

EXPERIMENT 4

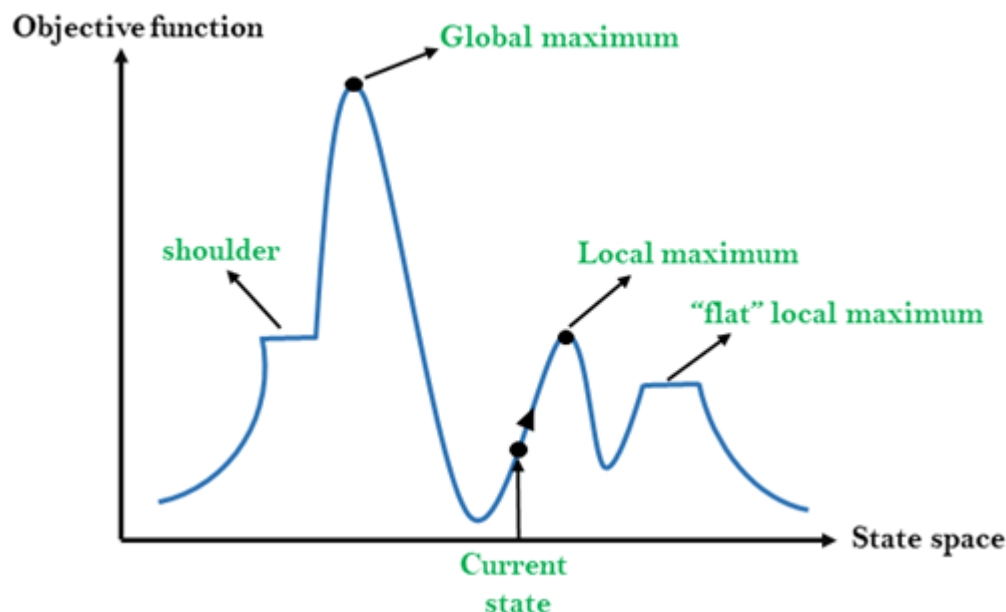
AIM: Program to implement Local Search algorithm using Hill climbing search

THEORY:

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 neighbour has a higher value.

Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.

A node of hill climbing algorithm has two components which are state and value.



Steepest-Ascent Hill Climbing :

It first examines all the neighboring nodes and then selects the node closest to the solution state as of next node.

Algorithm for Simple Hill climbing :

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

Step 2 : Repeat these steps until a solution is found or current state does not change

- a) Select a state that has not been yet applied to the current state.
- b) Initialize a new 'best state' equal to current state and apply it to produce a new state.
- c) Perform these to evaluate new state
 - i. If the current state is a goal state, then stop and return success.
 - ii. If it is better than best state, then make it best state else continue loop with another new state.
- d) Make best state as current state and go to Step 2: b) part.

Step 3 : Exit

CODE:

```
import java.util.*;
import java.util.Arrays;
import java.lang.Integer;
import java.lang.Math;

public class hill {

    public static int MAX, MIN;
    public static int[] hill = new int[10];
    public static int hill_pointer = 0;

    public static int heuristic(int[] arr, int end) {

        int total = 0;
        for (int i = 0; i < end; i++) {

            if (i > 0) {
                if (arr[i] != i) {
                    total -= i;

                } else {
                    total += i;
                }
            }
        }

        return total;
    }
}
```

```
public static void displayArr(int[] arr, int len) {
    System.out.print("[");
    for (int i = 0; i < len; i++) {
        System.out.print(" " + arr[i] + "");
    }
    System.out.print("]");
}

public static void displayPop(int[] arr, int len) {
    for (int i = 0; i < len; i++) {
        System.out.print(" [" + arr[i] + "]");
    }
}

public static void swap(int elem) {
    hill[hill_pointer] = elem;
}

public static void push(int elem) {
    hill_pointer++;
    hill[hill_pointer] = elem;
}

public static void pop() {
    hill_pointer--;
}

public static void calcHill(int[] arr, int num) {

    int[] popped = new int[num];
    int pointer = 0;
    int step = 0;
    int curHeuristic = heuristic(arr, arr.length);
    int end = arr.length - 1;

    // POPPING
    while (curHeuristic < MAX && end >= 0) {
        int nextHeuristic = heuristic(arr, end);

        if (step % 2 == 0) {
            System.out.print("\n\n Stack is: ");
            displayArr(arr, end + 1);
            displayPop(popped, pointer);
            System.out.print(" --> " + curHeuristic);
            // System.out.print("\n POPPED ARRAY : ") ;
        }
    }
}
```

```
if (nextHeuristic > curHeuristic) {
    popped[pointer] = arr[end];
    curHeuristic = nextHeuristic;
    end--;
    pointer++;

} else {

    break;
}

step++;

}

// COPY REST INTO HILL
System.arraycopy(arr, 0, hill, 0, end + 1);
hill_pointer = end;

int[] visited_popped = new int[num];

// END =0
if (end == 0) {

    int temp = hill[hill_pointer];
    for (int i = 0; i <= pointer; i++) {
        if (hill[hill_pointer] > popped[i]) {
            swap(popped[i]);
        }
    }

}

visited_popped[hill[hill_pointer]] = 1;
popped[pointer] = temp;
}
System.out.print("\n\n Stack is : ");
displayArr(hill, hill_pointer + 1);
System.out.print(" --> " + curHeuristic);
```

```
// PUSH
while (curHeuristic < MAX) {
    // All empty-- Find next push
    int maxHeuristic = Integer.MIN_VALUE;
    int pushVal = -1;
    int index = -1;
    hill_pointer++;

    for (int i = 0; i <= pointer; i++) {

        if (visited_popped[i] == 0) {
            swap(popped[i]);

            if (maxHeuristic < heuristic(hill, hill_pointer + 1)) {
                index = i;
                pushVal = popped[i];
                maxHeuristic = heuristic(hill, hill_pointer + 1);
            }
        }
    }
    // pop() ;
    swap(pushVal);
    visited_popped[index] = 1;
    curHeuristic = maxHeuristic;
    step++;
}

System.out.print("\n\n Stack is : ");
displayArr(hill, hill_pointer + 1);
System.out.print(" --> " + curHeuristic);
System.out.print("\n GOAL STATE REACHED!!!");

}

public static void main(String args[]) {

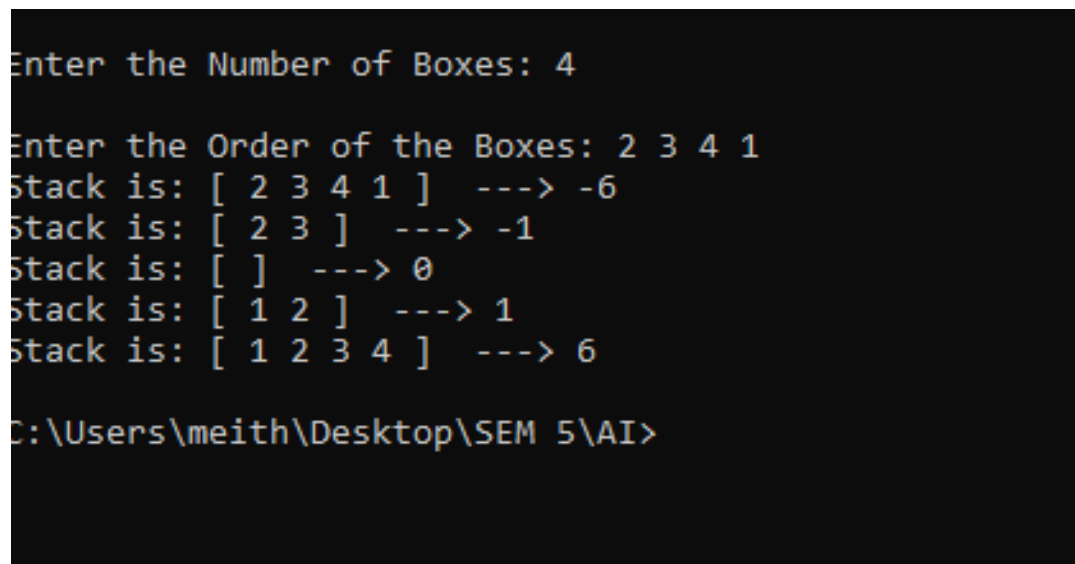
    Scanner sc = new Scanner(System.in);
    System.out.print("\n ENTER THE NUMBER OF BLOCKS : ");
    int num = sc.nextInt();

    int[] arr = new int[num];
    System.out.print("\n ENTER BLOCKS ORDER : ");

    for (int i = 0; i < num; i++) {
        arr[i] = sc.nextInt();
    }
}
```

```
MAX = num * (num - 1) / 2;  
MIN = MAX * (-1);  
  
calcHill(arr, num);  
  
sc.close();  
  
}  
  
}
```

OUTPUT:



```
Enter the Number of Boxes: 4  
  
Enter the Order of the Boxes: 2 3 4 1  
Stack is: [ 2 3 4 1 ] ---> -6  
Stack is: [ 2 3 ] ---> -1  
Stack is: [ ] ---> 0  
Stack is: [ 1 2 ] ---> 1  
Stack is: [ 1 2 3 4 ] ---> 6  
  
C:\Users\meith\Desktop\SEM 5\AI>
```

CONCLUSION:

In this experiment I implemented Hill Climbing Search using Steepest Ascend with step size equal to 2. Here I have calculated a Heuristic function to provide the heuristic value of each state with respect to the previous state, if the value is greater then we move ahead else the previous state was better than the next state. Finally, we reach the goal state. In steepest ascend instead of moving step wise we skip the smaller steps in the same direction, keeping in mind the the jump is not too large.