Lab Report
Assignment - 1

Ashutosh B21AI007

Question 1) :

Here we are implementing the search algorithms each of which are explained below :

Breadth First Search :

The algorithm begins by initializing a queue to store different states of the input vector and a set to keep track of visited states. Starting with the initial state, which is the input vector itself, the BFS algorithm explores neighboring states by iteratively swapping adjacent elements in the vector. The process continues until a sorted state is encountered, at which point the algorithm terminates. Throughout the traversal, the number of steps taken is recorded. This algorithm ensures that all possible permutations of the input vector are explored systematically while avoiding revisiting previously examined states, thanks to the use of a set to store visited states. The result of the algorithm is the number of steps required to sort the input vector, providing valuable insights into the complexity of the sorting process for different initial configurations of the vector.

Depth First Search :

The DFS approach differs from the Breadth-First Search (BFS) algorithm previously discussed in that it explores the state space by diving deep into a branch of possibilities before backtracking to explore other branches. The algorithm initializes a stack to maintain the state of the input vector and a set to record visited states. Starting with the initial state, which is the input vector itself, the DFS algorithm explores neighboring states by iteratively swapping adjacent elements in the vector, proceeding until a sorted state is encountered or all possibilities are exhausted. The number of steps taken is recorded throughout the process. Like BFS, DFS ensures that duplicate states are not revisited by using a set to store visited states. The result of the algorithm is the number of steps required to sort the input vector, shedding light on the intricacies of the sorting process for various initial vector configurations.

Iterative Deepening:

The algorithm utilizes a stack to maintain the state of the input vector, where each state is associated with a maximum exploration depth. A set is used to record visited states, ensuring that the algorithm avoids revisiting states during exploration.

The iterative deepening component dynamically increases the maximum depth, allowing the algorithm to explore deeper levels of the state space incrementally. At each iteration, the algorithm checks if the vector is sorted. If a sorted state is found or the maximum depth is reached, the algorithm terminates. Throughout the process, the number of steps taken is counted. This approach combines the advantages of DFS for exhaustive exploration with the systematic incrementation of depth to efficiently find the solution. The result of the algorithm is the number of steps required to sort the input vector, providing insights into the complexity of the sorting process for varying levels of exploration depth.

Uniform Cost Search :

The algorithm employs a priority queue to manage states, where each state is associated with a cost value. The priority queue ensures that states with lower costs are explored first. The algorithm initializes with the input vector and its cost set to zero, and a set to keep track of visited states. It systematically explores states by dequeuing the state with the lowest cost from the priority queue, swaps adjacent elements to generate new states, and calculates the cost for each state accordingly. Duplicate states are avoided using the visited state set. The algorithm continues until a sorted state is encountered, at which point it terminates, recording the number of steps taken. The result of the algorithm is the number of steps required to sort the input vector, providing insights into the sorting process's complexity while taking into account the cost associated with each state.

Heuristic function 1 :

This heuristic function serves as an informative measure of how far the input vector is from being fully sorted. The function operates by initializing a count variable to zero and employing two nested loops to systematically compare pairs of elements within the vector. The outer loop iterates over all elements from the beginning of the vector, while the inner loop iterates over the subsequent elements to the right. For each pair of elements compared, if the element on the right is smaller than the element on the left, the count variable is incremented. Ultimately, the function returns the count of inversions, providing a heuristic estimate of the vector's disorderliness. This heuristic can be particularly useful in informed search algorithms, such as A* search, where it guides the search process towards states that are closer to being sorted and away from states with a higher number of inversions.

The heuristic function heuristic_function1 is admissible because it yields a lower bound estimate of the cost to reach the goal state by counting the number of inversions in the input vector, a measure that inherently underestimates the true cost, as each inversion requires at least one swap. Moreover, assuming a cost of 1 for swapping neighboring elements is consistent with admissibility, as it ensures a conservative estimate of the actual cost in the context of search algorithms like A*, where admissible heuristics guide the search optimally by never overestimating the true cost to reach the goal.

Heuristic Function 2 :

This heuristic function has been used in hill climbing search where we are checking that the element is greater than its next element or not .
heuristic_function2 is an admissible heuristic as it yields a lower bound estimate of the cost to reach the goal state by counting the number of elements in the input vector that are out of their sorted order, which inherently underestimates the true cost, with each out-of-order element representing at least one swap. Additionally, assuming a cost of 1 for swapping neighboring elements maintains consistency with admissibility, as it guarantees that the heuristic remains optimistic and never overestimates the actual cost in the context of search algorithms like A*.

Greedy Search :

The Greedy Search paradigm, in this context, selects the most promising state for exploration at each step based on a heuristic estimate of how close that state is to the goal. The algorithm initializes a priority queue, utilizing a min-heap structure to prioritize states with the lowest estimated cost. The heuristic function heuristic_function1 is employed to estimate the number of inversions in the input vector, representing the number of adjacent elements that are out of their sorted order. This heuristic provides an underestimation of the true cost, ensuring admissibility, as each inversion necessitates at least one swap. The algorithm proceeds by dequeuing the state with the lowest estimated cost, exploring neighboring states obtained by swapping adjacent elements, and calculating new heuristic values for these states. Duplicate states are prevented through a set of visited states. The algorithm terminates when the goal state, a fully sorted vector, is reached, and the number of steps taken is recorded. The result is a count of steps needed to achieve the sorted state, and this information offers insights into the efficiency and effectiveness of Greedy Search in solving the sorting problem with the aid of a heuristic function.

A star Search :

The algorithm initializes a priority queue, utilizing a min-heap structure, to prioritize states based on their estimated total cost (heuristic cost plus actual cost). The heuristic function heuristic_function1 is employed to estimate the number of inversions in the input vector, which serves as an optimistic lower bound on the true cost, as each inversion requires at least one swap. The algorithm begins with an initial state represented by the input vector, an initial cost of 0, and the sum of the heuristic and cost as the priority key in the priority queue. Duplicate states are prevented through a set of visited states. The algorithm proceeds by dequeuing the state with the lowest estimated total cost, exploring neighboring states obtained by swapping adjacent elements, and calculating new costs and priorities. It terminates when the goal state, a fully sorted vector, is reached, and the number of steps taken is recorded. The result provides valuable insights into the efficiency and effectiveness of the A* algorithm in solving the sorting problem while considering both heuristic estimates and actual costs.

Hill Climbing Search :

Hill Climbing is a local search algorithm designed for optimization problems, aiming to find the best solution within a local neighborhood. Our algorithm starts with an initial vector and calculates its heuristic value using heuristic_function2, representing the number of out-of-order elements. The algorithm iteratively explores neighboring states by swapping adjacent elements in the vector and assessing their heuristic values. If a neighboring state yields a lower heuristic value, indicating fewer out-of-order elements, it becomes the new current state. The process continues until either a fully sorted vector is achieved or no better state can be found in the immediate neighborhood, signifying a local minimum. The number of steps taken to reach this state is recorded and returned as the result. Hill Climbing provides insights into its efficiency in sorting the vector by iteratively improving it based on the heuristic information.

<u>Results :</u>

Input array = { 4 , 6.3 , 9 , -3 }

Algorithm	Result from start to goal path	Number of steps

BFS	`4 6.3 9 -3`
	`6.3 4 9 -3`
	`4 9 6.3 -3`
	`4 6.3 -3 9`
	`6.3 9 4 -3`
	`6.3 4 -3 9`
	`9 4 6.3 -3`
	`4 9 -3 6.3`
	`4 -3 6.3 9`
	`9 6.3 4 -3`
	`6.3 9 -3 4`
	`6.3 -3 4 9`
	`9 4 -3 6.3`
	`4 -3 9 6.3`
	`-3 4 6.3 9`
		14

DFS	`4 6.3 9 -3`
	`4 6.3 -3 9`
		20

	4 -3 6.3 9
	4 -3 9 6.3
	4 9 -3 6.3
	9 4 -3 6.3
	9 4 6.3 -3
	9 6.3 4 -3
	9 6.3 -3 4
	9 -3 6.3 4
	-3 9 6.3 4
	-3 9 4 6.3
	-3 6.3 9 4
	-3 6.3 4 9
	6.3 -3 4 9
	6.3 -3 9 4
	6.3 9 -3 4
	6.3 9 4 -3
	9 -3 4 6.3
	-3 4 9 6.3
	-3 4 6.3 9	

Iterative Deepening	4 6.3 9 -3
	4 6.3 -3 9
	4 9 6.3 -3
	6.3 4 9 -3
	4 6.3 9 -3
	4 6.3 -3 9
	4 -3 6.3 9
	6.3 4 -3 9
	4 9 6.3 -3
	4 9 -3 6.3
	9 4 6.3 -3
	6.3 4 9 -3
	6.3 9 4 -3
	4 6.3 9 -3
	4 6.3 -3 9
	4 -3 6.3 9
	4 -3 9 6.3
	-3 4 6.3 9
		17

Uniform Cost Search	4 6.3 9 -3
	4 6.3 -3 9
	4 9 6.3 -3
	6.3 4 9 -3
	4 -3 6.3 9
	4 9 -3 6.3
	6.3 4 -3 9
	6.3 4 -3 9
	6.3 9 4 -3
	9 4 6.3 -3
	-3 4 6.3 9
		10

Greedy	4 6.3 9 -3
	4 6.3 -3 9
	4 -3 6.3 9
	-3 4 6.3 9
		3

A star Search	`4 6.3 9 -3`
	`4 6.3 -3 9`
	`4 -3 6.3 9`
	`-3 4 6.3 9`
		3

Hill Climbing	`Did Not reach the goal state`
	
	`Final State is :`
	`4 6.3 9 -3`
		0

After swapping the corresponding neighbors for each of the elements in the array, hill climb did not find any better heuristic so the number of steps is 0 in this case . It may work for some of the examples but not always .

Question 2 :

Results for randomly generated nodes

Value of n	Results
3	Average number of steps for breadth first search **1.9** Average number of steps for depth first search **2.75** Average number of steps for iterative_deepening **3.8** Average number of steps for uniform cost search **1.8** Average number of steps for greedy search **1.25** Average number of steps for A star search **1.25** Average number of steps for hill climbing search **0.45**
4	Average number of steps for breadth first search **12** Average number of steps for depth first search **12** Average number of steps for iterative_deepening **29.8** Average number of steps for uniform cost search **10.6** Average number of steps for greedy search **3.2** Average number of steps for A star search **3.2** Average number of steps for hill climbing search **0.8**
5	Average number of steps for breadth first search **65.6** Average number of steps for depth first search **56.2** Average number of steps for iterative_deepening **193.55** Average number of steps for uniform cost search **57.65** Average number of steps for greedy search **5.2** Average number of steps for A star search **5.2** Average number of steps for hill climbing search **0.9**

6	Average number of steps for breadth first search **297.9** Average number of steps for depth first search **345.05** Average number of steps for iterative_deepening **1344.1** Average number of steps for uniform cost search **260.85** Average number of steps for greedy search **6.7** Average number of steps for A star search **6.7** Average number of steps for hill climbing search **0.95**

Observations :

- UCS worked better than BFS. UCS explores nodes based on their path cost from the start node. It always selects the node with the lowest path cost to expand. In contrast, BFS explores nodes in the order they are encountered at each level, without considering the cost of reaching those nodes. In this case since the cost is uniform so both ucs and bfs are same in this case the only difference here is the order of nodes which is explored. In bfs we are exploring the just next node present in queue front while in ucs we are giving priority to the one which is of minimum cost.
- DFS and BFS workings differ for different cases. If the goal is in deep of the tree then dfs is to get it faster than bfs while in the other case if goal is in breadth then bfs gets it faster .
- Iterative deepening does not work better than bfs and dfs because it does some repetitive work. Overall for same cases bfs works better than dfs and for some dfs work better than bfs, and iterative deepening is a combination of both.
- By treating the cost of actions as zero in Greedy Search and adding a constant 'c' to each edge weight in A* Search, we preserve the same heuristic function ('h'). This means that only the 'g' part in the cost function 'f = g + h' changes by a constant 'c'. Consequently, if 'f(n1) < f(n2)' initially, then 'f(n1) + c < f(n2) + c' after adding 'c'. This consistency ensures that both algorithms select the same nodes from the fringe, resulting in the same path.
- Hill climbing is not able to find the goal as heuristic is checking just the neighbors and if by swapping each of the neighbors we get the same heuristic and hill climbing just checks the neighbors and if better is not found it stops the algorithm.