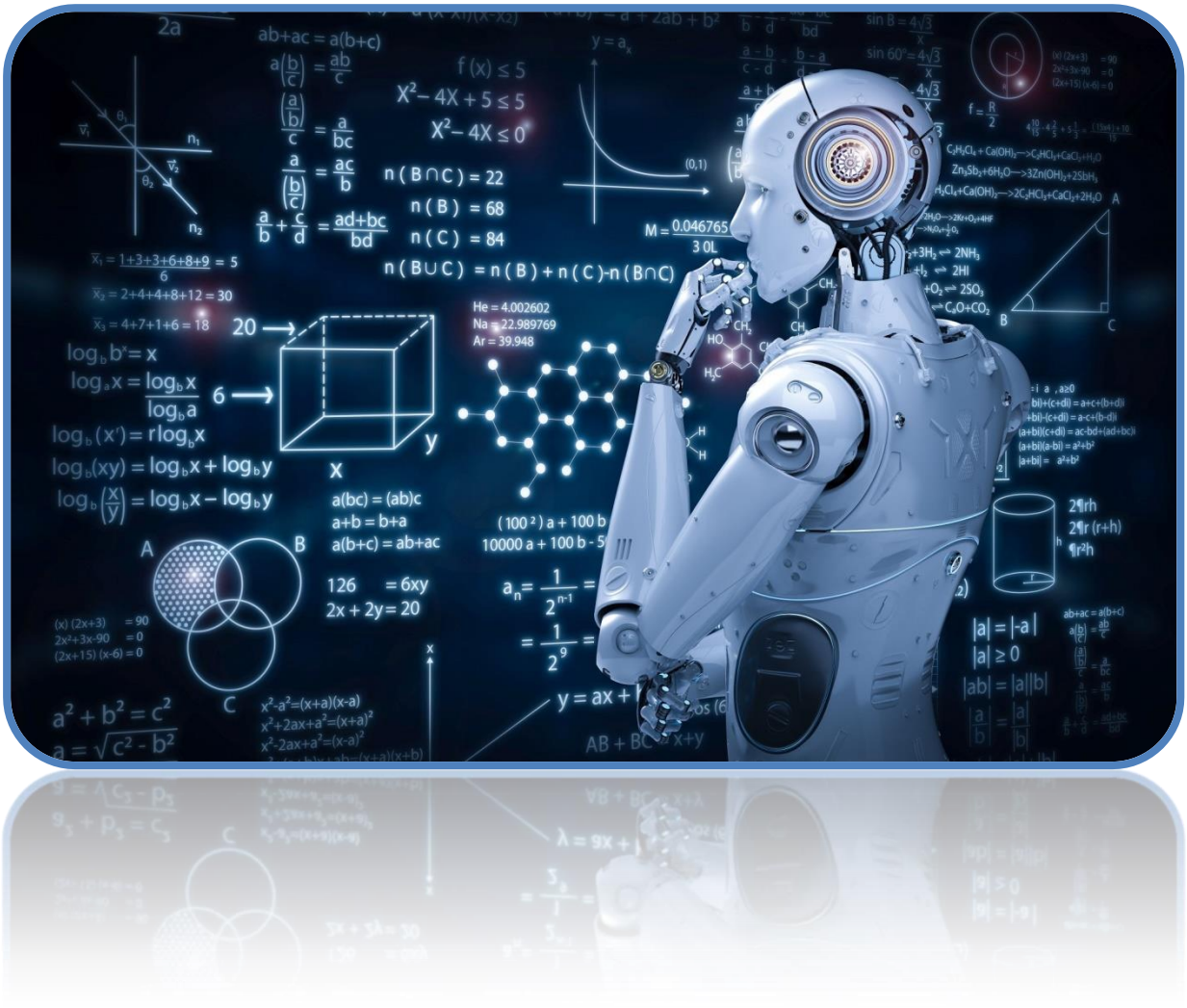
ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Master 1-SDIA

Implementation of optimization algorithms

Repport



Realized by:

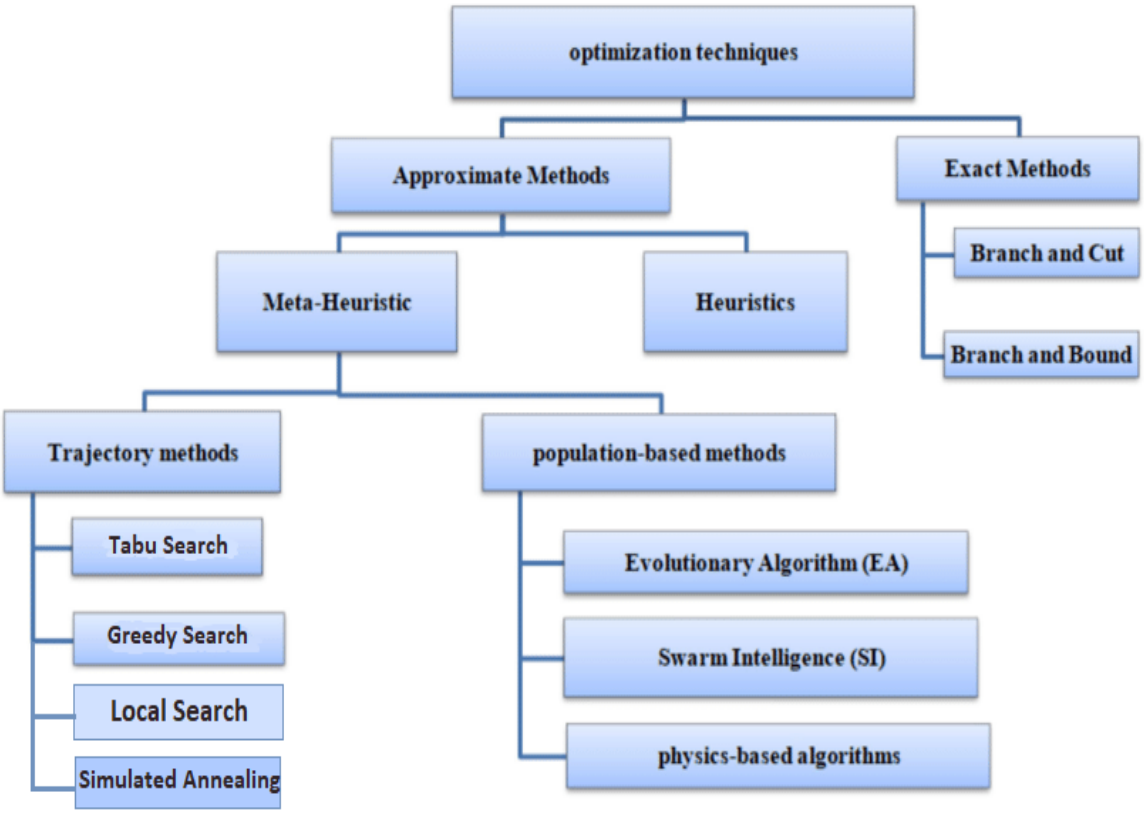
- **Abdelkarim AGOUJIL**
- **Hicham EL MOUDNI**

University Year: 2022-2023

Table of Contents:

I. General Introduction

Optimization algorithms are a class of algorithms that are used to find the best solution to a problem from among a set of possible solutions. The best solution is often characterized by the minimization or maximization of an objective function. The objective function is a mathematical expression that represents the performance of a particular solution, and the optimization algorithm searches for the set of input parameters that minimize or maximize this function. Optimization algorithms are widely used in many fields including machine learning, operations research, and engineering to find the best solution to problems such as function fitting, linear and nonlinear programming, and control system design. Some common optimization algorithm includes gradient descent, Tabu, and simulated annealing…



II. Local search algorithm (Descent)

1. Introduction

Local Search algorithm is an optimization method that improves a given solution by searching through nearby solutions in the solution space. It begins with an initial solution and repeatedly makes small changes to it in order to find a better solution. The algorithm stops when it reaches a local minimum, where no further improvements can be made in the immediate vicinity of the current solution.

2. Implementation

This function appears to perform a local search for the minimum value of a function within a given range. It does this by evaluating the function at a point **x**, and then at two neighboring points **x_left** and **x_right**. It then updates **x** and **fx** (the function value at **x**) to the point with the lowest function value, and continues this process for a maximum number of **max_iterations**. If the function value at neither of the neighboring points is lower than at **x**, the function terminates and returns the current values of **x** and **fx**.

```python
import numpy as np
import random
import math
import matplotlib.pyplot as plt

def local_search(f, x_prime, step, max_iterations):
  # un point x appartient a Omega
  x = x_prime
  fx = f(x)
  for i in range(max_iterations):
    # déterminer les positions voisines
    x_left = x - step
    x_right = x + step
    # évaluer la fonction en ces positions
    f_left = f(x_left)
    f_right = f(x_right)

    # si la fonction en x_left est plus petite que celle en x, mettre à jour x et fx
    if f_left < fx:
      x = x_left
      fx = f_left
```

```python
    # si la fonction en x_right est plus petite que celle en x, mettre à jour x et fx
    elif f_right < fx:
      x = x_right
      fx = f_right

    # sinon, la fonction a atteint un point minimum local
    else:
      break
  return x, fx
```

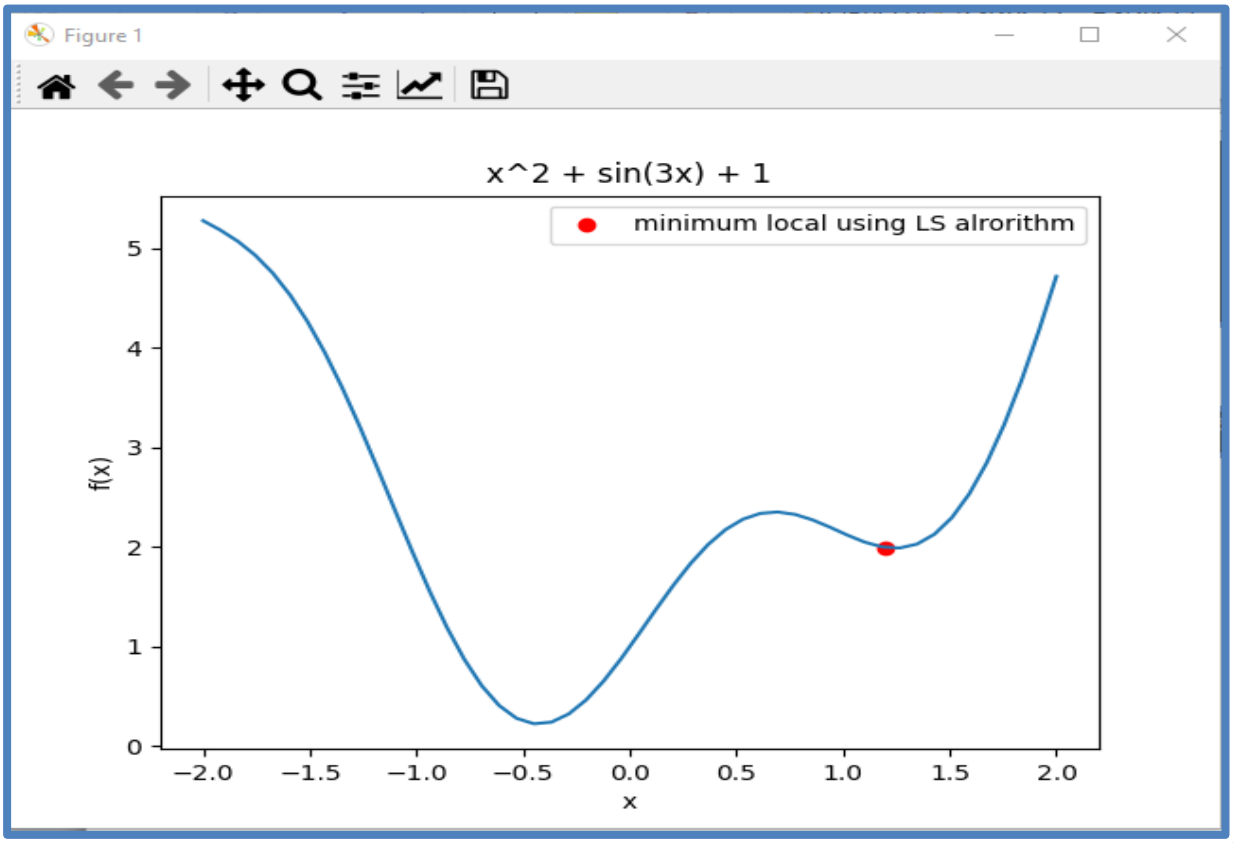
3. Testing

```
def f(x):
    return x**2 + np.sin(3*x) + 1

x_prime = 1.7
step = 0.5
max_iterations = 1000

minimum_x, minimum_fx = local_search(f, x_prime, step, max_iterations)

print("Le point minimum local est x =", minimum_x, "avec une valeur de f(x) =", minimum_fx)

x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = f(x)  # calculate y values for plotting
plt.plot(x, y)
plt.scatter(minimum_x, minimum_fx, color='red', label='minimum local using LS alrorithm')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('x^2 + sin(3x) + 1')
plt.legend()
plt.show()
```

Console:

```
Le point minimum local est x = 1.2 avec une valeur de f(x) = 1.997479556705148
```

Curve:



4. Complexity

The complexity of this algorithm is O(n), where n is the value of max_iterations. This is because the algorithm iterates through a loop that runs for a maximum of max_iterations times, and within each iteration, it performs a constant amount of operations (such as determining neighboring positions, evaluating the function in those positions, and updating the current position if a better solution is found). Therefore, the total number of operations performed is directly proportional to the value of max_iterations, and the complexity is O(n).

5. Advantages & Disadvantages

Advantages of local search :

- Simplicity: Local search algorithms are often simple to understand and implement.
- Flexibility: Local search can be applied to a wide range of optimization problems, including both continuous and discrete optimization problems.
- Efficiency: Local search algorithms can be very efficient, especially when the problem being solved has a large solution space.

disadvantages of local search :

- Convergence to local optima: Local search algorithms can get stuck in local optima, which are suboptimal solutions that are not the global optimal solution.
- Lack of any guarantees: Local search does not provide any guarantees that the solution will be global optimal.
- Requires a good starting point: Local search requires a good starting point for the algorithm, otherwise it can lead to poor results.
- Problem dependency: The performance of local search depends on the structure of the problem, some problems are harder to optimize using local search.

6. Conclusion

Local search is a powerful optimization technique that can be applied to a wide range of problems. It is simple to understand and implement, and can be very efficient. However, it has the potential to converge to local optima, which can be suboptimal solutions. Additionally, it does not provide any guarantees of global optimality, and requires a good starting point for the algorithm. The performance of local search also depends on the structure of the problem being solved.

III. Simulated Annealing algorithm

1. Introduction

Simulated Annealing is a optimization technique that is used to find the global minimum or maximum of a function. It is particularly useful for solving problems that have a large number of parameters, or when the solution space is complex and rugged. The algorithm is inspired by the annealing process used in metallurgy to harden metals, in which a material is heated to a high temperature and then cooled slowly to increase its strength and reduce defects.

2. Implementation

The basic idea behind simulated annealing is to start with a random initial solution and then iteratively modify it by making small random changes. The new solution is accepted or rejected based on a probability that depends on the difference in the value of the objective function between the old and new solutions, and the current "temperature" of the system. The temperature starts high and is gradually decreased over time, making it less likely that large changes will be accepted as the optimization proceeds. This mimics the annealing process in metallurgy, where the material is slowly cooled to reduce defects.

The algorithm was implemented by defining the objective function, the initial solution, and the parameters of the simulated annealing process.

```python
import numpy as np
import random
import math
import matplotlib.pyplot as plt

def simulated_annealing(f, x_init, T_init, alpha, T_min, max_iter):
    x_current = x_init
    T = T_init
    r = random.uniform(0, 1)
    best_x = x_current
    best_f = f(x_current)
    for i in range(max_iter):
        x_new = random.uniform(-10, 10)  # Generate new solution
        delta_f = f(x_new) - f(x_current)
        if delta_f < 0: # if new solution is better, accept it
            x_current = x_new
            if f(x_new) < best_f:
                best_f = f(x_new)
                best_x = x_new
```

```
        else:
            p = math.exp(-delta_f/T)
            if r < p: # accept worse solution with probability p
                x_current = x_new
        T = T*alpha
        if T < T_min:
            T = T_min
    return best_x, best_f
```

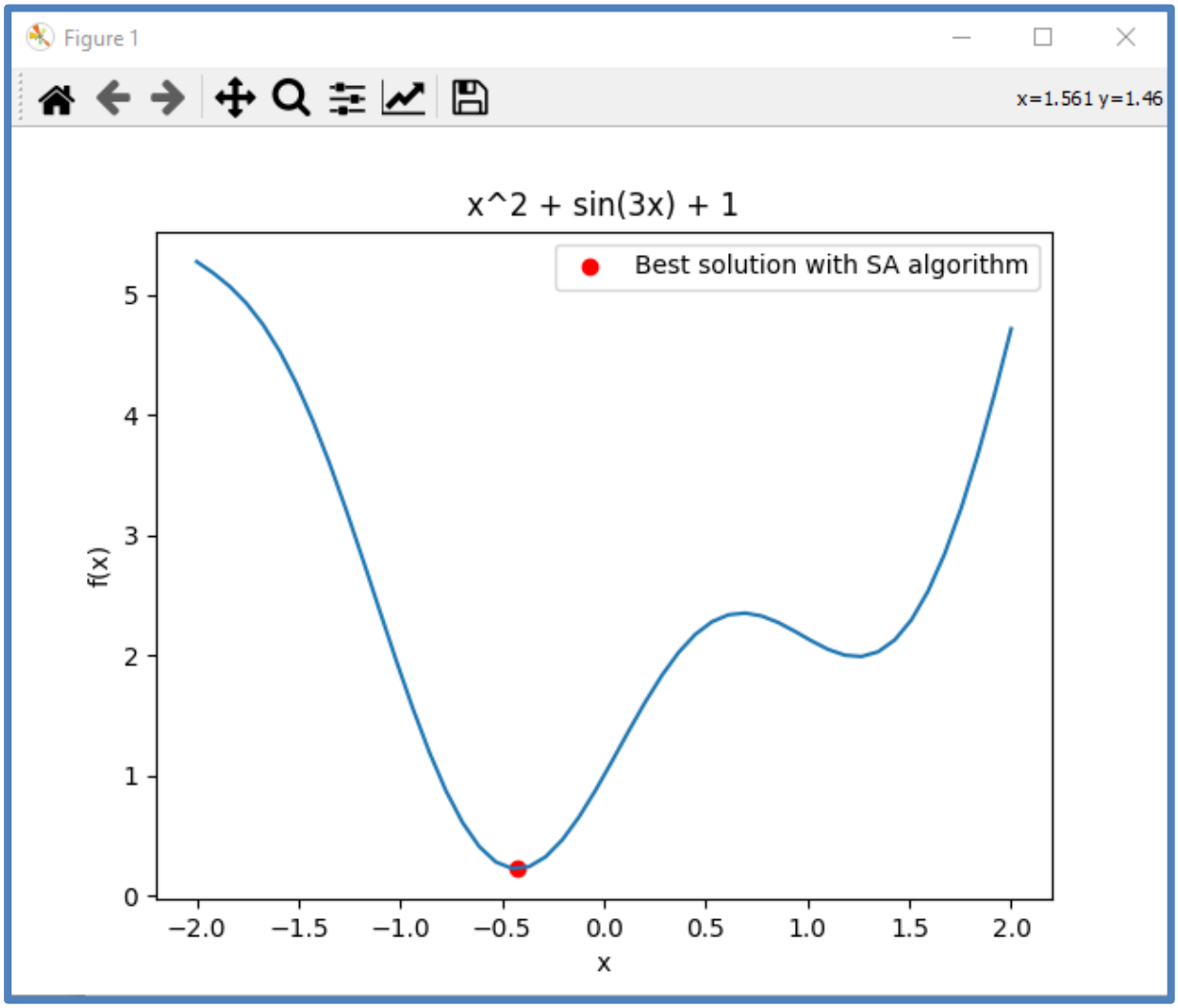
3. Testing

```
"""Test"""

# Example usage:
def f(x):
    return x**2 + np.sin(3*x) + 1

x_init = 3
T_init = 100
alpha = 0.95
T_min = 1e-6
max_iter = 10000

best_x, best_f = simulated_annealing(f, x_init, T_init, alpha, T_min, max_iter)
print(best_x, best_f)

x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = f(x)  # calculate y values for plotting

plt.plot(x, y)
plt.scatter(best_x, best_f, color='red', label='Best solution with SA algorithm')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('x^2 + sin(3x) + 1')
plt.legend()
plt.show()
```

Console:

```
best value of x :  -0.426207623967807 And best value of function f(x)=  0.22403283031483245
```

Curve:



4. Complexity

The complexity of this algorithm is O(n), where n is the value of max_iter. This is because the algorithm iterates through a loop that runs for a maximum of max_iter times, and within each iteration, it performs a constant amount of operations (such as generating a new solution, calculating delta_f, and updating the current and best solutions). Therefore, the total number of operations performed is directly proportional to the value of max_iter, and the complexity is O(n).

5. Advantages & Disadvantages

Advantages of Simulated Annealing:

- Global Optimization: Simulated Annealing is a global optimization algorithm, meaning it can find the global minimum of a function, unlike local optimization algorithms that can only find the local minimum.
- Handling Constraints: The algorithm can handle constraints on the solution space, making it useful for solving problems with complex constraints.
- Handling Multi-modal Functions: Simulated Annealing is particularly effective at finding the global minimum of multi-modal functions, which have multiple local minima.
- Handling Noise and Local Optima: The algorithm can handle noise and local optima by using a probabilistic acceptance function which allows a certain amount of exploration of the solution space.

Disadvantages of Simulated Annealing:

- Computationally Expensive: The algorithm can be computationally expensive, as it requires many iterations and random number generation, which can slow down the optimization process.
- Difficulty in Determining Parameters: Determining the proper values for the parameters (initial temperature, cooling rate, etc.) can be difficult and may require trial and error.
- Convergence to Local Minimum: The algorithm may converge to a local minimum, especially if the initial temperature is high, or if the cooling rate is too slow.
- Time-consuming: The algorithm takes a lot of time to run, especially when the function is complex or has many local minima, as it needs to search through a large solution space.

6. Conclusion

One of the key advantages of simulated annealing is that it is able to escape local minima or maxima, unlike other optimization techniques such as gradient descent, which can get stuck in these suboptimal solutions. This is because simulated annealing allows for larger moves in the solution space during the early stages of the optimization, when the temperature is high, which can lead to a more global minimum.

IV. Guided local search algorithm

1. Introduction

Guided Local Search (GLS) is a metaheuristic optimization algorithm that is a variation of Local Search (LS) algorithm. It is typically used for solving problems in combinatorial optimization, where the goal is to find the best solution from a finite set of solutions.

The main idea behind GLS is to guide the search process by incorporating a memory mechanism and intensification and diversification strategies, which help to escape from local optima and to explore the solution space more efficiently. The memory mechanism is used to keep track of the best solutions found so far, and the intensification and diversification strategies are used to focus the search around the best solutions and to explore new regions of the solution space, respectively.

The algorithm starts with an initial solution and repeatedly applies a neighborhood function to generate new solutions. The neighborhood function defines the set of solutions that can be reached from the current solution with a small number of changes. At each step, the algorithm selects the best solution from the current solution and its neighbors, and updates the current solution accordingly. The search process stops when a stopping criterion is met, such as reaching a maximum number of iterations or a satisfactory solution quality.

GLS algorithm is often used in conjunction with problem-specific heuristics or problem-specific knowledge to guide the search process in the direction of better solutions.

2. Implementation

This algorithm above defines a function called "**guided_local_search**" that takes another function as its input. The "**guided_local_search**" function is an implementation of a optimization technique called "**Guided Local Search**", which can be used to find the minimum value of a given function.

```python
import math
import random

def guided_local_search(function):
    max_iterations=1000
    neighborhood_size=10
    perturbation_size=0.1
    current_solution = random.uniform(-10, 10)
    best_solution = current_solution
    for i in range(max_iterations):
        neighborhood = [current_solution + random.uniform(-perturbation_size, perturbation_size)
                        for _ in range(neighborhood_size)]
        best_neighbor = min(neighborhood, key=function)
        if function(best_neighbor) < function(best_solution):
            best_solution = best_neighbor
        if function(best_neighbor) < function(current_solution):
            current_solution = best_neighbor
    return best_solution
```

The outermost defined function is "**function**" which takes a single variable x and returns **x^2 + sin(3x) + 1.**

The **guided_local_search** function starts by initializing a few variables: **max_iterations**, **neighborhood_size**, **perturbation_size** and **current_solution**.

max_iterations is the maximum number of iterations that the algorithm will run for before stopping.

neighborhood_size is the number of potential solutions to be generated in each iteration, **perturbation_size** is the range from which these potential solutions are generated, **current_solution** is the starting point for the algorithm and **best_solution** is the best solution found so far.

The for-loop within the **guided_local_search** function runs for **max_iterations** times. In each iteration, the neighborhood variable is assigned a list of **neighborhood_size** number of random solutions that are generated by adding a random value between **-perturbation_size** and **perturbation_size** to **the current_solution**. It uses this neighborhood as a list of candidates to find the **best_neighbor** which is the best of the neighborhood. Then it compares the **function(best_neighbor)** with **function(best_solution)**, if **function(best_neighbor)** is less than **function(best_solution)**, then it updates **best_solution**. If **function(best_neighbor)** is less than **function(current_solution),** then it updates **current_solution**.

After **max_iterations**, it returns **best_solution**.

3. Testing

first you have to import the following libraries to initialize the objective function and display the results in a curve

```python
import matplotlib.pyplot as plt
import numpy as np
```

Here are the main steps of the algorithm:

```
def main():
    def function(x):
        return x**2 + np.sin(3*x) + 1

    best_x = guided_local_search(function)
    print("Optimal solution: x = ",best_x)
    print("Optimal value: f(x) = ",function(best_x))

    x = np.linspace(-2, 2, 50)  # generate x values for plotting
    y = function(x)  # calculate y values for plotting

    plt.plot(x, y)
    plt.scatter(best_x, function(best_x), color='red', label='Best solution')
    plt.xlabel('x')
    plt.ylabel('function(x)')
    plt.title('Plot of the function f(x)')
    plt.legend()
    plt.show()
```

In the next line, the variable **best_x** is assigned the value returned by running the function **guided_local_search** with function as the input.

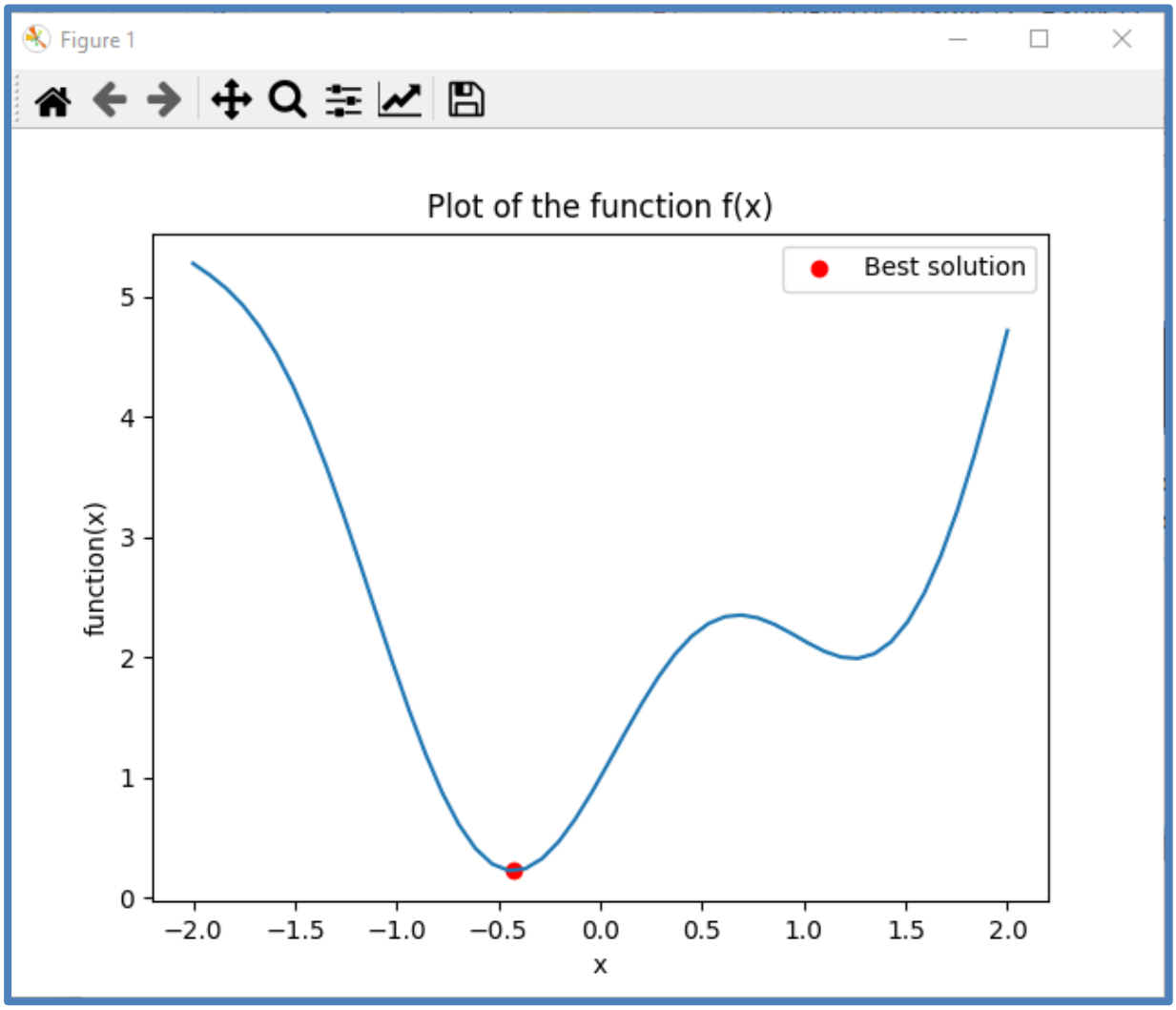
Then it prints the optimal solution (**best_x**) and the optimal value of the function (**function(best_x)**)

The next block of code is generating x and y values for plotting, it creates a np array x of 50 linearly spaced points in the interval [-2, 2] and then it assigns the result of function(x) to y.

Output:

```
Optimal solution: x =  -0.42731510707757175
Optimal value: f(x) =  0.22402640031125276
```

It also **plot** the best solution and the function that it is minimizing The key step that defines this method is the local search in a neighborhood around a current solution in order to find the best neighbor. Also, the current solution is updated only if the best neighbor solution is better than the current solution.



4. Complexity

The complexity of the guided_local_search function is O(n*m), where n is the number of iterations and m is the number of solutions evaluated in the neighborhood during each iteration.

In this specific implementation, max_iterations=1000, neighborhood_size=10, it will be O(1000*10)=O(10000) that means it grows linear with the input size.

It is important to note that the time complexity also depends on the function being optimized, as well as the initial solution, as the number of iterations required to find the global optimum can vary greatly depending on these factors.

5. Advantages & Disadvantages

Guided search is an algorithm used for searching through large spaces of possible solutions to a problem.

Advantages:

- ✓ It is able to use problem-specific knowledge to direct the search in a more efficient direction, which can reduce the number of steps required to find a solution.
- ✓ It is able to balance the trade-off between exploring different options and focusing on promising areas of the search space.

Disadvantages:

- It can be difficult to come up with an effective heuristic function to guide the search, which can limit the performance of the algorithm.
- It may not be able to find the optimal solution if the problem-specific knowledge used to guide the search is incomplete or incorrect.
- Guided search is memory intensive and not suitable for large search space.
- It can be computationally expensive, especially if the heuristic function is computationally expensive to evaluate.

It's important to note that, like most search algorithms, the effectiveness of guided search depends heavily on the specific problem and the quality of the heuristic function used to guide the search.

6. Conclusion

In conclusion, the guided search algorithm can be a useful tool for searching through large spaces of possible solutions to a problem. It can leverage problem-specific knowledge to direct the search in a more efficient direction, which can help to reduce the number of steps required to find a solution. However, the algorithm also has some disadvantages, such as being difficult to implement with a poor or incomplete heuristic function and being memory and computationally intensive. It's effectiveness will vary depending on the problem at hand and the quality of the heuristic function used.

V. Tabu search algorithm

1. Introduction

Tabu search is a metaheuristic optimization algorithm that is used to find approximate solutions to combinatorial optimization problems. The algorithm uses a memory structure called a "tabu list" to keep track of solutions that have recently been visited, in order to avoid cycling through the same solutions repeatedly. The tabu list acts as a form of "memory" for the algorithm, allowing it to make progress by moving to new solutions while avoiding solutions that it has already visited. The algorithm can also use a neighborhood function to define the set of solutions that can be reached from a given solution in one step. The algorithm's objective is to find solutions that are as good as or better than the current solution while also satisfying certain constraints.

Tabu Search is widely applied to various optimization problem, such as travelling salesman problem, n-queens problem, vehicle routing problem, job shop scheduling problem, and etc.

2. Implementation

The algorithm above defines a function "tabu_search" which takes as input an objective function, the length of the tabu list and the number of maximum iterations. The objective function is a mathematical function that takes an input and produces an output, the input and output are numbers.

```
import math
import random
import matplotlib.pyplot as plt
import numpy as np


def tabu_search(obj_func, tabu_list_length, max_iterations):
    # Initialize the best solution to a random value within the search space
    best_solution = random.uniform(-10, 10)
    best_obj_value = obj_func(best_solution)

    # Initialize the tabu list
    tabu_list = []

    for i in range(max_iterations):
        # Generate a random neighbor
        candidate_solution = random.uniform(-10, 10)
```

```
    # Check if the candidate solution is in the tabu list
    if candidate_solution in tabu_list:
        continue

    candidate_obj_value = obj_func(candidate_solution)

    # Update the best solution if the candidate solution is better
    if candidate_obj_value < best_obj_value:
        best_solution = candidate_solution
        best_obj_value = candidate_obj_value

    # Add the candidate solution to the tabu list
    tabu_list.append(candidate_solution)

    # If the tabu list is full, remove the oldest solution
    if len(tabu_list) > tabu_list_length:
        tabu_list.pop(0)

return best_solution
```

The **tabu_search** function starts by initializing the **best_solution** and **best_obj_value** with a random value within the search space (-10, 10) and the value returned by the objective function for that solution respectively. It then runs a loop for **max_iterations**, in each iteration it generates a random number between (-10, 10), and checks if it is in the **tabu_list** and if it is, it skips that iteration. If it is not in the tabu list then it checks if the **obj_value** of the **candidate_solution** is better than the **best_obj_value**, if it is, then the **candidate_solution** becomes the **best_solution**. Then it adds the **candidate_solution** to the **tabu_list**, and if the **tabu_list** is longer than **tabu_list_length**, it removes the oldest element.

3. Testing

First defines the **objective_function** as **x^2 + sin(4*x) + 1** and calls the **tabu_search** function to find the best solution and assigns the result to **best_x**

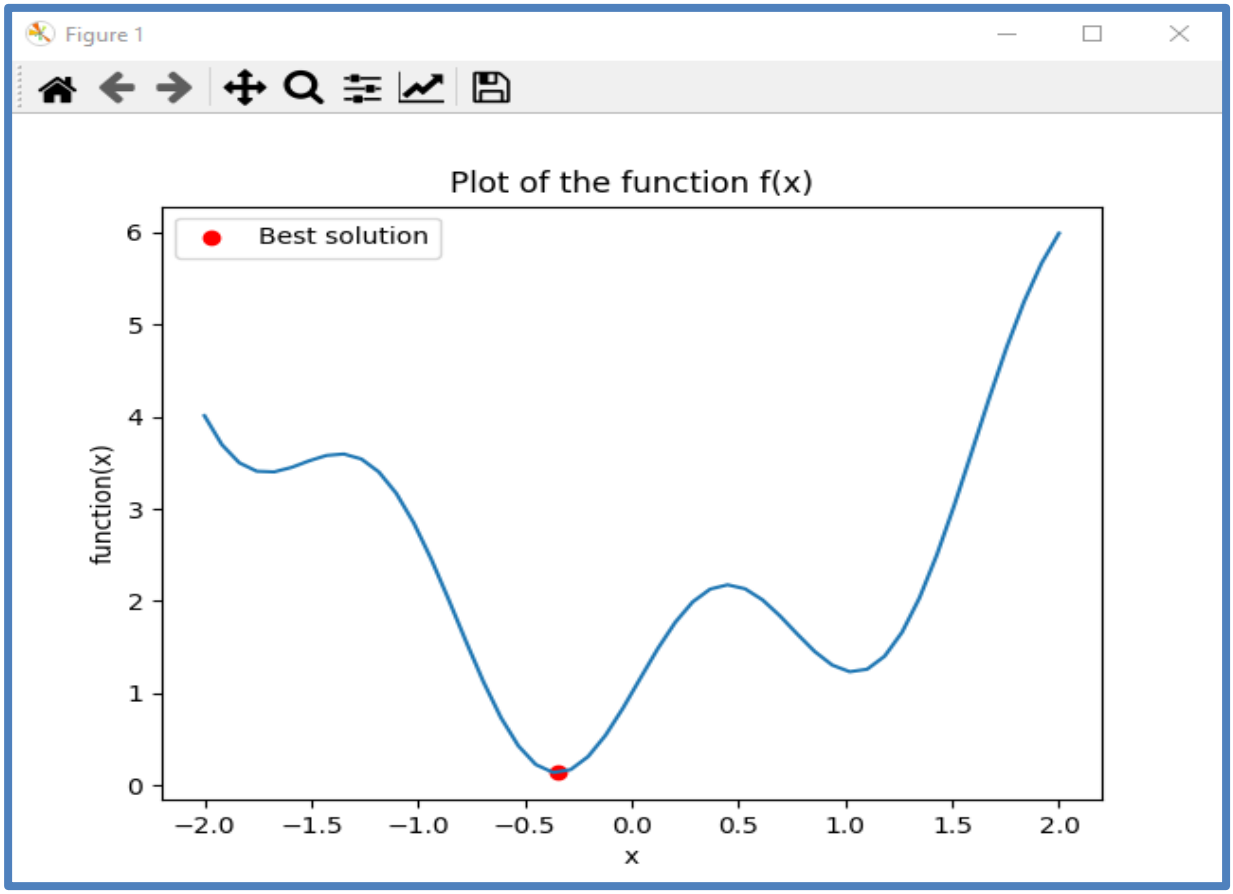
```
def objective_function(x):
    return x**2 + np.sin(4*x) + 1
best_x=tabu_search(objective_function, 5, 1000)
print("best x is : ",best_x)


x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = objective_function(x)  # calculate y values for plotting

plt.plot(x, y)
plt.scatter(best_x, objective_function(best_x), color='red', label='Best solution')
plt.xlabel('x')
plt.ylabel('function(x)')
plt.title('Plot of the function f(x)')
plt.legend()
plt.show()
```

Result:

```
best x is :  -0.34884453097410173
```

Then it plots the **objective_function** by generating x values and y values for the function and plotting it. Then it plots a red dot on the best_x, best_y for the best solution found by tabu_search. And then it shows the graph with the labels and title.

Result:



4. Complexity

The complexity of this code is determined by the running time of the tabu_search function, which consists of the for loop that runs for max_iterations and the operations performed inside the loop.

The running time of the for loop is O(max_iterations). Inside the loop, a random number is generated in O(1) time. Checking if an element is in a list takes O(n) time, where n is the length of the list, so in this case it takes O(tabu_list_length) time. Updating the best_solution, appending the element to the tabu_list, and removing the oldest element from the tabu_list each take O(1) time.

Therefore, the total running time of the tabu_search function is O(max_iterations * (1 + tabu_list_length)) and as the max_iterations dominates the other operations in the tabu_search function. It is fair to say the complexity of this code is O(max_iterations)

It is worth noting that the obj_func(x) could have a different complexity, the complexity of the code will be affected by the obj_func used.

5. Advantages & Disadvantages

Tabu Search is an optimization algorithm that can be used to find good solutions to complex problems. Here are some of the advantages and disadvantages of this algorithm:

Advantages:

- ✓ It can escape from local optima by using the tabu list to prevent cycling
- ✓ It can be used with a variety of objective functions, making it a versatile optimization algorithm
- ✓ It can handle constraints and incorporate domain-specific knowledge
- ✓ It's relatively simple to implement and understand

Disadvantages:

- It's not guaranteed to find the global optimum, it can get stuck in a local optimum
- It can be sensitive to the parameter settings, such as the length of the tabu list and the max_iterations
- It requires a lot of memory and computational resources, especially when the problem size is large
- It's not always easy to know when to stop the algorithm

It's important to note that the choice of optimization algorithm is problem-specific and this algorithm may or may not be the best one for a particular problem. However, it is a good choice if the optimization problem has a good structure that can be exploited with tabu lists and the exploration of different regions. Also it could be used in a problem where the use of other metaheuristics like Hill Climbing or Simulated Annealing are not able to find good solution and or get stuck in local optima.

6. Conclusion

Tabu Search algorithm is a powerful optimization method that is used to find good solutions to a wide range of problems. It is a type of local search algorithm that makes small changes to the current solution and uses a tabu list to prevent revisiting solutions that have been visited before. The algorithm also uses an aspiration criterion, which allows it to move to a new solution if it is significantly better than the current one. The performance of tabu search can be improved through careful design of the neighborhood structure, aspiration criteria, and tabu list. Additionally, it's not limited to specific problem domains and can be used to solve various optimization problems with good performance.

VI. Variable neighborhood search algorithm

1. Introduction

The VNS (Variable Neighborhood Search) algorithm is a metaheuristic optimization method that can be used to find the minimum of a function. It is inspired by the behavior of natural systems, where local optima are often surrounded by neighborhoods of worse solutions, but better solutions can be found by exploring different neighborhoods.

2. Implementation

The basic idea of the VNS algorithm is to start with an initial solution, and then repeatedly perturb this solution by changing the neighborhood it is in, until a local minimum is found. The algorithm can be summarized as follows:

1.Start with an initial solution

2.For a given number of iterations or until a stopping criterion is met:

a. Select a neighborhood of the current solution

b. Find the best solution in the selected neighborhood

c. If the best solution is better than the current solution, update the current solution

4.Return the best solution found

```python
import numpy as np
import random
import math
import matplotlib.pyplot as plt

def vns(f, x0, neighborhoods, max_iterations=100):
    x = x0
    best_x = x0
    best_f = f(x0)
    xs = [x0]  # list to store all solutions
    for i in range(max_iterations):
        for neighborhood in neighborhoods:
            x_new = neighborhood(x)
            f_new = f(x_new)
            if f_new < best_f:
                best_f = f_new
                best_x = x_new
                x = x_new
                xs.append(x)  # store new solution
    return best_x, f(best_x), xs
```

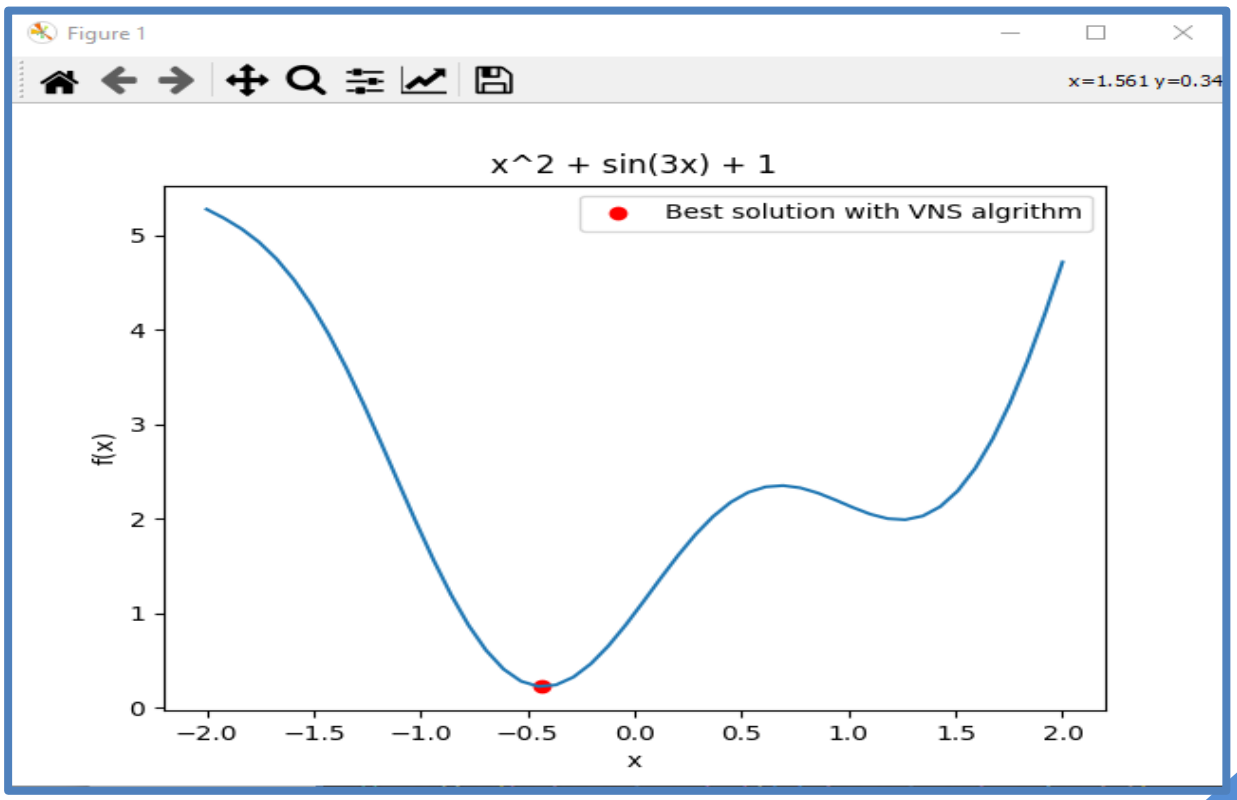
3. Testing

```
"""Test"""

def f(x):
    return x**2 + np.sin(3*x) + 1

def random_neighborhood(x):
    return x + random.uniform(-1, 1)

x0 = 0
neighborhoods = [random_neighborhood]
min_x, min_f,xs = vns(f, x0, neighborhoods)
print(min_x, min_f)

x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = f(x)  # calculate y values for plotting

plt.plot(x, y)
plt.scatter(min_x, min_f, color='red', label='Best solution with VNS algrithm')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('x^2 + sin(3x) + 1')
plt.legend()
plt.show()
```

Console:

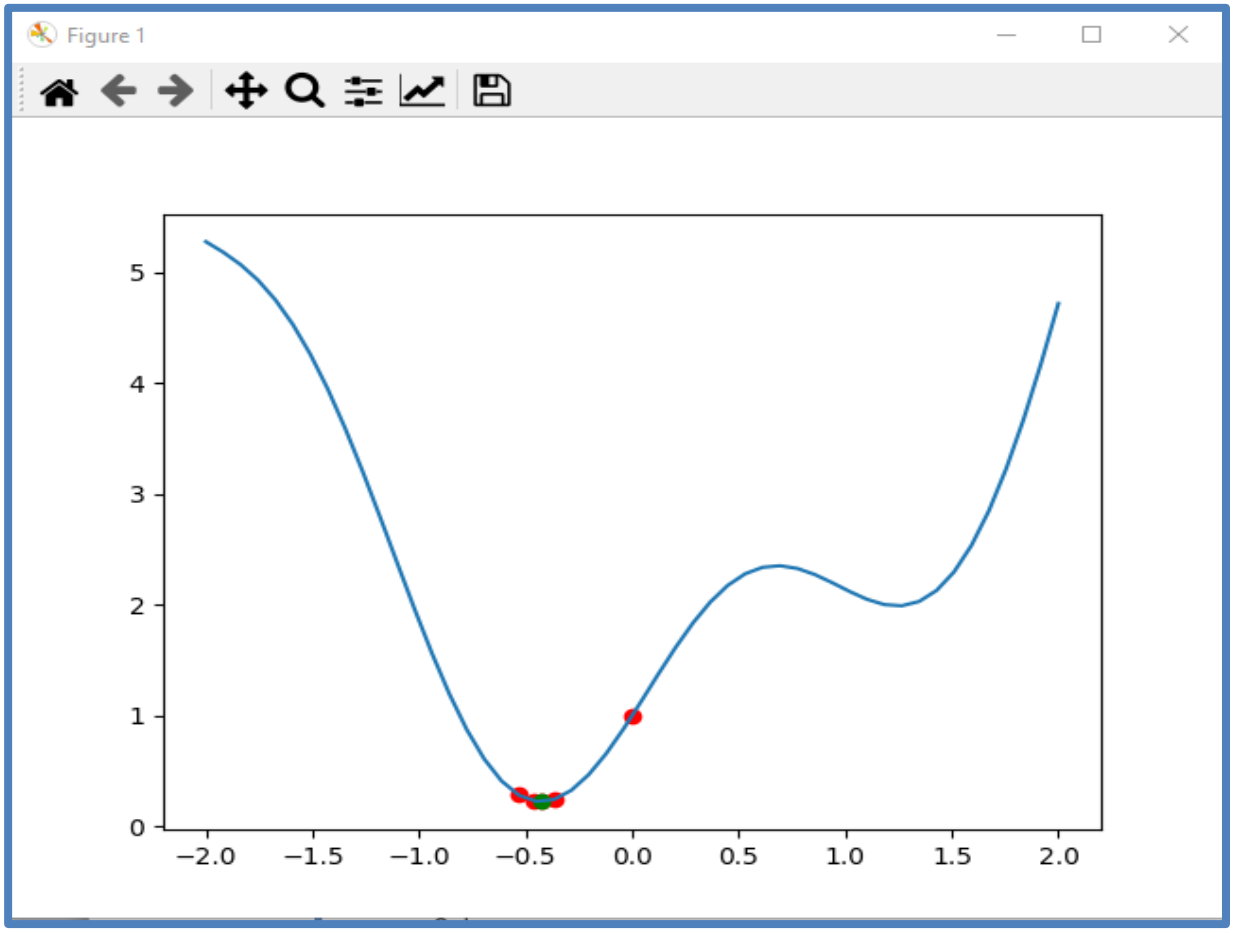
```
-0.4326356729133871 0.22417742015314468
```

Curve:



```
min_x, min_f, xs = vns(f, x0, neighborhoods)

# Plot function and solutions
x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = [f(xx) for xx in x]
plt.plot(x, y)
plt.scatter([x for x in xs], [f(x) for x in xs], color='r')
plt.scatter(min_x, min_f, color='green', label='Best solution with VNS algrithm')
plt.show()
```

Curve:



4. Complexity

The complexity of this algorithm is O(n * m), where n is the value of max_iterations and m is the number of neighborhoods. This is because the algorithm first iterates through a loop that runs for a maximum of max_iterations times. Within each iteration, it then iterates through another loop that runs for the number of neighborhoods. Within this second loop, it performs a constant amount of operations (such as generating a new solution and updating the current and best solutions). Therefore, the total number of operations performed is directly proportional to the product of max_iterations and the number of neighborhoods, making the complexity O(n * m).

5. Advantages & Disadvantages

Advantages of Variable Neighborhood Search :

- Global Optimization: VNS is a global optimization algorithm, meaning it can find the global minimum of a function, unlike local optimization algorithms that can only find the local minimum.
- Handling Constraints: The algorithm can handle constraints on the solution space, making it useful for solving problems with complex constraints.
- Handling Multi-modal Functions: VNS is particularly effective at finding the global minimum of multi-modal functions, which have multiple local minima.
- Flexibility: The algorithm is flexible as it allows for the use of multiple neighborhoods, which can be tailored to the specific problem at hand.

Disadvantages of Variable Neighborhood Search :

- Computationally Expensive: The algorithm can be computationally expensive, as it requires many iterations and random number generation, which can slow down the optimization process.
- Difficulty in Determining Parameters: Determining the proper values for the parameters, such as the number of iterations, the number of neighborhoods, and their structure, can be difficult and may require trial and error.
- Convergence to Local Minimum: The algorithm may converge to a local minimum, especially if the initial solution is not well chosen or if the neighborhoods are not properly designed.
- Time-consuming: The algorithm takes a lot of time to run, especially when the function is complex or has many local minima, as it needs to search through a large solution space with multiple neighborhoods.

6. Conclusion

Variable Neighborhood Search can be useful for optimization problems that have multiple local optima, where traditional optimization methods like gradient descent may get stuck in a poor local optimum.

VII. Gradient descent algorithm

1. Introduction

Gradient descent is an optimization algorithm used to minimize a function. It is commonly used in machine learning to adjust the parameters of a model in order to minimize the error or loss function.

The algorithm starts with an initial set of parameter values and then repeatedly updates the parameters in the opposite direction of the gradient of the loss function with respect to the parameters. The size of the update is determined by the learning rate hyper parameter. The process is repeated until a minimum of the loss function is reached or a preset number of iterations are reached.

There are different variants of Gradient descent like Stochastic Gradient Descent (SGD) and mini-batch gradient descent where in SGD we update the parameters after processing each sample, in mini-batch Gradient Descent, the parameters are updated after processing a small number of samples(batch size) instead of processing all the sample at once.

2. Implementation

This algorithm above is using gradient descent to find the minimum value of a Loss function (also called the objective function), represented by the cost function, which is defined as x**2 + np.sin(5*x) + 1.

```
import math
import matplotlib.pyplot as plt
import numpy as np


def Loss(x):
    return x**2 + np.sin(5*x) + 1

def function(x):
    return 2*x + 5*np.cos(5*x)

def gradient_descent(x_init, learning_rate=0.01, num_iterations=100):
    x_current = x_init
    for i in range(num_iterations):
        x_current = x_current - learning_rate * function(x_current)
    return x_current
```

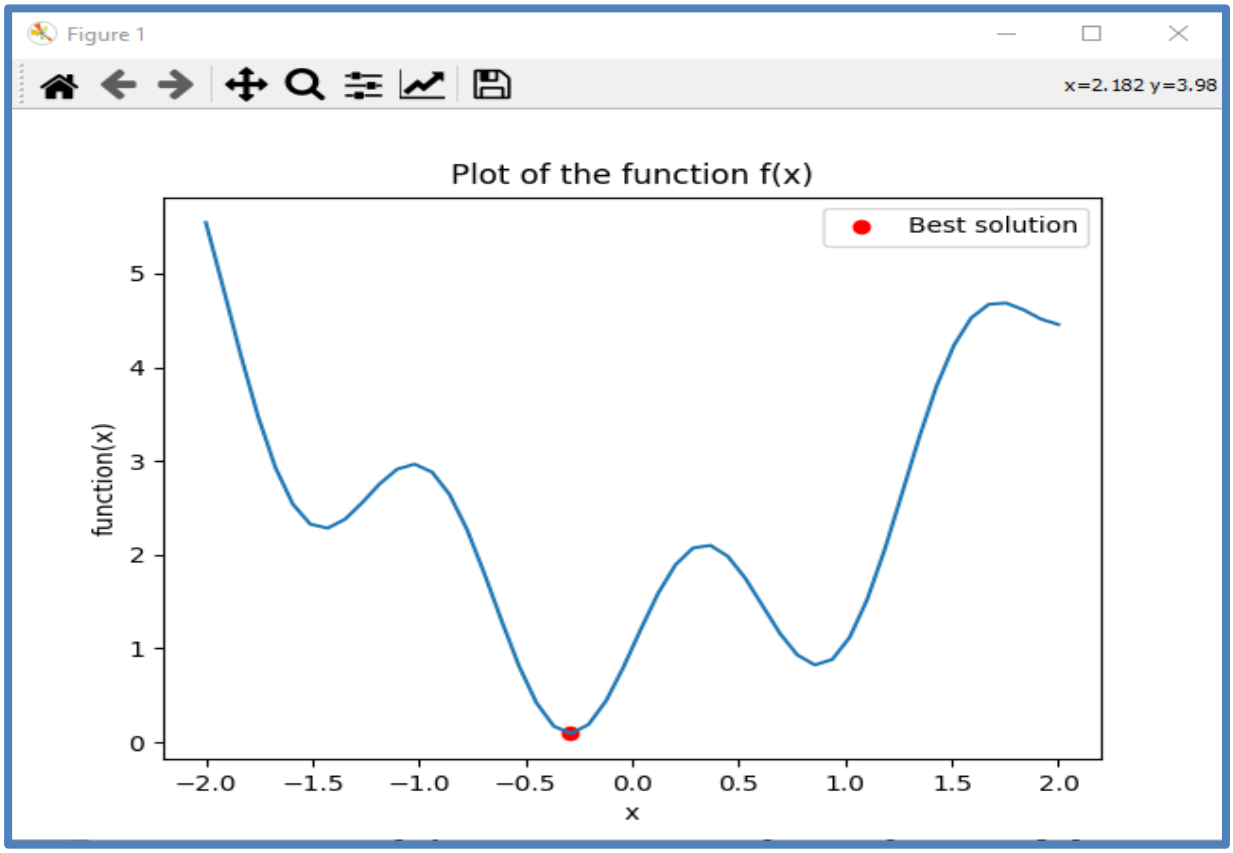
The gradient_descent function is used to find the minimum value of this function. It does this by first defining a starting point x_init and repeatedly moving in the opposite direction of the gradient of the function (hence the name "gradient descent") by a factor of the learning rate, until it reaches a minimum value or the number of iterations defined is reached.

The function is the derivative of cost defined as function(x): return 2*x + 5*np.cos(5*x)

3. Testing

```
x_min = gradient_descent(-1, num_iterations=1000)
print("Minimum found at", x_min)

x = np.linspace(-2, 2, 50)  # generate x values for plotting
y = Loss(x)  # calculate y values for plotting

plt.plot(x, y)
plt.scatter(x_min, Loss(x_min), color='red', label='Best solution')
plt.xlabel('x')
plt.ylabel('function(x)')
plt.title('Plot of the function f(x)')
plt.legend()
plt.show()
```

The cost function and the derivative of it both are plotted in the end using matplotlib library, x_min is calculated by the function gradient_descent and x_min is plotted with red dot in the graph.



It also displays the minimum found at a particular point of the graph in the console using print statement.

```
p\Nouveau dossier\Documents\GradianDecent\GradianDecent 1.py"
Minimum found at -0.2908393149953235
```

4. Complexity

The algorithm has a time complexity of O(num_iterations) and space complexity of O(1) assuming that the calculation of the Loss function, sin and cos also have O(1) time complexity. It uses one for loop which iterates num_iterations times, and within that loop there are three simple mathematical operations, so the time complexity is proportional to the number of iterations. It is using x_current variable, x_init variable and one iterator variable i, so the space complexity is O(1).

5. Advantages & Disadvantages

Here are some advantages and disadvantages of this algorithm:

Advantages:

- ✓ It is a widely used optimization algorithm and is well-suited for a wide range of problems.
- ✓ It can be used with a variety of different cost functions and can be applied to both linear and nonlinear problems.
- ✓ It is computationally efficient, particularly when the cost function is efficiently computable.
- ✓ With proper learning rate the algorithm will converge to the minimum value in relatively few steps.

Disadvantages:

- The algorithm may converge to a local minimum rather than the global minimum, which can be a problem for non-convex cost functions.
- It may be sensitive to the choice of the initial conditions and the learning rate.
- It is not suitable for online learning, it requires complete data in order to work.
- The convergence of the algorithm might be slow, especially for large dataset, the algorithm might get stuck in a flat region or plateaus.

6. Conclusion

Gradient descent is an iterative algorithm and the convergence speed and stability is heavily dependent on the choice of the learning rate. Even though gradient descent is a widely used optimization technique, it can get stuck in local minima, and that is where other optimization techniques like conjugate gradient,BFGS, and L-BFGS, which are second-order optimization methods that may provide better performance

VIII. General Conclusion

Optimization algorithms are a set of techniques used to find the best solution to a problem by minimizing or maximizing an objective function. There are many different optimization algorithms available, each with their own strengths and weaknesses, and the choice of algorithm will depend on the specific problem being solved and the requirements of the solution. Some common optimization algorithms include gradient descent, Newton's method, simulated annealing, and genetic algorithms. It is important to note that optimization algorithms are used in many different fields and applications, including machine learning, engineering, finance, and operations research. Choosing the best suited algorithm requires a good understanding of the problem, the underlying mathematical models and the requirement of solution.