Modern Iterative Algorithms and their Applications in Computer Engineering

Sadiq M. Sait and Habib Youssef

Introduction

Combinatorial optimization problems are encountered everywhere, in science, engineering, as well as in industrial management, economics, etc. Most Engineering and business schools offer several courses in algorithms and optimization. The advent of the digital computer is credited for all of this explosion in the amount of algorithmic solutions to combinatorial optimization problems. Such solution techniques were unthinkable before this magnificent invention.

In this chapter we are concerned with one class of combinatorial optimization algorithms: *general iterative non-deterministic algorithms.* The growing interest in this class of algorithms is attributed to their generality, ease of implementation, and mainly, the many success stories reporting very positive results. We shall limit ourselves to five dominant iterative non-deterministic algorithms, which, in order of popularity are: (1) Simulated Annealing (SA), (2) Genetic Algorithm (GA), (3) Tabu Search (TS), (4) Simulated Evolution, and (5) Stochastic Evolution. All five search heuristics have several important properties in common.

1. They are blind, in that they do not know when they reached the optimal solution. Therefore they must be told when to stop.

2. They are approximation algorithms, that is, they do not guarantee finding an optimal solution.

3. They have 'hill climbing' property, that is, they occasionally accept uphill (bad) moves.

4. They are easy to implement. All that is required is to have a suitable solution representation, a cost function, and a mechanism to traverse the search space.

5. They are all '*general*'. Practically they can be applied to solve any combinatorial optimization problem.

6. They all strive to exploit domain specific heuristic knowledge to bias the search toward "good" solution subspace. The quality of subspace searched depends to a large extent on the amount of heuristic knowledge used.

7. Although they asymptotically converge to an optimal solution, the rate of convergence is heavily dependent on the adequate choice of several parameters.

The last two properties are the hidden bone in the five combinatorial optimization strategies. Our goal in this chapter is to briefly introduce these five heuristics.

Most literature on computer algorithms mainly addresses deterministic heuristics. Recently, due to the increase in size and complexity of a large number of combinatorial optimization problems, there has been a growing interest in general iterative non-deterministic algorithms.

The chapter is organized into 6 sections. In the following 5 sections, we introduce the five iterative algorithms: namely, Simulated Annealing, Genetic Algorithm, Tabu Search, Simulated Evolution, and Stochastic Evolution. Only an intuitive discussion given and an essence of the heuristic is given. Pointers to other details such as convergence analysis, parallel implementation, applications etc., are given in bibliograpy.

1 Simulated Annealing

Simulated Annealing is one of the most well developed and widely used iterative techniques for solving optimization problems.

It is a general *adaptive* heuristic and belongs to the class of *non-deterministic* algorithms [NSS89]. It has been applied to several combinatorial optimization problems from various fields of science and engineering. These problems include TSP (traveling salesman problem), graph partitioning, quadratic assignment, matching, linear arrangement, and scheduling. In the area of engineering, simulated annealing has been applied to VLSI design (placement, routing, logic minimization, testing), image processing, code design, facilities layout [AK89], network topology design [EP93], etc.

One typical feature of simulated annealing is that, besides accepting solutions with improved cost, it also, to a limited extent, accepts solution with deteriorated cost. It is this feature that gives the heuristic the hill climbing capability. Initially the probability of accepting inferior solutions (those with larger costs) is large; but as the search progresses, only smaller deteriorations are accepted, and finally only good solutions are accepted. A strong feature of the simulated annealing heuristic is that it is both effective and robust. Regardless of the choice of the initial configuration it produces high quality solutions. It is also relatively easy to implement. Let us begin this section by first introducing annealing from an intuitive point of view.

1.1 Background

The term *annealing* refers to heating a solid to a very high temperature (whereby the atoms gain enough energy to break the chemical bonds and become free to move), and then slowly cooling the molten material in a controlled manner until it crystallizes. By cooling the metal at a proper rate, atoms will have an increased chance to regain proper crystal structure with perfect lattices. During this annealing procedure the free energy of the solid is *minimized*.

As early as 1953, Metropolis and his colleagues introduced a simple algorithm to simulate the evolution of a solid in a heat bath to its thermal equilibrium [M$^+$53]. Their simulation algorithm is based on *Monte Carlo techniques* and generates a *sequence of states* of the solid as follows. Given a current state S_i of the solid with energy E_i, a subsequent state S_j with energy E_j is generated by applying a perturbation mechanism. This perturbation transforms the current state into a next state with slight distortion. For instance a new state can be constructed by randomly selecting a particle and displacing it by some random amount. If the energy associated with the new state is lower than the energy of the current state, that is, $\Delta E = E_j - E_i \leq 0$, then the displacement is accepted, and the current state becomes the new state. However, if the energy of the new state is higher (the energy difference greater than zero), then the state S_j is accepted with a certain probability, which is given by

$$\text{Prob(accept)} = e^{-(\frac{\Delta E}{K_B T})} \tag{1}$$

where K_B is the Boltzmann constant and T denotes temperature. The acceptance rule described above is repeated a very large number of times. The acceptance criterion is known as the *Metropolis step*, named after its inventor, and the procedure is known as the *Metropolis algorithm*.

In the early eighties, thirty years after the idea of the Metropolis loop was introduced, a correspondence between annealing and combinatorial optimization was established, first by Kirkpatrick, Gelatt and Vecchi [KCGV83] in 1983, and independently by Černy [Čer85] in 1985. These scientists observed that there is a correspondence between, on one hand, *a solution to the optimization problem* and *a physical state of the material*, and between the *cost* of a solution of the combinatorial optimization problem and *free energy* in the molten metal. As a result of this analogy they introduced a solution method in the field of combinatorial optimization. This method is thus based on the simulation of the physical annealing process, and hence the name *simulated annealing* [KCGV83, Čer85].

As explained by Kirkpatrick et al. and Černy, a solution in combinatorial optimization is equivalent to a *state* in the physical system and the cost of the solution is analogous to the *energy* of that state. If we compare optimization to the annealing process, the attainment of global optimum is analogous to the attainment of a perfect crystal structure (a minimum energy state for the

material), and attainment of a structure with imperfections will correspond to getting trapped in a local optimum.

Every combinatorial optimization problem may be discussed in terms of a *state space*. A *state* is simply a configuration of the combinatorial objects involved. For example, consider the problem of partitioning a graph of $2n$ nodes into two equal sized subgraphs such that the number of edges with vertices in both subgraphs is minimized. In this problem, any division of $2n$ nodes into two equal sized blocks is a configuration. There is a large number of such configurations. Only some of these correspond to global optima, i.e., states with optimum cost.

An iterative improvement scheme starts with some given state, and examines a *local neighborhood* of the state for better solutions. A local neighborhood of a state S, denoted by $\aleph(S)$, is the set of all states which can be reached from S by making a small change to S. For instance, if S represents a two-way partition of a graph, the set of all partitions which are generated by swapping two nodes across the partition represents a local neighborhood. The iterative improvement algorithm moves from the current state to a state in the local neighborhood if the latter has a better cost. If all the local neighbors have larger costs, the algorithm is said to have *converged* to a *local optimum*. This is illustrated in Figure 1. Here, the states are shown along the x-axis, and it is assumed that

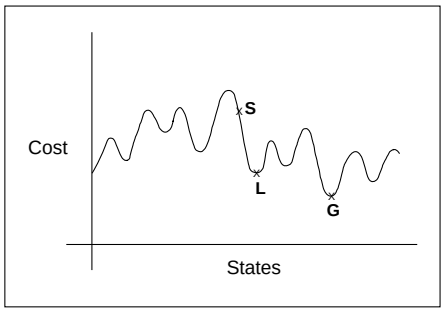

Figure 1: Local versus global optima.

two consecutive states are local neighbors. It is further assumed that we are discussing a *minimization* problem. The cost curve is *non-convex*, i.e., it has multiple minima. A greedy iterative improvement algorithm may start off with an initial solution such as **S** in Figure 1, then slide along the curve and find a local minimum such as **L**. There is no way such an algorithm can find the global minimum **G** of Figure 1, unless it "climbs the hill" at the local minimum **L**. In other words, an algorithm which occasionally accepts inferior solutions can escape from getting trapped in a local optimum. Simulated annealing is such a hill-climbing algorithm.

During annealing, a metal is maintained at a certain temperature T for a pre-computed amount of time, before reducing the temperature in a controlled manner. The atoms have a greater degree of freedom to move at higher temperatures than at lower temperatures. *The movement of atoms is analogous to the generation of new neighborhood states in an optimization process.* In order to simulate the annealing process, much flexibility is allowed in neighborhood generation at higher "temperatures", i.e., many 'uphill' moves are permitted at higher temperatures. The temperature parameter is lowered gradually as the algorithm proceeds. As the temperature is lowered, fewer and fewer uphill moves are permitted. In fact, at absolute zero, the simulated annealing algorithm turns greedy, allowing only downhill moves.

We can visualize simulated annealing by considering the analogy of a ball placed in a hilly terrain, as shown in Figure 2. The hilly terrain is nothing but the variation of the cost function over the configuration space, as shown by Figure 1. If a ball is placed at point **S**, it will roll down into a pit such as **L**, which represents a local minimum. In order to move the ball from the local minimum to the global minimum **G** we do the following. We enclose the hilly terrain in a box and place the box in a water bath. When the water bath is heated, the box begins to shake, and the ball has a chance to climb out of the local minimum **L**.

If we are to apply simulated annealing to this problem, we would initially heat the water bath to a high temperature, making the box wobble violently. At such high temperatures, the ball moves rapidly into and out of local minima. As time proceeds, we cool the water bath gradually. The lower the temperature, the gentler the movement of the box, and lesser the likelihood of the ball jumping out of a minimum. The search for a local minimum is more or less random at high temperatures; the search becomes more greedy as temperature falls. At absolute zero, the box is perfectly still, and the ball rolls down into a minimum, which, hopefully, is the global minimum **G**.

■

1.2 Simulated Annealing Algorithm

The simulated annealing algorithm is shown in Figure 3. The core of the algorithm is the *Metropolis* procedure, which simulates the annealing process at a given temperature T (Figure 4) [M$^+$53]. The *Metropolis* procedure receives as input the current temperature T, and the current solution $CurS$ which it improves through local search. Finally, *Metropolis* must also be provided with the value M, which is the amount of time for which annealing must be applied at temperature T. The procedure *Simulated_annealing* simply invokes *Metropolis* at decreasing temperatures. Temperature is initialized to a value T_0 at the beginning of the procedure, and is reduced in a controlled manner (typically in a geometric progression); the parameter α is used to achieve this cooling. The

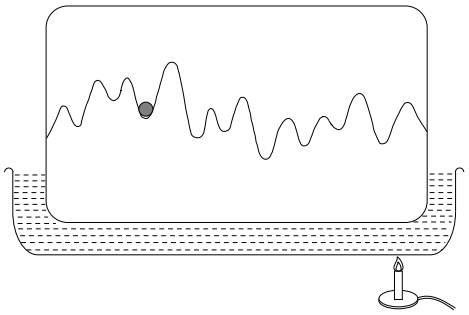

Figure 2: Design space analogous to a hilly terrain.

amount of time spent in annealing at a temperature is gradually *increased* as temperature is lowered. This is done using the parameter $\beta > 1$. The variable *Time* keeps track of the time being expended in each call to the *Metropolis*. The annealing procedure halts when *Time* exceeds the allowed time.

The *Metropolis* procedure is shown in Figure 4. It uses the procedure *Neighbor* to generate a local neighbor $NewS$ of any given solution S. The function *Cost* returns the cost of a given solution S. If the cost of the new solution $NewS$ is better than the cost of the current solution $CurS$, then the new solution is accepted, and we do so by setting $CurS$=$NewS$. If the cost of the new solution is better than the best solution ($BestS$) seen thus far, then we also replace $BestS$ by $NewS$. If the new solution has a higher cost in comparison to the original solution $CurS$, *Metropolis* will accept the new solution on a *probabilistic* basis. A random number is generated in the range 0 to 1. If this random number is smaller than $e^{-\Delta Cost/T}$, where $\Delta Cost$ is the difference in costs, ($\Delta Cost$= $Cost(NewS) - Cost(CurS)$), and T is the current temperature, the uphill solution is accepted. This criterion for accepting the new solution is known as the *Metropolis criterion*. The *Metropolis* procedure generates and examines M solutions.

The probability that an inferior solution is accepted by the *Metropolis* is given by $P(RANDOM < e^{-\Delta Cost/T})$. The random number generation is assumed to follow a *uniform distribution*. Remember that $\Delta Cost > 0$ since we have assumed that $NewS$ is uphill from $CurS$. At very high temperatures, (when $T \rightarrow \infty$), $e^{-\Delta Cost/T} \simeq 1$, and hence the above probability approaches 1. On the contrary, when $T \rightarrow 0$, the probability $e^{-\Delta Cost/T}$ falls to 0.

In order to implement simulated annealing on a digital computer we need to formulate a suitable cost function for the problem being solved. In addition, as in the case of local search techniques we assume the existence of a neighborhood structure, and need *perturb* operation or *Neighbor* function to generate new states (neighborhood states) from current states. And finally, we need a control

Algorithm Simulated_annealing($S_0, T_0, \alpha, \beta, M, Maxtime$);
(*S_0 is the initial solution *)
(*BestS is the best solution *)
(*T_0 is the initial temperature *)
(*α is the cooling rate *)
(*β a constant *)
(*$Maxtime$ is the total allowed time for the annealing process *)
(*M represents the time until the next parameter update *)

Begin
$T = T_0$;
$CurS$=S_0;
$BestS$=$CurS$;/* $BestS$ is the best solution seen so far */
$CurCost$=$Cost$($CurS$);
$BestCost$=$Cost$($BestS$);
$Time = 0$;
Repeat
Call Metropolis($CurS$, $CurCost$, $BestS$, $BestCost$, T, M);
$Time = Time + M$;
$T = \alpha T$;
$M = \beta M$
Until ($Time \geq MaxTime$);
Return ($BestS$)
End. (*of Simulated_annealing*)

Figure 3: Procedure for simulated annealing algorithm.

Algorithm Metropolis($CurS$, $CurCost$, $BestS$, $BestCost$, T, M);
Begin
Repeat
$NewS$= $Neighbor$($CurS$); /* Return a neighbor from $aleph$($CurS$) */
$NewCost$=$Cost$($NewS$);
$\Delta Cost$=($NewCost$−$CurCost$);
If ($\Delta Cost < 0$) **Then**
$CurS$=$NewS$;
If $NewCost < BestCost$ **Then**
$BestS$= $NewS$
EndIf
Else
If ($RANDOM < e^{-\Delta Cost/T}$) **Then**
$CurS$=$NewS$;
EndIf
EndIf
$M = M - 1$
Until ($M = 0$)
End. (*of Metropolis*)

parameter to play the role of temperature and a random number generator. The actions of simulated annealing are best illustrated with the help of an example. For the solution of the two way partitioning problem using SA, please refer to [**?**].

This all goes in one section, section 7 summarised in few sentences.

The convergence aspects of the simulated annealing algorithm have been the subject of extensive studies. For a thorough discussion of simulated annealing convergence we refer the reader to [AK89, AL85, OvG89].

put this in one sentence in the algorithm Parameters of the SA Algorithm

In the previous paragraphs we demonstrated that, if simulated annealing is allowed to run for an infinitely long time, starting with a high value of T, and allowing $T \rightarrow 0$, then it will find a desired optimal configuration. In practice, however, simulated annealing is only run for a finite amount of time. A finite time implementation can be realized by generating homogeneous Markov chains of finite lengths for a sequence of decreasing values of temperature. To achieve this, a set of parameters that govern the convergence of the algorithm must be specified. This set of parameters is commonly referred to as the "cooling schedule" [AK89, OvG89, KCGV83].

Again cite the book and summarize the above para.

It is customary to determine the schedule by trial and error. However, some researchers have proposed cooling schedules that rely on some mathematical rigor [**?**].

SA Requirements? SA Applications

TimberWolf3.2

A popular package that uses simulated annealing for VLSI standard-cell placement and routing is the TimberWolf3.2 package [SSV86].

Parallelization of SA

Conclusions and Recent Work

2 Genetic Algorithms

Genetic Algorithm (GA), is a powerful, domain-independent, search technique that was inspired by Darwinian theory. It emulates the natural process of evolution to perform an efficient and systematic search of the solution space to progress toward the optimum. It is based on the theory of *natural selection* that assumes that individuals with certain characteristics are more able to survive, and hence pass their characteristics to their offsprings.

Genetic algorithm is an *adaptive* learning heuristic. Similar to simulated annealing, it also belongs to the class of general *non-deterministic* algorithms. Several variations of the basic algorithm (modified to adapt to the problem at hand) exist. We will henceforth refer to this set as genetic algorithms (in plural).

Genetic algorithms (GAs) operate on a *population* (or set) of *individuals* (or solutions) encoded as strings. These strings represent points in the search space. In each iteration, referred to as a generation, a new set of strings that represent solutions (called offsprings) is created by crossing some of the strings of the current generation [Gol89]. Occasionally new characteristics are injected to add diversity. GAs combine information exchange along with *survival of the fittest* among individuals to conduct the search.

Since their appearance, GAs have been applied to solve several combinatorial optimization problems from various fields of science, engineering and business (see Section **??**).

Let us begin this section with a brief introduction to background and terminology and subsequently present the basic genetic algorithm.

Schema Theorem

In living organisms, as members of the population mate, they produce offsprings that have a significant chance of retaining the desirable characteristics of their parents, and sometimes even combine or inherit the 'best' characteristics of both parents. By establishing a correspondence between, on one hand, *a solution to the optimization problem* and the element of the population (represented by the *chromosome*), and between the *cost* of a solution and the *fitness* of an individual in the population, a solution method in the field of combinatorial optimization is introduced. The method thus simulates the process of natural evolution based on Darwinian principles, and hence the name *Genetic Algorithm*[Gol89, Hol75].

Genetic algorithms (GAs) were invented by John Holland and his colleagues [Hol75] in the early 1970s. Holland incorporated features of natural evolution to propose a *robust*, computationally simple, and yet powerful technique for solving difficult optimization problems.

When employing GAs to solve a combinatorial optimization problem one has to find an efficient representation of the solution in the form of a chromosome (encoded string). Associated with each chromosome is its *fitness value*. If we simulate the process of natural reproduction, combined with the biological principle of survival of the fittest, then, as each generation progresses, better

and better individuals (solutions) with higher fitness values are expected to be produced.

Robustness

Genetic algorithms, on the other hand, are both effective and *robust* [Dav91, Gol89, **?**]. The main characteristics of GA are listed below:

They work with coding of parameters: GAs work with a coding of the parameter set, not the parameters themselves. Therefore, one requirement when employing GAs to solve a combinatorial optimization problem is to find an *efficient* representation of the solution in the form of a chromosome (encoded string).

They search from a set of points: In other optimization methods such as simulated annealing or tabu search we move from a single point in the search space, using some transition rule, to the next point. This type of point to point movement most often causes trapping in local optima. In contrast, GAs simultaneously work from a rich collection of points (a population of solutions). Therefore, the probability of getting trapped in false valleys (in case of minimization problem) is reduced.

They only require objective function values: GAs are not limited by assumptions about the search space (such as continuity, existence of derivatives, etc.,), and they do not need or use any auxiliary information. To perform an effective search for better and better structures, they **only** require *objective* (cost) *function* values.

They are non-deterministic: GAs use probabilistic transition rules, not deterministic rules. Mechanism for choice of parents to produce offsprings, or for combining of genes in various chromosomes are probabilistic.

They are blind: They are blind in the sense that they do not know when they hit the optimum, and therefore they must be told when to stop.

The structure that encodes how the organism is to be constructed is called a *chromosome*. One or more chromosomes may be associated with each member of the population. The complete set of chromosomes is called a *genotype* and the resulting organism is called a *phenotype*. Similarly, the representation of a solution to the optimization problem in the form of an encoded string is termed as a *chromosome*. In most combinatorial optimization problems a single chromosome is generally sufficient to represent a solution, that is, the genotype and the chromosome are the same.

The symbols that make up a chromosome are known as *genes*. The different values a gene can take are called *alleles*.

The fitness value of an individual (genotype or a chromosome) is a *positive* number that is a measure of its goodness. When the chromosome represents a solution to the combinatorial optimization problem, the fitness value indicates

the cost of the solution. In the case of a minimization problem, solutions with lower cost correspond to individuals that are more fit.

This is not the only way in which one can map costs to fitness values. There are other effective schemes which we discuss later in Section **??**.

GAs work on a population of solutions. An *initial population constructor* is required to generate a certain predefined number of solutions. The quality of the final solution produced by a genetic algorithm depends on the size of the population and how the initial population is constructed. The initial population generally comprises random solutions. Later we elaborate on other schemes to construct the initial population (see Section **??**).

GAs work on chromosomes or pairs of chromosomes to produce new solutions called offsprings. Common genetic operators are *crossover* (χ) and *mutation*. They are derived by analogy from the biological process of evolution. Crossover operator is applied to pairs of chromosomes. The two individuals selected for crossover are called *parents*. Mutation is another genetic operator that is applied to a single chromosome. The resulting individuals produced when genetic operators are applied on the parents are termed as *offsprings*.

Choice of Parents The choice of parents for crossover from the set of individuals that comprise the population is probabilistic. In keeping with the ideas of natural selection, we assume that stronger individuals, that is those with higher fitness values, are more likely to mate than the weaker ones. One way to simulate this is to select parents with a probability that is directly proportional to their fitness values. That is, the larger the fitness of a certain chromosome, the greater is its chance of being selected as one of the parents for crossover.

To accomplish this type of selection we may use the *roulette-wheel method*. In this method a wheel is constructed on which each member of the population is given a sector whose size is proportional to the relative fitness of that individual. To select a parent the wheel is spun, and whichever individual comes up becomes the selected parent. Therefore, in this method, individuals with lower fitness values also have a finite but lower probability of being selected for crossover [Gol89].

Crossover (χ) is the main genetic operator. It provides a mechanism for the offspring to inherit the characteristics of both the parents. It operates on two parents (P_1 and P_2) to generate *offspring*(s).

There are several crossover operators that have been proposed in the literature. Depending on the combinatorial optimization problem being solved some are more effective than others. One popular crossover that will also help illustrate the concept is the *simple crossover*. It performs the "cut-catenate" operation. It consists of choosing a *random* cut point and dividing each of the two chromosomes into two parts. The offspring is then generated by catenating the segment of one parent to the left of the cut point with the segment of the second parent to the right of the cut point.

Mutation (μ) produces incremental random changes in the offspring by randomly changing allele values of some genes. In case of binary chromosomes it

corresponds to changing single bit positions. It is not applied to all members of the population, but is applied probabilistically only to some. Mutation has the effect of perturbing a certain chromosome in order to introduce *new* characteristics not present in any element of the parent population. For example, in case of binary chromosomes, toggling some selected bit produces the desired effect.

A *generation* is an iteration of GA where individuals in the current population are selected for crossover and offsprings are created. Due to the addition of offsprings, the size of population increases. In order to keep the number of members in a population fixed, a constant number of individuals are selected from this set which consists of both the individuals of the initial population, and the generated offsprings. If $\mathbf{M}$ is the size of the initial population and N_o is the number of offsprings created in each generation, then, before the beginning of next generation, we select $\mathbf{M}$ new parents from $\mathbf{M} + N_o$ individuals. A greedy selection mechanism is to choose the best $\mathbf{M}$ individuals from the total of $\mathbf{M} + N_o$.

We will now summarize the main aspects of the basic genetic algorithm.

2.1 Genetic Algorithm

In order to implement the genetic algorithm on a digital computer, one of the most important steps is to encode the solution of the combinatorial optimization as a string of symbols, also known as chromosome. This encoding must be amenable to genetic operations. In addition to this, unlike in other search techniques, GAs do not operate on one solution but a collection of solutions termed population. An *initial population constructor* is required to generate a certain predefined number of solutions. The quality of final solution depends upon the size of the population and how the initial population is constructed. The population comprises random solutions, or, a combination of random solutions and those produced using known constructive heuristics. We also need a mechanism to generate offsprings from parent solutions.

During each generation of the genetic algorithm a set of offsprings are produced by the application of the *crossover* operator. The crossover operator ensures that the offsprings generated have a mixture of parental properties. In order to introduce new alleles into the chromosome, with a certain probability, *mutation* is also applied. Following this, from the entire pool comprising both the parents and their offsprings, a fixed number of individuals are chosen that form the population of the new generation. If the $\mathbf{M}$ best individuals are chosen from this pool, then the fitness of the best individual, will be the same or better than the fitness of the best individual of the previous generation. Similarly, the average fitness of the population will be the same or higher than the average fitness of the previous generation. Thus the fitness of the entire population and the fitness of the best individual increase in each generation. The structure of the simple genetic algorithm is given in Figure 5.

```
Procedure (Genetic_Algorithm)
  M= Population size.                (*# Of possible solutions at any instance.*)
  N_g= Number of generations.        (*# Of iterations.*)
  N_o= Number of offsprings.         (*To be generated by crossover.*)
  P_μ= Mutation probability.         (*Also called mutation rate M_r.*)
  P ← Ξ(M)                           (*Construct initial population P. Ξ is population constructor.*)
  For j = 1 to M                     (*Evaluate fitnesses of all individuals.*)
      Evaluate f(P[j])               (*Evaluate fitness of P.*)
  EndFor
  For i = 1 to N_g
      For j = 1 to N_o
          (x, y) ← φ(P)              (*Select two parents x and y from current population.*)
          offspring[j] ← χ(x, y)     (*Generate offsprings by crossover of parents x and y.*)
          Evaluate f(offspring[j])   (*Evaluate fitness of each offsprings.*)
      EndFor

      For j = 1 to N_o               (*With probability P_μ apply mutation.*)
          mutated[j] ← μ(y)
          Evaluate f(mutated[j])
      EndFor
  P ← Select(P, offsprings)          (*Select best M solutions from parents & offsprings.*)
  EndFor
  Return highest scoring configuration in P.
End
```

Figure 5: Structure of a simple genetic algorithm.

3 Schema Theorem and Implicit Parallelism

In this section we throw more light on what is processed by GAs and show how this processing will lead to optimal results in our optimization problems. We will see how crossover, the critical accelerator of the search process combines parts of good solutions from diverse chromosomes [Gol89, Hol75, **?**].

To study the what and how of GAs performance, we resort to the notion of *schema*.

4 GA Convergence Aspects

One of the desirable properties that a stochastic iterative algorithm should possess is the convergence property, i.e., the guarantee of converging to one of the global optima if given enough time. In this section, we examine the convergence properties of the GA heuristic. Convergence aspects of GA using Markovian analysis has been addressed by several researchers [GS87, NV93, EAH90, DP91, DP93, Mah93, Rud94]. Fogel [Fog95] provides a concise treatment of the main GA convergence results.

5 GA In Practice

Before we look into more examples and case studies of GAs in the field of science and engineering,

Inversion

Inversion is the third operator of GA and like mutation it also operates on a single chromosome. Its basic function is to laterally invert the order of alleles between two randomly chosen points on a chromosome.

Other Issues GA Applications Genetic algorithms applications Parallelization of GA

GAs may suffer from the problem of premature convergence. Their effectiveness can be increased by including some features of other heuristics. For example, GAs are combined with simulated annealing to introduce more diversity into the population thereby preventing premature convergence. A complete section in is dedicated to this issue, where we discuss combination of GAs with other heuristics (such as tabu search and simulated annealing) discussed in this chapter [**?**].

6 Conclusions

In this section we presented the basics of genetic algorithms. These algorithms emulate the natural process of evolution. Unlike other search heuristics, they conduct the search by operating on a set of solutions called the population. They work with chromosomal representations (encoded strings) of solutions, require only objective function values, and search from a set of points. The basic idea is to combine solutions called parents to produce new solutions called offsprings, with the objective that the offsprings will inherit some parental characteristics. To accomplish this, crossover is used. It is the crossover operator that distinguishes GAs from other optimization algorithms. We discussed several crossover operators. Mutation is another operator that is used to inject new characteristics in the individuals.

In this section, we also shed some light on the fundamental theorem of genetic algorithms, the schema theorem.

Several variations of the basic technique, convergence related issues, and practical considerations for implementation of GAs on a digital computer, were discussed. Implementation aspects of this powerful iterative heuristics were presented with applications and case studies. A brief survey of various problems to which GAs have been successfully applied was presented. We also touched upon techniques for parallelizing GAs, and summarized several recent related issues.

All five heuristics described in this chapter constitute very general and effective optimization techniques. Recently, SA, GA, and TS has been designated by the Committee of the Next Decade of Operations Research as 'extremely promising' for the future treatment of practical applications[1]. It is our belief that Simulated Evolution and Stochastic Evolution are equally promising techniques to a wide array of combinatorial optimization problems.

7 Tabu Search

In the previous section we discussed simulated annealing, which was inspired by the cooling of metals, and genetic algorithms, which imitate the biological phenomena of evolutionary reproduction. In this section we present a more recent optimization method called Tabu Search (TS) which is based on selected concepts of artificial intelligence (AI).

Tabu search was introduced by Fred Glover [Glo89, Glo90b, GTdW93, GL97] as a general iterative heuristic for solving combinatorial optimization problems. Initial ideas of the technique were also proposed by Hansen [Han86] in his *steepest ascent mildest descent* heuristic.

Tabu search is conceptually simple and elegant. It is a form of local neighborhood search. Each solution $S \in \Omega$ has an associated set of neighbors $\aleph(S) \subseteq \Omega$. A solution $S' \in \aleph(S)$ can be reached from S by an operation called a *move* to S'. Normally, the neighborhood relation is assumed symmetric. That is, if S' is a neighbor of S then S is a neighbor of S'.

Tabu search is a generalization of local search. At each step, the local neighborhood of the current solution is explored and the best solution in that neighborhood is selected as the new current solution. Unlike local search which stops when no improved new solution is found in the current neighborhood, tabu search continues the search from the best solution in the neighborhood even if it is worse than the current solution. To prevent cycling, information pertaining to the most recently visited solutions are inserted in a list called *tabu list*. Moves to tabu solutions are not allowed. The tabu status of a solution is overridden when certain criteria (aspiration criteria) are satisfied. One example of an aspiration criterion is when the cost of the selected solution is better than the best seen so far, which is an indication that the search is actually not cycling back, but rather moving to a new solution not encountered before.

Tabu search is a *metaheuristic*, which can be used not only to guide search in complex solution spaces, but also to direct the operations of *other* heuristic procedures. It can be superimposed on any heuristic whose operations are characterized as performing a sequence of *moves* that lead the procedure from one trial solution to another. In addition to several other characteristics, the attractiveness of tabu search comes from its ability to escape local optima.

[1] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

Tabu search differs from simulated annealing or genetic algorithm which are "memoryless", and also from branch-and-bound, A* search, etc., which are rigid memory approaches. One of its features is its systematic use of *adaptive* (flexible) memory. It is based on very simple ideas with a clever combination of components, namely [Glo90a, SM93]:

1. a short-term memory component; this component is the core of the tabu search algorithm,

2. an intermediate-term memory component; this component is used for regionally **intensifying** the search, and,

3. a long-term memory component; this component is used for globally **diversifying** the search.

As will be elaborated in this section, the central idea underlying tabu search is the exploitation of the above three memory components. Using the short-term memory, a *selective history* $\mathbf{H}$ of the states encountered is maintained to guide the search process. Neighborhood $\aleph(S)$ is replaced by a modified neighborhood which is a function of the history $\mathbf{H}$, and is denoted by $\aleph(\mathbf{H}, S)$. History determines which solutions may be reached by a move from S, since the next state S is selected from $\aleph(\mathbf{H}, S)$. The short-term memory component is implemented through a set of *tabu* conditions and the associated *aspiration criterion*.

The major idea of the short-term memory component is to classify certain search directions as tabu (or *forbidden*). By doing so we avoid returning to previously visited solutions. Search is therefore forced away from recently visited solutions, with the help of a memory known as *tabu list* $\mathbf{T}$. This memory contains *attributes* of some k most recent moves. The size of the tabu list denoted by k is the number of iterations for which a move containing that attribute is forbidden after it has been made. The tabu list can be visualized as a window on accepted moves as shown in Figure 6. The moves which tend to undo previous moves within this window are forbidden.

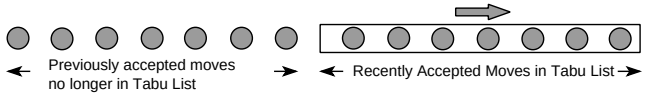

Figure 6: The tabu list can be visualized as a window over accepted moves.

Cycling back to previously visited solutions is prevented by the tabu list(s). However, since only move attributes (not complete solutions) are stored in tabu lists, these tabu moves may also prevent the consideration of some solutions which were *not* visited earlier. To relax the actions of tabu lists, aspiration

criteria are introduced. Then, solutions that are the result of moves having attributes found in the tabu list are also considered *if* they satisfy the aspiration criteria. A flow chart illustrating the basic short-term memory tabu search algorithm is given in Figure 7. Intermediate-term and long-term memory processes are used to intensify and diversify the search respectively, and have been found to be very effective in increasing both quality and efficiency [Glo95, Glo96, DV93].

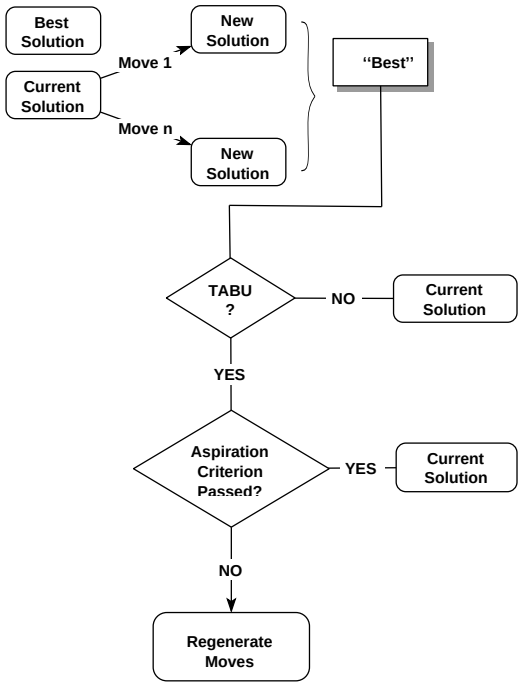

Figure 7: Flow chart of the tabu search algorithm.

We first introduce the basic tabu search algorithm based on the short-term memory component (Section 8). Then we will build the necessary background and the required terminology. Following this we present some practical issues of the tabu search algorithm for implementation on a digital computer (Section **??**). Limitations of short-term memory, uses of intermediate and long-term memories, and strategies for diversifying the search are explained in Sections **??** and **??**.

8 Tabu Search Algorithm

An algorithmic description of a simple implementation of the tabu search is given in Figure 8. The procedure starts from an initial feasible solution S (current

```
Ω      :  Set of feasible solutions.
S      :  Current solution.
S*     :  Best admissible solution.
Cost   :  Objective function.
ℵ(S)   :  Neighborhood of S ∈ Ω.
V*     :  Sample of neighborhood solutions.
T      :  Tabu list.
AL     :  Aspiration Level.

       Begin
1.     Start with an initial feasible solution S ∈ Ω.
2.     Initialize tabu lists and aspiration level.
3.     For  fixed number of iterations Do
4.           Generate neighbor solutions V* ⊂ ℵ(S).
5.           Find best S* ∈ V*.
6.           If  move S to S* is not in T   Then
7.                 Accept move and update best solution.
8.                 Update tabu list and aspiration level.
9.                 Increment iteration number.
10.          Else
11.                If  Cost(S*) < AL Then
12.                      Accept move and update best solution.
13.                      Update tabu list and aspiration level.
14.                      Increment iteration number.
15.                EndIf
16.          EndIf
17.    EndFor
       End.
```

Figure 8: Algorithmic description of short-term Tabu Search (TS).

solution) in the search space Ω. A neighborhood $\aleph(S)$ is defined for each S. A sample of neighbor solutions $\mathbf{V}^* \subset \aleph(S)$ is generated. An extreme case is to generate the entire neighborhood, that is to take $\mathbf{V}^* = \aleph(S)$. Since this is generally impractical (computationally expensive), a small sample of neighbors ($\mathbf{V}^* \subset \aleph(S)$) is generated called *trial* solutions ($|\mathbf{V}^*| = n \ll |\aleph(S)|$). From these trial solutions the best solution, say $S^* \in \mathbf{V}^*$, is chosen for consideration as the next solution. The move to S^* is considered even if S^* is worse than S, that is,

$Cost(S^*) > Cost(S)$. A move from S to S^* is made provided certain conditions are satisfied.

Selecting the best move in $\mathbf{V}^*$ is based on the assumption that good moves are more likely to reach optimal or near-optimal solutions. As mentioned above, the best candidate solution $S^* \in \mathbf{V}^*$ *may* or *may not* improve the current solution, but is still considered. It is this feature that enables *escaping* from local optima. However, even with this strategy, it is possible to reach a local optimum, ascend (in case of a minimization problem) since moves with $Cost(S^*) > Cost(S)$ are accepted, and then in a later iteration return back to the same local optimum. That is, there is a possibility of *cycling* by returning back to previously visited solutions. This may cause the search to go through the same subset of solutions for ever.

A tabu list is maintained to prevent returning to previously visited solutions. This list contains information that to some extent forbids the search from returning to a previously visited solution. It is *not* a list of solutions, since storing previously visited solutions, even a small number of them, and comparing them with newly generated ones would be expensive both in terms of computation time and memory requirement. Instead, selected move attributes are stored in the tabu list. Tabu restrictions therefore may also forbid moves to attractive *unvisited* solutions. For example, if a move is made tabu in iteration i and its reversal comes in iteration j, where $j = i + l$ and $1 < l < |\mathbf{T}|$, then it is possible that the reverse move, although tabu, may take the search into a new region because of the effects of $l - 1$ intermediate (previous) moves. It is therefore necessary to relax the actions of the tabu list and overrule the tabu status of moves in certain situations. This is done with the help of the notion of *aspiration criterion*.

Aspiration criterion is a device used to override the tabu status of moves whenever appropriate. It temporarily overrides the tabu status if the move is sufficiently good. The aspiration criterion must make sure that the reversal of a recently made move (i.e., a move in the tabu list) leads the search to an unvisited solution, generally a better one.

Several aspiration criteria have been suggested and used in the literature. The customary one, also the simplest and most commonly used, overrides the tabu status if the reversal of a move in the tabu list produces a solution better than the best obtained thus far during the search. This is also known as *best solution* aspiration criterion. Other aspiration criteria will be discussed later (see Section **??**).

Refering again to the algorithmic description in Figure 8, initially the current solution is the best solution. Copies of the current solution are perturbed with moves to get a set of new solutions. The best among these is selected and if it is not tabu then it becomes the current solution. If the move is tabu its aspiration criterion is checked. If it passes the aspiration criterion then it becomes the current solution. If the move to the next solution is accepted, then the move or some of its attributes are stored in the tabu list. Otherwise moves

are regenerated to get another set of new solutions. If the current solution is better than the best seen thus far, then the best solution is updated. Whenever a move is accepted the iteration number is incremented. The procedure continues for a fixed number of iterations, or until some pre-specified stopping criterion is satisfied.

Tabu restrictions and aspiration criterion have a symmetric role. The order of checking for tabu status and aspiration criterion may be reversed, though most applications check if a move is tabu before checking for aspiration criterion [Glo90c].

Below we explain some phrases and terms frequently used in this section. We will illustrate the working of the basic tabu search algorithm with the help of an example. Following this, we discuss various implementation related issues.

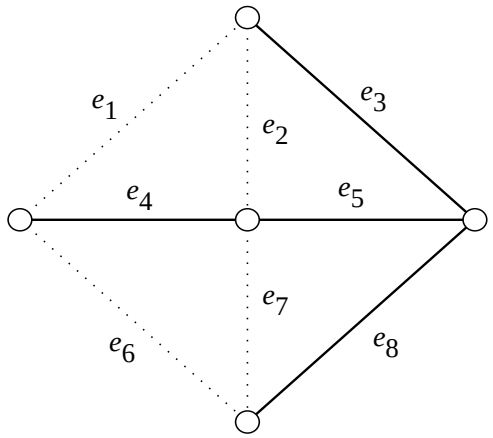

Figure 9: Graph for Problem in Example **??**. Cost of tree = 9+1+3+7+2×100=220.

Implementation Related Issues In the previous example we illustrated some basic characteristics of the short-term tabu search heuristic. There are several variations of the above method that can be incorporated. Below we give a flavor of some of these variations which are related to moves and their attributes, tabu lists and their sizes, possible data structures, aspiration criteria, etc. [GL95]

8.1 Move Attributes

8.2 Tabu List and Tabu Restrictions

8.3 Data Structure to Handle Tabu-lists

8.4 Other Aspiration Criteria

9 Limitations of Short-Term Memory

In many applications, the short-term memory component by itself has produced solutions superior to those found by alternative procedures, and usually the use of intermediate-term and long-term memory is bypassed. However, several studies have shown that intermediate and long-term memory components can improve solution quality and/or performance [MGPO89, Rya89, IE94, DV93].

9.1 Intermediate-Term Memory (Search Intensification)

The basic role of the intermediate-term memory component is to *intensify* the search. By its incorporation, the search becomes more aggressive. As the name suggests, memory is used to intensify the search.

This type of intensification strategy is useful for solving large problems because the search focuses on generating solutions that are good, and only a subset of decision elements are incorporated in these solutions.

9.2 Long-Term Memory (Search Diversification)

The goal of long-term memory component is to *diversify* the search. The principles involved here are just the *opposite* of those used by the intermediate-term memory function. Instead of more intensively focusing the search with regions that contain previously found good solutions, the function of this component is to drive the search process into new regions that are different from those examined thus far.

Diversification is used to explore new regions of the solution space. Most heuristic search techniques use or have a built in mechanism for diversifying the search. Without diversification, the search can become localized in a small area of the solution space, eliminating the possibility of finding a global optimum.

The introduction of randomization to achieve diversification is common among search procedures such as simulated annealing and genetic algorithms. Simulated annealing incorporates randomization to make diversification a function of temperature. Genetic algorithms also use randomization in crossover, mutation, and, selection, to diversify the search. Diversification strategies in tabu search are designed and used in a number of ways. The long-term memory component is used to incorporate diversification. Generally there appears to be a hidden assumption that diversification must tantamount to randomization

[KLG94]. However, in tabu search, deterministic diversification is employed with the help of short- and long-term memories.

Diversification using long-term memory in tabu search can be accomplished by creating an evaluator whose task is to take the search to new starting points [Glo89]. For example, in the TSP, a simple form of long-term memory is to keep a count of the number of times each edge has appeared in the tours previously generated. Then, an evaluator can be used to penalize each edge on the basis of this count, thereby favoring the generation of "other hopefully good" starting tours that tend to avoid those edges most commonly used in the past. This sort of approach is viewed as a **frequency** based tabu criterion in contrast to the **recency** based (tabu list) illustrated earlier. Such a long-term strategy can be employed by means of a long-term tabu list (or any other appropriate data structure) which is periodically activated to employ tabu conditions of increased stringency, thereby forcing the search process into new territory.

It is easy to create and test the short-term memory component first, and then incorporate the intermediate/long components for additional refinements. Below we illustrate this by two examples. One is an optimization problem based on the short-term memory component. The other example illustrates how frequency can be accommodated in the previous example to diversify the search. Examples of other techniques used for diversification are discussed in Section **??**.

Penalizing Frequent Moves

We now illustrate how the long-term memory can be incorporated to diversify the search process. To do this, we use the same matrix data structure used in our earlier example (Example **??**, Page **??**). Recall that the upper diagonal matrix was used to store the recency information. We will use the lower diagonal matrix to store the frequency of moves made. That is, each entry (i, j) in the lower diagonal matrix stores the number of times the swap (i, j) was made. We can then use this information to define a move evaluator $\mathcal{E}(\mathbf{H}, S)$, which is a function of both the cost of the solution, and the frequency of the swaps stored. Our objective is to diversify the search by giving more consideration to those swaps that have not been made yet, and to penalize those that frequently occurred [LG93]. Therefore, the design of the evaluator must be such that moves that most frequently occurred in the past are given less consideration. For example, if a swap (i, j) was made to take the solution from current state S to a new state S^*, and the term $\text{Freq}(i, j)$ is the number of times swap (i, j) was made, then the evaluation of the move can be expressed as follows:

$$\mathcal{E}(\mathbf{H}, S^*) = \begin{cases} Cost(S^*) & Cost(S^*) \leq Cost(S) \\ Cost(S^*) + \alpha \times \text{Freq}(i, j) & Cost(S^*) > Cost(S) \end{cases}$$

α is constant which depends on the range of the objective function values, the number of iterations, the span of history considered, etc. Its value (α's) is such that cost and frequency are appropriately balanced.

Examples of Diversifying Search TS Convergence Aspects TS Applications Parallelization of TS Other Parallel Implementations:

10 Conclusions

In this section we presented the basics of tabu search heuristic. Several implementation issues such as moves and their attributes, tabu-lists (static/dynamic) and tabu restrictions, data structures to handle tabu lists, and various aspiration criteria were presented with examples (Section **??**).

Tabu search is different from other search techniques in several respects. One, of course, is the use of memory. In addition, reasonably sized subset of neighborhood is explored and the best move amongst these is chosen. Further, unlike other search techniques where 'best' generally refers to the best cost of the solution, in tabu search best refers to change in the evaluation function which depends not only on the objective/cost function, but also on the search history, region being searched, etc.

The core of the tabu search algorithm is the short-term memory component, implemented with the help of tabu_list(s) and aspiration criterion. Intermediate and long-term memory components are also used for intensification and diversification (Sections 9.1 and 9.2). Diversification techniques that penalize frequently occurring moves, and others that are suitable for permutation problems, were discussed in Section **??**.

Convergence aspects were discussed in Section **??**. Deterministic tabu search is not guaranteed to converge, on the other hand probabilistic tabu search would converge if run for a large amount of time.

In the section on tabu search applications (Section **??**) we illustrated how tabu search can be engineered to solve several hard combinatorial optimization problems. Two such applications have been discussed in detail.

Tabu search has been able to find optimal solutions for many relatively small problems instances in reasonably small time. The runtime can grow to unacceptable proportion for large problem size. Several parallelization strategies have been proposed for tabu search All of them resulted in significant speed up (Section **??**).

Other important, recent, and often neglected issues were discussed in Section **??**. Target analysis presented in Section **??** helps in designing suitable evaluators useful when applying diversification strategies. Neighborhood has a different meaning in tabu search than in other search methods. Often, the term neighboring solutions refers to a solution that is obtained by means of a small perturbation to the current solution. However, in tabu search, when intensification and diversification are applied using intermediate-term and long-term

memory processes, $\aleph(\mathbf{H}, S)$ may contain solutions not in $\aleph(S)$. For example, these may include high quality local optima (elite solutions) encountered at various points during the search process.

A common phenomenon in most combinatorial optimization problems is that the amount of computational effort needed to generate all the available moves grows faster than linearly with increase in the problem dimension. Complete examination of all alternatives then becomes computationally expensive. In such cases, the number of solutions examined can be restricted using what is known as 'candidate list strategies'. Various proposed candidate list strategies that help increase the efficiency of search were discussed (Section **??**). Using such strategies, search can also be restricted to certain regions, thereby causing intensification.

Strategic oscillation is a critical component in some tabu search applications [KGA93]. The general concept (Section **??**) is one of varying the weights applied to different parts of the problem when evaluating moves [Glo89, Glo77]. In the study presented in [KGA93], the importance of *feasibility* during the search procedure is dynamically varied. In tabu search, sometimes infeasible neighboring solutions are also considered. By allowing feasible as well as *infeasible* solutions to occur, the search is able to traverse more of the solution space and locate better solutions in the process.

In order to generate new starting solutions, path relinking is employed (see Section **??**). In path relinking, some elite solutions are selected, one of them serves as an initiating solution. Then, smallest number of moves are made that take the initiating solution to the remaining solutions (guiding solutions). The intermediate points (or solutions) can be used as new starting solutions. New elite solutions may also be found in this process, since the process is similar to that of combining good characteristics of various solutions.

---

11 Conclusion

Simulated annealing, like all other iterative techniques, is very greedy with respect to run time. The acceleration of simulated annealing has been an extensive area of research since the introduction of the algorithm. Among the widely researched acceleration techniques is parallelization. The various parallelization strategies of simulated annealing are also discussed in this chapter (Section **??**).

All five iterative algorithms are very greedy with respect to execution time no matter how well tuned the parameters are. The proliferation of a large number of parallel computers has forced extensive research on the parallelization of these algorithms. For each technique, a section is dedicated to this issue of parallelization. A bibliography is provided at the end of each chapter,

In Section xx we provide a comparative analysis of the five algorithms such as similarities, differences, solution qualities, and look into hybridization aspects.

We also provide a brief introduction to fuzzy logic and neural networks, and show how fuzzy logic can help ease the formulation of multi-criteria optimization problems.

Convergence related issues are discussed in Section **??**. In Section **??** we discuss some engineering applications, with case studies and examples, that further illustrate the implementation aspects of this powerful iterative technique. Parallelization related issues are discussed in Section **??**. Other important and neglected issues such as target analysis, candidate list strategies, strategic oscillation, path relinking, etc., are discussed in Section **??**.

Finally, we acknowledge the support provided by King Fahd University of Petroleum and Minerals under Project Code # COE/xxxxx/187.

References

[AK89] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, 1989.

[AL85] E. H. L. Aarts and P. J. N. Van Laarhoven. Statistical cooling: A general approach to combinatorial optimization problem. *Philips Journal of Research*, 40(4):193–226, January 1985.

[Čer85] V. Černy. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Application*, 45(1):41–51, January 1985.

[Dav91] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, NY, 1991.

[DP91] T. E. Davis and J. C. Principe. A Simulated Annealing Like Convergence Theory for the Simple Genetic Algorithm. *Proceedings of the 4th International Conference on Genetic Algorithm*, pages 174–181, 1991.

[DP93] T. E. Davis and J. C. Principe. A Markov Chain Framework for the Simple Genetic Algorithm. *Proceedings of the 4th International Conference on Genetic Algorithm*, 13:269–288, 1993.

[DV93] F. Dammeyer and Stefan Voβ. Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*, 41:31–46, 1993.

[EAH90] A. H. Eiben, E. H. L. Aarts, and K. M. Van Hee. Global convergence of genetic algorithms: A markov chain analysis. In H. P. Schwefel and Männer, editors, *In Paralle Problem Solving from Nature*, pages 4–12. Berlin: Springer-Verlag, 1990.

[EP93] C. Ersoy and S. S. Panwar. Topological design of interconnected LAN/MAN networks. *IEEE Journal on Selected Areas in Communications*, 11(8):1172–1182, 1993.

[Fog95] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence* . IEEE Press, 1995.

[GL95] F. Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill Book Co., Europe, 1995.

[GL97] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, MA, 1997.

[Glo77] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[Glo89] F. Glover. Tabu search- Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[Glo90a] F. Glover. Artificial intelligence, heuristic frameworks and tabu search. *Managerial and Decision Economics*, 11:365–375, 1990.

[Glo90b] F. Glover. Tabu search- Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[Glo90c] F. Glover. Tabu search: A tutorial. *Technical Report, University of Colorado, Boulder*, February 1990.

[Glo95] F. Glover. Tabu search fundamentals and uses. *Technical Report, Graduate School of Business Administration, University of Colorado at Boulder*, June 1995.

[Glo96] F. Glover. Tabu search and adaptive memory programming – advances, applications and challenges. *Technical Report, College of Business, University of Colorado at Boulder*, 1996.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., 1989.

[GS87] D. E. Goldberg and P. Segrest. Finite markov chain analysis of genetic algorithms. *Genetic Algorithms and Their Applications: Proceedings of 2nd International Conference on GAs*, pages 1–8, 1987.

[GTdW93] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[Han86] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. *Congress on Numerical Methods in Combinatorial Optimization*, 1986.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, Michigan, 1975.

[IE94] O. Icmeil and S. Selcuk Erenguc. A tabu search procedure for the resource constrained project scheduling problem with discounted cash flows. *Computers & Operations Research*, 21(8):841–853, 1994.

[KCGV83] S. Kirkpatrick, Jr. C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):498–516, May 1983.

[KGA93] J. P. Kelly, B. L. Golden, and A. A. Assad. Large-scale controlled rounding using tabu search with strategic oscillation. *Annals of Operations Research*, 41:69–84, 1993.

[KLG94] J. P. Kelly, M. Laguna, and F. Glover. A study of diversification strategies for the quadratic assignment problem. *Computers Ops Research*, 21(8):885–893, 1994.

[LG93] M. Laguna and F. Glover. Bandwidth packing; a tabu search approach. *Management Science*, 39(4):492–500, 1993.

[M+53] N. Metropolis et al. Equation of state calculations by fast computing machines. *Journal of Chem. Physics*, 21:1087–1092, 1953.

[Mah93] S. W. Mahfoud. Finite markov chain models of an alternative selection strategy for the genetic algorithm. *Complex systems*, 7:155–170, 1993.

[MGPO89] M. Malek, M. Guruswamy, M. Pandya, and H. Owens. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, 21:59–84, 1989.

[NSS89] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. *International Journal of Computer-Aided VLSI Design*, 1(1):1–23, 1989.

[NV93] A. E. Nix and M. D. Vose. Modeling genetic algorithms with markov chains. *Annals of Mathematics and Artificial Intelligence*, pages 79–88, 1993.

[OvG89] R.H.J.M. Otten and L.P.P.P. van Ginneken. *The Annealing Algorithm*. Kluwer Academic Publishers, MA, 1989.

[Rud94] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5:1:96–101, 1994.

[Rya89] J. Ryan, editor. *Heuristics for Combinatorial Optimization*. June 1989.

[SM93] Minghe Sun and P. G. McKeown. Tabu search applied to the general fixed charge problem. *Annals of Operations Research*, 41:405–420, 1993.

[SSV86] C. Sechen and A. L. Sangiovanni-Vincentelli. Timberwolf3.2: A new standard cell placement and global routing package. *Proceedings of 23rd Design Automation Conference*, pages 432–439, 1986.