is to be made in one or more of n possible investments. Let p_j be the profit expected from investment j , and w_j the required investment. It is self-evident that the optimal solution of the knapsack problem above will indicate the best possible choice of investments.

A naive approach to solve the knapsack problem would be to examine all possible binary vectors x , selecting the best of those that satisfy the constraint. A full enumeration consists of 2^n vectors and thus, for a computer with a clock frequency of 3.6 GHz ($3.6 \cdot 10^9$ instructions per second, i.e. 1 instruction in $0.28 \cdot 10^{-9}$ s), the time, t , to compute 2^n instructions is given by:

$$t = 2^n \cdot (0.28 \cdot 10^{-9})_{sec} = \frac{1}{365 \cdot 24 \cdot 3600} \cdot 0.28 \cdot 2^n \cdot 10^{-9} \text{ years};$$

For $n = 60$ vectors, for example, $t \approx 10$ years, and for $n = 61$ more than 20 years, almost 4 centuries for $n = 65$, and so on.

However, the (N.F.L.T.) No-Free-Lunch-Theorem (Martello and Toth, 1990) states that, if an algorithm performs well on a particular class of problems, having been designed to exploit the specific characteristics of the problem in question, then it necessarily pays for that with degraded performance on the other problems. The theorem is used as an argument against using generic searching algorithms (e.g. genetic algorithms and simulated annealing) without exploiting as much domain knowledge as possible. Alternatively, the theorem establishes that “a general-purpose universal optimization strategy is not possible, and the only way one strategy can outperform another is if it is specially adapted to the problem under consideration” (Ho and Pepyne, 2002).

So combinatorial optimization algorithms are custom-strategies; that is, they are optimization techniques specialized to the particular kind of problem being solved. *Complete (or exact) algorithms* are guaranteed to find, for every instance of a specified CO problem of finite size, an optimal solution in bounded time (with proof of its optimality), while in *approximate methods*, the guarantee of finding an optimal solution is sacrificed for the sake of getting good solutions in a significantly reduced amount of time.

Combinatorial optimization problems can be classified as **P**-problems and **NP-hard** problems. In computational complexity theory, the class **P** consists of all those decision problems that can be solved on a “deterministic sequential Turing-machine” in an amount of time that is polynomial $p(x)$ in the size of the input x ; the class **NP** consists of all those decision problems whose positive solutions $|x|$ can be verified in polynomial-time $p(|x|)$ given the right information, or equivalently, whose solution can be found in polynomial-time p on a “non-deterministic Turing-machine”. The term **NP-hard** (Non-deterministic Polynomial-time hard) refers to the class of problems that contains all problems H , such that for every decision problem L in NP there exists a “polynomial-time many-one reduction” to H , written $L \propto H$. Formally, H is NP-hard if

- There exists an L in NP and
- $L \propto H$ (i.e. The NP-problem L is reducible to H) in a polynomial-time.

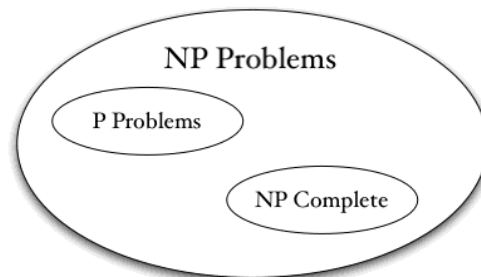


FIG 1: Diagram of complexity classes provided that $P \neq NP$. If $P = NP$, then all three classes are equal.

In computational complexity theory, a *many-one reduction* from L to H is a reduction that converts instances of the decision problem L into instances of the decision problem H . If there is an algorithm that solve instances of H , it is possible to use it to solve instances of L in:

- the time needed for N plus the time needed for the reduction;
- the maximum of the space needed for N plus the space needed for the reduction.

Many-one reductions are a special case and a weaker form of Turing reductions; the latter are sometimes more convenient for designing reduction algorithms, but their power also causes several important classes, such as NP , to be not closed under this kind of reduction. Formally, a many-one reduction from L to H (or L many-one reducible to H) is a totally computable function $f : D_f \rightarrow C_f$, which is a function that can be calculated using a mechanical calculation device (i.e. it is possible to construct an algorithm that halts after a finite amount of time and decides whether or not a given element belongs or not to the function domain), such as

$$\text{if } \alpha \in L \Leftrightarrow f(\alpha) \in H, \quad \forall \alpha \in D_f.$$

Therefore, the “polynomial-time many-one reduction” from L to H is a many-one reduction which is computable by a deterministic Turing-machine in polynomial-time; while the term “ C_1 reducible to C_2 ” (written $C_1 \propto C_2$) means that for every instance $a \propto C_1$ there is a deterministic algorithm which transforms it into an instance $b \propto C_2$, such that the answer of the “deterministic Turing-machine” to b is YES *if and only if* the answer to a is YES.

So, the notion of NP-hardness plays an important role in the discussion about the relationship between the complexity classes P and NP , because if it is possible to find an algorithm that solves one of these problems H in polynomial-time, it should be possible construct a polynomial-time algorithm for any problem L in NP by first performing the reduction from L to H and then running the polynomial algorithm. This would be equivalent stating “ $P = NP$ ”, and thus solving the biggest open question in theoretical computer science concerning the relationship between these two classes. Indeed, a common mistake is to think that the word “NP” in “NP-hard” stands for “non-polynomial” but although it is widely suspected that there are no polynomial-time algorithms for these problems, this has never been proved.

Summarizing, if a NP-problem L is many-one reducible in polynomial-time to H , the H is called NP-hard. This doesn't mean that H is in NP: therefore, if it is also the case that H is in NP, then H is called NP-complete (**NP-C**). FIG 1 shows the diagram of the complexity classes NP, P and NP-C, assuming that $P \neq NP$. In a more formal way, a decision problem H is NP-complete if

- it is in NP and
- it is NP-hard (i.e. every other problem in NP is reducible to it).

A consequence of this definition is that if we had a polynomial-time algorithm for H , we could solve all problems in NP in polynomial-time (“ $P = NP$ ”). The NP-complete complexity class contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. For more details see Garey and Johnson (1979) in which many NP-Complete problems are classified.

Numerous instances of problems arising in Operational Research and other fields are too large for an exact solution to be found in reasonable time. Thousands of problems are NP-hard and no algorithm with a number of steps polynomial in the size of the instances exists. Moreover, in some cases where a problem admits a polynomial algorithm, the power of this polynomial may be so large that realistic size instances cannot be solved in reasonable time in most of the cases (these are called long-term P problems). In this context (NP-C problems, long-term P problems, or in general problems difficult to solve), making use of exact algorithms is ineffective. So approximate algorithms should deal them in order to find “quickly” (reasonable run-times), with “high” probability, provable “good” solutions (low error from the real optimal solution). In the last 20 years, a new kind of approximate algorithms commonly called metaheuristics have emerged in this class, which basically try to combine heuristics in high level frameworks aimed at efficiently and effectively exploring the search space.

In the next section the basic concepts of metaheuristics will be outlined, allowing different kinds of classification between them.

In the sections 3 and 4, the most important single-point and population-based methods will be presented in order to analyze their similarities and differences, advantages and disadvantages and components from a conceptual point of view.

In the section 5, the two very significant forces of intensification and diversification, that mainly determine the behavior of metaheuristics, will be pointed out, concluding by exploring the importance of hybridization and integration methods in section 6.

2. CONCEPTS AND CLASSIFICATION OF META-HEURISTICS

Approximate algorithms can be divided in two main classes: *local-search algorithms* and *constructive algorithms*. In short, the first class starts from an initial solution and it tries to replace the current solution at each step with a “better” one in a neighborhood of the current solution. Here the *neighborhood* of a current solution s is defined as a function $N(s): S \rightarrow 2^S$, which assigns to every $s \in S$ a set of neighborhoods $N(s) \subseteq S$, where S is the search space. With the introduction of a neighborhood structure it is possible to define the concept of *locally minimal solution (or local minimum)* with respect to a neighborhood structure $N(s)$, as a solution \underline{s} such that $\forall s \in S \rightarrow f(\underline{s}) \leq f(s)$.

The second class of approximate methods, constructive algorithms, builds a solution at each step, simply adding components to the solution of the previous step, until the constraints are satisfied. These methods are usually faster than local-search methods, but they also tend to be of lower quality.

In recent years, there have been significant advances in the theory and application of meta-heuristics to the approximate solution of hard optimization problems. The term *metaheuristic* derives from the composition of two Greek words: “Heuristic” (from the verb *heuriskein*) that means “to find”; and the suffix “Meta” that means “beyond, in an upper level”. Before this term was largely adopted, metaheuristics were often called modern heuristics (Rayward-Smith V.J., Osman I.H., Reeves C.R. and Smith G.D, 1996). As illustrated in (FIG 2), this family includes, but it is not limited to, Tabu Search (TS), Simulated Annealing (SA), Explorative Local Search Methods including Greedy Randomized Adaptive Search Procedures (GRASP), Iterated Local Search (ILS), Variable Neighborhood Search (VNS) and Guided Local Search (GLS), Evolutionary Computation (EC) including Genetic Algorithms (GA) and Ant Colony Optimization (ACO).

As Voss S., Martello S., Osman I.H. and Roucairol C., (1999) state, “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”.

Summarizing, metaheuristics are strategies, approximate and usually non deterministic, that guide the search process to efficiently explore the search space in order to find (near-) optimal solutions, using techniques which range from simple local search procedures to complex learning processes. They are not problem-specific, can incorporate mechanisms to avoid “traps” (local optima), may use domain-specific knowledge to explore the best promising areas and finally they can memorize the search experience in order to guide the future search (long/short-time form of memory).

It is very important to clarify the concepts of diversification and intensification used in metaheuristics; the first term means the exploration of the search space while the latter one the exploitation of the accumulated search-experience. When the search process starts, it needs to compute the value of very different points in the search domain in order to find the promising areas (diversification). Then the algorithm needs to investigate promising zones to find the local-optimum (intensification). The best local optimum found in the different areas will be the candidate solution, hoping to be as near as possible to the optimum that the algorithm is looking for. The terms “diversification” and “intensification” are mainly used in methods based on the concept of memory, such as Tabu Search. Conversely the terms “exploration” and “exploitation” are used in strategies that don’t require explicit usage of memory, such as evolutionary computation. Finding a good balance between diversification (exploration) and intensification (exploitation) is essential for a metaheuristic in order to quickly identify regions in the search space with high quality solutions, without wasting too much time in regions with a low quality. Metaheuristics can be classified in different ways depending on the specific point of view of interest.

A first classification can be made by considering *nature-inspired* algorithms, such as Genetic Algorithms and Ant Algorithms, and *non-nature inspired* ones, such as Tabu Search and Iterated Local Search. This classification is not very meaningful because many

recent hybrid algorithms fit both classes at the same time and also because sometimes it is not possible to clearly attribute an algorithm to one of the two classes.

A second classification can be *population-based*, like Genetic Algorithms, and *single point search* methods, such as Tabu Search, Iterated Local Search and Simulated Annealing. The algorithms of this latter class are often called also *trajectory methods* because they work on a single solution at each time-step describing a curve (trajectory) in the search space during the progress of the search; they encompass local search-based metaheuristics. On the other hand, population-based metaheuristics compute simultaneously a set of points at each time-step of the search process, describing the evolution of an entire population in the search domain (FIG 2).

Metaheuristics can also be classified according to the way they make use of the objective function. If, during the search, the objective function is altered by trying to incorporate information collected during the search process (for example to escape from local minima), then the metaheuristic is said to have a *dynamic objective function*, as with the Guided Local Search (GLS). Techniques that keep the objective function as it is given by the problem belong to the class of metaheuristic with a *static objective function*.

Moreover, most metaheuristic algorithms work on one *single neighborhood structure*, i.e. the fitness landscape topology does not change in the course of the search process. Other metaheuristics, such as Variable Neighborhood Search (VNS), use a set of neighborhood structures (*various neighborhood structures*), diversifying the search by swapping between different fitness landscapes.

Finally, a very important feature to classify metaheuristics is the usage of a memory during the search history, because it is one of the fundamental elements of a powerful metaheuristic. In *memory-less algorithms* the next state depends only on the information accumulated in the current state of the search process, as a Markov process, while in *memory-usage algorithms* there is a usage of a short-term and/or a long-term memory. Usually, the first keeps track of recently visited solutions (moves), while the second is a huge storage of information about the entire search process.

The classification of metaheuristics in trajectory and population based methods permits a clear distinction between these kinds of algorithms. In the following sections, the most important single-point and population-based methods will be presented in order to analyze their similarities and differences, advantages and disadvantages and components from a conceptual point of view.

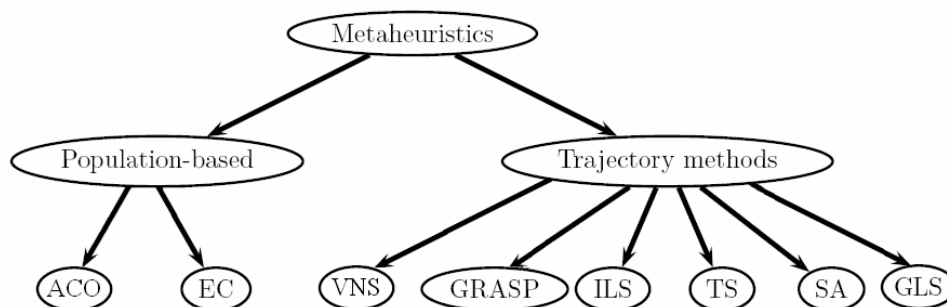


FIG 2: Main metaheuristics and their principal classification in trajectory and population-based methods.

3. TRAJECTORY METAHEURISTICS

Trajectory methods are so called because the search process designs a trajectory in the search space, starting from an initial state and dynamically adding a new better solution to the curve in each discrete time-step. So, this process can be seen as the evolution in time of a discrete dynamical system in the state space. The generated trajectory is useful because it provides information about the behavior of the algorithm and its dynamics in order to choose the most effective method to solve the problem instance under consideration. The system dynamics are the result of the combination of algorithms (i.e. chosen strategy), problem representation (i.e. definition of the search landscape) and problem instance. Trajectory shape depends on the strategy used: simple algorithms generate a trajectory composed of a *transient* phase followed by an *attractor* (a fixed point, a cycle or a complex attractor); advanced algorithms generate more complex trajectories comprising more different phases, representing the dynamic tuning between diversification and intensification during the search process. These continuous oscillations provide alternate phases in the designed trajectory, trying to find an optimal balance between these fundamental strengths. The main trajectory search methods are described below.

3.1 BASIC LOCAL SEARCH (OR ITERATIVE SEARCH)

Basic Local Search is the simplest trajectory search technique, and is often used in conjunction with other techniques. The concept is simple: every “move” from the current solution to the candidate solution is only performed if the objective function value given by the candidate solution is smaller than the value given by the current solution (in the case of a minimization problem). A *move* is the choice of a solution s' from the neighborhood $N(s)$ of the previous solution s , that is $s' \in N(s)$. The algorithm halts when a better solution can't be found (i.e. the current solution is a local minimum). Formally, in pseudo code, the procedure may be specified as follows (Blum C. and Roli A., 2003):

```
 $s$  = Generate-Initial-Solution();  
repeat  
   $s$  = Improve( $N(s)$ );  
until no improvements are possible
```

The procedure Improve($N(s)$) can be in the extremes either a *first improvement*, or a *best improvement* function, or any intermediate option. The former scans the neighborhood $N(s)$ and chooses the first solution that is better than s , the latter exhaustively explores the neighborhood and returns one of the solution with the lowest objective function value. Both methods stop at local minima. Therefore, their performance strongly depends on the definition of the search space S , of the objective function f and of neighborhood structure $N(\cdot)$.

The effectiveness of a Basic Local Search tends to be highly unsatisfactory for combinatorial optimization problems, because it often becomes trapped in a local minimum. Some extra mechanisms have been developed to enable the procedure to escape from a local minimum, but they add computational complexity to the entire procedure.

Therefore, rather than as a stand-alone algorithm, Basic Local Search is usually used as a component in hybrid metaheuristics in order to improve performance to try solving a specific CO problem.

Possible termination conditions of a Basic Local Search could be:

- Reaching the maximum cpu time;
- Reaching the maximum total numbers of iterations;
- Finding a solution s with an objective function $f(s)$ smaller than a threshold value;
- Achieving a maximum number of iterations without improvements.

3.2 SIMULATED ANNEALING (S.A.)

Simulated Annealing (SA) is possibly the oldest probabilistic metaheuristic for global optimization problems, and surely one of the first to explicitly provide a way to escape from local traps. It was independently invented by Kirkpatrick S., Gelatt C. D. and Vecchi M. P. (1983), and by V. Cerny (1985). The SA metaheuristic performs a stochastic search of the neighborhood space. In the case of a minimization problem, modifications to the current solution that increase the value of the objective function are allowed in SA, in contrast to classical descent methods where only modifications that decrease the objective value are possible.

The name and inspiration of this method come from the process of annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling provides an opportunity to find configurations with lower internal energy than the initial one.

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called *temperature*), that is gradually decreased during the process (cooling process).

The dependency is such that the current solution changes arbitrarily in the search domain when T is large, i.e. at the beginning of the algorithm, through *uphill moves* (or *random walks*) that saves the method from becoming stuck at a local minimum. Afterwards, the temperature T is gradually decreased, intensifying the search process in the specific promising-zone of the domain (*downhill moves*).

More precisely, the current solution is always replaced by a new one if this modification reduces the objective value, while a modification increasing the objective value by Δ is only accepted with a probability $e^{-\Delta/T}$. At a high temperature, the probability of accepting an increase to the objective value is high (uphill moves: high diversification and low intensification). Instead, this probability gets lower as the temperature is decreased (downhill moves: high intensification and low diversification).

The process described is memory-less because it follows a trajectory in the state space in which the successor state is chosen depending only on the incumbent one, without taking into account the past of the search process.

This strategy is specified in pseudo code as follows (Blum C. and Roli A., 2003):

```

s = s0; //Generate-Initial-Solution()
T = T0;
e = e0; //e0 > emax
k = 0;
while k < kmax and e > emax do
    s' = neighbor(s); //Pick-At-Random(N(s))
    if f(s') < f(s) then
        s = s'; //s' replaces s
    else
        
$$\frac{f(s')-f(s)}{T}$$

        if rand() < e then
            s = s'; //Accepting a worse s' as new solution with a
                given probability
        endif
    endif
    Update(T);
    k = k + 1;
Endwhile //termination conditions met
return s;
    
```

The initial temperature value, the number of iterations to be performed, the temperature value at each step, the cooling (reduction) rate of T , and the stopping criterion are determined by the so-called “SA cooling schedule”, generally specified by the following rule:

$$T_{k+1} = \text{funct}(T_k, K);$$

Theoretical results on non-homogeneous Markov chains (Aarts E. H. L., Korst J. H. M. and Laarhoven P. J. M. V., 1997) state that under particular conditions on the cooling schedule, the algorithm converges in probability to a global minimum for $k \rightarrow \infty$. More precisely, calling p_k the probability to find a global minimum after k steps:

$$\exists \Gamma \in \mathfrak{R} \ / \ \text{iif} \ \sum_{k=1}^{\infty} e^{\frac{\Gamma}{T_k}} \rightarrow \infty \Leftrightarrow \lim_{k \rightarrow \infty} p_k = 1; ;$$

So different cooling schedules, all satisfying this hypothesis of convergence, may be considered, such as a *logarithmic* cooling law:

$$T_{k+1} = \frac{\Gamma}{\lg(K + K_0)};$$

$$\text{Indeed } \sum_{k=0}^{\infty} e^{-\frac{\Gamma}{T_{k+1}}} = \sum_{k=0}^{\infty} e^{-\frac{\Gamma}{\lg(K+K_0)}} = \sum_{k=0}^{\infty} e^{-\lg(K+K_0)} = \sum_{k=0}^{\infty} \frac{1}{(K+K_0)} \rightarrow \infty;$$

The drawback of this logarithmic cooling law is that it is too slow for practical purposes; therefore faster cooling schedule techniques are adopted in real applications, such as a geometric cooling law, which is a cooling rule with an exponential decay of the temperature:

$$T_{k+1} = \alpha \cdot T_k; \quad \text{where } \alpha \in [0, 1];$$

Other complex cooling techniques can be used in order to improve the performance of the SA algorithm. For example, to have an optimal balance between diversification and intensification, the cooling rule may be updated during the search process. At the beginning T can be constant or linearly decreasing to have a high diversification factor for a larger exploration of the domain; after that, T can follow a fast rule, such as the geometric one, to converge quickly to a local optimum.

Other successful variants are *non-monotonic* cooling schedules that alternate phases of cooling and reheating, providing an oscillating balance between diversification and intensification.

Concluding, simulated annealing has been applied to several combinatorial problems, such as (Blum C. and Roli A., 2003):

- Quadratic Assignment Problem (QAP);
- Job Shop Scheduling (JSS) problem.

Rather than as a stand-alone algorithm, it is nowadays used as a component in hybrid metaheuristics to improve performance as in Threshold Accepting and Great Deluge Algorithms (Blum C. and Roli A., 2003).

3.3 TABU SEARCH (T.S.)

Tabu Search (TS) method, introduced by Glover (1986), is one of the most widely used metaheuristics. It shares with SA the ability to guide the search avoiding traps in poor local optima, but in a deterministic way rather than a stochastic one, modeling human memory processes.

Memory is implemented by the implicit recording of previously seen solutions using a simple data structure. This consists of a *tabu list* of moves which have been made in the recent past of the search, and which are forbidden (tabu) for a certain numbers of iterations. This helps to avoid cycling, and serves also to promote a diversified search of the solution, trying to escape from local minima.

At each iteration, TS moves to the best admissible neighbour restricted to the solutions that do not belong to the tabu list, referring to this set as the *allowed set*. The best solution from the allowed set is chosen as the new current solution, it is added to the tabu list and the oldest element of the tabu list removed (FIFO queue). Due to this dynamic

restriction of allowed solutions in a neighborhood, TS can be considered as a dynamic neighborhood search technique, with a short term memory implemented by the tabu list.

The entire algorithm stops when a termination condition is met or the allowed set is empty, as specified in the following procedure (Blum C. and Roli A., 2003):

```
s = Generate-Initial-Solution();  
TabuList =  $\emptyset$ ;  
while termination conditions not met do  
  s = Choose-Best-Of( $N(s) \setminus$  TabuList);  
  Update(TabuList);  
Endwhile
```

Generally, usage of memory in metaheuristics can be described in terms of four “dimensions” in the search: recency, frequency, quality and influence, in which the first two are the most important.

Recency records the most recent iteration in which a solution was involved. In TS the most recent moves are forbidden and the length of the tabu list, called “tabu tenure”, represents the recency principle. The tabu tenure is either fixed or dynamically updated during the search process. If its value is small, there is a high exploitation of the domain, but not many uphill moves to differentiate the search. Otherwise, if the tabu tenure is large, the exploration of new areas is encouraged because it forbids revisiting a large number of solutions.

Obviously, it is more convenient varying the tabu tenure dynamically. For example, the tabu tenure could be periodically reinitialized at random between a minimum value and a maximum value. Otherwise, it could be manually increased if there are many solution repetitions (i.e. a larger diversification factor is needed), while it could be decreased if no improvements are obtained and more intensification is required.

It is often beneficial to focus on some components or *attributes* of a move rather than on the complete move itself, avoiding managing a list on entire solutions that could make TS inefficient and not practical. Attributes are stored in different tabu lists defining the *tabu conditions*, which are used to filter the neighborhood of a solution and generate the allowed set. A neighbouring solution is considered forbidden and deemed not admissible if it has attributes on a tabu list.

Storing attributes rather than complete solutions is much more efficient, but also it may cause some non-tabu solutions, because forbidding an attribute means assigning the tabu status to probably more than one solution. To correct such errors *aspiration criteria* are defined, enabling the introduction of a solution in the allowed set even if it is forbidden by tabu conditions. The most commonly used aspiration criterion selects elements that are better than the current solution.

If recency simulates the short-term memory, a long-term memory can be implemented by the use of a variety of *frequency* measures, as “residence” measures and “transition” measures. The former is related to the number of times a particular attribute is observed, while the latter relates to the number of times an attribute changes from one value to another. In each case, the frequency measures are usually employed to generate penalties, which modify the objective function. Thereby, diversification is encouraged by the generation of solutions embodying combinations of attributes significantly different

from those previously encountered. Conversely, intensification is promoted by incorporating attributes of solutions from selected subsets of elements, called *elite subsets*, implicitly focussing the search in sub-regions defined relative to these subsets.

After discussing the concepts of recency and frequency, it may be also helpful to provide a brief reiteration of the basic notions of quality and influence.

Quality in TS usually refers to those solutions with good objective function values. A collection of such elite solutions may stimulate a more intensive search in the most promising regions of the search area.

Influence is roughly a measure of the degree of change induced in solution structure, commonly expressed in terms of the distance of a move from one solution to the next. It is an important aspect of the use of aspiration criteria, and is also relevant to the development of candidate list strategies. 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.

The Tabu Search heuristic is a rich source of ideas. Many of these ideas together with the corresponding strategies have been, and are currently, adopted by other metaheuristics. From a practical point of view, a recency-based approach with a simple neighborhood structure, searched using a restricted candidate list strategy, will often provide very good results.

TS has been applied to most CO problems; examples of successful applications are (Blum C. and Roli A., 2003):

- Robust Tabu Search to the QAP;
- Reactive Tabu Search to the MAXSAT problem;
- multidimensional Knapsack problem;
- cutting and packing problems;
- assignment problems;
- Job Shop Scheduling (JSS) problems (TS dominates completely this area);
- vehicle routing.

3.4 EXPLORATIVE LOCAL SEARCH METHODS

Explorative Local Search Methods are a family of trajectory algorithms recently developed. The most important ones, such as GRASP (Greedy Randomized Adaptive Search Procedure), VNS (Variable Neighborhood Search), GLS (Guided Local Search), and ILS (Iterated Local Search), will be briefly explained below.

3.4.1 GRASP: GREEDY RANDOMIZED ADAPTIVE SEARCH PROCEDURE

GRASP is a recently exploited method combining the power of greedy heuristics, randomization, and local search. It mainly consists of a construction phase and a local search improvements phase, as specified in the following pseudo code procedure (Blum C. and Roli A., 2003):

GRASP ALGORITHM:

```
while termination conditions not met do  
   $s$  = Construct-Greedy-Randomized-Solution(); // 1° phase  
  Apply-Local-Search( $s$ ); // 2° phase  
  Memorize-Best-Found-Solution();  
Endwhile
```

1° phase – Construct Greedy Randomized Solution():

```
while termination conditions not met do  
   $s$  = Construct-Greedy-Randomized-Solution();  
  Apply-Local-Search( $s$ ); // 2° phase  
  Memorize-Best-Found-Solution();  
Endwhile  
  
 $s = 0$ ; //  $s$  denotes a partial solution in this case  
 $\alpha$  = Determine-Candidate-List-Length(); //def. of RCL length  
while solution not complete do  
   $RCL_\alpha$  = Generate-Restricted-Candidate-List( $s$ );  
   $x$  = Select-Element-At-Random( $RCL_\alpha$ );  
   $s = s \cup \{x\}$ ;  
  Update-Greedy-Function( $s$ ); //update of the heuristic values  
Endwhile
```

2° phase – Apply Local Search(s):

```
Solution-Improvement(); //e.g. S.A., T.S.
```

The *solution construction* mechanism is characterized by a dynamic constructive heuristic and by randomization: each solution is randomly produced step-by-step by uniformly adding one new element from a candidate list (RCL_α) to the current solution. RCL_α is the restricted candidate list of length α that contains the best α elements of the search space. The elements are ranked by means of a heuristic criterion that gives them a score as a function of the benefit if inserted in the current partial solution.

These values can be either static values (fixed from the starting point to the end of the entire algorithm) or dynamic values (updated at each step depending on the current partial solution). The length α of the restricted candidate list is a very important parameter because it determines the strength of the heuristic bias, and also influences the sampling of the search space. In the extreme cases:

- $\alpha = 1$; only the best element would be added; construction mechanism is equivalent to a deterministic Greedy Heuristic;
- $\alpha = n$; completely random construction mechanism; random-choice of elements from RCL_α .

The simplest scheme to define α is updating it at each step, either randomly or by means an adaptive scheme.

After the solution construction phase, a *local search* is applied (such as S.A., T.S., iterative improvements) to try to improve the best current solution. The best element found

since the starting iteration is memorized, and the algorithm continues until the user termination conditions are reached.

Basic GRASP does not use history-memory of the search process and, for this reason, it is often outperformed by other metaheuristics. For its characteristics of simplicity and high speed, it is able to produce good solutions in a short amount of time. For example, it is a useful method for generating good starting points for other hybrid metaheuristics.

GRASP can be effective if the solution construction mechanism samples the most promising regions of the domain (by using an effective constructive heuristic and an appropriate value of α), and if the resulting solutions from the constructive heuristic belong to regions associated with different local minima (by using an effective constructive heuristic and an appropriate local search).

Successful applications of GRASP are (Blum C. and Roli A., 2003):

- Graph Planarization problems;
- grouping and clustering problems;
- production planning;
- vehicle routing;
- assignment problems;
- Job Shop Scheduling (JSS) problems.

3.4.2 VNS: VARIABLE NEIGHBORHOOD SEARCH

Variable Neighborhood Search is a new and widely applicable metaheuristic that makes use of a strategy based on dynamically changing neighborhood structures during the search process (Hansen and Mladenović, 1999, 2001, 2003, 2005). VNS provides a general framework and many variants exist for specific requirements. VNS doesn't follow a trajectory, but it searches for new solutions in increasingly distant neighborhoods of the current solution, jumping only if they are better than the current best solution.

At the starting point it is required to define arbitrarily the neighborhood structure. The simplest and more ordinary choice is a structure in which the neighborhoods have increasingly cardinality: $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$ (Nevertheless, with this sequence a large number of solutions could be revisited, at the cost of increased computational time. Today, attempts to improve the scanning of the landscape are made through more complex neighborhood structures).

The process of changing neighborhoods in the 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 VNS approach can be summarized in the sentence: "One Operator, One Landscape", meaning that promising zones on the search space given by a specific neighborhood may not be promising for other neighborhoods (landscape). Nevertheless, a local optimum with respect to a given neighborhood may not be locally optimal with respect to another neighborhood.

The VNS basic procedure is specified below (Blum C. and Roli A., 2003):

VNS ALGORITHM:

```

Select a set of neighborhood structures  $N_k(s)$ , with  $k = 1 \dots max$ ;
// 0. Initialization phase, e.g.  $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$ 
 $s = \text{Generate-Initial-Solution}()$ ;
while termination conditions not met do
     $k = 1$ ;
    while  $k < max$  do // Inner loop
         $s' = \text{Pick-At-Random}(N_k(s))$ ; // 1. Shaking phase
         $\hat{s} = \text{Local-Search}(s')$ ; // 2. Local search phase
        if  $(f(\hat{s}) < f(s))$  then
             $s = \hat{s}$ ; // 3. Move phase
             $k = 1$ ;
        else
             $k = k + 1$ ;
        endif
    Endwhile // end inner loop
Endwhile
    
```

The shaking phase consists of the random selection of a point s' in the neighborhood of the current solution $N_k(s)$. It provides a good starting point for the local search phase because s' may belong to a different basin of the current solution s (inner the $N_k(s)$), but maintaining some good characteristics of it. The random point s' is generated in order to avoid cycling, which might occur if any deterministic rule was used. The succeeding local search is not restricted to $N_k(s)$ but any neighborhood structure can be used. Afterwards, if no improvements are obtained ($f(\hat{s}) > f(s)$) in the move phase, the neighborhood structure is changed ($k = k + 1$) giving a progressive diversification (in the case of increasing cardinality neighborhoods $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$). As usual, the stopping conditions may be to reach either the maximum allowed CPU time, the maximum number of iterations, or the maximum number of iterations between two succeeding improvements.

Numerous variants in VNS have been found. Experimentally, VNS performance can be improved if s' is not just picked at random from $N_k(s)$, but it is achieved by performing an iterative search in the shaking phase between a random selection of points. Moreover, setting $k = k + k_{step}$ instead of $k = k + 1$, and $k = k_{min}$ instead of $k = 1$, gives an easy and natural way to drive the intensification and diversification of the search. It is also possible to remove the local search step (RVNS: Reduced VNS) for very large instances for which it is costly, making it similar to the classic Monte-Carlo method. Other more successful variants can be achieved and they will be described below.

A first important successful variant of VNS is the *Variable Neighborhood Descent (VND)* algorithm. It is based on the fundamental concept: the properties of a neighborhood are in general different from those of other neighborhoods and, therefore, a search strategy performs differently on them (Hansen and Mladenović, 2005). As Simulated Annealing, VND is a descent-ascent method because it may accept a worse candidate solution \hat{s} , following a certain probability $e^{-(f(\hat{s})-f(s))}$ inserting in the move phase (Conversely, VNS is a descent method because it only accepts a candidate solution if it is better than the current one).

Moreover, instead of a first improvement strategy (used in the VNS shaking phase), VND makes use of a best improvement search in $N_k(s)$ in order to improve its performance. The VND algorithm can be obtained by substituting the inner loop of the VNS procedure, as specified in the following (Blum C. and Roli A., 2003):

VND ALGORITHM:

```

Select a set of neighborhood structures  $N_k(s)$ , with  $k = 1 \dots max$ ;
// 0. Initialization phase, e.g.  $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$ 
 $s = \text{Generate-Initial-Solution}()$ ;
while termination conditions not met do
     $k = 1$ ;
    while  $k < max$  do // Inner loop
         $s' = \text{Chose-Best-Of}(N_k(s))$ ;
        if  $(f(s') < f(s))$  then
             $s = s'$ ; // Move phase
        else
            if  $\text{rand}() < e^{-(f(s')-f(s))}$  then
                 $s = s'$ ; // Flexible move phase: Accepting a worse  $s'$  as new
                // solution with probability  $e^{-(f(s')-f(s))}$ 
            else
                 $k = k + 1$ ; // no improvements: local minimum reached;
                // let's investigate in another neighbor
        endif
    endif
Endwhile // end inner loop
Endwhile
    
```

The best improvement local search is applied in the current neighborhood (Chose-Best-Of ($N_k(s)$)) and, in case a local minimum is found (i.e. no other improvements in that neighbor), the search proceeds investigating if the solution found is also a local optimum for the successive neighborhood ($k = k + 1$). Conversely, if a move is performed ($s = s'$), the search will proceed with the same neighborhood structure until a minimum is reached. As in the classical VNS, the search will stop if either the maximum CPU time, the maximum number of iterations, or the maximum number of iterations between two improvements is reached. The choice of the neighborhood structures is the critical point in VNS and VND, because the neighborhoods should exploit different properties and characteristics of the search space. So another variant of VNS, called *Variable Neighborhood Decomposition Search (VNDS)*, selects the neighborhoods producing a decomposition of the problem instance (Hansen and Mladenović, 2005). VNDS follows the usual VNS scheme, but the neighborhood structures and the local search are defined on sub-problems of each solution, All attributes (variables) of the current solution are kept fixed with the exception of k of them, which define a neighborhood structure N_k . Local search only regards changes on the variables belonging to the sub-problem it is applied to.

VNDS procedure can be obtained by substituting the inner loop of the VNS algorithm, as specified in the following (Blum C. and Roli A., 2003):

VNDS ALGORITHM:

```

Select a set of neighborhood structures  $N_k(s)$ , with  $k = 1 \dots max$ ;
// 0. Initialization phase, e.g.  $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$ 
 $s = \text{Generate-Initial-Solution}()$ ;
while termination conditions not met do
     $k = 1$ ;
    while  $k < max$  do // Inner loop
         $s' = \text{Pick-At-Random}(N_k(s))$ ; // 1. Shaking phase
         $\hat{s} = \text{Local-Search}(s', k \text{ variables})$ ; // 2. Local search phase
        if  $(f(\hat{s}) < f(s))$  then
             $s = \hat{s}$ ; // 3. Move phase
             $k = 1$ ;
        else
             $k = k + 1$ ;
        endif
    Endwhile // end inner loop
Endwhile
    
```

In the shaking phase the current solution s and the incumbent one s' differ only in k attributes (variables); in the local search phase the new solution \hat{s} is found by just allowing movements involving these k attributes. If a better solution \hat{s} is reached, the algorithm will start again setting $k = 1$ (i.e. the first neighborhood). Conversely, if no improving solutions are reached ($f(\hat{s}) > f(s)$) it means that the current solution s is a local minimum for k variables; so the algorithm will increase the number of the variables ($k = k + 1$) and will go on the search. The algorithm will stop if the usual stop conditions are met.

VNS, VND, and VNDS are steepest descent-oriented algorithms and often they are unsuitable to effectively explore the search space. So another variant has been developed called *Skewed VNS (SVNS)*, which extends VNS providing a more flexible acceptance criterion (Hansen and Mladenović, 2005). As an alternative to only accepting solution improvements, worse solutions s'' can be accepted if they differ from the current one less than the value of $\alpha \cdot \rho(s, s'')$, where $\rho(s, s'')$ is the distance (previously defined) between s and s'' , and α is a weight parameter in the acceptance criterion. The SVNS procedure is specified in the following pseudo code (Blum C. and Roli A., 2003):

SVNS ALGORITHM:

```

Select a set of neighborhood structures  $N_k(s)$ , with  $k = 1 \dots max$ ;
// 0. Initialization phase, e.g.  $|N_1(s)| < |N_2(s)| < \dots < |N_{max}(s)|$ 
 $s = \text{Generate-Initial-Solution}()$ ;
while termination conditions not met do
     $k = 1$ ;
    while  $k < max$  do // Inner loop
         $s' = \text{Pick-At-Random}(N_k(s))$ ; // 1. Shaking phase
         $s'' = \text{Local-Search}(s')$ ; // 2. Local search phase
        if  $(f(s'') - f(s) < \alpha \cdot \rho(s, s''))$  then // new accept. criterion
             $s = s''$ ; // 3. Move phase: improvements + worse
        endif
    Endwhile
Endwhile
    
```

```
                                // solutions in the range  $\alpha \cdot \rho(s, s')$   
     $k = 1;$   
    else  
         $k = k + 1;$   
    endif  
    Endwhile // end inner loop  
Endwhile
```

Variable Neighborhood Search and its variants have been successful applied to many CO problems (Hansen P. and Mladenović N., 2003), such as:

- Traveling Salesman Problem (TSP);
- vehicle routing;
- location and clustering problems;
- weighted MAXSAT problem;
- graph and network based problems;
- Job Shop Scheduling (JSS) problems.

3.4.3 GLS: GUIDED LOCAL SEARCH

The Guided Local Search (GLS) approach gradually moves (to guide the search) away from local minima by changing the search landscape. In contrast to Tabu Search and VNS strategies, the set of solutions and the neighborhood structure are kept fixed while the objective function f is dynamically changed, in order to make the current local optimum less desirable and trying to escape from it (FIG 3).

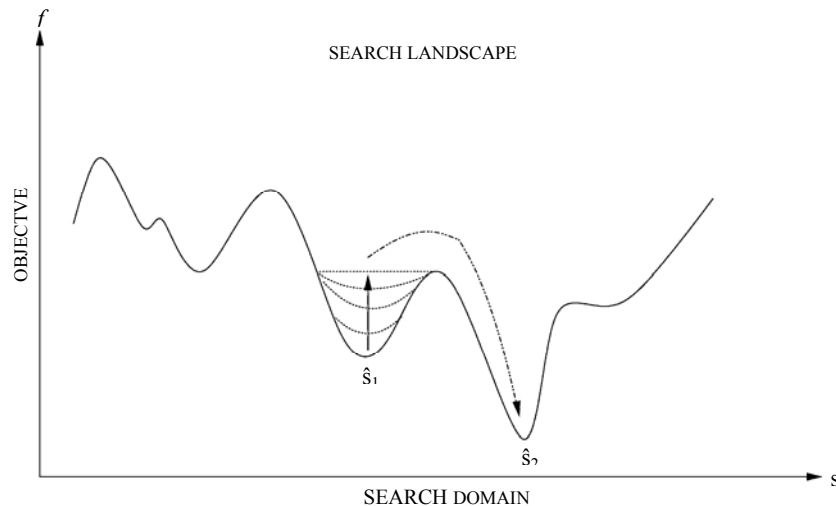


FIG 3: Guided Local Search strategy: escaping from traps increasing the relative objective function value, (Blum C. and Roli A., 2003).

This strategy is based on the definition of *solution features*, which may be any kind of properties or characteristics that can be used to discriminate between solutions (e.g. in TSP they could be arcs between pairs of cities, while in the MAXSAT problem the number of unsatisfied clauses could be considered).

An indicator function $I_i(s)$ is defined to show if the feature i is present in a specific solution s :

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

The new objective function f' is equal to the sum of the current objective function f and a term depending on the m features:

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

where λ is the *regulation parameter* balancing the importance of features I respect the original $f(s)$; p_i are the *penalty parameters* weighting the importance of the specific feature i . Before the actual search starts, the algorithm initializes all penalties parameters to zero, and assigns the variables uniformly at random. After each search phase, the penalties of all features with maximal utility are incremented by one, where the utility of a solution s under feature i is defined as:

$$Util(s, i) = \begin{cases} I_i(s) \cdot \frac{c_i}{1 + p_i} & \text{if feature } i \text{ is present in solution } s \\ 0 & \text{otherwise} \end{cases}$$

where c_i is the *cost* assigned to feature i , obtained from an heuristic evaluation of the relative importance of features with respect to others (the higher the costs, the higher are the utilities of the associated features). Nevertheless, the cost is scaled by the penalty parameter to prevent the algorithm from being totally biased toward the cost, making it sensitive to the search history. The GLS procedure is specified in the following pseudo code (Blum C. and Roli A., 2003):

```

s = Generate-Initial-Solution();
while termination conditions not met do
  for "all features  $I$  with Max Util( $s, i$ )" do
     $p_i = p_i + 1$ ; // or a variant is  $p_i = \alpha \cdot p_i$  with  $\alpha \in [0, 1]$ ;
  end for
  Update( $f'$ ,  $\mathbf{p}$ );
Endwhile
    
```

where $\mathbf{p} = [p_1, p_2, \dots, p_m]$ is the vector of penalties, updated in every search cycle, together with the incumbent objective function f' .

A variant is to apply the penalties update rule (i.e. the multiplicative rule) with a lower frequency than with an incrementing rule (for example every few hundreds of

iterations), in order to smooth the weights of penalized features and to prevent the landscape from becoming too rugged. The penalty update rules are often very sensitive to the problem instance. Another extension of GLS uses an additional mechanism for bounding the range of the penalties: if after the updating process, the maximum penalty exceeds a given max threshold, all penalties are uniformly decayed, improving the performance of the algorithm and its efficacy to solve large and hard structured instances of problems.

Successful applications of GLS are (Blum C. and Roli A., 2003):

- Traveling Salesman Problem (TSP);
- vehicle routing;
- weighted MAXSAT problem;
- Quadratic Assignment Problem (QAP).

3.4.4 ILS: ITERATED LOCAL SEARCH

Iterated Local Search (ILS) mainly consists of two steps, the first to reach local optima performing a walk in the search space, while the second to efficiently escape from local optima. The aim of this strategy is to prevent getting stuck in local optima of the objective function. Iterated Local Search is probably the most general scheme among the explorative strategies. It is often used as framework for other metaheuristics or can be easily incorporated as subcomponents in some of them to build effective hybrids. Formally, in pseudo code its procedure may be specified as following (Blum C. and Roli A., 2003):

```

 $s_0$  = Generate-Initial-Solution();
 $\hat{s}$  = Local-Search( $s_0$ );
while termination conditions not met do
     $s'$  = Perturbation ( $\hat{s}$ , history);
     $\hat{s}'$  = Local-Search ( $s'$ );
    if ( $f(\hat{s}') < f(\hat{s})$ ) then // Move phase
         $\hat{s} = \hat{s}'$ ; // improvements
    else
         $\hat{s} = \text{Apply-Acceptance-Criterion}(\hat{s}, \hat{s}', \textit{history})$ ; // acceptance
        // criterion for worse solutions
    endif
Endwhile // end inner loop
    
```

The algorithm initializes the search by selecting an initial candidate solution s_0 . The construction of s_0 should be both computationally not expensive and a good starting point for local search. The fastest way is to generate randomly the initial solution. However constructive heuristics may also be adopted in order to quickly find high-quality starting points. Afterwards, a locally optimal solution \hat{s} is achieved by applying a local search procedure, whose characteristics have a considerable influence on the performance of the entire algorithm.

The core of the overall algorithm mainly consists of the following three phases:

1. A “perturbation” applied to the current candidate solution \hat{s} ;
2. Another “local search” performed to the modified solution s' in order to find a local optimum s'' ;
3. The application of an “acceptance criterion” to decide which of the two local optima, s'' or \hat{s} , has to be chosen to continue the search process.

The specific steps have to be properly designed and set to find a good tradeoff between intensification and diversification, and so achieving high performance in the difficult CO area. Both the perturbation and the acceptance criterion mechanisms can use aspects of the search history (long- or short-term memory). For example, stronger perturbation should be applied when the same local optima are repeatedly encountered.

The role of the perturbation (usually probabilistic to avoid cycling) is to modify the current candidate solution to help the search process to effectively escape from local minima, in order to eventually find different better points. Typically, the strength of perturbation has a strong influence on the length of the subsequent local search phase. It can be either fixed (independently of the problem size) or variable. However, the latter one is in general more effective because the bigger the problem size is, the larger should be the strength. A more sophisticated adaptive strength scheme is also possible in which the perturbation strength is increased when more diversification is needed, and decreased when intensification seems preferable (VNS and its variants belong to this category).

The acceptance criterion has also a strong influence on the behavior and performances of ILS. The two extremes are:

- Accepting the new local optimum only in case of improvement (strong intensification: iterative improvement mechanism);
- Always accepting the new solution (high diversification: random walk in the search space).

Between these extremes, there are several intermediate choices. It is possible, for example, to adopt a kind of annealing schedule: accepting all the improving candidate solutions and also the non-improving ones with a probability that is a function of the temperature T and the difference of objective function values (Blum C. and Roli A., 2003):

$$prob(\hat{s}, \hat{s}', history, T) = \begin{cases} e^{-\frac{f(\hat{s}') - f(\hat{s})}{T}} & \text{if } f(\hat{s}') > f(\hat{s}) \\ 1 & \text{otherwise} \end{cases}$$

As in simulated annealing, the cooling schedule for the temperature T can be either monotonic (non-increasing in 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: instead of constantly decreasing the temperature, it is increased when more diversification seems to be required.

Successful applications of GLS are (Blum C. and Roli A., 2003):

- Traveling Salesman Problem (TSP);
- Single Machine Total Weighted Tardiness (SMTWT) problem;
- Quadratic Assignment Problem (QAP).

4. POPULATION-BASED METAHEURISTICS

Population-based methods deal at each step with a set of solutions (or a population) rather than with a single one, providing a natural and intrinsic way to explore the search space. Their performance strongly depends on the way the populations are manipulated. The main population-based methods in combinatorial optimization are Evolutionary Computation (EC) and Ant Colony Optimization (ACO). In EC methods, specific recombination and mutation operators modify sets of individuals, while in ACO a colony of artificial ants is used to construct solutions guided by the pheromone trails and by heuristic information (as it will be specified in the following sections).

4.1 EVOLUTIONARY COMPUTATION

The field of natural evolution applied to optimization algorithms is at a stage of tremendous growth. The main idea consists of the survival of the best element in natural evolution processes. There are currently three well-defined paradigms, which have served as the basis for much of the research in this field:

- Genetic Algorithms (GA);
- Evolution Strategies (ES);
- Evolutionary Programming (EP).

Each of these emphasizes a different aspect of natural evolution. In general, they have foundation on the following evolutionary operators:

- a) *Recombination* or *crossover*, which recombines two or more individuals (ancestors) to produce new individuals (children);
- b) *Mutation* or *modification*, which causes a self-adaptation of individuals;
- c) *Selection* of individuals based on their *fitness* (value of an objective function or some measure of the quality of solutions), which is the driving force in evolutionary algorithms. Individuals with a higher fitness have a higher probability to be chosen as members of the next population (or as parents for the generation of new individuals). This is an analogy with the principle of survival of the fittest in natural evolution, i.e. the capability of nature to adapt itself to a changing environment.

Formally, in pseudo code, the general EC procedure is specified as follow (Blum C. and Roli A., 2003):

```
P = Generate-Initial-Population ();  
Evaluate (P);  
while termination conditions not met do  
    P' = Recombine (P);  
    P'' = Mutate (P');  
    Evaluate (P'');  
    P = Select ( $P \cup P''$ );  
Endwhile // end inner loop
```

EC methods often outperform classical optimization algorithms when applied to difficult real-world problems.

It is generally accepted that any EC algorithm must have five basic components (Michalewicz, 1996):

- 1) A genetic representation of problem solutions;
- 2) A way to create the initial population;
- 3) An evaluation function rating the solutions in terms of their fitness;
- 4) Genetic operators (already mentioned);
- 5) Values for the specific parameters (population size, probabilities of applying genetic operators, etc).

The data structure used to represent the solutions and the set of genetic operators, constitute the skeleton of each EC algorithm. Historically, there are associations between GA and binary string representations, between ES and vectors of real numbers (in order to perform numerical optimizations), and between EP and finite state machines (in order to build predictive systems). The EP and ES communities have emphasized a reproduction mechanism based on mutation. By contrast, the GA community emphasized reproduction based on recombination and mutation.

Genetic Algorithms have their origins from the studies of cellular automata conducted by John Holland (1975, 1992) and his colleagues, but only recently their potential for solving combinatorial optimization problems has been exploited. The term "Genetic Algorithms" is due to their genetic make-up representation and manipulation of individuals, rather than using a phenotypic representation. The basic idea is to maintain a population of candidate solutions, which evolves under a selective pressure helping the survival of the fittest individual. Hence, they are a class of local search methods employing solution generation, which operates on attributes of a set of solutions rather than attributes of a single solution. Genetic Algorithms work on finite populations each of which as N chromosomes (solutions). The chromosomes are fixed strings with binary values (alleles) at each position (locus). An allele is the 0 or 1 value in the bit string, while the "locus" is the position at which the 0 or 1 value is placed in the chromosome. Chromosomes are evaluated according to a specified fitness function, and are selectively interbred in pairs to produce offspring, through the genetic operators. The resulting offspring inherit properties directly from their parents. The fitter a chromosome is, the more likely it is to produce offspring. The offspring are evaluated and placed in the new population replacing the weaker members. The GA mechanism consists of three phases: evaluation of the fitness of each chromosome, selection of the parent chromosomes, and applications of mutation and recombination (crossover) operators to the parent chromosomes. The process is repeated until the system ceases to improve. The survival of the fittest ensures that the overall solution quality increases as the algorithm proceeds from one generation to the next one.

Differently from GA, Evolution Strategies were developed mainly to build systems capable of solving real-valued parameter optimization problems. Its natural representation of the individuals consists of a vector of real numbers in order to help mutation operators and gene manipulation. Generally, ES emphasize behavioral changes by mutation at the level of the individual.

The third EC approach is Evolutionary Programming, which stresses behavioral change at the level of the species. The phenotypes of individuals are represented as finite

state machines capable of reacting to environmental stimulation, and to develop operators (primarily mutation) for reflecting structural and behavioural change over time.

The Evolutionary Computation characteristics are summarized as follows (Blum C. and Roli A., 2003):

- **DESCRIPTION OF INDIVIDUALS:** EC deals with a population of individuals commonly represented as bit strings. In GA, the single individuals are called genotypes, while the solutions formed by combinations of individuals are called phenotypes;
- **EVOLUTION PROCESS:** In each evolution process, the selection operator is fundamental in choosing the individuals to enter the next population of each step. If they are chosen exclusively from the offspring, it is a case of a so-called “generational replacement” evolution process. Instead, if it is also allowed to transfer individuals of the current population into the next one, it is a case of a so-called “steady state” evolution process. Moreover, EC may work with a population of fixed or variable size;
- **NEIGHBORHOOD STRUCTURE:** EC can also deal with an unstructured population, in which any individual may be recombined with any other one to create an offspring (e.g. Basic GA). If any individual can be recombined with only those included in a particular set (e.g. Parallel GA), it is a case of a structured population;
- **INFORMATION SOURCES:** If the information sources for the crossover operations are just a couple of individuals, it is a case of a two-parents crossover scheme. Otherwise, if the offspring are produced by some recombination of more than two parents, it is the case of a multi-parent crossover. Recently clever crossover schemes were developed, such as Gene Pool Recombination (using population statistics to generate the individuals of the next population), or the Bit-Simulated Crossover (using a probability distribution over the search space given by the current population to generate the next one);
- **INFEASIBILITY:** there are three different ways to handle infeasible solutions generated by the genetic operators. Infeasible individuals could be simply “rejected”, “penalized” (by assigning them an additional bad fitness value, so that they will have difficulty in being reselected in the succeeding steps to create offspring), or just “repaired” (but it is not always possible);
- **INTENSIFICATION STRATEGY:** Some EC methods, including mechanisms to improve the exploitation of the fitness function, were designed in recent years. They have been shown to be useful in many practical applications. Memetic Algorithms, for example, apply a local search to every individual of a population to quickly identify promising areas in the search space (Moscato P., 1999). While the use of a population ensures the diversification of the search, the use of local search techniques improves the intensification factor on the promising zones. Another strategy performed by the so-called Linkage Learning and Building Block Learning algorithms, guides the search to promising areas, combining each parts of individuals with good properties (see (Goldberg D.E. at all, 1991), (Van Kemenade C. H. M., 1996), (Watson R.A. at all, 1998), (Harik G., 1999) as examples). Moreover, generalized recombination operators incorporating the notion of “neighborhood search” into EC, have been recently proposed in other novel methods (Rayward-Smith V. J., 1994);
- **DIVERSIFICATION STRATEGY:** A problem to avoid in EC algorithms is the premature convergence toward sub-optimal solutions. To try to avoid premature convergence, the simplest mechanism is the use of the mutation operator, which just

performs a small random perturbation on individuals (noise). Other strategies are the “crowding” mechanism, the novel “fitness sharing” and “niching”. They reduce the allocation of reproductive fitness to an individual in the population, proportionally to the number of other individuals sharing the same region in the search space.

Evolutionary Computation algorithms have been applied to most CO problems, such as (Blum C. and Roli A., 2003):

- multi-objective optimization;
- growing Bioinformatics area;
- evolvable hardware.

In the following two sections, three further populations-based methods that are sometimes also regarded as being EC algorithms, will be introduced.

4.1.1 SCATTER SEARCH AND PATH RELINKING

Scatter Search (SS) and its generalization called Path Relinking (PR) are novel evolutionary methods compatible with randomized implementations, but not based on randomization as in the case with the other evolutionary approaches (Glover F., Laguna M. and Martí R., 2000). They join solutions by generalized path constructions (in both Euclidean and neighborhood spaces) and utilizing strategic designs, instead of exclusively using randomization. Scatter Search and Path Relinking embody strategies still not emulated by other evolutionary methods. The approach has been shown to be advantageous for solving a variety of complex optimization problems (see (Glover F., Laguna M. and Martí R., 2000) for more details).

The SS process captures information not separately contained in the original vectors. It takes advantage of auxiliary heuristic methods both for selecting the elements to be combined and for generating new vectors. It linearly combines solutions from a set, called the *reference set*, in order to create new ones.

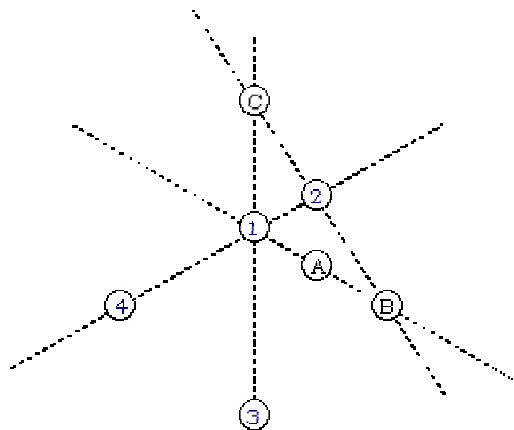


FIG 4: Example of reference set (Glover F., Laguna M. and Martí R., 2000).

In the example specified in (FIG 4), the original reference set consists of the solutions labeled A, B and C, (Glover F., Laguna M. and Martí R., 2000). After a non-convex combination of the reference solutions A and B, a number of new solutions in the line segment defined by A and B are created; in the example only solution 1 is introduced into the reference set. In a similar way, other convex and non-convex combinations between original and newly created reference solutions, produce points 2, 3 and 4. Finally, the resulting reference set is composed of 7 solutions (or elements).

Scatter Search does not leave solutions in a raw form after the combination mechanism, but applies heuristic improvements to the candidates for entry into the reference set. Unlike a “population” in Genetic Algorithms, the reference set of solutions in Scatter Search is relatively small. A typical GA population size consists of 100 elements, which are randomly sampled to create combinations. In contrast, Scatter Search systematically chooses two or more elements of the reference set to create new solutions. If the reference set consists of b solutions, experimentally the procedure will examine around $(3b-7) \cdot b/2$ elements, and so there is a practical need for keeping the cardinality of the reference set small. Typically, the reference set in Scatter Search has 20 solutions or less. Moreover, Genetic Algorithms need large populations to maintain a good level of diversification (for the random sampling embedded in its search mechanisms), while Scatter Search systematically injects diversity to the reference set. To limit the scope of the search to a selective group, a mechanism for controlling the number of possible combinations in a given reference set can be used. The reference set is divided into “tiers” and combined solutions must include at least one of the elements from each of them.

The Scatter Search approach may be outlined as follows (Glover F., Laguna M. and Martí R., 2000):

- 1) Generate a starting set of solution vectors to guarantee a critical level of diversity. Apply custom heuristic processes to try to improve these solution vectors. The reference solutions will be a subset of the best vectors. A solution may be added to the reference set if the diversification factor of the set improves, even if its objective value is inferior to other solutions competing for admission into the set.

- 2) Create new solutions consisting of structured combinations of subsets of the current reference solutions. These combinations are chosen to produce points both inside and outside the convex regions spanned by the reference solutions, and they are modified to become acceptable solutions.

- 3) Apply the heuristic processes (already used to generate the reference set) to improve the solutions created. These heuristic processes must be able also to operate on infeasible solutions to restore the feasibility if possible.

- 4) Extract a collection of the “best” improved solutions from last step and add them to the reference set. The notion of “best” is once again broad, as in the step 1. Steps 2, 3 and 4 are repeated until the reference set does not change. Moreover, the reference set is periodically diversified restarting from step 1. When reaching a specified iteration limit the algorithm will stop.

The goal of structured combinations in Scatter Search is to create weighted centres of the selected sub-regions. This adds non-convex combinations external to the original reference solutions (e.g., solution 3 in FIG 4). Another important feature relates to the construction of new solutions “within” and “across” clusters of points. Finally, Scatter

Search employs subordinate mechanisms to improve infeasible solutions, in order to make it possible for them to be included into the reference set.

The main general behavior of Scatter Search, also applicable to Path Relinking, is specified in the following routines (Blum C. and Roli A., 2003):

- *Seed-Generation*: One or more seed trial solutions are created to initialize the algorithm;
- *Diversification-Generator*: a procedure to generate a collection of diverse trial solutions from an arbitrary seed as input;
- *Improvement*: a local search method transforms a trial solution into one or more enhanced ones. (Neither the input nor the output solutions are required to be feasible);
- *Reference-Set-Update*: this method builds the reference set, consisting of the “best” found solutions (typically small values, e.g. no more than 20). Solutions gain membership of the reference set according to their quality or their diversity values;
- *Subset-Generation*: operates on the reference set and produces a subset of its solutions as a basis for creating combined solutions;
- *Solution-Combination*: transforms a given subset of solutions into one or more combined solution vectors.

From a spatial orientation, in Scatter Search new solutions are created by linear combinations of reference solutions using both positive and negative weights. The resulting points can be both inside and outside the convex region spanned by the reference set. By natural extension, such combinations may be paths, generated between and beyond selected solutions in neighborhood space rather than in Euclidean space. This SS extension is called Path Relinking. A path between solutions in a neighborhood space will produce new solutions sharing a subset of attributes contained in the parent solutions. The attributes vary according to the path selected and the location on the path. Such paths are specified by the solution attributes that are added, dropped or modified by the moves executed in neighborhood space. To generate the desired paths starting from an initiating solution, the moves must progressively introduce (or subtract) attributes by a guiding solution. This step consists of the incorporation of attributes from elite parents in partially or fully constructed solutions by means of heuristic methods. It is carried out by isolating assignments occurring frequently or influentially in high quality solutions, and then introducing them into other solutions (implicit form of frequency-based memory). Moreover, the possibilities of multiparent path generation emerge in Path Relinking. Typically, the generation of such paths “relinks” previous points in the neighborhood space in ways not achieved from the search history (hence giving the approach its name). Path Relinking is often used as a hybrid component in metaheuristics, such as Tabu Search and GRASP.

The evolutionary Scatter Search and Path Relinking have proved unusually effective for solving diverse optimization problems, from both classical and real world settings. It has been applied to a lot of problems such as (Blum C. and Roli A., 2003):

- multiobjective assignment and quadratic assignment;
- vehicle routing;
- financial product design;
- Job Shop Scheduling (JSS);
- mixed integer programming.

4.1.2 Estimation of Distribution Algorithms (EDA)

Genetic Algorithms are optimization techniques based on selection and recombination of promising solutions. GA behavior depends on the choice of the genetic operators: selection, crossover, mutation, probabilities of crossover and mutation, population size, rate of generational reproduction, number of generations etc. Interactions among the variables are only rarely considered. Moreover, the fixed two parents recombination and evolution sometimes provide low quality solutions. Two parents crossover can be replaced by generating new solutions according to a probability distribution (Toyon Kumar P. and Hitoshi I., 2002). This new approach is the Estimation of Distribution Algorithm (EDA).

In EDA, interactions among variables are explicitly expressed through the joint probability distribution calculated from a database of selected individuals. Sampling the probability distribution of the previous generation creates the offspring for the next one. Selections methods of EDA are the same of those used in Genetic Algorithms, but neither crossover nor mutation is applied. Formally, in pseudo code, Estimation of Distribution Algorithm may be specified as follows (Blum C. and Roli A., 2003):

```

P = Initialize-Population ();
while termination criteria not met do
    Psel = Select (P);           // Psel ⊂ P
    p(x) = p(x | Psel) = Estimate-Probability-Distribution();
    P = Sample-Probability-Distribution ();
Endwhile
    
```

The algorithm starts generating an initial population P of M size; N ($\leq M$) individuals are selected to form P_{sel} . The next step is to calculate the joint probability distribution of the selected individuals P_{sel} using one of the EDA methods. Just sampling it will generate offspring, and the old population will be replaced according to the specific replacement strategy. The algorithm goes on until the termination conditions are satisfied.

Different EDA approaches can be exploited according to the dependencies among the problem specific variables. Univariate Marginal Distribution Algorithm (UMDA), Population Based Incremental Learning (PBIL) and Compact Genetic Algorithm (CGA) do not consider interaction among variables (univariate); it is the easiest way to calculate the probability distribution. Indeed, the joint probability distribution may be simply computed as the product of the marginal probabilities of the single variables.

In UMDAs the joint probability distribution is factorized as a product of independent univariate marginal distributions, estimated from marginal frequencies.

Instead, in PBILs a vector of probabilities represents the population of individuals:

$$p_l(x) = (p_l(x_1), \dots, p_l(x_i), \dots, p_l(x_n))$$

where $p_l(x_i)$ refers to the probability of obtaining a 1 in the i_{th} component of the population in the l_{th} generation (Toyon Kumar P. and Hitoshi I., 2002). At each step, M individuals are generated by sampling $p_l(x)$. The best N individuals are selected to update the probability vector by a rule, which shifts the vector towards the best of generated individuals.

In CGA the probabilities are initialized to a 0,5 value for each variable. Then using this vector of probabilities, the method randomly generates new individuals. An evaluation of their objective function values provides a ranking. Then, the probability vector $p_i(x)$ is shifted toward the generated solution vector(s) with highest quality. The distance that the probability vector is shifted depends on the learning rate parameter. Then, a mutation operator is applied to the probability vector. After that, the cycle is repeated until the vector of probabilities converges to an optimum. The probability vector can be regarded 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. (Topon Kumar P. and Hitoshi I., 2002).

EDA provides better results for variables with no significant interactions among each other. To solve problems of pairwise interactions (bivariate dependencies), other procedure should be applied, such as Mutual Information Maximizing Input Clustering (MIMIC) Algorithm, Combining Optimizers with Mutual Information Tress (COMIT) or Bivariate Marginal Distribution Algorithm (BMDA).

Instead, for real world problems where multiple interactions occur, the followed algorithms are used: Factorized Distribution Algorithm (FDA), Extended Compact Genetic Algorithm (ECGA), Bayesian Optimization Algorithm (BOA), Estimation of Bayesian Network Algorithm (EBNA).

BOA, for example, estimates the joint probability distributions of selected individuals using modeling data from Bayesian Networks (Topon Kumar P. and Hitoshi I., 2002). The Bayesian metric, used to measure the goodness of each structure, has the property that structures reflecting the same conditional dependency or independency have the same scores. In order to reduce the cardinality of the search space, BOA imposes restrictions on the number of parents a node may have (for problems where a node may have more than 2 parents, the situation is complicated to solve).

The field of EDA is still quite young and much of the research effort is focused on methodology rather than high-performance applications. EDA has been applied to various CO problems such as (Blum C. and Roli A., 2003):

- Knapsack problems;
- Job Shop Scheduling (JSS) problem.

4.1.3 Quantum-inspired Genetic Algorithms (QGA)

Quantum-inspired Genetic Algorithms (QGA) are a family of novel evolutionary computing methods based on concepts and principles of quantum mechanics (e.g. standing waves, interference, coherence). They are applied to successful evolutionary computing methods, particularly to genetic algorithms, in order to increase their performance. A quantum-inspired computational method generates candidate solutions to the problem instance, and a classical algorithm checks if these solutions are in fact feasible. In order to continue a clear comprising of QGA, it is necessary to underline some basic principles of quantum mechanics.

An atom consists of a nucleus (containing particles called protons (positive charges) and neutrons (neutral charges)) and electrons (negative charges), surrounding the nucleus

through wave orbits (not-planar). There are different types of orbit, depending on two factors: angular momentum and energy level. An electron around a nucleus jumps states in discrete quanta by absorbing photons (from a low energy orbit to an higher energy one) or releasing it (high level to a lower one): the term “quantum” means that in-between states or orbits don’t exist, while a “photon” is the smallest unit of energy.

A quantum particle’s *location* can be described by a *quantum state vector* $|\Psi\rangle$, representing a *linear superposition* (i.e. a weighted sum) of the particle given individual quantum state vectors $|A\rangle, |B\rangle, |C\rangle, \dots$, respectively of the possible positions A, B, C, ... (Narayan A., 1999):

$$|\Psi\rangle = \alpha \cdot |A\rangle + \beta \cdot |B\rangle + \gamma \cdot |C\rangle + \dots;$$

where the weighting factors $\alpha, \beta, \gamma, \dots$ are complex numbers, which represent the probability that the particle is in a specific location ($\text{prob}_A = |\alpha|^2, \text{prob}_B = |\beta|^2, \text{prob}_C = |\gamma|^2, \dots$ respectively).

From Heisenberg’s uncertainly principle, both the position and momentum of a particle cannot be simultaneously known at any particular moment. Thus, if there are n locations given by n state vectors, the particle is said to be at all n locations at the same time. However, in the act of observing a quantum state (or wave function), it collapses to a single one (many universes interpretation of Everett: all quantum systems exist in parallel universes; it is not possible to view a quantum system in all these universes but only in a single one). For example, in the case of two universes, the probability P_{12} of arrival of the particle in a specific point is the square of the height of its *quantum amplitude* a_{12} (Narayan A. and Moore M., 1998):

$$P_{12} = a_{12}^2;$$

From the analogy with the water waves theory, the total amplitude a_{12} is the sum of the wave amplitude of each single universe (Narayan A. and Moore M., 1998):

$$a_{12} = a_1 + a_2;$$

So the probability P_{12} is given by:

$$P_{12} = a_{12}^2 = (a_1 + a_2)^2 = a_1^2 + a_2^2 + 2 \cdot a_1 \cdot a_2 = P_1 + P_2 + 2 \cdot a_1 \cdot a_2;$$

that is, the probability P_{12} of arrival of the particle in a specific point is the sum of the probability to have the particle in each single universe, adding of an *interference factor*, $2 \cdot a_1 \cdot a_2$, due by the scrambling between the universes.

Recently it was proved that a quantum system could be used to perform computations and could simulate quantum processes, impossible to compute efficiently on a conventional calculator (Narayan A., 1999). The “many universes” interpretation was

used by Shor (1994) in his quantum computing method for extracting prime factors of very large integers. A memory register was placed in a superposition of all possible integers it could contain, followed by a different calculation being performed in each ‘universe’. This result was used to deal with cryptography algorithms, in which key production methods are based on the seeming intractability of finding the prime factors of very large integers. So, although there are no known fast classical algorithms for factorizing large numbers into primes, Shor’s method uses known fast algorithms for taking a candidate prime factor and determining whether it is in fact a prime factor.

Quantum principles were applied to genetics algorithms giving an initial basic methodology to design quantum algorithms. The following guidelines explain how to develop a Quantum-inspired Genetic Algorithm (Narayan A., 1999):

1. Express the problem in a numerical form through specific conversion methods;
2. Determine the initial configuration;
3. Define the terminating conditions;
4. Divide the problem instance into smaller sub-problems;
5. Identify the number of required universes;
6. Assign an universe to each sub-problem;
7. Compute in parallel in the different universes;
8. There must be a form of interaction (interference) between all the universes, which yields a solution or new useful information for the universes.

An important difference between the classical GA and QGA is the representations of the elementary information unit. If GA are based on bits, QGA are based on *QuBits*, derived by the superposition principle of quantum mechanics. The QuBit does not represent only the value 0 or 1, but a superposition of the two bits. Its state is represented as follows:

$$|\Psi\rangle = \alpha \cdot |0\rangle + \beta \cdot |1\rangle ;$$

where $|0\rangle$ and $|1\rangle$ are the classical bit values 0 and 1, and α and β are complex numbers, respectively standing for the probability to measure the value 0 ($\text{prob}_0 = |\alpha|^2$) and that to measure 1 ($\text{prob}_1 = |\beta|^2$). That is:

$$|\alpha|^2 + |\beta|^2 = 1 ;$$

In the case of multiple QuBits, as in a quantum system, the resulting state space grows exponentially with respect to the number of particles. For example, in the case of n QuBits, the state space has 2^n dimensions, and its representation is defined as follows (Narayan A. and Moore M., 1998):

$$\begin{bmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \beta_1 & \beta_2 & \dots & \beta_n \end{bmatrix}; \quad \text{where } |\alpha_i|^2 + |\beta_i|^2 = 1, \quad \text{with } i = 1, 2, \dots, n;$$

Each chromosome is encoded as a matrix of $2 \times n$ QuBits. This allows a chromosome to encode not only one solution but all the possible solutions by using the superposition principle. Again, $|\alpha_i|^2$ and $|\beta_i|^2$ are the probabilities to measure respectively the value $|0\rangle$ and the value $|1\rangle$ for the QuBit i of a certain chromosome. Each quantum operation regards in parallel all the states present within the superposition (characteristic of diversity; for more details see (Narayan A. and Moore M., 1998)). Only one QuBit chromosome is enough to represent c states, while in a classical bit representation at least c chromosomes are needed. This means that the QuBit representation possesses simultaneously the two characteristics of exploration and exploitation. If $|\alpha_i|^2$ or $|\beta_i|^2$ converges to 1 or 0, the QuBit chromosome stretches to a single state and the property of diversity disappears gradually.

The procedure for a general Quantum-inspired Genetic Algorithm, starting from an initial population, applies four quantum operators (Quantum crossover interference, Classical crossover operator, Mutation, Shifting) and an evaluation. Formally, it may be specified as follows (Talbi H., Draa A. and Batouche M., 2004):

```

m = Choose-Chromosomes-Number ();
n = Choose-QuBits-Number ();
P = Generate-Initial-Population (m, n);
Measure&Evaluate (P);
while termination conditions not met do
    PI = Interference (P);    // Quantum crossover interference
    PII = Recombine (PI);    // Classical crossover operator
    PIII = Mutate (PII);
    PIV = Shift (PIII);
    Measure&Evaluate (PIV);
    P = Select-New-Population (PIV | m);
Endwhile    // end inner loop
    
```

At the beginning, the initial population P , composed of m “quantum chromosomes”, is randomly generated. After the four operators of Quantum crossover interference, Classical crossover operator, Mutation, and Shifting are applied .

The first operation is a *quantum interference (crossover)* that allows a shift of each QuBit in the direction of the corresponding bit value in the best solution. That is performed by rotating the specific QuBit (α_i and β_i) of an angle, K , function of the value of the corresponding bit in the best solution (FIG 5).

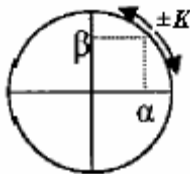


FIG 5: Quantum Interference: rotation operator (Talbi H., Draa A. and Batouche M., 2004).

α	β	Reference bit value	Rotation angle K
>0	>0	1	+ K
>0	<0	0	- K
>0	>0	1	- K
>0	<0	0	+ K
<0	>0	1	- K
<0	<0	0	+ K
<0	>0	1	+ K
<0	<0	0	- K

TABLE 1: Example of rotation angle of a QuBit in function of the bit values of the best solution (Talbi H., Draa A. and Batouche M., 2004).

TABLE 1 shows an example of the sign of the K rotation angle of a chosen QuBit in function of the reference bit value of the best solution (α and β are the component of K).

The second operation is a *classical crossover* performed between each pair of the m chromosomes at a random position, obtaining an overall population of $m \times m$ chromosomes. The third operation is a random *mutation* over some chromosomes. It depends on the probability of applying permutations on a chromosome, and the probability of permuting each column in the chromosome with another column. The fourth operation consists of a random *shifting* of a chromosome, in order to increase the diversification of the search process.

After these four quantum operators, a special kind of measurement is applied to the resulting solutions in order to extract the best one. In Quantum Mechanics, only states containing exactly one QuBit with the value 1 in each line, and exactly one QuBit having the value 1 in each column (coherent solution) are possible. Conversely, in QGA, the final measurement does not destroy the states superposition, keeping all the possible solutions for the following iterations. After the evaluation of the solutions, a new population for the next iteration is selected if the termination conditions are not reached. The new population will consist of the best $(m - 1)$ chromosomes from the current population, plus one chromosome randomly selected among the other ones (in order to maintain a good diversity).

The increased performance of Quantum-inspired Genetic Algorithms with respect to classical Genetic Algorithms may be attributed mainly to interference crossover and to the multiple superpositions of individuals. Interference crossover provides a larger number of chromosomes to choose for the next generation, while the multiple superpositions of individuals allow losing less good solutions during each step.

If progress continues at this rate, future computer circuits will be based on nanotechnology and the behaviour of such circuits will have to be given in quantum mechanical terms rather than in terms of classical physics (since on the atomic scale matter obeys the laws of quantum mechanics). Such compilers will require less translation to machine language than the classical ones, so carrying really efficiency benefits (Narayan A. and Moore M., 1998). Even if it is currently not clear how true quantum computation

algorithms will be related to quantum hardware (e.g. quantum logic gates), quantum-inspired computing could help quantum hardware platforms to be feasible.

The increased performance of Quantum-inspired Genetic Algorithms with respect to classical Genetic Algorithms has been proved for some classical combinatorial optimization problems, such as the (Narayan A. and Moore M., 1998):

- Traveling Salesman Problem (TSP);
- Knapsack Problems;
- depth-two and/or tree problem.

4.2 ANT COLONY OPTIMIZATION (ACO)

Ant colony optimization (ACO) was first proposed in the early 90's by Marco Dorigo and colleagues. As Dorigo M. and Stützle T. (2004) state, its inspiring source is the foraging behaviour of real ants. When searching for food, ants initially explore the area surrounding their nest in a random manner. As soon as an ant finds a food source, it evaluates quantity and quality of the food and carries some of the found food to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone deposited, which may depend on the quantity and quality of the food, will guide other ants to the food source. The indirect communication between the ants via the pheromone trails allows them to find shortest paths between their nest and food sources (FIG 6). This functionality of real ant colonies is exploited in artificial models in order to solve discrete optimization problems.

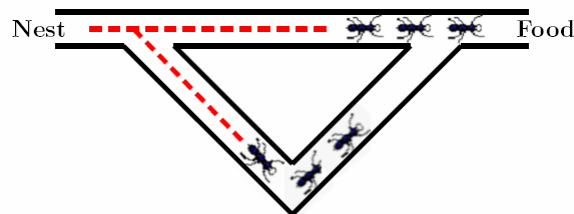


FIG 6: Foraging behaviour of real ants.

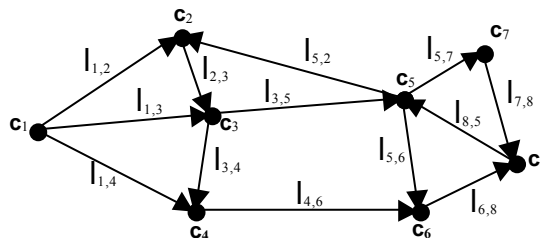


FIG 7: Example of a *decision (or constructive) graph*.

ACO algorithms use the parameterized probabilistic *pheromone model* to include the chemical pheromone trails: ants incrementally construct solutions by adding opportunely defined solution components to a partial solution under consideration. In doing that, artificial ants perform randomized walks on a completely connected *construction* (or *decision*) graph $G = (C, L)$, whose vertices c_i are the solution components C and the set L are the connections l_{ij} (FIG 7).

Pheromone trail parameters, T_i and T_{ij} , are associated, respectively, with every node c_i and each arch l_{ij} , which also have assigned the *pheromone values* τ_i and τ_{ij} . Furthermore, *a priori or run time values*, η_i and η_{ij} , are associated with every node c_i and each arch l_{ij} respectively (Dorigo M. and Stützle T., 2004).

It is possible to define the following useful parameter sets:

- $T = \{T_i ; T_{ij}\} \Leftrightarrow$ *set of pheromone trail parameters*
- $H = \{\eta_i ; \eta_{ij} ; \tau_i ; \tau_{ij}\} \Leftrightarrow$ *set of all the heuristic values*

All these values are used by the ants to take probabilistic decisions on how to move on the decision graph. The probabilities involving in moving on the construction graph are commonly called *transition probabilities*.

The simplest ACO algorithm is the *Ant System (AS)*, which is only based on the pheromone trail parameters T and the set of the heuristic values H . Considering a as a single ant, A as the set of all ants a_i , and s_a as the solution constructed by $a \in A$, a basic description of this procedure is specified as follows (Blum C. and Roli A., 2003):

```

Initialize-Pheromone-Values ( $T, H$ );
while termination conditions not met do
  for all ants  $a \in A$  do
     $s_a =$  Construct-Solution( $T, H$ );
  endfor
  Apply-Online-Delayed-Pheromone-Update ( $T, \{s_a / a \in A\}$ );
Endwhile
    
```

After the initialization (Initialize-Pheromone-Values (T, H)) of the pheromone parameters T and heuristic values H ($\eta_i = \eta_{ij} = \tau_i = \tau_{ij} = \text{ph} > 0$), each ant constructs a solution of the problem at each step of the algorithm. In this phase an ant incrementally builds a solution by adding probabilistic-chosen components (by means of transition probabilities) to the partial solution constructed so far. Only feasible solution components can be added to the current partial solution, avoiding the infeasible ones.

This mechanism is called *state transition rule* (Dorigo M. and Stützle T., 2004):

$$\text{prob}(c_r / s_a[c_k]) = \begin{cases} \frac{[\eta_r]^\alpha \cdot [\tau_r]^\beta}{\sum_{c_u \in J(s_a[c_k])} [\eta_u]^\alpha \cdot [\tau_u]^\beta} & \text{if } c_r \in J(s_a[c_k]); \\ 0 & \text{otherwise.} \end{cases}$$

where the above parameters have the following meaning:

- $prob(c_r / s_a[c_k])$ is the probability of adding the component c_r to the partial solution, $s_a[c_k]$, constructed so far (c_k is the last node added);
- α , β adjust, respectively, the relative importance of heuristic information and pheromone values;
- $J_a(s_a[c_k])$ is the set of solution components allowed to be added to the partial solution $s_a[c_k]$.

Once all ants have constructed a solution, the *online delayed pheromone update rule* (Apply-Online-Delayed-Pheromone-Update ($T, \{s_a / a \in A\}$)) increases the pheromone of solution components that have been found in high-quality solution (Dorigo M. and Stützle T., 2004):

$$\tau_j = (1 - \rho) \cdot \tau_j + \sum_{a \in A} \Delta \tau_j^{s_a} \quad \forall \tau_j \in T$$

$$\text{with } \Delta \tau_j^{s_a} = \begin{cases} F(s_a) & \text{if } c_j \text{ is a component of } s_a; \\ 0 & \text{otherwise.} \end{cases}$$

and where:

- $F(\cdot) : S \rightarrow \Re$ is the *quality function* satisfying:
 $\forall s \in S, \forall s' \in S / s \neq s', \quad f(s) < f(s') \Rightarrow F(s) > F(s')$;
- ρ is the *pheromone evaporation rate*, with value included between 1 and 0 ($0 < \rho \leq 1$).

An extension of AS is the more general *Ant Colony Optimization (ACO)* metaheuristic (FIG 8). It consists of three parts assembled in the Schedule-Activities construct and synchronized following the decisions of the designer.

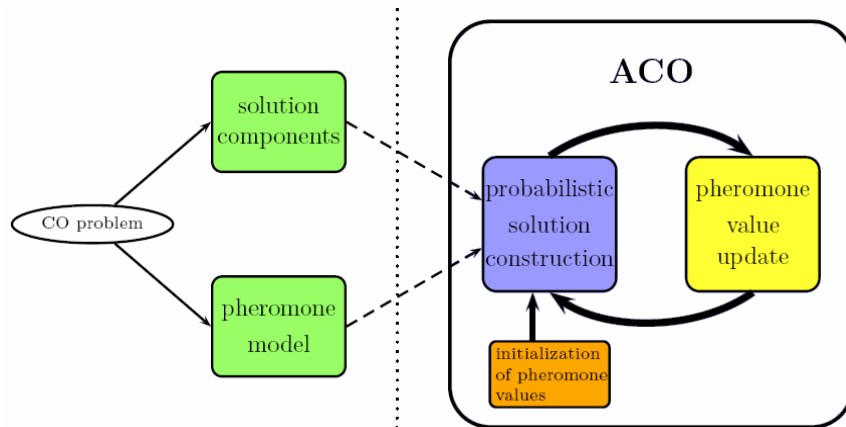


FIG 8: Overview of the Ant Colony Optimization metaheuristic (Blum, 2005).

Formally, in pseudo code, the procedure may be specified as follow (Blum C. and Roli A., 2003):

```
while termination conditions not met do  
  Schedule-Activities  
    Ant-Based-Solution-Construction();  
    Pheromone-Update();  
    Daemon-Actions();    //optional  
  end Schedule-Activities  
Endwhile
```

In the Ant-Based-Solution-Construction method, each ant builds a solution moving through the decision graph G , following the same state transition rule pointed out in AS. In this phase, this mechanism makes use of a sort of memory because each ant keeps the partial solution it has built in terms of path, allowing to retrace backward the same way. Instead, the Pheromone-Update method improves AS including two kinds of different update processes:

1) *Online step-by-step pheromone update (real time)*: When an ant is walking on connection l_{ij} in order to reach component c_j to add to the current partial solution, it updates step-by-step the pheromone trail parameters and their values, τ_i and τ_{ij} ;

2) *Online delayed pheromone update (offline)*: If the quality of the built solution is good, the ant retraces the path backward and updates the pheromone trail parameters and their values, τ_i and τ_{ij} , and/or connections l_{ij} , according to the degree of excellence of this solution.

ACO includes also the mechanism of *Pheromone evaporation*. The pheromone values, τ_i , decrease with time to avoid rapid convergence to local minima, due to the nature of the online delayed pheromone update rule. It represents a form of “forgetting”, allowing an easier exploration of new areas of the search domain.

It is also possible to apply a centralized action on the algorithm by means Daemon-Actions. For example, a daemon entity may collect global information about the path found by each ant, and can decide whether to apply additional weight to the pheromone of the components used by the ant that built the best solution. Pheromone updates performed by the daemon are called *offline pheromone updates*.

Currently, the best performing implementations of ACO are (Dorigo M. and Stützle T., 2004):

- *Ant Colony System (ACS)*, created by Dorigo M.;
- *MAX-MIN Ant System (MMAS)*, first designed by Stützle T.

Ant Colony System (ACS) metaheuristic has been introduced to improve the performance of AS. Its main differences with AS are:

1) *Daemon offline pheromone updates*: It is the same optional mechanism carried out by Daemon-Action entity in ACO, but here it becomes mandatory. At the end of an iteration, every ant builds a solution. Then, pheromone bias is added to the pheromone values, τ_i and τ_{ij} , of the arcs within the best solution.

2) *Pseudo random-proportional rule*: Ants use two mechanisms to decide where to move next in the decision graph. The first mechanism involves deterministic moves (in a greedy manner) to intensify the search around high-quality solutions. The second one includes random movements, chosen by the usual decision rule, to diversify the search process.

3) *Online step-by-step pheromone update*: As outlined, the pheromone trail parameters and their values, τ_i and τ_{ij} , of the ant walking on arcs $l_{i,j}$ to reach component c_j , are update in real-time. No online delayed pheromone updates are allowed.

Instead, *MAX-MIN Ant System (MMAS)* improves AS through the following strategies:

1) *Daemon offline pheromone updates*: as in ACS, additional pheromone is added to the arcs used by the best ant within each iteration.

2) *Bounded pheromone values*: the pheromone values, τ_i and τ_{ij} , are bounded to vary in a finite interval $[\tau_{\min}, \tau_{\max}]$, after being initialized to τ_{\max} . In this way, the probability of constructing a solution can't exceed a minimum threshold value (a lower bound ≥ 0), previously fixed. So, solutions apparently of medium/low-quality have the chance to find a global optimum, increasing the diversification factor of the search.

3) *Application of restarts*: they are periodic re-initialization of the pheromone values in order to encourage the diversification of the search.

Current research tries to use ACO with other metaheuristics in order to create hybridizations. Similarities between ACO and probabilistic learning algorithms have been found, such as with Estimation of Distribution Algorithms (EDA), with Population-Based Incremental Learning (PBIL) algorithms, with Stochastic Gradient Descent (SGD) algorithms, and with the Univariate Marginal Distribution Algorithms (UMDA). For more details see (Blum C. and Roli A., 2003).

Successful applications of ACO include a lot of applications to combinatorial optimization problems (Blum C. and Roli A., 2003):

- Knapsack problems;
- Job Shop Scheduling (JSS) problem;
- routing in communication networks;
- Sequential Ordering Problem (SOP);
- Resource Constraint Project Scheduling (RCPS).

In the following sections, two significant forces of intensification and diversification, that mainly determine the behavior of metaheuristics, will be pointed out, concluding by exploring the importance of hybridization and integration methods.

5. INTENSIFICATION AND DIVERSIFICATION FORCES

As shown in the previous section, the utilization of randomness in local search algorithms can lead to significant increases in their performance and robustness. However, with this potential comes the need to balance randomized and goal-directed components of the search strategy, a trade-off that is often characterized as 'diversification vs intensification'. As already stated, the term diversification means the exploration of the search space while the term intensification means the exploitation of the accumulated

search-experience. When the search process starts, it needs to compute the value of different points in the search domain in order to find the promising areas (diversification). Then the algorithm needs to investigate promising zones to find the local-optimum (intensification). The best local optimum found in the different areas will be the candidate solution, hoping to be as near as possible to the global optimum that the algorithm is looking for. The terms “diversification” and “intensification” are mainly used in methods based on the concept of memory, such as Tabu Search. Conversely the terms “exploration” and “exploitation” are used in strategies that don’t require explicit usage of memory, such as evolutionary computation. Finding a good balance between diversification (exploration) and intensification (exploitation) is essential for a metaheuristic in order to quickly identify regions in the search space with high quality solutions, without wasting too much time in regions with a low quality.

Balancing properly these two strengths is a crucial issue in heuristics and so a large variety of techniques have been proposed in recent years. These techniques are often based on intuition and experience rather than on theoretically or empirically derived principles. In this context, both problem specific knowledge and a solid understanding of the properties and characteristics of the different metaheuristics are crucial for achieving peak performance and robustness. Intensification and diversification are not contradictory options. Each feature contains aspects of the other.

A framework may be helpful in providing a unified view on these critical components in order to focus this concept and to underline similarities and differences among the different metaheuristic approaches (Blum C. and Roli A., 2003). *I&D component* is defined as any algorithmic or functional component (operators, actions, or strategies) that has an intensification and/or a diversification effect on the search process (e.g.: genetic operators, perturbations of probability distributions, the use of tabu lists, changes in the objective function...). Any of these components contains either an intensification or a diversification effect, as can be shown in the *I&D frame* (FIG 9). The *I&D frame* compares each other the I&D components of different metaheuristics (Blum C. and Roli A., 2003).

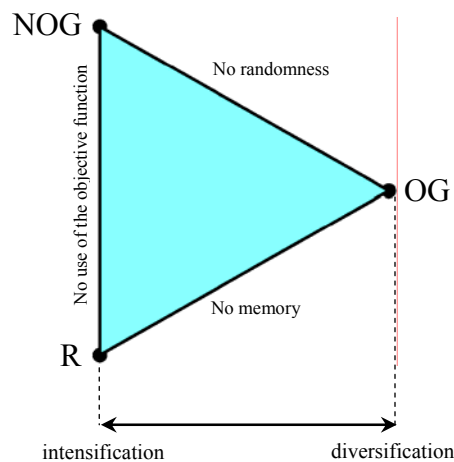


FIG 9: The *I&D frame*, a unified view on intensification and diversification (Blum C. and Roli A. 2003).

The space of all I&D components is drawn as a triangle with the three extreme corners:

1) **OG (Objective function Guided)**, which corresponds to components solely guided by the objective function of the problem (max intensification, min diversification). For example, the steepest descent choice rule in local search is placed in this corner;

2) **NOG (Not only Objective function Guided)**, which covers all components guided by one or more functions other than the objective function without using of any randomness (max diversification, min intensification). The deterministic restart mechanism based on global frequency counts of solution components is an example of such a component;

3) **R (Randomness guided)**, which comprises completely random I&D components, guided by nothing (max diversification, min intensification). An example is a restart with random individuals in EC algorithms.

The less an I&D component uses the objective function, the further away from corner OG it has to be located. In the same manner, the less randomness is involved, the further away from corner R it has to be located. Furthermore, a third gradient describes the influence of criteria different from the objective function. These criteria generally come from the exploitation of the search history, stored in some form of memory.

In the following table (TABLE 2), some basic I&D components, intrinsic to the classic metaheuristics, are listed.

Metaheuristic	I&D component
SA	acceptance criterion+ cooling schedule
TS	neighbor choice (tabu lists)
	aspiration criterion
EC	recombination
	mutation
	selection
ACO	pheromone update
	probabilistic construction
ILS	black box local search
	kick-move
	acceptance criterion
VNS	black box local search
	neighborhood choice
	shaking phase
	acceptance criterion
GRASP	black box local search
	restricted candidate list
GLS	penalty function

TABLE 2: I&D components intrinsic to the basic metaheuristics (Blum C. and Roli A. 2003).

6. HYBRIDIZATION OF METAHEURISTICS

A current trend is the integration of single point search methods with population-based ones. In this section, a brief description of the most important hybridization approaches is given. It is possible to divide hybrid methods into three classes.

The first class, called *Components Exchange Among Metaheuristics*, consists of methods including components from different metaheuristics, usually from a trajectory method and a population-based one. The strength of population-based methods is the concept of recombining solutions, explicitly in EC algorithms and Scatter Search by recombination operators, implicitly in ACO and EDA for the nature of their mechanisms.

The recombination follows the criterion to mix high-quality solutions in the hope of finding better ones, on the followed direction. The recombination in population-based methods allows “big” guided steps in the search space, usually larger than the ones performed by trajectory methods. Some trajectory methods, such as ILS and VNS, also perform “big” steps, but resulting from random mechanisms called “kick move” or “perturbation”, indicating the absence of guidance.

Instead, the strength of trajectory methods is based on a local search, to strictly explore a promising region in the search space. In this way, the danger of being close to good solutions but “missing” them is not as high as in population-based methods. Most of the successful applications of EC and ACO make use of local search procedures. Summarizing, population-based methods are better at identifying promising areas in the search space, whereas trajectory methods are superior in exploring specific zones of the domain. Thus, hybrid metaheuristics, combining the advantages of population-based methods with the power of trajectory methods, are often very successful today (Blum C. and Roli A., 2003).

The second form of hybridization, *Cooperative Search*, consists of a search carried out by different algorithms, approximate or complete ones, exchanging information about states, models, entire sub-problems, solutions or other search space characteristics. Cooperative algorithms can be either different search techniques, or instances of the same algorithm with different settings of the model or the parameters, or algorithms in parallel execution with a variable level of communication. Cooperative Search also receives much attention as a result of the rapid growth of parallel implementations of metaheuristics. Research on parallelization is focused on the re-design of metaheuristics in order to make them suitable for parallel implementations.

The latter hybridization class is the *Integration of Metaheuristics and Systematic (or complete) Search Methods*, producing very effective algorithms for real-world applications (e.g. the successful integration of metaheuristics in Constraint Programming (CP)). There are three main approaches for the integration of metaheuristics (especially trajectory methods) and systematic techniques (CP and tree search).

The first consists in their sequential application and/or their interleaved execution. For example, the metaheuristic may produce solutions which are then improved by systematic search (or vice-versa, the systematic algorithm may generate partial solutions which are completed by the metaheuristic). This procedure can also be viewed as a loose form of cooperative search.

The second approach uses a complete method to efficiently explore the neighborhood, instead of randomly sampling it or simply enumerating the neighbors. This

approach is particularly effective when the neighborhood to explore is very large, because it combines the advantages of a fast exploration, by using a metaheuristic, with efficient neighborhood exploitation, performed by a systematic method.

The third possibility consists of introducing concepts or strategies from classes of algorithms into others. A typical case is a probabilistic backtracking instead of a deterministic one into a search-tree algorithm. This is carried out through the introduction of the concepts of tabu list and aspiration criteria into a search-tree algorithm, in order to manage the list of open nodes to explore.

7. SUMMARY AND CONCLUSIONS

This report presents a review of metaheuristic methods, covering both theoretical aspects of metaheuristics and applications to Combinatorial Optimization problems.

Metaheuristics are classified as trajectory-based (e.g. Basic Local Search, Simulated Annealing, Tabu Search, GRASP, Variable Neighborhood Search, Guided Local Search, Iterated Local Search) and as population-based (e.g. Evolutionary Computation, Scatter Search, Path Relinking, Estimation of Distribution Algorithms, Quantum-inspired Genetic Algorithms, Ant Colony Optimization). The use of intensification and diversification with both categories of metaheuristics is extremely analyzed in this report. Finally, the promising area of metaheuristic hybridization is explored.

An example of hybridization is the attempt to introduce the concept of memory in SA, which is a memory-less method, through the integration with other usage-memory metaheuristics, such as Tabu Search (Aarts E. H. L., Korst J. H. M. and Laarhoven P. J. M. V., 1997). In general, the TS field is a rich source of ideas, which have been and are currently adopted by other metaheuristics. Another promising research direction in TS is to create more advanced ways to adapt the tabu tenure dynamically (Glover F., 1986).

As TS, also GRASP may be successfully integrated into other search techniques, due to its simplicity and high speed (generally). However, a basic GRASP does not use the history of the search process: the only memory requirement is for storing the problem instance and for keeping the best so-far solution. This is one of the reasons why GRASP is often outperformed by other metaheuristics. So, another promising research direction is trying to introduce the concept of memory in GRASP through other usage-memory methods (Blum C. and Roli A., 2003).

Instead, current research area in VNS is huge. A systematic study of moves (or neighborhoods) for whole classes of problems together with the data-structures most adequate for their implementation is one promising direction. Another one is trying to consider more sophisticated distributions of neighborhood. Introduction of memory, parallel VNS, hybridizing VNS within exact algorithms, enhancing graph theory are all research directions for VNS (Hansen P. and Mladenović N., 2005).

In Evolutionary Computation, recent successes were obtained in the rapidly growing bioinformatics area, in multiobjective optimization and in evolvable hardware (Blum C. and Roli A., 2003). Path relinking is often used as a component in metaheuristics such as Tabu Search and GRASP. Differently, the field of EDA is still quite young and much of the research effort is focused on methodology rather than high performance applications. Recently, researchers have been dealing with finding similarities between ACO algorithms

and probabilistic learning algorithms such as EDA. An important step into this direction was the development of the Hyper-Cube Framework for Ant Colony Optimization (HC-ACO) (Blum C. et al., 2001). Furthermore, connections of ACO algorithms to Stochastic Gradient Descent (SGD) algorithms represent a research area of growing interest. Instead, current research in Quantum-inspired Genetic Algorithms is trying to make quantum hardware platforms feasible, even if it is currently not clear how true quantum computation algorithms will be related to quantum hardware (Narayan A. and Moore M., 1998).

In conclusion, there is a need for the hybridization of metaheuristics to be examined in detail in order to be able to produce hybrid metaheuristics that perform better than their “pure” parents.

BIBLIOGRAPHY

- Aarts E. H. L., Korst J. H. M. and Laarhoven P. J. M. V., (1997). 'Simulated annealing'. In Aarts E. H. L. and Lenstra J. K. (1997). *Local Search in Combinatorial Optimization*, Chichester, England: in Wiley-Interscience, 91-120.
- Blum C., Roli A. and Dorigo M., (2001). 'HC-ACO: The hyper-cube framework for ant colony optimization'. In *Proceedings of MIC'2001-Meta-heuristics International Conference*, Vol. 2., Porto, Portugal, 399-403.
- Blum C. and Roli A., (2003). 'Metaheuristics in combinatorial optimization: Overview and conceptual comparison', *ACM Computing Surveys*, 35(3): 268-308.
- Blum C. (2005). 'Ant Colony Optimization: Introduction and recent advances', *HIS 2005 Rio de Janeiro, Brazil*.
- Burke E.K. and Kendall G., (2006). *Search methodologies. Introductory tutorials in optimization and decision support techniques*, Germany: in Springer-Verlag Berlin Heidelberg.
- Cerny V., (1985). 'A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm', *Journal of Optimization Theory and Applications*, 45: 41-51.
- Dorigo M. and Stützle T., (2004). *Ant Colony Optimization*, Boston, Massachusetts: in MIT Press.
- Garey M. and Johnson D., (1979). *Computers and intractability; a guide to the theory of NP-Completeness*, New York: in W. H. Freeman.
- Glover F., (1986). 'Future paths for integer programming and links to artificial intelligence', *Computers and Operations Research*, 13: 533.
- Glover F., Laguna M. and Marti R., (2000). 'Fundamentals of Scatter Search and Path Relinking', *Control and Cybernetics Journal*, 39, 3: 653-684.
- Glover F. and Kochenberger G.A. (2003). *Handbook of metaheuristics*, Norwell, Massachusetts: in Kluwer Academic Publishers.
- Goldberg D. E., Deb K. and Korb B., (1991). 'Don't worry, be messy', In *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan-Kaufmann, La Jolla, CA.
- Hansen P. and Mladenović N., (1999). 'An introduction to variable neighborhood search'. In Voss S., Martello S., Osman I.H. and Roucairol C., *Meta-Heuristics. Advanced and Trends Local Search Paradigms for Optimization*, Norwell, Massachusetts: in Kluwer Academic Publishers, Chapter 30, 433-458.
- Hansen P. and Mladenović N., (2001). 'Variable neighborhood search: Principles and applications'. *European Journal of Operational Research*, 130: 449-467.
- Hansen P. and Mladenović N., (2003). 'Variable neighborhood search'. In Glover F. and Kochenberger G.A. (2003). *Handbook of metaheuristics*, Norwell, Massachusetts: in Kluwer Academic Publishers, Chapter 6, 145-184.
- Hansen P. and Mladenović N., (2005). 'Variable neighborhood search'. In Burke E.K. and Kendall G., (2006). *Search methodologies. Introductory tutorials in optimization and decision support techniques*, Germany: in Springer-Verlag Berlin Heidelberg, Chapter 8, 211-238.
- Harik G., (1999). 'Linkage learning via probabilistic modeling in the ECGA', *University of Illinois IlliGAL*, Tech. Rep. No. 99010, ,

- Ho Y.C. and Pepyne D.L., (2002). 'Simple Explanation of the No-Free-Lunch Theorem and Its Implications', *Journal of Optimization Theory and Applications*, 115: 549.
- Holland J. H., (1975). *Adaption in natural and artificial systems*, Michigan in: The University of Michigan Press, Ann Harbor.
- Holland J. H., (1992). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, Cambridge, London in: MIT Press.
- Hoos H.H. and Stützle T., (2005). *Stochastic local search. Foundations and applications*, Elsevier, San Francisco, CA: Morgan Kaufmann Publishers.
- Kirkpatrick S., Gelatt C. D. and Vecchi M. P., (1983). 'Optimization by simulated annealing', *Science*, 220, 4598: 671-680.
- Korte B. and Vygen J., (2006). *Combinatorial Optimization, theory and algorithms*, Germany: in Springer-Verlag Berlin Heidelberg.
- Martello S. and Toth P., (1990). *Knapsack problem. Algorithms and computer implementations*, Chichester: in John Wiley & sons.
- Michalewicz Z., (1996). 'Evolutionary Computation and Heuristics'. In Osman I.H. and Kelly J.P., (1996). *Meta-Heuristics: Theory and Applications*, Norwell, Massachusetts: in Kluwer Academic Publishers, Chapter 3, 37-52.
- Moscato P., (1999). 'Memetic algorithms: A short introduction'. In Corne F. G. D. and Dorigo M., *New Ideas in Optimization*, Eds. McGraw-Hill.
- Narayan A. and Moore M., (1998). 'Quantum-Inspired Genetic Algorithms', *IEEE Transactions on Evolutionary Computation*, Vol. 3: 2238.
- Narayan A., (1999). 'Quantum Computing for Beginners', Technical Report 344, Department of Computer Science, University of Exeter, England.
- Osman I.H., and Kelly J.P., (1996). *Meta-Heuristics: Theory and Applications*, Norwell, Massachusetts: in Kluwer Academic Publishers.
- Osman I.H. and Laporte G., (1996). 'Meta-Heuristics: A bibliography', *Annals of Operations Research*, 63: 513-623.
- Rayward-Smith V. J., (1994). 'A unified approach to tabu search, simulated annealing and genetic algorithms', In Rayward-Smith V.J., *Applications of Modern Heuristics*, Ed. Alfred Waller Limited Publishers.
- Rayward-Smith V.J., Osman I.H., Reeves C.R. and Smith G.D., (1996). *Modern heuristic search methods*, Chichester: in John Wiley & sons.
- Shor P. W., (1994). 'Algorithms for quantum computation: Discrete logarithms and factoring', *35th Annual Symposium on Foundations of Computer Science*, IEEE Press.
- Talbi H., Draa A. and Batouche M., (2004). 'A New Quantum-Inspired Genetic Algorithm for Solving the Traveling Salesman Problem', *International Conference on Industrial Technology*, IEEE Press.
- Topon Kumar P. and Hitoshi I., (2002). 'Linear and Combinatorial Optimizations by Estimation of Distribution Algorithms', *9th MPS Symposium on Evolutionary Computation, IPSJ*, 1-8.
- Van Kemenade C. H. M., (1996). 'Explicit filtering of building blocks for genetic algorithms'. In Voigt H.M., Ebeling W., Rechenberg I. and Schwefel H.P., *Proceedings of the 4th Conference on Parallel Problem Solving from Nature - PPSN IV*, Eds. Lecture Notes in Computer Science, vol. 1141. Springer, Berlin, 494-503.

- Voss S., Martello S., Osman I.H. and Roucairol C., (1999). *Meta-Heuristics. Advanced and Trends Local Search Paradigms for Optimization*, Norwell, Massachusetts: in Kluwer Academic Publishers.
- Watson R.A., Hornby G.S. and Pollack J.B., (1998). 'Modeling building-block interdependency'. In Koza J. R., (1998). *Late Breaking Papers at the Genetic Programming 1998 Conference*, Ed. Stanford University Bookstore, University of Wisconsin, Madison, Wisconsin, USA.
- Wikipedia website, (2006), <http://en.wikipedia.org>. Visited in January 2006.
- Wolpert D.H. and Macready W.G., (1995). 'No Free Lunch Theorems for Search', Santa Fe Institute Technical Report, SFI-TR-95-02-010.
- Wolpert D.H. and Macready W.G., (1997). 'No Free Lunch Theorems for Optimization', *IEEE Transactions on Evolutionary Computation*, 1: 67-82.