Assignment – Nirbhay Gandhi

Searching Methods

Index	
	Page No.

GitHub link: https://github.com/Nirbhay-Gandhi/Python-Programming/tree/Prod/Assignment

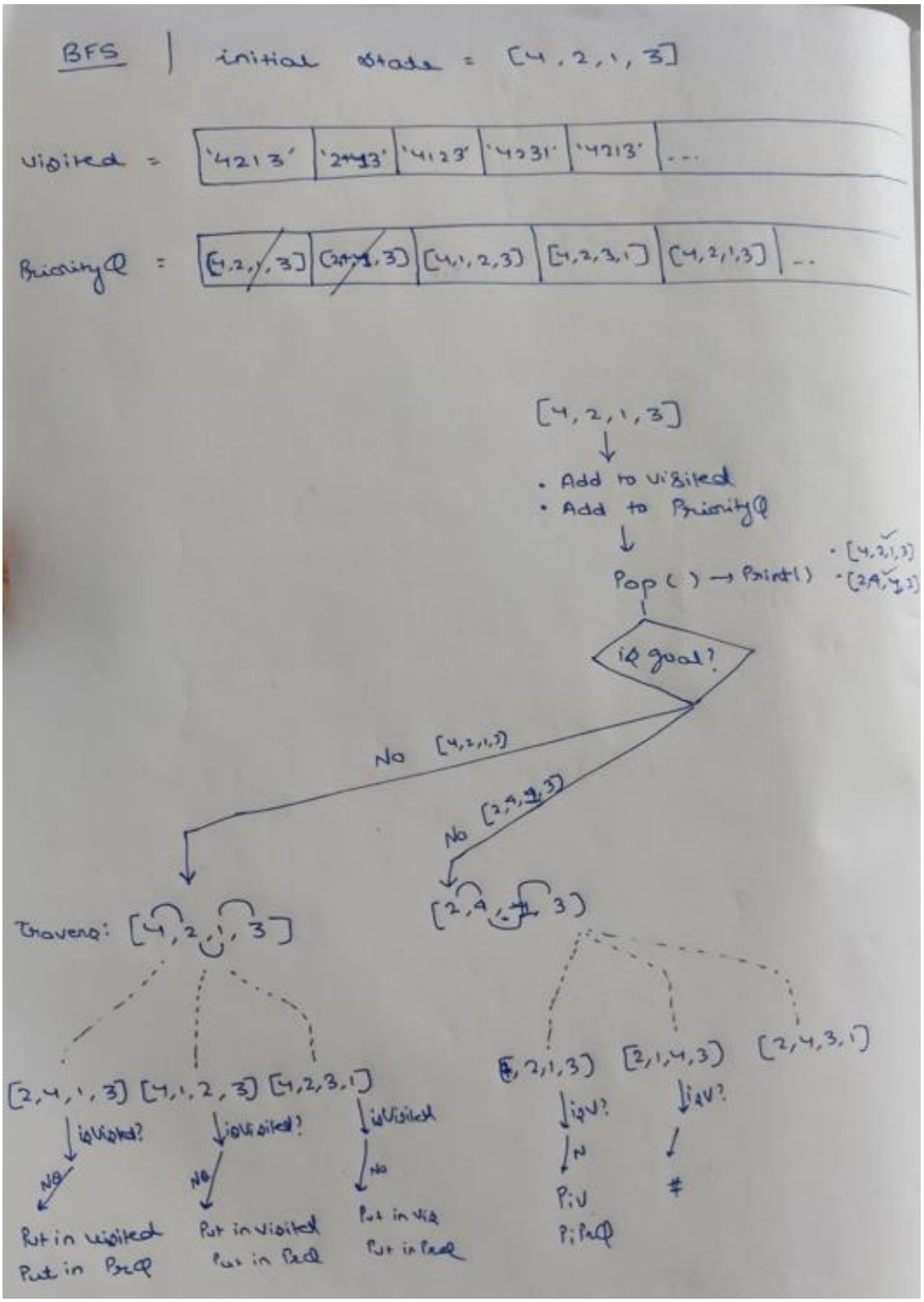
Initial Considerations

- **start_state :** unsorted list of double type (user input)
- **successive_state :** list with swapped neighboring element
- **goal_state :** sorted list in ascending order
- **cost to travel a successive state :** 1 unit per action

Data Structures in use

- Two data structures in use :
  - Queue - Replicated by List
  - Visited Array – To store all the visited states, replicated by using Set

Algorithms Overview



1. Breadth First Search (BFS) – Uninformed searching techniques

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Queue: In BFS to add the element in the Queue, we will simple **Push initial_state** into the **queue** any mark it visited
3. Removal of the element from Queue: In BFS for popping (removal) of the element, will be simply based on the **FIFO(First in First out) principle.**

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

****Note:** A similar approach of algorithm will be followed in rest all the Searching algorithms.
Only difference will be in Step-2 (Addition of Element) and Step-3 (Removal of Element). Different methods are followed that will illustrate in that particular algorithm section.

2. Depth First Search (DFS) – Uninformed searching techniques

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Stack: In DFS to add the element in the Stack, we will simple **Push initial_state** into the **stack** any mark it visited.
3. Removal of the element from Stack: In DFS for popping (removal) of the element, will be based on the **LIFO(Last in First out) principle.**

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

3. Uniform Cost Search (UFS) – Uninformed searching techniques

- In uniform cost search, we always use the path that has the least total cost to reach the goal state. We blindly choose the action that gives us the immediate minimum cost

-> how we'll find the optimal cost path?

we attch a priority with every state. that priority will act as a action cost required from one state to next state

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Queue: In UFS to add the element in the Queue, we will simple **Push initial_state along with priority as tuple** into the **queue** any mark it visited.
3. Removal of the element from Queue: In UFS for popping (removal) of the element, will be based on the **least priority first out principle.** Where the popped priority element, will be our operations cost.

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

4. Greedy Best First Search (Greedy) – Informed searching techniques

- In uniform cost search, we always use the path that has the least total cost to reach the goal state. We blindly choose the action that gives us the immediate minimum cost

-> how we'll find the optimal cost path?

we attch a priority with every state. that priority will act as a action cost required from one state to next state

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Queue: In Greedy to add the element in the Queue, we will simple **Push initial_state along with its priority defined by heuristic as tuple** into the **queue** any mark it visited.
3. Removal of the element from Queue: In Greedy for popping (removal) of the element, will be based on the **least heuristic first out principle.** Where the popped priority element, will be our operations cost.

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

4. Greedy Best First Search (Greedy) – Informed searching techniques

- In uniform cost search, we always use the path that has the least total cost to reach the goal state. We blindly choose the action that gives us the immediate minimum cost

-> how we'll find the optimal cost path?

we attch a priority with every state. that priority will act as a action cost required from one state to next state

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Queue: In Greedy to add the element in the Queue, we will simple **Push initial_state along with its priority defined by heuristic as tuple** into the **queue** any mark it visited.
3. Removal of the element from Queue: In Greedy for popping (removal) of the element, will be based on the **least heuristic first out principle.** Where the popped priority element, will be our operations cost.

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

5. S star Search (A*) – Informed searching techniques

f(n) = h(n)+g(n)

g(n) = actual cost from start node to n

h(n) = estimation cost from n to goal node

- here, every node will have 2 properties:

(a) actual cost, which we will keep the priority over here

(b) the heuristic value, that we'll calculate in every step

- jo action hume min f(n) deta hai, hum usko expand karte hai

Algorithm

1. **Queue & Visited array** both to be initialized **empty.**

2. Adding of the element in Queue: In Greedy to add the element in the Queue, we will simple **Push f(n) along with its priority defined by heuristic as tuple** into the **queue** any mark it visited.
   f(n) initially will be heuristic(n)+operational cost. Operational cost will be 0 initilaly.
3. Removal of the element from Queue: In Greedy for popping (removal) of the element, will be based on the **least f(n) first out principle.** Where the popped priority element, will be our operations cost.

4. **Check**, is the given_state equal to the **goal state**, if so algorithm comes to end. Else proceed next steps
5. **Generate the neighboring states** : Each neighboring state is generated by **swapping the ith and (i+1)th element** of the initial_state.
6. Visit the **unvisited neighbors.**
7. Repeat from step **2**.

6. Hill Climb Search (HC) – Informed searching techniques

we can proceed ahed only, if the state is better than the current state

Algorithm

1. **Neighbor array** to be initialized **empty.**
2. **Visit all the successors** and **append it** into the neighbors array.
3. **Select** that neighbor which is giving us the **least heuristic** value.
4. **Proceed ahead** only if the **heuristic** of the **best selected neighbor** is **less than** the **heuristic** of the **current state**.
5. Stop the algorithm if heuristic of the best selected neighbor in step-4 is greater than the heuristic of current neighbor.

Helper Functions Overview

1. swap(array, i, j) : swaps arr[i] with arr[j]

2. is_goalstate(input_arr) : is input_arr ==? Sorted(input_arr)

3. Heuristic Function Calculation

Heuristic function calculated on the basis of the **Manhatten Distance method**.

Algorithm

- Create a copy of the input state and sort it.
- Compute the heuristic sum by iterating through each element in the original state:
    - For each element, calculate the absolute difference between its index in the sorted state and its index in the original state.
    - Sum up these absolute differences to get the heuristic sum.
- Return the heuristic sum.

4. Finding the best Heuristic neighbor

Algorithm

- create list of heuristics from the input list of neighbors <List<List>>, which contains the heuristic of all the neighbors.
- find the index of the min heuristic value from the heuristics as min_index.
- return the element from neighbor at index min_index.

5. Remove Duplicates

Removes duplicates from the array, by preserving the order of the elements.

Results

```
PS D:\DELL\DSA\DSA Questions\Python DSA practice\Python-Programming> & C:
stions/Python DSA practice/Python-Programming/Assignment/MainApp.py"
Enter Array: 4 4 6.3 9 -3
Searching Methods
1. Breadth First Search:
2. Depth First Search:
3. Uniform First Search:
4. Greedy Best First Search:
5. A* Search:
6. Hill Climb Search:
7. All Algorithm Compairison
Algorithm Choice:-
```

Breadth First Search

```
Breadth First Search
[0]: [4.0, 6.3, 9.0, -3.0]
[1]: [6.3, 4.0, 9.0, -3.0]
[2]: [4.0, 9.0, 6.3, -3.0]
[3]: [4.0, 6.3, -3.0, 9.0]
[4]: [6.3, 9.0, 4.0, -3.0]
[5]: [6.3, 4.0, -3.0, 9.0]
[6]: [9.0, 4.0, 6.3, -3.0]
[7]: [4.0, 9.0, -3.0, 6.3]
[8]: [4.0, -3.0, 6.3, 9.0]
[9]: [9.0, 6.3, 4.0, -3.0]
[10]: [6.3, 9.0, -3.0, 4.0]
[11]: [6.3, -3.0, 4.0, 9.0]
[12]: [9.0, 4.0, -3.0, 6.3]
[13]: [4.0, -3.0, 9.0, 6.3]
[14]: [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```

Depth First Search

```
Depth First Search
[0]: [4.0, 6.3, 9.0, -3.0]
[1]: [4.0, 6.3, -3.0, 9.0]
[2]: [4.0, -3.0, 6.3, 9.0]
[3]: [4.0, -3.0, 9.0, 6.3]
[4]: [4.0, 9.0, -3.0, 6.3]
[5]: [9.0, 4.0, -3.0, 6.3]
[6]: [9.0, 4.0, 6.3, -3.0]
[7]: [9.0, 6.3, 4.0, -3.0]
[8]: [9.0, 6.3, -3.0, 4.0]
[9]: [9.0, -3.0, 6.3, 4.0]
[10]: [-3.0, 9.0, 6.3, 4.0]
[11]: [-3.0, 9.0, 4.0, 6.3]
[12]: [-3.0, 6.3, 9.0, 4.0]
[13]: [-3.0, 6.3, 4.0, 9.0]
[14]: [6.3, -3.0, 4.0, 9.0]
[15]: [6.3, -3.0, 9.0, 4.0]
[16]: [6.3, 9.0, -3.0, 4.0]
[17]: [6.3, 9.0, 4.0, -3.0]
[18]: [9.0, -3.0, 4.0, 6.3]
[19]: [-3.0, 4.0, 9.0, 6.3]
[20]: [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```

Uniform First Search

```
Algorithm Choice:- 3
Uniform Cost Search
[0]: Cost 0  [4.0, 6.3, 9.0, -3.0]
[1]: Cost 1  [4.0, 6.3, -3.0, 9.0]
[2]: Cost 1  [4.0, 9.0, 6.3, -3.0]
[3]: Cost 1  [6.3, 4.0, 9.0, -3.0]
[4]: Cost 2  [4.0, -3.0, 6.3, 9.0]
[5]: Cost 2  [4.0, 9.0, -3.0, 6.3]
[6]: Cost 2  [6.3, 4.0, -3.0, 9.0]
[7]: Cost 2  [6.3, 9.0, 4.0, -3.0]
[8]: Cost 2  [9.0, 4.0, 6.3, -3.0]
[9]: Cost 3  [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```


Greedy Best First Search

```
Algorithm Choice:- 5
A star Search
[0]: Cost 0  [4.0, 6.3, 9.0, -3.0]
[1]: Cost 1  [4.0, 6.3, -3.0, 9.0]
[2]: Cost 2  [4.0, -3.0, 6.3, 9.0]
[3]: Cost 3  [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```

A* Search

```
Algorithm Choice:- 6
Hill Climb Search
[0]: [4.0, 6.3, 9.0, -3.0]
[1]: [4.0, 6.3, -3.0, 9.0]
[2]: [4.0, -3.0, 6.3, 9.0]
[3]: [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```

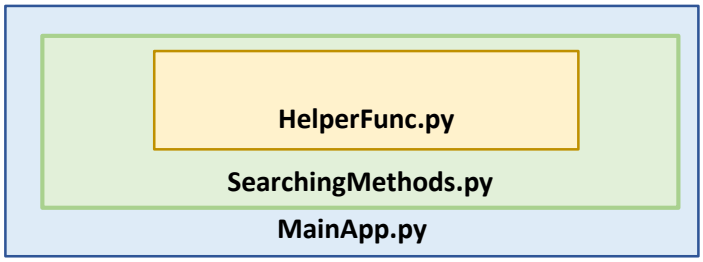
Prod* ⊗ 0 ⚠ 0 0

Hill Climb Search

```
Algorithm Choice:- 6
Hill Climb Search
[0]: [4.0, 6.3, 9.0, -3.0]
[1]: [4.0, 6.3, -3.0, 9.0]
[2]: [4.0, -3.0, 6.3, 9.0]
[3]: [-3.0, 4.0, 6.3, 9.0]
-1 to terminate, anyother key to continue:
```

Codes

Code flow:



Module: HelperFunctions.py

```
class HelperFunc:

    def swap(lst, i, j):
        lst[i], lst[j] = lst[j], lst[i]

    def is_goalstate(lst):
        lst_sorted = sorted(lst)
        return lst == lst_sorted

    def list_to_str(lst):
        temp = list(map(str, lst))
        str_joined = ','.join(temp)
        return str_joined

    """
    Heuristic function calculated on the basis of the Manhatten Distance method:
    means, it is the cummulative of how much distance each element has to travell inorder
to reach
    the correct place
    """
    def heuristic(state):
        sorted_sate = state[:]
        sorted_sate.sort()
        heuristic_sum = 0
        for element in state:
            heuristic_sum = heuristic_sum + abs(sorted_sate.index(element) -
state.index(element))
        return heuristic_sum

    def cumm_heuristic(state):
        return HelperFunc.heuristic(state)

    def min_heurstic_neighbour(neighbours):
        heuristics = list(map(HelperFunc.cumm_heuristic,neighbours))
        min_index = heuristics.index(min(heuristics))
        return neighbours[min_index]
```

Module: SearchingMethods.py

```
from HelperFunctions import HelperFunc as Hf
from queue import PriorityQueue

class UnInformedSearchMethods:
    def Breadth_first_search(initial_state):
        print("Breadth First Search")
        visited = set()
        queue = []

        queue.append(initial_state)
        visited.add(Hf.list_to_str(initial_state))

        iteration = 0
        while queue:
            #pop the queue element
            curr_state = queue.pop(0)
            print(f"[{iteration}]: {curr_state}")

            if Hf.is_goalstate(curr_state):
                return

            for i in range(len(curr_state) - 1):
                next_state = curr_state[:]
                Hf.swap(next_state, i, i + 1)
                if Hf.list_to_str(next_state) not in visited:
                    #visit the unvisited neighbours & put them into the queue
                    visited.add(Hf.list_to_str(next_state))
                    queue.append(next_state)
            iteration +=1
        return iteration


    def Depth_first_search(initial_state):
        print("Depth First Search")
        visited = set()
        stack = []

        stack.append(initial_state)
        visited.add(Hf.list_to_str(initial_state))

        iteration = 0
        while stack:
            #pop the queue element
            curr_state = stack.pop()
            print(f"[{iteration}]: {curr_state}")

            if Hf.is_goalstate(curr_state):
                return

            for i in range(len(curr_state) - 1):
                next_state = curr_state[:]
                Hf.swap(next_state, i, i + 1)
```

```
                if Hf.list_to_str(next_state) not in visited:
                    #visit the unvisited neighbours & put them into the queue
                    visited.add(Hf.list_to_str(next_state))
                    stack.append(next_state)
            iteration +=1
        return iteration


    """
    - uniform cost search me hum goal state par reach karne ke liye hamesha wahi path use
karte hai,
    jiska path cost sabse jyda kam hota hai
    - blindly jo action hume immediate min cost deta hai, hum usko choose karte hai

    -> how we'll find the optimal cost path?
    we attch a priority with every state. that priority will act as a action cost required
from one state to
    next state
    """
    def Uniform_cost_search(initial_state):
        print("Uniform Cost Search")
        visited = set()
        priority_queue = PriorityQueue()

        priority_queue.put((0, initial_state))
        visited.add(Hf.list_to_str(initial_state))

        total_cost = 0
        iteration = 0
        while priority_queue:
            opern_cost, curr_state = priority_queue.get()
            print(f"[{iteration}]: Cost {opern_cost}  {curr_state}")

            if Hf.is_goalstate(curr_state):
                return

            #exploring out successors steps
            for i in range(len(curr_state) - 1):
                next_state = curr_state[:]
                Hf.swap(next_state, i, i + 1)
                if Hf.list_to_str(next_state) not in visited:
                    total_cost += 1  #Cost: 1 unit per action (given)
                    #visit the unvisited neighbours & put them into the queue
                    visited.add(Hf.list_to_str(next_state))
                    priority_queue.put((opern_cost+1, next_state))
            iteration+=1
        return iteration


class InformedSearchMethods:
    def Greedy_best_first_search(initial_state):
        print("Greedy Best First Search")
        visited = set()
        priority_queue = PriorityQueue()
```

```
    priority_queue.put((Hf.heuristic(initial_state), initial_state))
    visited.add(Hf.list_to_str(initial_state))

    iteration = 0
    while priority_queue:
        heurstc, curr_state = priority_queue.get()
        print(f"[{iteration}]: {curr_state}")

        if Hf.is_goalstate(curr_state):
            return

        for i in range(len(curr_state) - 1):
            next_state = curr_state[:]
            Hf.swap(next_state, i, i + 1)
            if Hf.list_to_str(next_state) not in visited:
                #visit the unvisited neighbours & put them into the queue
                visited.add(Hf.list_to_str(next_state))
                priority_queue.put((Hf.heuristic(next_state), next_state))
        iteration+=1
    return iteration


"""
f(n) = h(n)+g(n)
g(n) = actual cost from start node to n
h(n) = estimation cost from n to goal node

- here, every node will have 2 properties:
(a) actual cost, which we will keep the priority over here
(b) the heuristic value, that we'll calculate in every step

- jo action hume min f(n) deta hai, hum usko expand karte hai
"""
def A_star_search(initial_state):
    print("A star Search")
    visited = set()
    priority_queue = PriorityQueue()

    priority_queue.put((Hf.heuristic(initial_state), 0, initial_state))
    visited.add(Hf.list_to_str(initial_state))

    iteration = 0
    while not priority_queue.empty():
        heurstc, opern_cost, curr_state = priority_queue.get()
        print(f"[{iteration}]: Cost {opern_cost}  {curr_state}")

        if Hf.is_goalstate(curr_state):
            return

        for i in range(len(curr_state) - 1):
            next_state = curr_state[:]
            Hf.swap(next_state, i, i + 1)
            if Hf.list_to_str(next_state) not in visited:
```

```
                    #visit the unvisited neighbours & put them into the queue
                    visited.add(Hf.list_to_str(next_state))
                    nextOpern = opern_cost + 1
                    priority_queue.put((nextOpern+Hf.heuristic(next_state), nextOpern,
next_state))
            iteration+=1


    """
    proceed ahed with the best heuristic neighbor only
    """
    # Hill-Climbing Search
    def Hill_climb_search(state):
        print("Hill Climb Search")
        iteration = 0
        while True:
            print(f"[{iteration}]: {state}")
            neighbors = []

            for i in range(len(state) - 1):
                next_state = state[:]
                Hf.swap(next_state, i, i + 1)
                neighbors.append(next_state)

            best_neighbor_state = Hf.min_heurstic_neighbour(neighbors)
            #we can proceed ahed only, if the state is better than the current state
            if Hf.heuristic(best_neighbor_state) >= Hf.heuristic(state):
                break
            state = best_neighbor_state
            iteration +=1


def main():
    str_nums = input()
    nums = str_nums.split()
    start_state = [float(num) for num in nums]
    print("Breadth First Search:")
    UnInformedSearchMethods.Breadth_first_search(start_state)
    print("Depth First Search:")
    UnInformedSearchMethods.Depth_first_search(start_state)
    print("Uniform First Search:")
    UnInformedSearchMethods.Uniform_cost_search(start_state)
    print("Greedy Best First Search:")
    InformedSearchMethods.Greedy_best_first_search(start_state)
    print("A* Search:")
    InformedSearchMethods.A_star_search(start_state)
    print("Hill Climb Search:")
    InformedSearchMethods.Hill_climb_search(start_state)


if __name__ == "__main__":
    main()
```

Module: SearchingMethods.py

```
from SearchingMethods import InformedSearchMethods as ISM
from SearchingMethods import UnInformedSearchMethods as USM
from HelperFunctions import HelperFunc as Hf

choice = 0
str_nums = input("Enter Array: ")
nums = str_nums.split()
start_state = [float(num) for num in nums]
start_state = Hf.remove_duplicates(start_state)

while(choice != -1):
    print("Searching Methods")
    print("1. Breadth First Search:")
    print("2. Depth First Search:")
    print("3. Uniform First Search:")
    print("4. Greedy Best First Search:")
    print("5. A* Search:")
    print("6. Hill Climb Search:")
    print("7. All Algorithm Compairison")

    choice = int(input("Algorithm Choice:- "))

    match choice:
        case 1:
            USM.Breadth_first_search(start_state)
        case 2:
            USM.Depth_first_search(start_state)
        case 3:
            USM.Uniform_cost_search(start_state)
        case 4:
            ISM.Greedy_best_first_search(start_state)
        case 5:
            ISM.A_star_search(start_state)
        case 6:
            ISM.Hill_climb_search(start_state)
        case _:
            print("Invalid choice")

    choice = int(input("-1 to terminate, anyother key to continue: "))
```