SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

SCHOOL OF COMPUTING

DEPARTMENT OF DATA SCIENCE AND BUSINESS SYSTEMS

18CSC305J ARTIFICIAL INTELLIGENCE

MINI PROJECT REPORT

Searching Techniques in AI

Name:	**Akshat Gaur**
Register Number:	**RA1911027010051**
Mail ID:	**ag5287@srmist.edu.in**
Department:	**CSE**
Specialization:	**Big Data Analytics**
Semester:	**VI**

Team Members -

Name	**Registration Number**
Akshunn Sharma	**RA1911027010041**
Ishan Gupta	**RA1911027010043**
A Prithivi	**RA1911027010052**

CONTENTS

ABSTRACT

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

We are mainly going work in the 2 types of search algorithms mainly – Uninformed search and Informed search. The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

1 – INTRODUCTION AND MOTIVATION

We have been studying the different ways we can search an element in a tree or graph. Based on different scenarios and different requirements we choose different search algorithm to get the desired output in the most efficient way. In AI we have studied about theoretical concepts of some complex search algorithms, which are mainly classified into 4 categories- Uninformed Search, Informed Search, Local Search and Adversarial Search, and their most commonly used algorithms.

Understanding all the algorithms and their working is very interesting yet confusing and complex. Students most of the time get confused when it comes to implementation, and in AI, practical knowledge is way more important than theoretical knowledge, because we only understand the algorithm and functioning of an algorithm thoroughly when we implement them.

Also, all algorithms has their own values in different scenarios, its not necessary that one algorithm will work optimally and efficiently in every case. And there comes the role of knowing which algorithm is best for which case, only then one can use the best algorithm for the problem.

To understand how all these algorithms work and how we pick best algorithm for a given problem statement, we are planning to make a program which takes a graph as an input (directed/undirected, heuristic/non-heuristic & weighted/non-weighted) and run all the search algorithms on them to see how each algorithm approach to a solution, what are its limits and what are its advantages. We are trying to make this mini-project easy-to-understand and thorough with logic so that students could easily grasp the knowledge and learn these algorithms on another level.

<u>2 – REVIEW OF EXISTING METHODS AND THEIR LIMITATIONS</u>

There are very good lectures available on the web, explaining about all the searching algorithms which are used in AI, but a very few of them explain the implementation of the code, also when they implement the code, the explanation is not very good or easy to understand.

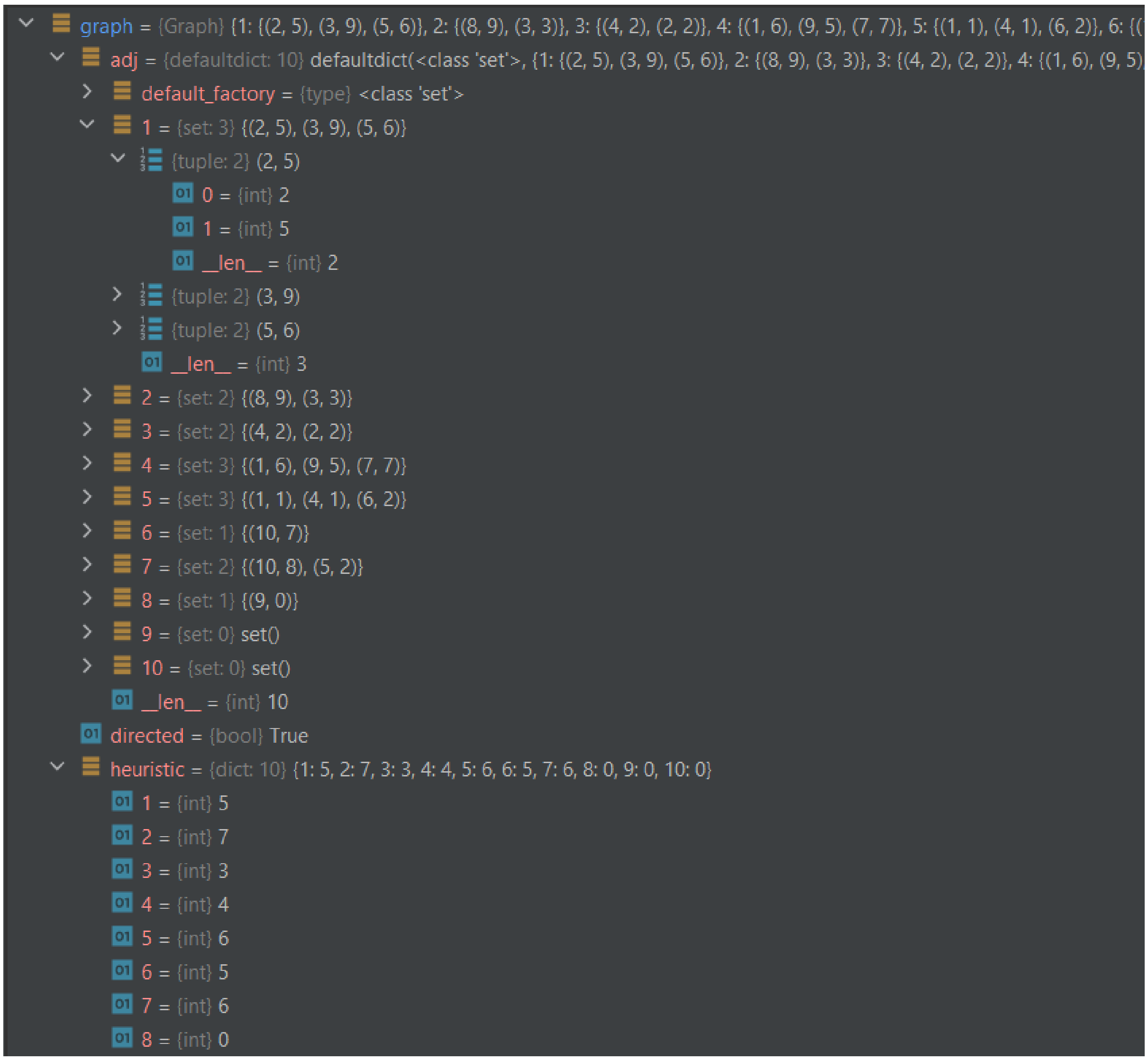
It will be easy to understand, if all the codes are applied on same input graph, with same structure of graph, in general the all codes online have their own input style, code pattern, which makes it difficult to relate and compare with other algorithms, if there is one single structure of graph is followed in all the algorithm implementation, one can easily understand the differences and similarities between different algorithms and it won’t take time to understand the basic code again and again, student, after understanding the format once, can directly jump to the algorithm implementation part of the code.

But there is no such formatted code available online.

3 – PROPOSED METHOD WITH ARCHITECTURE

For our code implementation, we are going to create a class graph, which will be having out graph object and all the search algorithms as functions.

From the given screenshot we can get an idea of what our graph object looks like, this object will be created as soon as the class is initialized.



```
graph = {Graph} {1: {(2, 5), (3, 9), (5, 6)}, 2: {(8, 9), (3, 3)}, 3: {(4, 2), (2, 2)}, 4: {(1, 6), (9, 5), (7, 7)}, 5: {(1, 1), (4, 1), (6, 2)}, 6: {(
  adj = {defaultdict: 10} defaultdict(<class 'set'>, {1: {(2, 5), (3, 9), (5, 6)}, 2: {(8, 9), (3, 3)}, 3: {(4, 2), (2, 2)}, 4: {(1, 6), (9, 5)
    default_factory = {type} <class 'set'>
    1 = {set: 3} {(2, 5), (3, 9), (5, 6)}
      {tuple: 2} (2, 5)
        0 = {int} 2
        1 = {int} 5
        __len__ = {int} 2
      {tuple: 2} (3, 9)
      {tuple: 2} (5, 6)
      __len__ = {int} 3
    2 = {set: 2} {(8, 9), (3, 3)}
    3 = {set: 2} {(4, 2), (2, 2)}
    4 = {set: 3} {(1, 6), (9, 5), (7, 7)}
    5 = {set: 3} {(1, 1), (4, 1), (6, 2)}
    6 = {set: 1} {(10, 7)}
    7 = {set: 2} {(10, 8), (5, 2)}
    8 = {set: 1} {(9, 0)}
    9 = {set: 0} set()
    10 = {set: 0} set()
    __len__ = {int} 10
  directed = {bool} True
  heuristic = {dict: 10} {1: 5, 2: 7, 3: 3, 4: 4, 5: 6, 6: 5, 7: 6, 8: 0, 9: 0, 10: 0}
    1 = {int} 5
    2 = {int} 7
    3 = {int} 3
    4 = {int} 4
    5 = {int} 6
    6 = {int} 5
    7 = {int} 6
    8 = {int} 0
```

About our graph object- its stored in variable "adj" as defaultdict(), where the *key* is the node number and the *value* is list of all the nodes connected to it (if weighted, list will be having the values as tuple, where 2nd parameter will denote weight of the edge), our class initializer will convert the input list to a proper graph as per user request (directed/undirected, heuristic/non-heuristic & weighted/non-weighted). For heuristic values there's another separate dictionary, with *key* is node and *value* is heuristic value.

Once we are done with the inputs, we one-by-one call all the search algorithms, execute them and print their path traversals in the console.

Note: For Local search algorithms, the input formats are totally different (will be explained in module description thoroughly), so we will be implementing in different file.

4 – MODULES WITH DESCRIPTION

Graph Definition:

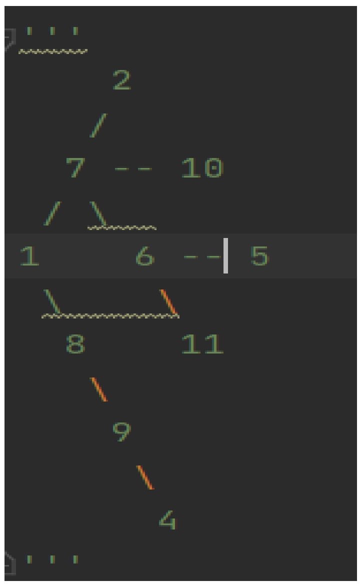
The graph object will have 5 parameters –

➢ Adj	[defaultdict()]	(contains node index and edge weight)
➢ Weighted	[bool]	
➢ Directed	[bool]	
➢ Heuristic	[bool]	
➢ heuristic	[dict]	(contains heuristic values of each node)

To initialize the graph we get a nested list as input “edgelist”, where

Edgelist = [[first_node, second_node, edge_weight], […]]

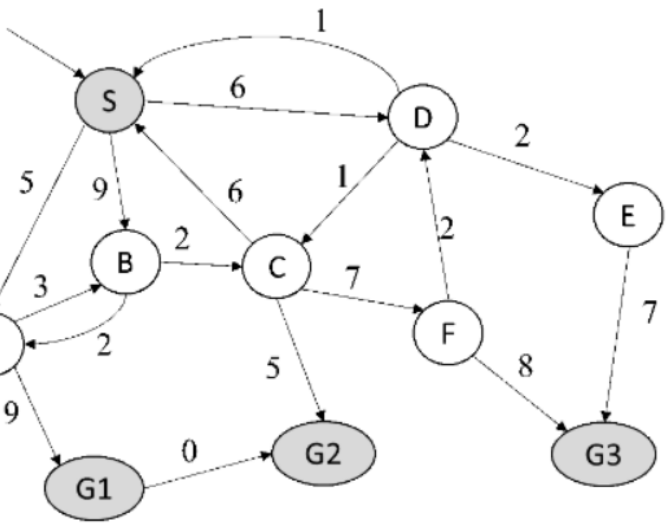
With the help of _initialize(), add(), generate_heuristics() we initialize our graph.



```
'''
        2
       /
      7 -- 10
     / \
  1       6 --| 5
     \       \
      8       11
       \
        9
         \
          4
'''
```

UNDIRECTED GRAPH

(input1)



DIRECTED GRAPH

(input2)

BFS (Breadth First Search):

Definition:

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

Algorithm:

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state)

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Type of graph used:

- Data Structure: Undirected Graphs or Trees
- Weighted: No
- Search style: Uninformed

Complexity and Optimality:

- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$
- Completeness: Yes
- Optimality: Gives Optimal Solution

Best suit for: Mostly used for finding shortest possible path

Output: [consider input1 (undirected graph)]

- Path: 1-> 8-> 7-> 9-> 2-> 10-> 6-> 4

DFS (Depth First Search):

Definition:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Algorithm:

The DFS algorithm works as follows:

- Start by putting any one of the graph's vertices on top of a stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
- Keep repeating steps 2 and 3 until the stack is empty

Type of graph used:

- Data Structure: Undirected Graphs or Trees
- Weighted: No
- Search style: Uninformed

Complexity and Optimality:

- Time Complexity: $O(b^m)$ [b = number of nodes & m = maximum depth of any node]
- Space Complexity: O(bm)
- Completeness: Complete [within a finite state space]
- Optimality: Non-Optimal [may take large no. of steps before reaching goal]

Best suit for:
Mostly topological sorting, scheduling problems, cycle detection in graphs, and solving puzzles with only one solution, such as a maze or a sudoku puzzle

Output:
[consider input1 (undirected graph)]

- Path: 1-> 8-> 9-> 4

DLS (Depth Limit Search):

Definition:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

DLS is mostly used when we know the search domain, and there exists a prior knowledge of the problem and its domain while this is not the case for uninformed search strategy. Typically, we have little idea of the goal node's depth unless one has tried to solve it before and has found the solution.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cut-off failure value: It defines no solution for the problem within a given depth limit.

Algorithm:

- We start with finding and fixing a start node.
- Then we search along with the depth using the DFS algorithm.
- Then we keep checking if the current node is the goal node or not.

Type of graph used:

- Data Structure: Undirected Graphs or Trees
- Weighted: No
- Search style: Uninformed

Complexity and Optimality:

- Time Complexity: $O(b^l)$ [b = number of nodes & l = limit set]
- Space Complexity: O(b * l)
- Completeness: No
- Optimality: Non-Optimal

Best suit for: Any DFS application where depth of goal node is known.

Output: [consider input1 (undirected graph), with *depth limit set to '4'*]

- Path: 1-> 8-> 9-> 4-> 7-> 2-> 10-> 6-> 11-> 4

IDDFS (Iterative Deepening Depth First Search):

Definition:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found. This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found. This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Algorithm:

```
➢ function IDDFS(root) is
    o for depth from 0 to ∞ do
        ▪ found, remaining ← DLS(root, depth)
        ▪ if found ≠ null then return found
            • else if not remaining then return null
➢ function DLS(node, depth) is
    o if depth = 0 then
        ▪ if node is a goal then return (node, true)
        ▪ else return (null, true)    (Not found, but may have children)
    o else if depth > 0 then
        ▪ any remaining ← false
        ▪ foreach child of node do
            • found, remaining ← DLS(child, depth−1)
            • if found ≠ null then return (found, true)
            • if remaining then any remaining ← true
        ▪ return (null, any_remaining)
```

Type of graph used:

- Data Structure: Undirected Graphs or Trees
- Weighted: No
- Search style: Uninformed

Complexity and Optimality:

- Time Complexity: $O(b^d)$[b= number of nodes & d= depth of the shallowest solution]
- Space Complexity: O(bd)
- Completeness: Complete [If the branching factor is finite]
- Optimality: Optimal [optimal if path cost is a non- decreasing function of ‘d’]

<u>Best suit for:</u> We use IDDFS when we do not know the depth of our solution and have to search a very large state space.

<u>Output:</u> [consider input1 (undirected graph), with depth limit set to 4]

- Path: 1-> 8-> 9-> 4-> 7-> 2-> 10-> 6-> 11-> 4

Informed Search Algos (Introduction):

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function (denoted by h(n)):

Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

A* Search:

Definition:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a *fitness number*.

Algorithm:

- Place the starting node in the OPEN list.
- Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- Return to Step 2

Type of graph used:

- Data Structure: Directed Graphs or Trees
- Weighted: Yes
- Search style: Informed

Complexity and Optimality:

- Time Complexity: O(b^d) [b = branching factor & d = depth of solution]
- Space Complexity: O(b^d)
- Completeness: Yes [only when 'b' is finite & cost at every action is fixed]
- Optimality: Optimal

Best suit for: Any DFS application where depth of goal node is known.

Output: [consider input2 (directed graph), looking for 11th node]

- Path: 1-> 2-> 3-> 4-> 5-> 6-> 11

Hill Climbing:

Definition:

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

Types of Hill Climbing Algorithm:

- Simple hill Climbing
- Steepest-Ascent hill-climbing
- Stochastic hill Climbing

Algorithm:

- Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- Loop until a solution is found or the current state does not change.
  - Let SUCC be a state such that any successor of the current state will be better than it.
  - For each operator that applies to the current state:
    - Apply the new operator and generate a new state
    - Evaluate the new state
    - If it is goal state, then return it and quit, else compare it to the SUCC
    - If it is better than SUCC, then set new state as SUCC
    - If the SUCC is better than the current state, then set current state to SUCC

Type of graph used:

Data Structure:	Undirected Graphs or Trees
Weighted:	No
Search style:	Uninformed

Complexity and Optimality:

Time Complexity:	O(∞) [∞ is infinity]
Space Complexity:	O(b)
Completeness:	No
Optimality:	Non-Optimal

Best suit for: Used in optimising Travelling salesmen type of problems.

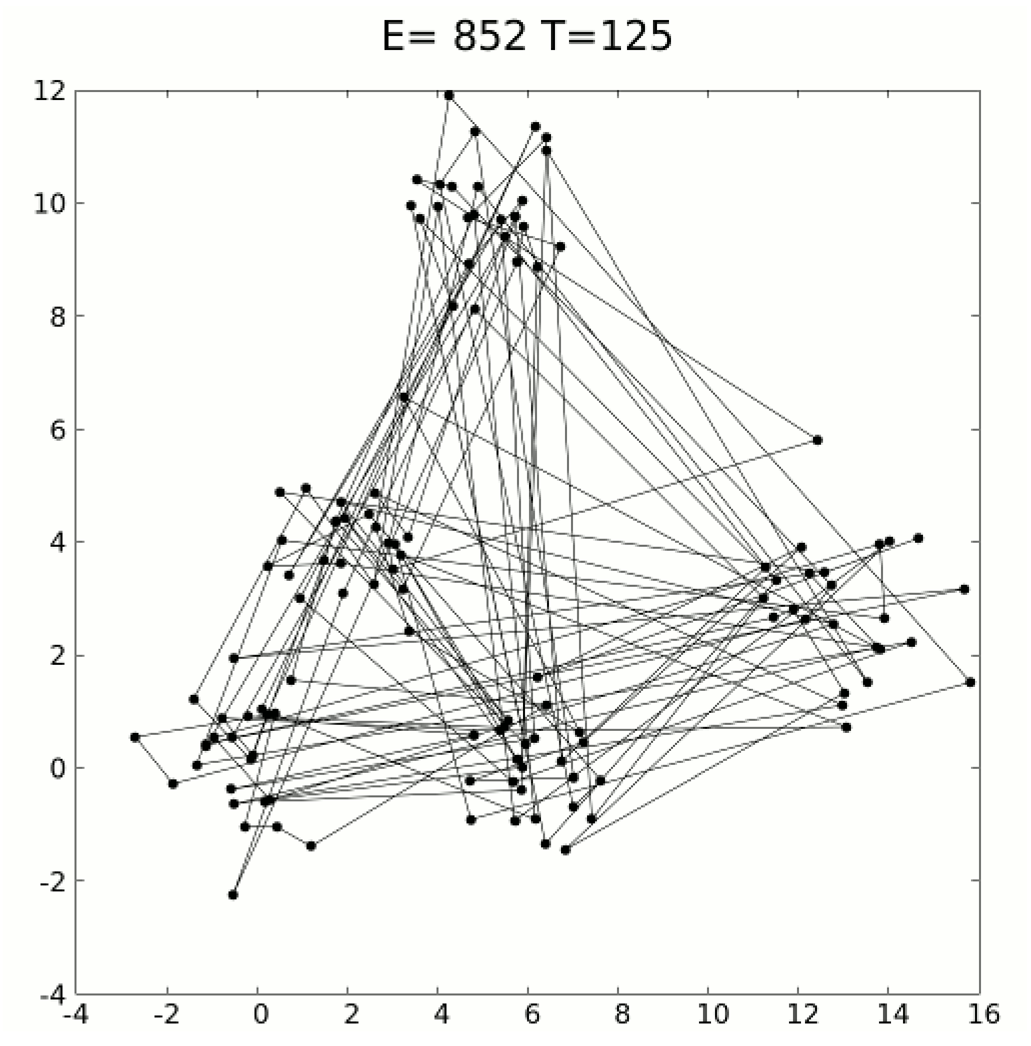
Simulated Annealing:

Definition:

Simulated annealing is a method for solving unconstrained and bound-constrained optimization problems. The method models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy.

At each iteration of the simulated annealing algorithm, a new point is randomly generated. The distance of the new point from the current point, or the extent of the search, is based on a probability distribution with a scale proportional to the temperature. The algorithm accepts all new points that lower the objective, but also, with a certain probability, points that raise the objective. By accepting points that raise the objective, the algorithm avoids being trapped in local minima, and is able to explore globally for more possible solutions. An annealing schedule is selected to systematically decrease the temperature as the algorithm proceeds. As the temperature decreases, the algorithm reduces the extent of its search to converge to a minimum.

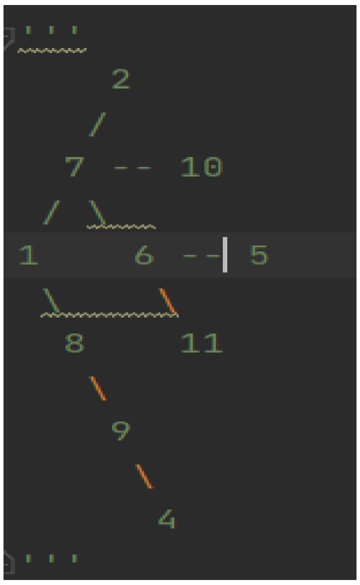
<u>Used for:</u> Used overcome the limits of Hill climbing algorithm.



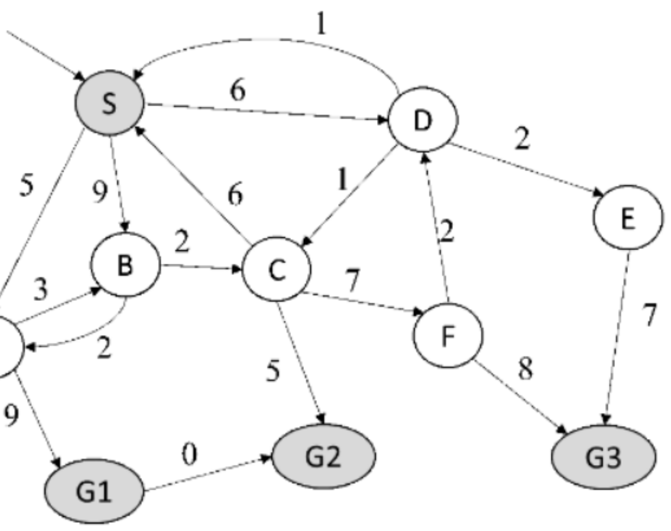
6 – Output SCREENSHOTS

Informed and uninformed Search-

Input graph:



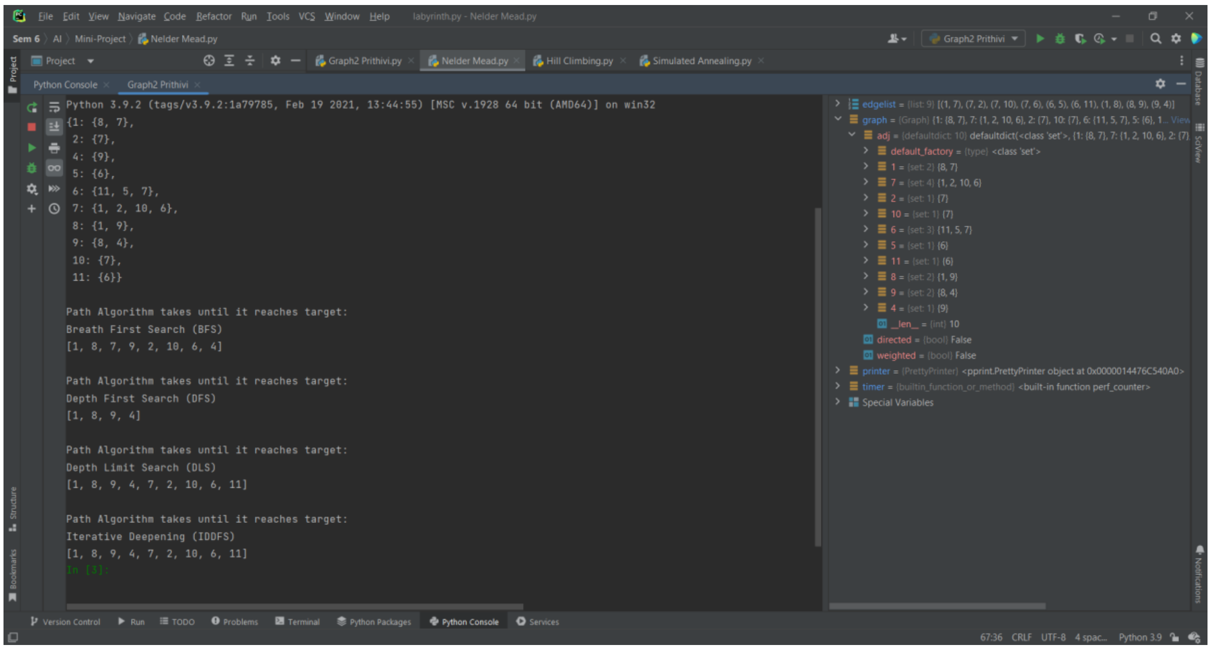
```
'''
        2
       /
     7 -- 10
    / \
1      6 --| 5
    \_______\
     8      11
      \
       9
        \
         4
'''
```



UNDIRECTED GRAPH (input1)

DIRECTED GRAPH (input2)

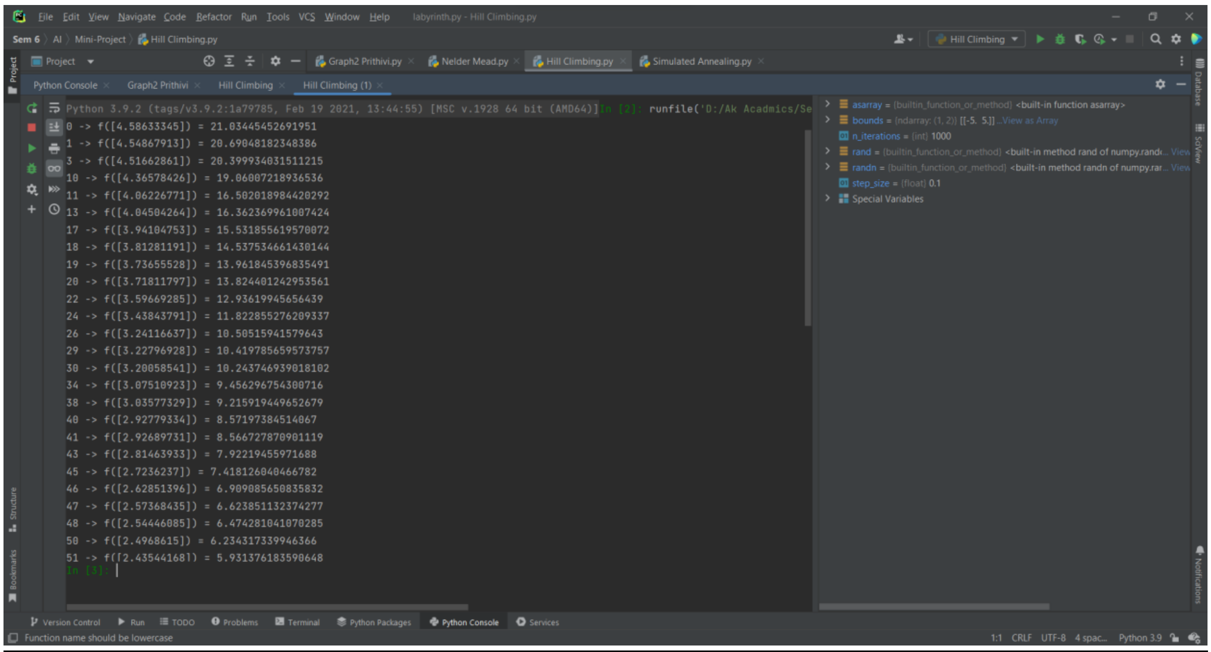
Output graph:



```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
{1: {8, 7},
 2: {7},
 4: {9},
 5: {6},
 6: {11, 5, 7},
 7: {1, 2, 10, 6},
 8: {1, 9},
 9: {8, 4},
 10: {7},
 11: {6}}

Path Algorithm takes until it reaches target:
Breath First Search (BFS)
[1, 8, 7, 9, 2, 10, 6, 4]

Path Algorithm takes until it reaches target:
Depth First Search (DFS)
[1, 8, 9, 4]

Path Algorithm takes until it reaches target:
Depth Limit Search (DLS)
[1, 8, 9, 4, 7, 2, 10, 6, 11]

Path Algorithm takes until it reaches target:
Iterative Deepening (IDDFS)
[1, 8, 9, 4, 7, 2, 10, 6, 11]
In [3]:
```

```
edgelist = {list: 9} [(1, 7), (7, 2), (7, 10), (7, 6), (6, 5), (6, 11), (1, 8), (8, 9), (9, 4)]
graph = {Graph} {1: {8, 7}, 7: {1, 2, 10, 6}, 2: {7}, 10: {7}, 6: {11, 5, 7}, 5: {6}, 1... View
  adj = {defaultdict: 10} defaultdict(<class 'set'>, {1: {8, 7}, 7: {1, 2, 10, 6}, 2: {7},
    default_factory = {type} <class 'set'>
    1 = {set: 2} {8, 7}
    7 = {set: 4} {1, 2, 10, 6}
    2 = {set: 1} {7}
    10 = {set: 1} {7}
    6 = {set: 3} {11, 5, 7}
    5 = {set: 1} {6}
    11 = {set: 1} {6}
    8 = {set: 2} {1, 9}
    9 = {set: 2} {8, 4}
    4 = {set: 1} {9}
    __len__ = {int} 10
  directed = {bool} False
  weighted = {bool} False
printer = {PrettyPrinter} <pprint.PrettyPrinter object at 0x0000014476C540A0>
timer = {builtin_function_or_method} <built-in function perf_counter>
Special Variables
```

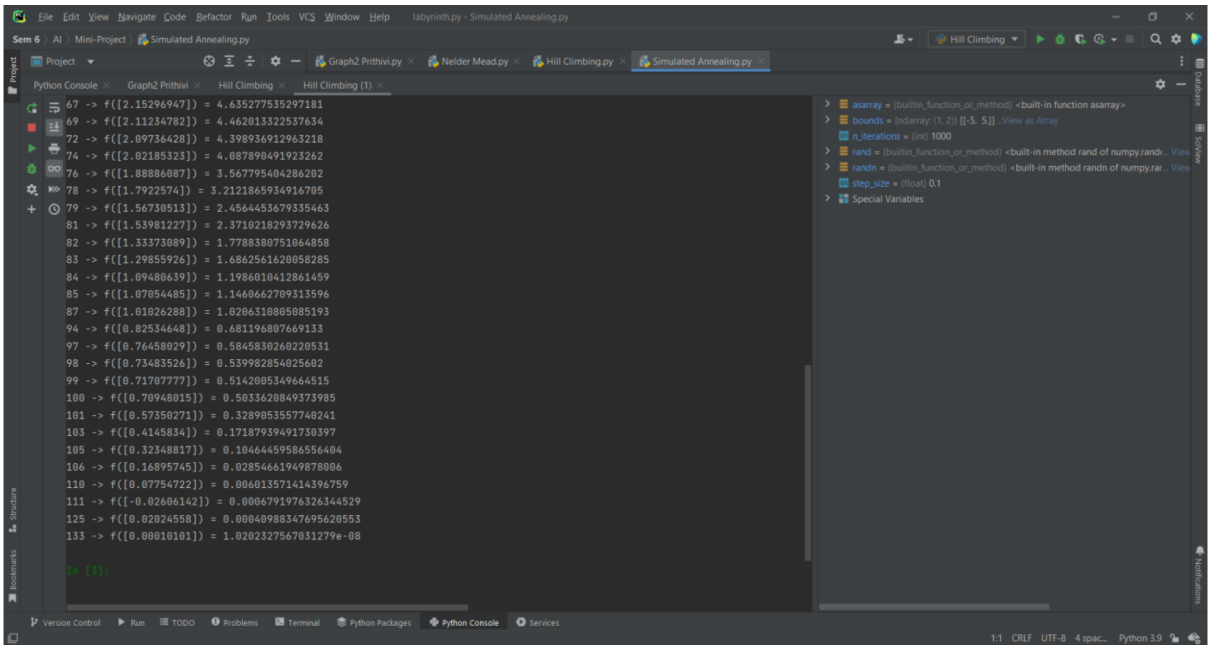
Hill Climbing-



```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]In [2]: runfile('D:/Ak Acadmics/Se
0 -> f([4.58633345]) = 21.03445452691951
1 -> f([4.54867913]) = 20.69048182348386
3 -> f([4.51662861]) = 20.399934031511215
10 -> f([4.36578426]) = 19.06007218936536
11 -> f([4.06226771]) = 16.502018984420292
13 -> f([4.04504264]) = 16.362369961007424
17 -> f([3.94104753]) = 15.531855619570072
18 -> f([3.81281191]) = 14.537534661430144
19 -> f([3.73655528]) = 13.961845396835491
20 -> f([3.71811797]) = 13.824401242953561
22 -> f([3.59669285]) = 12.93619945656439
24 -> f([3.43843791]) = 11.822855276209337
26 -> f([3.24116637]) = 10.50515941579643
29 -> f([3.22796928]) = 10.419785659573757
30 -> f([3.20058541]) = 10.243746939018102
34 -> f([3.07510923]) = 9.456296754300716
38 -> f([3.03577329]) = 9.215919449652679
40 -> f([2.92779334]) = 8.57197384514067
41 -> f([2.92689731]) = 8.566727870901119
43 -> f([2.81463933]) = 7.92219455971688
45 -> f([2.7236237]) = 7.418126040466782
46 -> f([2.62851396]) = 6.909085650835832
47 -> f([2.57368435]) = 6.623851132374277
48 -> f([2.54446085]) = 6.474281041070285
50 -> f([2.4968615]) = 6.234317339946366
51 -> f([2.43544168]) = 5.931376183590648
In [3]:
```

```
asarray = {builtin_function_or_method} <built-in function asarray>
bounds = {ndarray: (1, 2)} [[-5.  5.]] ...View as Array
n_iterations = {int} 1000
rand = {builtin_function_or_method} <built-in method rand of numpy.rand... View
randn = {builtin_function_or_method} <built-in method randn of numpy.rar... View
step_size = {float} 0.1
Special Variables
```

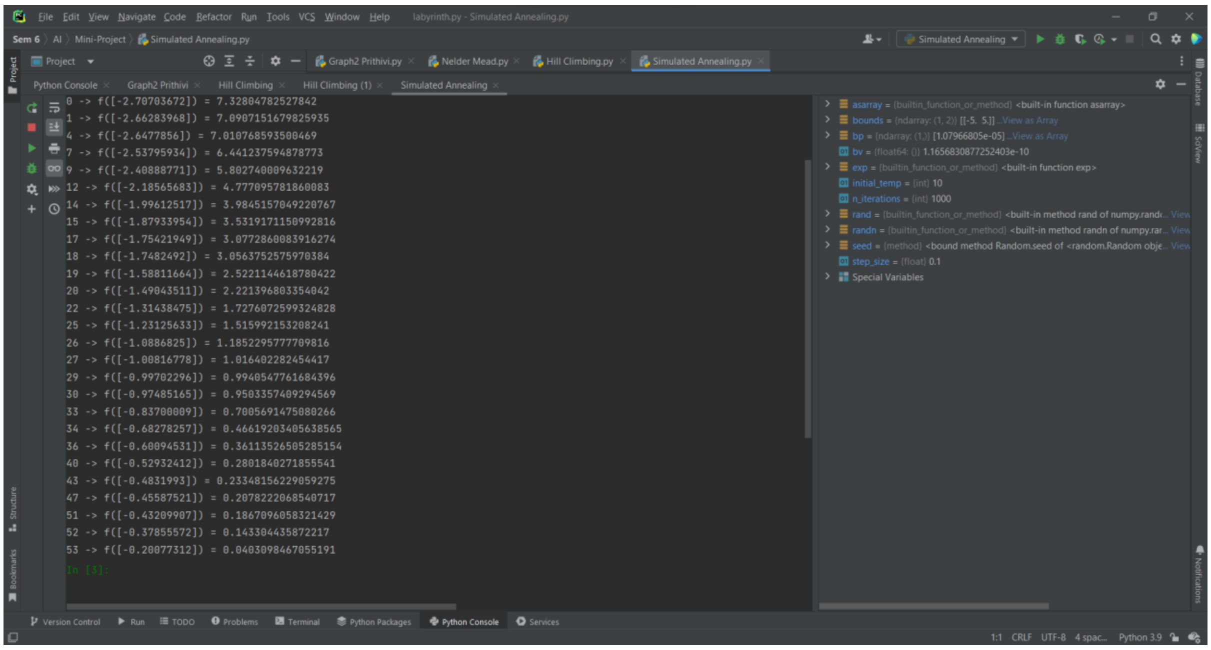
Function name should be lowercase



```
67 -> f([2.15296947]) = 4.635277535297181
69 -> f([2.11234782]) = 4.462013322537634
72 -> f([2.09736428]) = 4.398936912963218
74 -> f([2.02185323]) = 4.087890491923262
76 -> f([1.88886087]) = 3.567795404286202
78 -> f([1.7922574]) = 3.2121865934916705
79 -> f([1.56730513]) = 2.4564453679335463
81 -> f([1.53981227]) = 2.3710218293729626
82 -> f([1.33373089]) = 1.7788380751064858
83 -> f([1.29855926]) = 1.6862561620058285
84 -> f([1.09480639]) = 1.1986010412861459
85 -> f([1.07054485]) = 1.1460662709313596
87 -> f([1.01026288]) = 1.0206310805085193
94 -> f([0.82534648]) = 0.681196807669133
97 -> f([0.76458029]) = 0.5845830260220531
98 -> f([0.73483526]) = 0.539982854025602
99 -> f([0.71707777]) = 0.5142005349664515
100 -> f([0.70948015]) = 0.5033620849373985
101 -> f([0.57350271]) = 0.3289053557740241
103 -> f([0.4145834]) = 0.17187939491730397
105 -> f([0.32348817]) = 0.10464459586556404
106 -> f([0.16895745]) = 0.028546619498780006
110 -> f([0.07754722]) = 0.006013571414396759
111 -> f([-0.02606142]) = 0.0006791976326344529
125 -> f([0.02024558]) = 0.00040988347695620553
133 -> f([0.00010101]) = 1.0202327567031279e-08

In [3]:
```

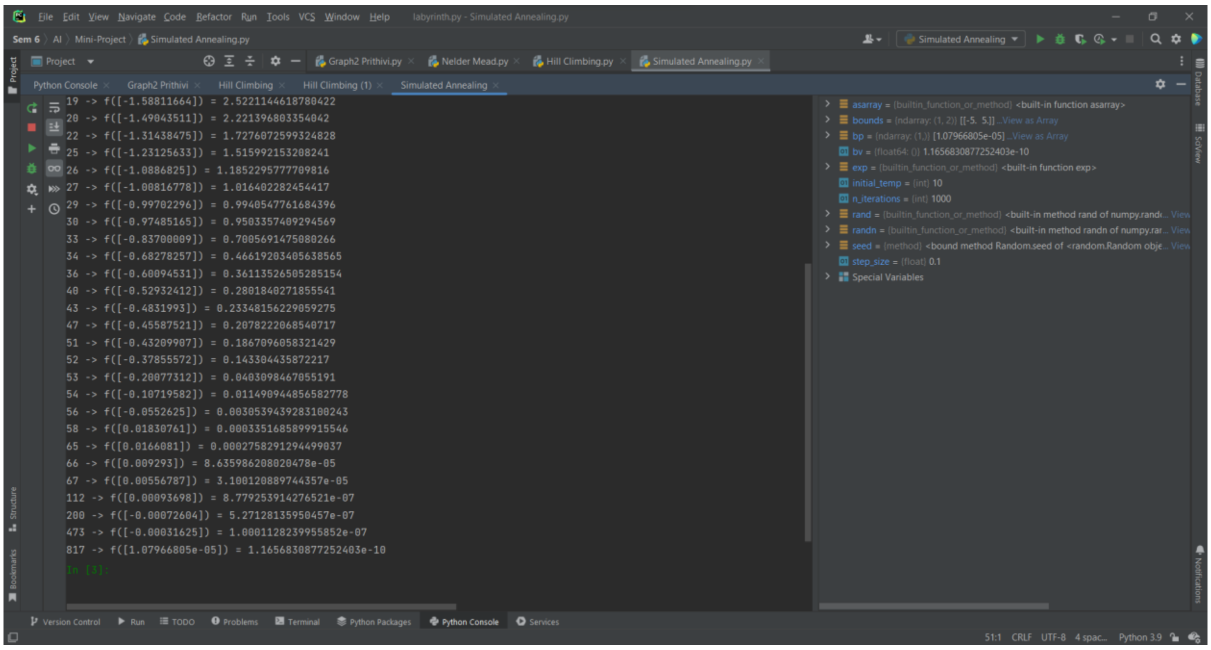
```
asarray = {builtin_function_or_method} <built-in function asarray>
bounds = {ndarray: (1, 2)} [[-5.  5.]] ...View as Array
n_iterations = {int} 1000
rand = {builtin_function_or_method} <built-in method rand of numpy.rand... View
randn = {builtin_function_or_method} <built-in method randn of numpy.rar... View
step_size = {float} 0.1
Special Variables
```

Simulated Annealing-



```
0 -> f([-2.70703672]) = 7.32804782527842
1 -> f([-2.66283968]) = 7.0907151679825935
4 -> f([-2.6477856]) = 7.010768593500469
7 -> f([-2.53795934]) = 6.441237594878773
9 -> f([-2.40888771]) = 5.802740009632219
12 -> f([-2.18565683]) = 4.777095781860083
14 -> f([-1.99612517]) = 3.9845157049220767
15 -> f([-1.87933954]) = 3.5319171150992816
17 -> f([-1.75421949]) = 3.0772860083916274
18 -> f([-1.7482492]) = 3.0563752575970384
19 -> f([-1.58811664]) = 2.5221144618780422
20 -> f([-1.49043511]) = 2.221396803354042
22 -> f([-1.31438475]) = 1.7276072599324828
25 -> f([-1.23125633]) = 1.515992153208241
26 -> f([-1.0886825]) = 1.1852295777709816
27 -> f([-1.00816778]) = 1.016402282454417
29 -> f([-0.99702296]) = 0.9940547761684396
30 -> f([-0.97485165]) = 0.9503357409294569
33 -> f([-0.83700009]) = 0.7005691475080266
34 -> f([-0.68278257]) = 0.46619203405638565
36 -> f([-0.60094531]) = 0.36113526505285154
40 -> f([-0.52932412]) = 0.2801840271855541
43 -> f([-0.4831993]) = 0.23348156229059275
47 -> f([-0.45587521]) = 0.2078222068540717
51 -> f([-0.43209907]) = 0.1867096058321429
52 -> f([-0.37855572]) = 0.143304435872217
53 -> f([-0.20077312]) = 0.0403098467055191

In [3]:
```

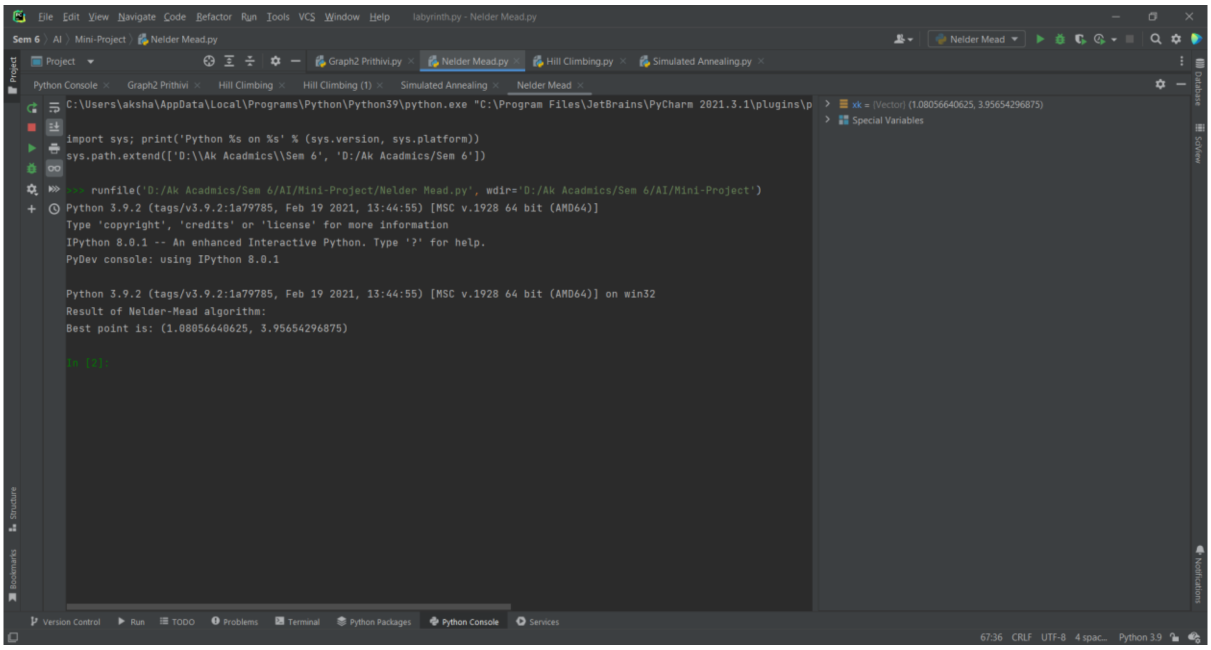
```
asarray = {builtin_function_or_method} <built-in function asarray>
bounds = {ndarray: (1, 2)} [[-5.  5.]] ...View as Array
bp = {ndarray: (1,)} [1.07966805e-05] ...View as Array
bv = {float64: ()} 1.1656830877252403e-10
exp = {builtin_function_or_method} <built-in function exp>
initial_temp = {int} 10
n_iterations = {int} 1000
rand = {builtin_function_or_method} <built-in method rand of numpy.rand... View
randn = {builtin_function_or_method} <built-in method randn of numpy.rar... View
seed = {method} <bound method Random.seed of <random.Random obje... View
step_size = {float} 0.1
Special Variables
```



```
19 -> f([-1.58811664]) = 2.5221144618780422
20 -> f([-1.49043511]) = 2.221396803354042
22 -> f([-1.31438475]) = 1.727607259932482 8
25 -> f([-1.23125633]) = 1.515992153208241
26 -> f([-1.0886825]) = 1.1852295777709816
27 -> f([-1.00816778]) = 1.016402282454417
29 -> f([-0.99702296]) = 0.9940547761684396
30 -> f([-0.97485165]) = 0.9503357409294569
33 -> f([-0.83700009]) = 0.7005691475080266
34 -> f([-0.68278257]) = 0.46619203405638565
36 -> f([-0.60094531]) = 0.36113526505285154
40 -> f([-0.52932412]) = 0.2801840271855541
43 -> f([-0.4831993]) = 0.23348156229059275
47 -> f([-0.45587521]) = 0.2078222068540717
51 -> f([-0.43209907]) = 0.1867096058321429
52 -> f([-0.37855572]) = 0.143304435872217
53 -> f([-0.20077312]) = 0.0403098467055191
54 -> f([-0.10719582]) = 0.011490944856582778
56 -> f([-0.0552625]) = 0.003053943928310024 3
58 -> f([0.01830761]) = 0.00033516858999155 46
65 -> f([0.0166081]) = 0.0002758291294499037
66 -> f([0.009293]) = 8.635986208020478e-05
67 -> f([0.00556787]) = 3.100120889744357e-05
112 -> f([0.00093698]) = 8.779253914276521e-07
200 -> f([-0.00072604]) = 5.27128135950457e-07
473 -> f([-0.00031625]) = 1.0001128239955852e-07
817 -> f([1.07966805e-05]) = 1.1656830877252403e-10

In [3]:
```

```
asarray = {builtin_function_or_method} <built-in function asarray>
bounds = {ndarray: (1, 2)} [[-5.  5.]] ...View as Array
bp = {ndarray: (1,)} [1.07966805e-05] ...View as Array
bv = {float64: ()} 1.1656830877252403e-10
exp = {builtin_function_or_method} <built-in function exp>
initial_temp = {int} 10
n_iterations = {int} 1000
rand = {builtin_function_or_method} <built-in method rand of numpy.rand... View
randn = {builtin_function_or_method} <built-in method randn of numpy.rar... View
seed = {method} <bound method Random.seed of <random.Random obje... View
step_size = {float} 0.1
Special Variables
```

Nelder-Mead algorithm-



```
C:\Users\aksha\AppData\Local\Programs\Python\Python39\python.exe "C:\Program Files\JetBrains\PyCharm 2021.3.1\plugins\p

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['D:\\Ak Acadmics\\Sem 6', 'D:/Ak Acadmics/Sem 6'])

>>> runfile('D:/Ak Acadmics/Sem 6/AI/Mini-Project/Nelder Mead.py', wdir='D:/Ak Acadmics/Sem 6/AI/Mini-Project')
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.0.1 -- An enhanced Interactive Python. Type '?' for help.
PyDev console: using IPython 8.0.1

Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Result of Nelder-Mead algorithm:
Best point is: (1.08056640625, 3.95654296875)

In [2]:
```

```
xk = {Vector} (1.08056640625, 3.95654296875)
Special Variables
```

CONCLUSION

We have learned about all informed and uninformed search algorithms, how they work, their algorithms. Also we have learned about their complexities, their optimality and completeness, and we have understood why are the constraints when the algorithms give an optimal and complete solution. With that we have also discovered in which case which algorithm will give best results.

Additionally, we have also explored about Local search and its algorithm and working, and uses, we have analysed algorithms like hill climbing, and how simulated annealing helps covering up the limitations of hill climbing algorithm.

In the implementation, we have created a graph class where we have created the different functions and helper functions, to implement all informed and uninformed search algorithms in the same code and dynamic creation of graph based on given input preferences (i.e. heuristic/non-heuristic, weighted/non-weighted, directed/undirected).

RESOURCES

- https://www.javatpoint.com/ai-uninformed-search-algorithms#:~:text=A%20depth%2Dlimited%20search%20algorithm,has%20no%20successor%20nodes%20further
- https://en.wikipedia.org/wiki/Simulated_annealing
- https://iq.opengenus.org/iterative-deepening-search/
- https://www.youtube.com/channel/UCM-yUTYGmrNvKOCcAl21g3w

Appendix I – Source Code

Informed And Uninformed search:

```
from collections import defaultdict
import pprint
from timeit import default_timer as timer

class Graph():

    '''Graph Class -> Can be directed or undirected(default)
                   -> Can be weighted or unweighted(default)
                   -> Can have a heuristic or not(default)'''

    def __init__(self, edgelist, directed=False, weighted=False,
heuristic=False):

        '''
        Initialize graph using a list of edges.
        If Unweighted-> edgelist=(Node 1, Node2)
        If Weighted-> edgelist=(Node 1, Node2, Edge Weight)

        :param edgelist: Node to start search from
        :param directed: Set if graph is directed or undirected
(default=Flase)
        :param weighted: Set if graph is weighted or unweighted
(default=Flase)
        :param heuristic: Set if graph requires a heuristic or not
(default=Flase)

        :return: None
        '''

        self.adj = defaultdict(set)
        self.directed = directed
        self.weighted = weighted
        self._initialize(edgelist) # fill out adjacency list
        if heuristic:
            self.heuristic = {}
            self.generate_heuristics() # Generate heuristic f option is set

    def _initialize(self,edgelist):

        if not self.weighted: # unweighted edges
            for source,dest in edgelist:
                self.add(source,dest)
        else:
            for source,dest,weight in edgelist: # weighted edges
                self.add(source,dest,weight)

    def add(self,node1,node2,weight=None):

        '''
        Function used to add edges to the graph.

        :param node1: Node to start search from
        :param node2: Node to search for
        :param weight: Weights of the edge for weighted Graphs
(defaut=None)

        :return: None
        '''

        if weight is None:
            self.adj[node1].add(node2)
            if not self.directed: # addedge from a-b to b-a also if graph
is undirected
                self.adj[node2].add(node1)
        else:
            self.adj[node1].add((node2,weight)) # add pairs node1 :
(node2,weight)
            if not self.directed:
                self.adj[node2].add((node2,weight))


    def generate_heuristics(self):
        # Can be modified to generate heuristic, but we will use a hard
coded one  for now
        self.heuristic = {1: 5, 2: 7, 3: 3, 4: 4, 5: 6, 6: 5, 7: 6, 8: 0,
9: 0, 10: 0}

    def bfs(self, start, target):

        '''
        Breath First Search where given a source and target node using
breath
        first method we attempt to find the path to target node.

        :param start: Node to start search from
        :param target: Node to search for

        :return: Path to node if found
        '''

        visited=set()
        q=[]
        traversal=[] # remember path taken
        visited.add(start)
        q.append(start)
        while q:
            node=q.pop(0)
            traversal.append(node)
            if node==target: # stop when goal node is reached
                return traversal
            for other in self.adj[node]: # for every neighbour if not
visited add to queue and mark visited
                if other not in visited:
                    visited.add(other)
                    q.append(other)
        return "Node not Found"

    def dfs(self, start, target):

        '''
        Deapth First Search where given a source and target node using
depth
        first method we attempt to find the path to target node.
        :param start: Node to start search from
        :param target: Node to search for

        :return: Path to node if found
        '''

        result=[]
        visited=set()
        self._helper_dfs(start,visited,result,target)
        if target in result:
            return result
        else:
            return"Node not Found"

    def _helper_dfs(self, node, visited, result, target):
        if result and result[-1]==target: # if goal node is visited then
returns
            return
        visited.add(node)
        result.append(node)

        for other in self.adj[node]: # for every neighbour if not visited
run dfs on it
            if other not in visited:
                self._helper_dfs(other,visited,result,target)

    def dls(self, start, target, max_depth):

        '''
        Deapth Limited Search where given a maximum depth path will be
        returned only if node is found at or before the maximum depth.

        :param start: Node to start search from
        :param target: Node to search for
        :param max_depth: Maximum depth to limit search space <= maximum
depth

        :return: Path to node if found
        '''

        result=[]
        visited=set()
        depth=defaultdict()
        depth[start]=0
        self._helper_dls(start,visited,result,target,depth,max_depth)
        if target in result:
            return result
        else:
            return"Node not Found"

    def _helper_dls(self, node, visited, result, target, depth, max_depth):

        if depth[node]>max_depth: # check maximum depth condition
            return
        if result and result[-1]==target: # check if goal node is found
            return
        visited.add(node)
        result.append(node)
        for other in self.adj[node]:
            if other not in visited:
                depth[other]=depth[node]+1 # keeps track of depth ->depth
```

```
of child is parent+1

self._helper_dls(other,visited,result,target,depth,max_depth)

    def iterative_deepening(self, start, target, max_depth):

        '''
        Iterative Deepening Search where given a maximum depth we will
search
        iteratively in every depth from 0-max_depth and return path if
found.

        :param start: Node to start search from
        :param target: Node to search for
        :param max_depth: Maximum depth to limit search space <= maximum
depth

        :return: Path to node if found
        '''

        flag=0
        for current_depth in range(max_depth+1): # for every depth from 0
to maximum depth run a depth limited search
            result=self.dls(start,target,current_depth)
            if not isinstance(result,str):
                return result
        return "Node not Found"

    def bi_directional(self):
        pass

    def Astar(self,start, target):
        result = set()
        visited = set()
        pqueue = []
        star= defaultdict()
        star[start]= 0
        target_found =False
        self._helper_Astar(star, visited, start, target, pqueue, result,
target_found)
        if target in result:
            return result
        else:
            return"Node not Found"

    def _helper_Astar(self, star, visited, node, target, pqueue, result,
target_found):
        if target_found:
            return
        visited.add(node)
        result.add(node)
        for other in self.adj[node]:
            if other in visited:
                continue
            visited.add(other)
            pqueue.append(other)
            if target == other[0]:
                target_found = True
                return

        while len(pqueue)!=0:
            x = min(pqueue)[1]
            for i in range(len(pqueue)):
                if pqueue[i][1]== x:
                    x=i
                    break
            if pqueue[x] in result: result.pop(pqueue[x])
            node2 = pqueue.pop(x)
            # if node2 in visited: continue
            self._helper_Astar(star, visited, node2[0], target, pqueue,
result, target_found)


    def __str__(self):
        return f'{dict(self.adj)}'



'''
    2
   /
  7 -- 10
 / \
1    6 --    5
 \    \
  8    11
   \
    9
     \
      4
'''

edgelist = [(1,7),(7,2),(7,10),(7,6),(6,5),(6,11),(1,8),(8,9),(9,4)]

# https://media.cheggcdn.com/media/cb6/cb6e15aa-4815-47a4-b2ff-
ca9086722f40/phpj6mHyD
# s1  a2  b3  c4  d5  e6  f7  g1 8  g2 9  g3 10

#
edgelist=[(1,2,5),(1,3,9),(1,5,6),(2,3,3),(2,8,9),(3,2,2),(3,4,2),(4,1,6),(
4,7,7),(4,9,5),(5,1,1),(5,4,1),(5,6,2),(6,10,7),(7,5,2),(7,10,8),(8,9,0)]
# graph=Graph(edgelist, True, True, True)

graph=Graph(edgelist)
printer=pprint.PrettyPrinter()
printer.pprint(dict(graph.adj))
# print(graph)

'''
start = timer()

bfs=graph.bfs('A','K')

end = timer()
print(end - start)
print(bfs)
'''

print('\nPath Algorithm takes until it reaches target: \nBreath First
Search (BFS)')
print(graph.bfs(1,4))
print('\nPath Algorithm takes until it reaches target: \nDepth First Search
(DFS)')
print(graph.dfs(1,4))

print('\nPath Algorithm takes until it reaches target: \nDepth Limit Search
(DLS)')
print(graph.dls(1,11, 4))

print('\nPath Algorithm takes until it reaches target: \nIterative
Deepening (IDDFS)')
print(graph.iterative_deepening(1,11, 4))

# print('\nPath Algorithm takes until it reaches target: \nA-star (A*)')
# print(graph.Astar(1,10))
```

Hill Climbing:

```
from numpy import asarray
from numpy.random import randn, rand


def objective(x):
    return x[0] ** 2.0


def hillclimbing(objective, bounds, n_iterations, step_size):
    initial_point = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] -
bounds[:, 0])
    initial_val = objective(initial_point)
    for i in range(n_iterations):
        new_point = initial_point + randn(len(bounds)) * step_size
        new_val = objective(new_point)
        if new_val <= initial_val:
            initial_point, initial_val = new_point, new_val
            print(f'{i} -> f({initial_point}) = {initial_val}')
    return [initial_point, initial_val]


def Evolution_Global_Optimization():
    pass


def Basin_Hopping_Optimization():
    pass


def Simulated_Annealing():
    pass


def BFGS():
    pass


def Nelder_Mead():
    pass
```

```
def Particle_Swarm_Optimization():
    pass


def Simple_Genetic_Algorithm():
    pass


bounds = asarray([[-5.0, 5.0]])
n_iterations = 1000
step_size = 0.1
hillclimbing(objective, bounds, n_iterations, step_size)
```

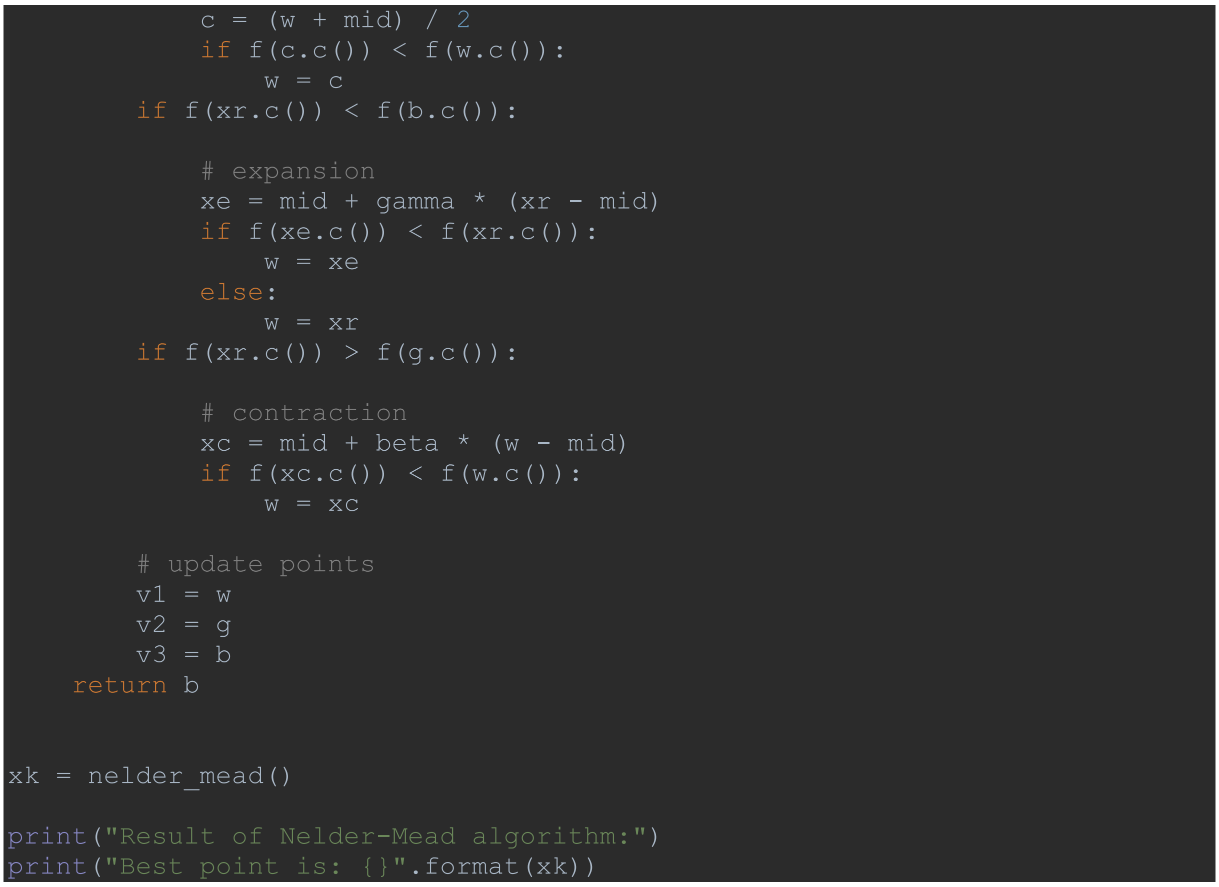
Simulated Annealing:

```
from numpy.random import rand,randn
from numpy import asarray
from random import seed
from math import exp

def objective(x):
    return x[0]**2.0

def annealer(objective, bounds, n_iterations, step_size,initial_temp):
    best_point=bounds[:,0] + rand(len(bounds)) * (bounds[:,1] - 
bounds[:,0])
    best_value=objective(best_point)
    current_point,current_value=best_point,best_value

    for i in range(n_iterations):
        new_point=current_point + randn(len(bounds)) * step_size
        new_value=objective(new_point)

        if new_value < best_value:
            best_point,best_value=new_point,new_value
            print(f'{i} -> f({best_point}) = {best_value}')

        difference=new_value-current_value
        temperature=initial_temp/(i+1)
        metropolis_acceptance=exp(-difference/temperature)
        if difference < 0 and rand() < metropolis_acceptance:
            current_point,current_value=new_point,new_value
    return [best_point,best_value]

bounds = asarray([[-5.0, 5.0]])
n_iterations = 1000
step_size = 0.1
initial_temp=10
bp,bv=annealer(objective, bounds, n_iterations, step_size,initial_temp)
print(bp,bv)
```

Nelder Mead:

```


class Vector(object):
```

```python
    def __init__(self, x, y):
        """ Create a vector, example: v = Vector(1,2) """
        self.x = x
        self.y = y

    def __repr__(self):
        return "({0}, {1})".format(self.x, self.y)

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __sub__(self, other):
        x = self.x - other.x
        y = self.y - other.y
        return Vector(x, y)

    def __rmul__(self, other):
        x = self.x * other
        y = self.y * other
        return Vector(x, y)

    def __truediv__(self, other):
        x = self.x / other
        y = self.y / other
        return Vector(x, y)

    def c(self):
        return (self.x, self.y)


# objective function
def f(point):
    x, y = point
    return x ** 2 + x * y + y ** 2 - 6 * x - 9 * y


def nelder_mead(alpha=1, beta=0.5, gamma=2, maxiter=10):
    # initialization
    v1 = Vector(0, 0)
    v2 = Vector(1.0, 0)
    v3 = Vector(0, 1)

    for i in range(maxiter):
        adict = {v1: f(v1.c()), v2: f(v2.c()), v3: f(v3.c())}
        points = sorted(adict.items(), key=lambda x: x[1])

        b = points[0][0]
        g = points[1][0]
        w = points[2][0]

        mid = (g + b) / 2

        # reflection
        xr = mid + alpha * (mid - w)
        if f(xr.c()) < f(g.c()):
            w = xr
        else:
            if f(xr.c()) < f(w.c()):
                w = xr
```



```
            c = (w + mid) / 2
            if f(c.c()) < f(w.c()):
                w = c
        if f(xr.c()) < f(b.c()):

            # expansion
            xe = mid + gamma * (xr - mid)
            if f(xe.c()) < f(xr.c()):
                w = xe
            else:
                w = xr
        if f(xr.c()) > f(g.c()):

            # contraction
            xc = mid + beta * (w - mid)
            if f(xc.c()) < f(w.c()):
                w = xc

        # update points
        v1 = w
        v2 = g
        v3 = b
    return b


xk = nelder_mead()

print("Result of Nelder-Mead algorithm:")
print("Best point is: {}".format(xk))
```

Appendix II – GitHub

https://github.com/akshatgaur2/Searching-Algorithms-in-AI-Comparision