Optimazation

Analyse and Implements of Local Search & Metaheuristic Methodes

EL MAJJODI Abdeljalil
ALLOUCHE Salaheddine

MESTARI Mohammed

Distribution System and Artificial Intelligence Master's
Department of Mathematics and Informatics
University of Hassan 2 Casablanca

January 17, 2023

Abstract

Optimization is the process of finding the best solution among a set of possibilities, by adjusting the parameters and variables in a given problem. It is a central concept in many fields, including mathematics, engineering, and computer science. Optimization problems can take many forms, such as finding the minimum or maximum value of a mathematical application (e.g. function) or finding a solution that satisfies certain constraints. Many different techniques can be used to solve optimization problems, such as gradient descent, Newton's method, and genetic algorithms. These methods have varying levels of applicability and performance depending on the problem at hand.

Contents

List of Figures

Chapter 1

Introduction

In this study, we will look at different ways to find the best solution by adjusting the settings of the problem. The methods that used in this study will be implemented using Python Programmation language. Those are: descent, simulated annealing, guided local search, Tabu search, and genetic algorithms. In addition we will show a comparison of those methods in terms of differences, pros , and cons of each method

1.1 Motivation

Recently, the field of artificial intelligence is progressing rapidly and achieving great results thanks in part to the use of advanced optimization algorithms. These methods allow us to find the best solutions to complex problems and are crucial in many aspects of AI, such as training neural networks. While we may not have unlocked the full potential of these techniques yet, the results we are seeing are truly impressive. This serves as a motivation to continue researching and developing new optimization methods, in order to push the boundaries of what is possible with AI.

1.2 Objectives

Regarding our objectives in this study which just to analyze and implement each optimization algorithm and know how it works and when we can use it.

Chapter 2

Descent Search

2.1 What is descent search

The descent method is an optimization algorithm that belongs to the family of local search, it is an iterative method that starts with a random solution to a problem, then tries to find the better solution, and repeats this operation until no further improvement can be found.
Descent methods can get stuck in a local minimum, which is a point that is a minimum within a certain region, but not the global minimum (the overall lowest point) of the function. If the initial guess is not close enough to the global minimum, the algorithm may converge to a local minimum instead.

2.2 Implementation

2.2.1 General schema of a descent algorithm

Figure 2.1 summarizes the pseudo steps of the descent algorithm in order to find a better solution.

2.2.2 Implementation with Python

```
1    def descent(x,f,maxIter=100):
2      x1=x[0]
3      i=0
4      while i<maxIter:
5        x2=somePoint(x)
6        if  f(x1)>f(x2):
7          x1=x2
8          plot2(x1,x,i) #function plot the new minimum
9        i+=1
```

Listing 2.1: Descent Algorithm

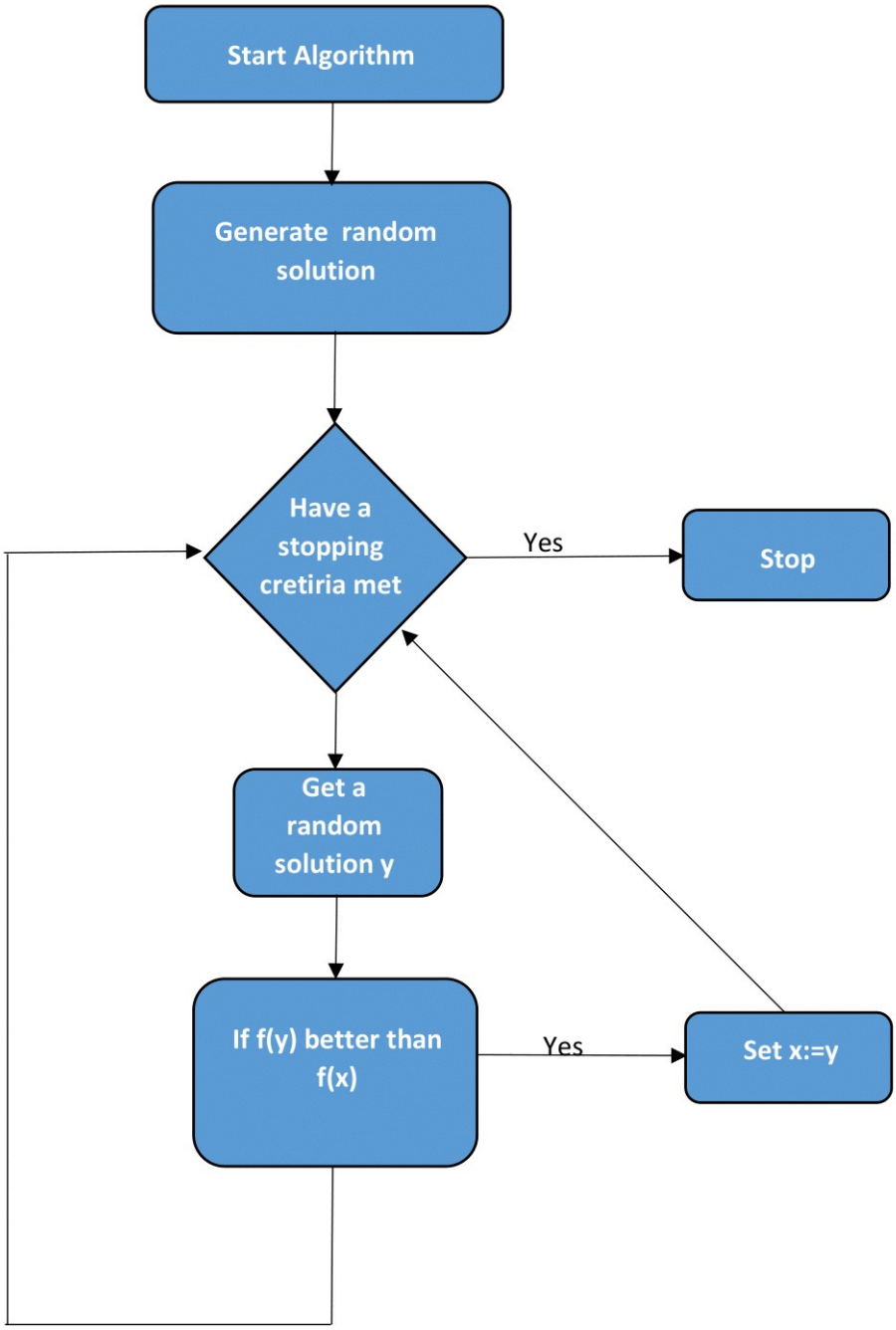

Figure 2.1: Descent algorithm Schema

2.2.3 Test

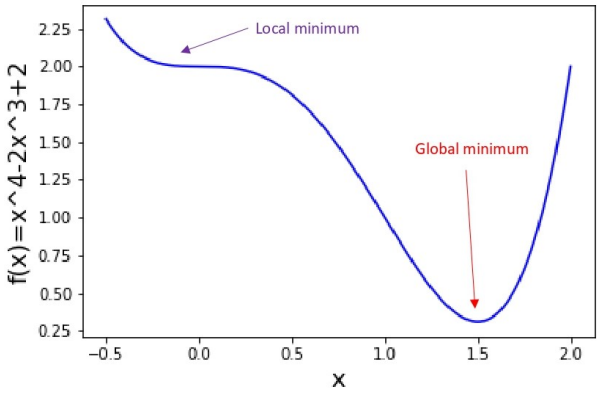

Figure 2.2: Test function

Let us test the method with a function $f(x) = x^4 - 2x^3 + 2$ and 100 and 100 iterations as a condition of stopping This function has a local minimum and global maximum and we try to get one of them with the descent method.

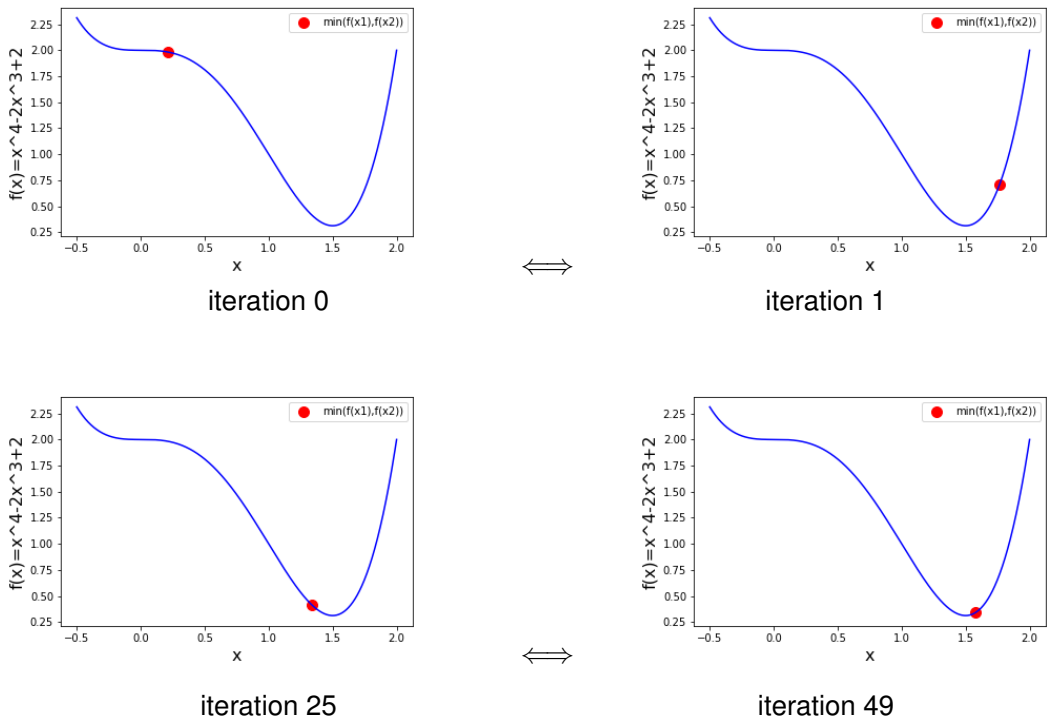

Figure 2.3: Test iterations of Descent

2.3 Analyse

Like all methods of optimization, the descent method also has limits; this method as mentioned can get stuck in a local minimum, which is not a global minimum of the function, the method can converge slowly if the function is highly non-linear or has a

large number of local minimal and maximal, this method not guaranteed to converge to a global minimum or even a local minimum.
The time complexity of this algorithm depends on the specific implementation, but in general, it can be considered a kind of greedy algorithm. It has a time complexity of O(n) where n is the number of iterations or steps required to reach a local optimum, assuming that each step takes constant time. However, in practice, the number of steps required to reach a local optimum can be much larger than the number of iterations, and it can also depend on the initial point.
In addition to time complexity, the algorithm also has a space complexity of O(1), as it only needs to store the current point, and the value of the function at that point, and it does not require any additional memory to store other points or information.
It is worth noting that, the actual performance of this algorithm depends heavily on the structure of the problem, and the quality of the heuristics used in the algorithm. In some cases, the algorithm may be very fast and efficient, while in other cases it may be very slow and inefficient.

Chapter 3

Variable Neighborhood Search

3.1 What is Variable Neighborhood Search

Variable Neighborhood Search (VNS) is a metaheuristic optimization algorithm that is used to find the global optimum of a function. It is particularly useful for solving problems that have many local optima and are difficult to solve using traditional optimization methods.
The basic idea behind VNS is to explore the solution space by moving from one neighborhood to another, where a neighborhood is defined as a set of solutions that can be reached from the current solution by making a small change. The algorithm starts with an arbitrary solution, and then repeatedly perturbs the solution by moving to a different neighborhood. The new solution is then accepted if it is better than the current solution, otherwise, the algorithm stays in the current solution. The process is repeated until a global optimum is found or a stopping criterion is met.
The key feature of VNS is the use of different neighborhoods, which allows the algorithm to escape from local optima and explore a wider range of solutions. The algorithm has the flexibility to choose different neighborhoods at different steps of the search, hence the name variable neighborhood search.

3.1.1 Implementation

3.1.2 General schema of VNS

this schema 3.1 gives a general idea of how the algorithm work.

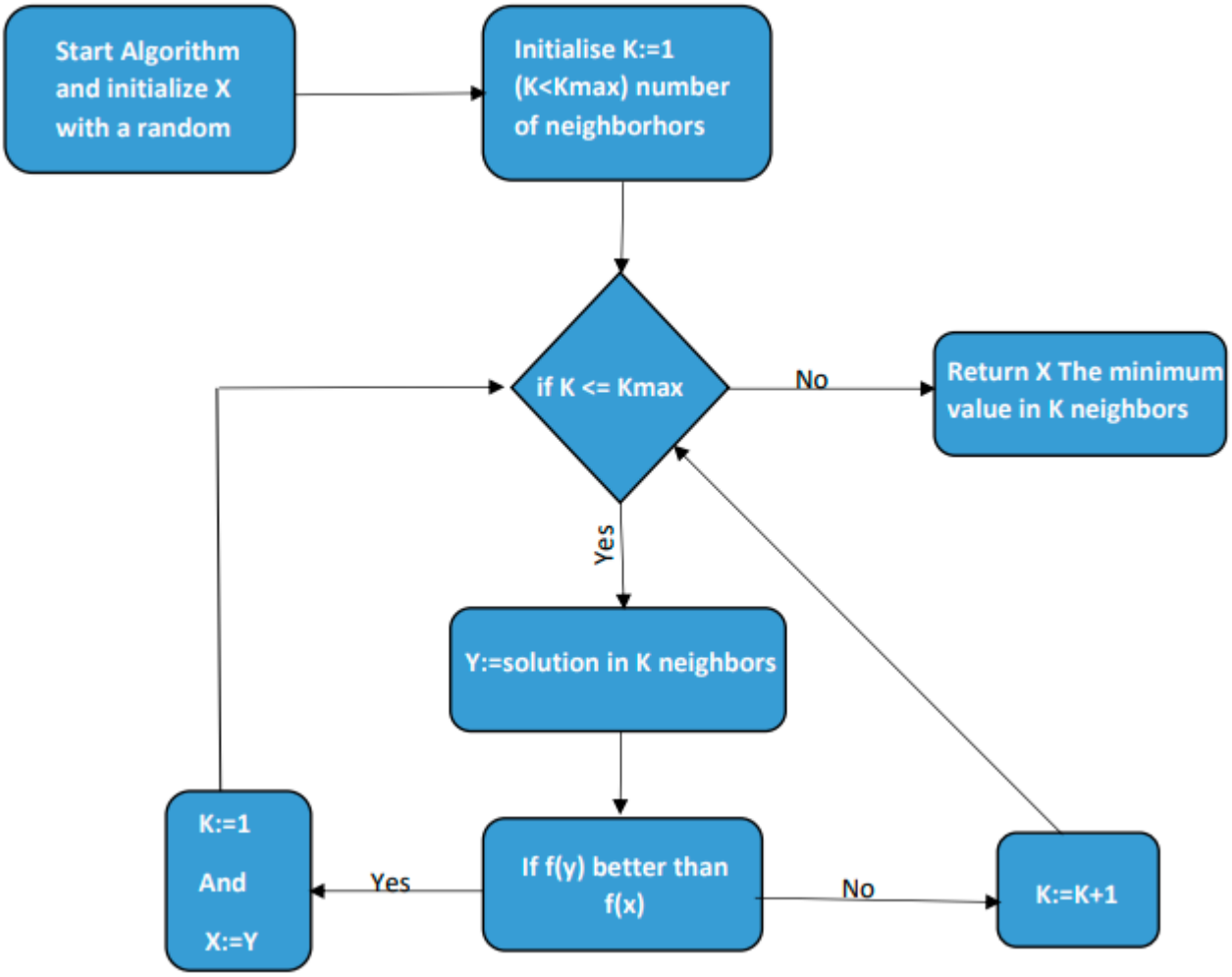

Figure 3.1: VNS algorithm schema

3.1.3 Implementation with Python

```
def VNS(sol,f,Kmax):
    x=np.random.choice(sol) #initialize X with a random
    k=1
    plot(x,sol)
    while k<=Kmax:
        y=SolKNeighbour(x,k); #get a solution in K neighbors and put it
    in y
        if f(x)>f(y):
          k=1
          x=y
          plot(x,sol) #for plot the new minimum
        else:
          k+=1
    return x
```

Listing 3.1: VNS Algorithm

3.1.4 Test

we will test this algorithm with the same function $f(x) = x^4 - 2x^3 + 2$ with a maximum of neighbors $Kmax := 80$ and will try to analyze the result.

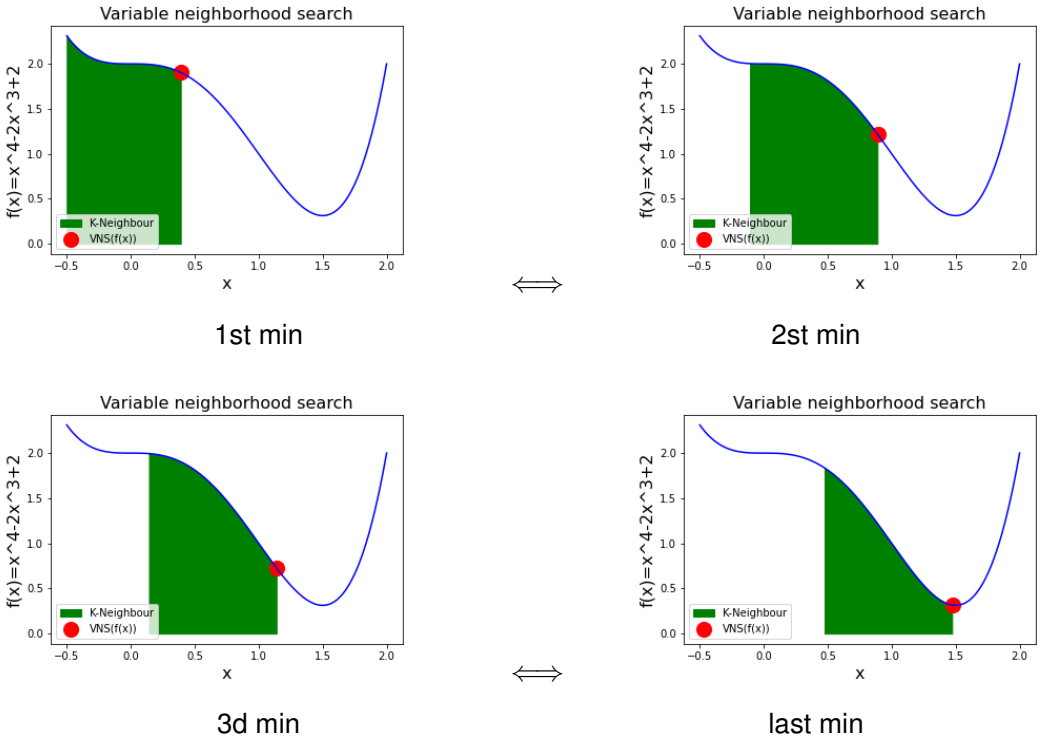

Figure 3.2: Test iterations of VNS in each of the K-neighbors

3.2 Analyse

Variable neighborhood search can reach the global minimum but that's not enough to say it could reach it in complex problems, this algorithm though is powerful but still have a lot of problems as the time complexity can be high, especially for problems with a large number of neighborhoods or large neighborhood sizes. this can make the algorithm impractical for large-scale or real-time applications, also VNS is a metaheuristic algorithm, which means that it does not guarantee finding the global optimum. the quality of the solution found by VNS depends on the quality of the heuristics used in the algorithm, the neighborhoods defined, and the stopping criterion.
Like other optimization algorithms, VNS is also sensitive to the initial solution, which means that it may converge to different solutions depending on the starting point.

Chapter 4

Simulated Annealing

4.1 What is Simulated Annealing

Simulated annealing (Cernyt', 1985; Kirkpatrick et al., 1983) is inspired by an analogy between the physical annealing of solids (crystals) and combinatorial optimization problems. In the physical annealing process, a solid is first melted and then cooled very slowly, spending a long time at low temperatures, to obtain a perfect lattice structure corresponding to a minimum energy state. SA transfers this process to local search algorithms for combinatorial optimization problems. It does so by associating the set of solutions to the problem attacked with the states of the physical system, the objective function with the physical energy of the solid, and the optimal solutions with the minimum energy states.
Simulated annealing randomly chooses a solution x' in the neighborhood of the current solution x. If x' is better than x, then x' is accepted and becomes the current solution, whereupon the process repeats. If x' is no better than x, x' is nonetheless accepted with a certain probability p.

$$p = \frac{1}{1 + \exp \frac{x - x'}{Temperature}} \tag{4.1}$$

If x' is not accepted, another solution x' is chosen randomly from the neighborhood of x.

4.2 Implementation

4.2.1 Parameters

To implement an SA algorithm, the following hyperparameters have to be specified:

Parameter	Sinification
$X0$	Initial solution
$T0$	The temperature
M	The number of iterations to be performed at each temperature
N	How many times do you want to search your neighborhood
$Alpha$	By how much do you want to decrease the temp
K	Helps reduce the step-size in the case if we have from solution to a solution much bigger

4.2.2 General schema of Simulated Annealing

The schema4.1 of this algorithm explains all sides of the algorithm

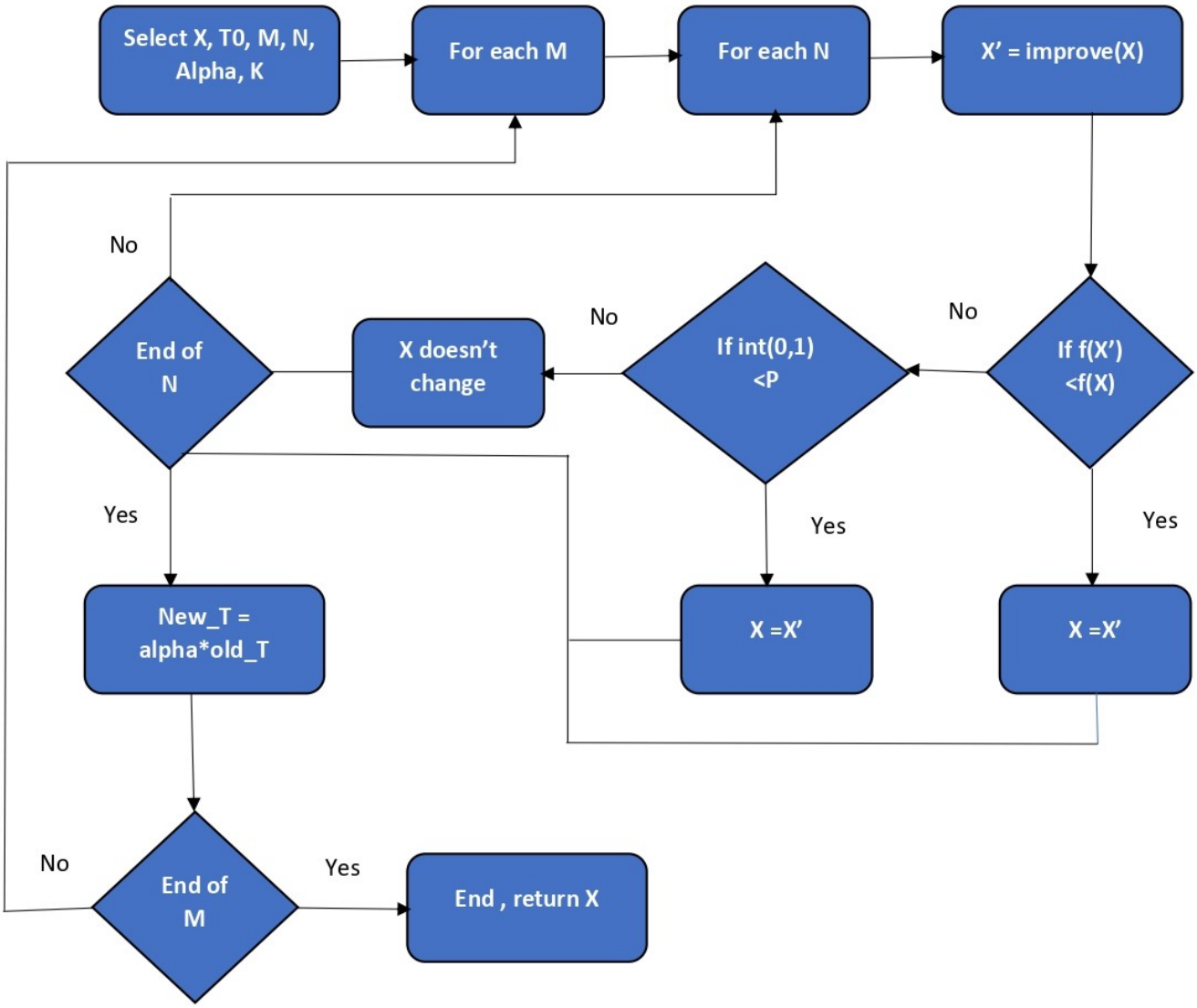

Figure 4.1: Simulated Annealing schema

4.2.3 HIMMELBLAU function

We're going to apply this algorithm to the HIMMELBLAU function 4.2, it is a multi-modal function, used to test the performance of optimization algorithms. Were going to try finding the x and y for f equal to 0.
The function is defined:

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{4.2}$$

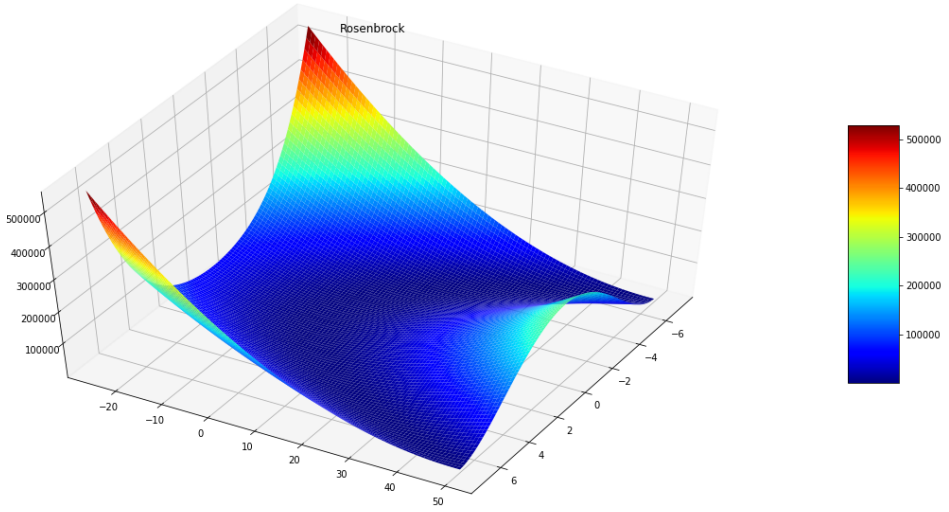

Figure 4.2: HIMMELBLAU function

4.2.4 Implementation with Python

```
def Simulated_Annealing(*,x=2,y=1,T0=1000,M=300,N=15,alpha=0.85,k=0.1)
    :

    start_time = time.time() # start time (timing purposes)

    temp = []      # to plot the temprature
    obj_val = []   # to plot the obj val reached at the end of each m (
    small M)

    for i in range(M): # how many times to decrease the temp.
        for j in range(N): # for each m, how many neighborhood searche

            x_temporary,y_temporary=fc.improve(x,y,k)

            x,y,f=fc.update(x,y,x_temporary,y_temporary,T0)

        temp.append(T0)
```

```
16          obj_val.append(f)
17
18          T0 = alpha*T0
19
20      print("Execution Time in Seconds:",time.time() - start_time) # exec
    . time
21
22      return (x,y,obj_val,temp)
```

Listing 4.1: SA Algorithm

4.2.5 Test:

We try to get the value of x and y when the value of the function4.2 is equal to 0.

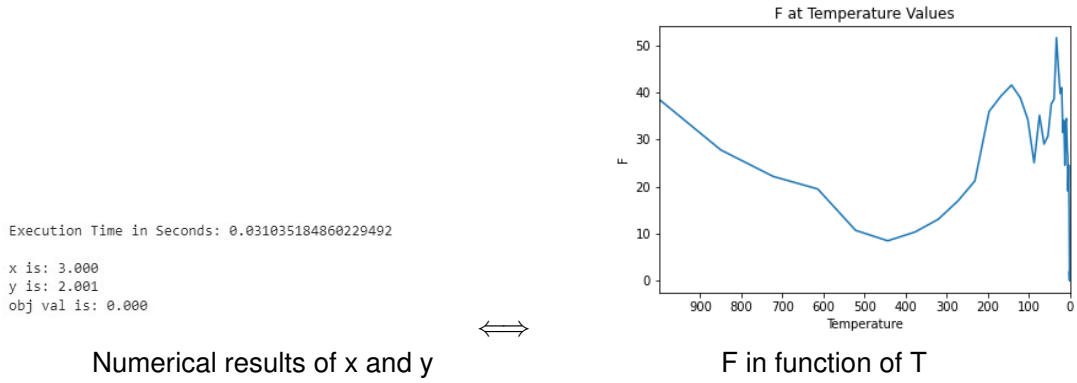

Figure 4.3: Result of SA

4.3 Analyse

There are three important differences between SA and local search:

- There is a difference in how the procedures halt. SA is executed until some external termination condition is satisfied as opposed to the requirement of local search to find an improvement.
- The function "improve" doesnt have to return a better point from the neighborhood of x'. It just returns an accepted solution from the neighborhood of X, where the acceptance is based on the current temperature T.
- In SA, the parameter T is updated periodically, and the value of this parameter influences the outcome of the procedure improve? This feature doesnt appear in local searches. Note also that the above sketch of SA is quite simplified so as to match the simplifications we made for the local search. For example, we omitted the initialization process and the frequency of changing the temperature parameter T.

Again, the main point here is to underline similarities and differences.

Chapter 5

Tabu Seach

5.1 What is Tabu Search

Tabu search (Glover, 1989, 1990; Glover and Laguna, 1997) relies on the systematic use of memory to guide the search process. It is common to distinguish between short-term memory, which restricts the neighborhood N(s) of the current solution s to a subset N'(s), and long-term memory, which may extend N(S) through the inclusion of additional solutions (Glover and Laguna, 1997).
TS uses a local search that, at every step, makes the best possible move from s to a neighbor solution s' even if the new solution is worse than the current one; in this latter case, the move that least worsens the objective function is chosen. To prevent the local search from immediately returning to a previously visited solution and, more generally, to avoid cycling, TS can explicitly memorize recently visited solutions and forbid moving back to them. More commonly, TS forbids reversing the effect of recently applied moves by declaring tabu that solution attributes that change in the local search. The tabu status of solution attributes is then maintained for a number tt of iterations; the parameter it is called the tabu tenure or the tabu list length. Unfortunately, this may forbid moves toward attractive, unvisited solutions. To avoid such an undesirable situation, an aspiration criterion is used to override the tabu status of certain moves. Most commonly, the aspiration criterion drops the tabu status of moves leading to a better solution than the best solution visited so far.
To increase the efficiency of simple TS, long-term memory strategies can be used to intensify or diversify the search.

- Intensification strategies are intended to explore more carefully promising regions of the search space either by recovering elite solutions (the best solutions obtained so far) or attributes of these solutions.

- Diversification refers to the exploration of new search space regions through the introduction of new attribute combinations.

Many long-term memory strategies in the context of TS are based on the memorization of the frequency of solution attributes. For a detailed discussion of techniques exploiting long-term memory, see Glover & Laguna (1997).

5.2 Implementation

5.2.1 Parameters

To implement a TS algorithm, the following hyperparameters have to be specified:

Parameter	Sinification
$X0$	Initial solution
M	The number of iterations to be performed
N	How many times do you want to search your neighborhood
L	Length of tabu list
Segma	The move operator

5.2.2 General schema of Tabu Search

the schema5.1 of the tabu algorithm is very important to understand it.

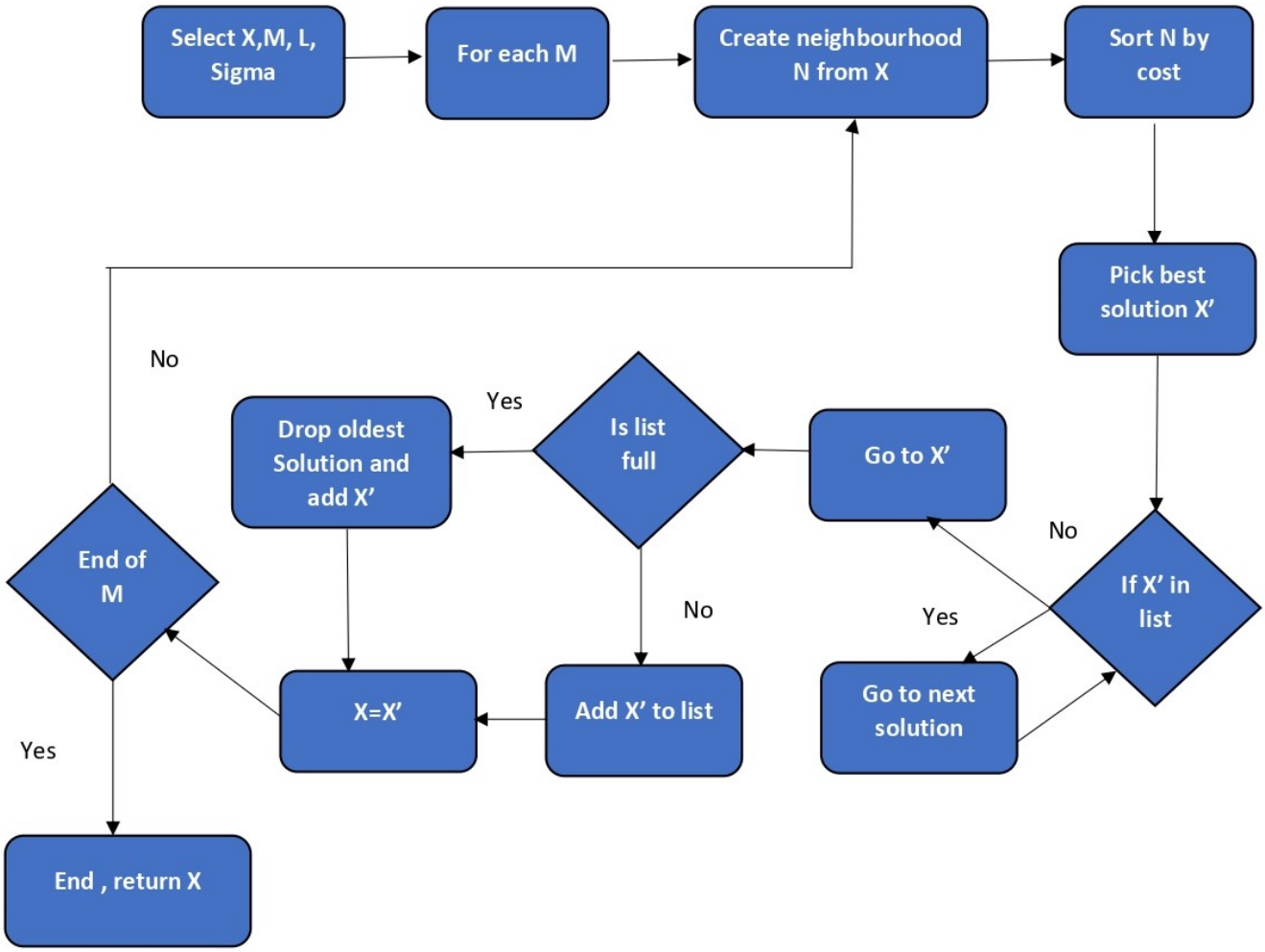

Figure 5.1: Tabu Search schema

5.2.3 Quadratic Assignment Problem (QAP)

We're going to apply this algorithm to the Quadratic Assignment Problem (QAP), which is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. The problem models the following real-life problem (the facilities' location problems):

- There are a set of n (8 for this example) facilities and a set of n locations. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (the number of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows.

5.2.4 Implementation with Python

```
def Tabu(distance,flow,x,*,M=60,L=10):
    start_time = time.time() # start time (timing purposes)
    Tabu_List = np.empty((0,len(x)+1))
    One_Final_Guy_Final = []
    Save_Solutions_Here = np.empty((0,len(x)+1))
    Iterations = 1
    for i in range(M):
        All_N_for_i = fc.creat_neighbours(x)
        OF_Values_all_N=fc.neighbours_cost(x,distance,flow,All_N_for_i)

        # Ordered OF of neighborhood, sorted by OF value
        OF_Values_all_N_Ordered = np.array(sorted(OF_Values_all_N,key=
    lambda y: y[0]))

        # Check if solution is already in Tabu list, if yes, choose the
     next one
        Current_Sol,Tabu_List=fc.best_solution(OF_Values_all_N_Ordered,
    Tabu_List,L)

        Save_Solutions_Here = np.vstack((Current_Sol,
    Save_Solutions_Here)) # Save solutions, which is the best in each
    run

        # In order to "kick-start" the search when stuck in a local
    optimum, for diversification
        L,x,Iterations=fc.dynamic_list(x,Current_Sol,Iterations,L)

     One_Final_Guy_Final = fc.final_soluction(Save_Solutions_Here)[np.
    newaxis]

    print("Execution Time in Seconds:",time.time() - start_time)
    return One_Final_Guy_Final[0]
```

Listing 5.1: Tabu Search Algorithm

5.2.5 Test

As we referred previously we will be trying to solve the QA Problem5.2.3 with Tabu algorithm.

```
Execution Time in Seconds: 1.0990087985992432

DYNAMIC TABU LIST

Initial Solution: ['D', 'A', 'C', 'B', 'G', 'E', 'F', 'H']
Initial Cost: 310

Min in all Iterations: ['C' 'D' 'H' 'B' 'A' 'E' 'F' 'G']
The Lowest Cost is: 214
```

Figure 5.2: QAP solution with Tabu

5.3 Analyse

Tabu search is almost identical to simulated annealing with respect to the structure of the algorithm. As in SA, the function "improve" returns an accepted solution from the neighborhood of x, which need not be better than x, but the acceptance is based on the history of the search H. Everything else is the same (at least, from a high-level perspective).

Chapter 6

Genetic Algorithm

6.1 What is Genetic Algorithm

Genetic algorithm (Holland, 1975; Goldberg, 1989) is population-based algorithms that use operators inspired by population genetics to explore the search space (the most typical genetic operators are reproduction, mutation, and recombination), it mimics evolution by natural selection. It begins with a set of solutions (a population) and allows some pairs of solutions, perhaps the best ones, to mate. A crossover operation produces an offspring that inherits some characteristics of the parent solutions. At this point, the fewer desirable solutions are eliminated from the population so that only the fittest survive. The process repeats for several generations, and the best solution for the resulting population is selected. Indicate how this algorithm can be viewed as examining a sequence of problem restrictions. In what way does the generation of offspring produce a relaxation of the current restriction? What is the role of the selection criterion? Why is relaxation bounding unhelpful in this algorithm? Hint: Relaxation bounding is helpful when it obviates the necessity of solving the current restriction. Think about how the current relaxation is obtained.
As in constrained biological systems, the best individuals in the population are those that have a better chance of reproducing and passing on some of their genetic heritage to the next generation. A new population, or generation, is then created by combining the genes of the parents. It is expected that some individuals of the new generation will have the best characteristics of both parents, and therefore they will be better and will be a better solution to the problem. The new group (the new generation) is then subjected to the same selection criteria and subsequently generates its own offspring. This process is repeated several times until all individuals have the same genetic heritage. Members of this latest generation, who are usually very different from their ancestors, have genetic information that corresponds to the best solution to the problem.
The basic genetic algorithm has three simple operations that are no more complicated than algebraic operations:

- Selection: It is processes or individuals copied according to the value of their objective function. We can describe the function f as a measure of profit, utility or quality that we want to maximize (minimize). If we copy individuals according to their f-value, this implies that individuals with higher values have a greater probability of contributing offspring to the next generation. This is an artificial

version of Darwin's "survival of the fittest".

- Crossover: It is the process where new individuals are formed from parents. These new individuals, the offspring, are formed by making a cross between two parents. We choose a random position k between $[1, l-1]$ where l is the length of the individual. The crossing is done by exchanging the bits from position $k+1$ to l.
- Mutation: It is a random process where a bit changes value. This process plays a secondary role in the genetic algorithm, but it is still important. The mutation ensures that no point in the search space has a zero probability of being reached.

6.2 Implementation

6.2.1 Parameters

To implement a GA algorithm, the following hyperparameters have to be specified:

Parameter	Sinification
M	The number of generations
N	Population size
Pc	Probability of crossover
Pm	Probability of Mutation
K	Number of contestants(tournament selection)
L	Chromosome length

$$Precision = \frac{b-a}{2^L - 1} \tag{6.1}$$

$$Decoding = (\sum_{i=1} bit * 2^i) * Precision + a \tag{6.2}$$

6.2.2 General schema of GA

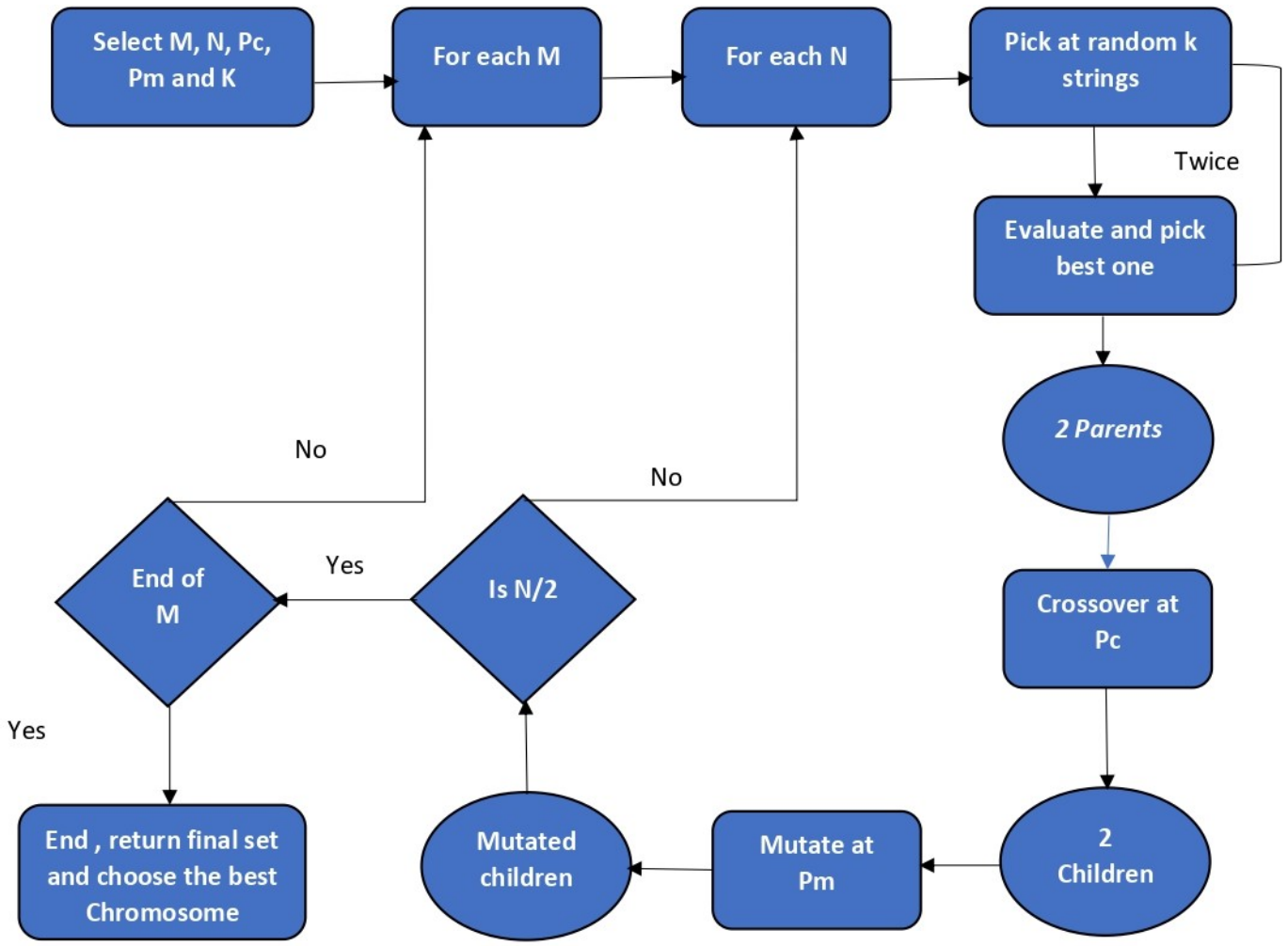

Figure 6.1: Genetic Algorithm schema

6.2.3 Implementation with Python

```
def Genetic(x_y_string,*,M=80,N=120,pc=1,pm=0.3):
    start_time = time.time() # start time (timing purposes)
    # create an empty array to store a solution from each generation
    best_of_a_generation = np.empty((0,len(x_y_string)+1))
    # so now, pool_of_solutions, has n (population) chromosomes
    pool_of_solutions = fc.initialisez(x_y_string,N)
    for i in range(M): # do it n (generation) times
        new_population,new_population_with_obj_val=fc.create_population
    (x_y_string,N,pool_of_solutions,pc,pm)
        # we replace the initial (before) population with the new one (
    current generation)
        pool_of_solutions = new_population
        # for each generation we want to find the best solution in that
     generation
        sorted_best_for_plotting = np.array(sorted(
    new_population_with_obj_val,key=lambda x:x[0]))
```

```
13          # since we sorted them from best to worst the best in that
    generation would be the first solution in the array
14          best_of_a_generation = np.vstack((best_of_a_generation,
    sorted_best_for_plotting[0]))
15
16      # for this array of last population (convergence), there is a best
    solution so we sort them from best to worst
17      sorted_last_population = np.array(sorted(
    new_population_with_obj_val,key=lambda x:x[0]))
18      sorted_best_of_a_generation = np.array(sorted(best_of_a_generation,
    key=lambda x:x[0]))
19      print("Execution Time in Seconds:",time.time() - start_time)
20      return (sorted_last_population,sorted_best_of_a_generation,
    best_of_a_generation,sorted_best_of_a_generation)
```

Listing 6.1: GA Algorithm

6.2.4 Test

for this section, we used the same function (Himmelblau Function 4.2) used in SA and we get the next results:

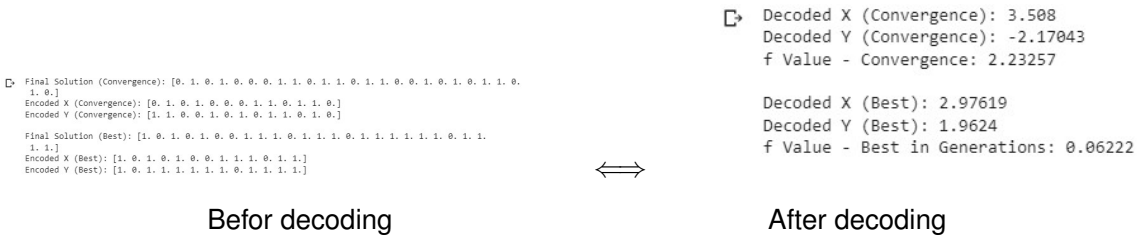

Figure 6.2: GA results

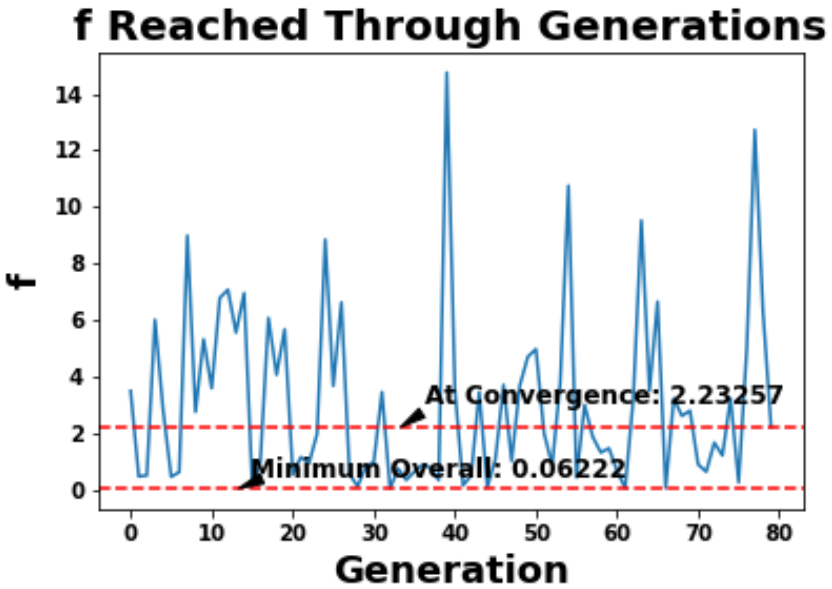

Figure 6.3: Evolution of solution

6.3 Analyse

GA typically use binary or discrete-valued variables to represent information in individuals and they favor the use of recombination, while evolution strategies and evolutionary programming often use continuous variables and put more emphasis on the mutation operator. Nevertheless, the difference between the different paradigms is becoming more and more blurred

Conclusions

A metaheuristic is a high-level problem-solving strategy that uses heuristic methods to find approximate solutions to optimization and search problems. Local search is a type of metaheuristic that focuses on finding solutions that are "locally" optimal, rather than globally optimal. In local search, an algorithm starts with an initial solution and iteratively makes small changes to improve it, until it reaches a locally optimal solution. Local search is often used in combination with other metaheuristics such as simulated annealing and genetic algorithms.