Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

Name:Divyesh Khunt **Sapid:60009210116** **Batch:D12**

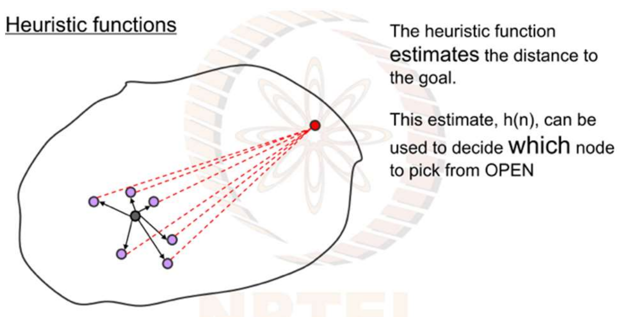
AY: 2023-24

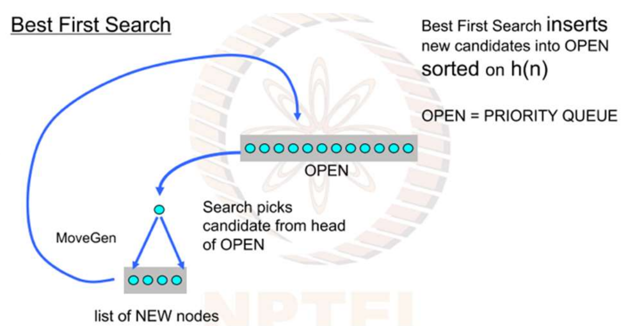
Experiment 3

(Heuristic Search)

Aim: Comparative analysis of Heuristic based methods.

Theory:





Algorithm for Best First Search

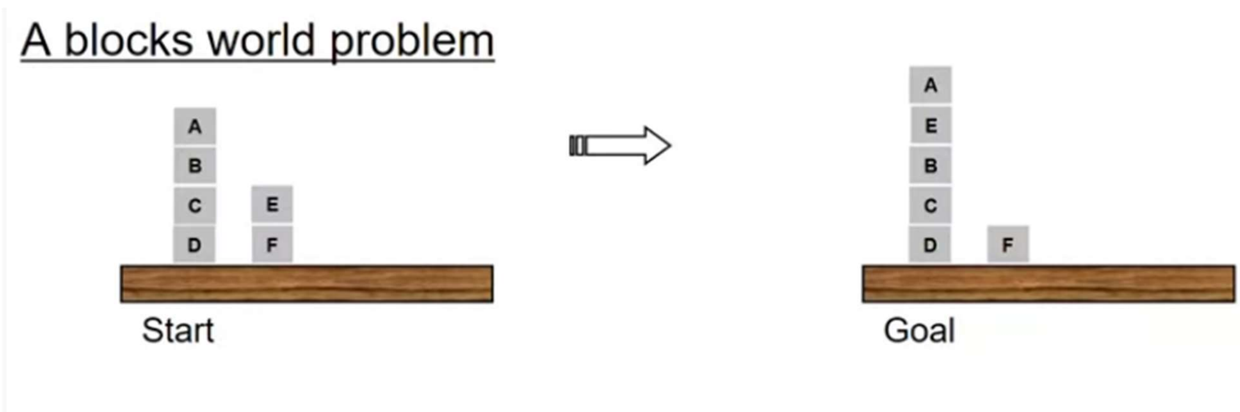
```
Best-First-Search(S)
OPEN ← (S, null, h(S)) []
CLOSED ← empty list
while OPEN is not empty
nodePair ← head OPEN
(N, , ) ← nodePair
if GoalTest(N) = true
return ReconstructPath(nodePair, CLOSED)
else CLOSED ← nodePair CLOSED
neighbours ← MoveGen(N)
newNodes ← RemoveSeen(neighbours, OPEN, CLOSED)
newPairs ← MakePairs(newNodes, N)
OPEN ← sorth( newPairs ++ tail OPEN )
return empty list
```

Algorithm Hill climbing

Hill-Climbing(S)
1 N ← S
2 do bestEver ← N
3 N ← head sorth MoveGen(bestEver)
4 while h(N) is better than h(bestEver)
5 return bestEver

Lab Assignment to do:

1. Design any two different heuristics for a given blocks world problem and show that one is better than another using Hill Climbing and Best First Search.



```
l1=[]
l2=[]
table=[l1, l2]

print("Enter the blocks from bottom to top")

n1 = int(input("Enter the number of blocks in l1: "))
for i in range(n1):
    l1.append(input())

n2 = int(input("Enter the number of blocks in l2: "))
for i in range(n2):
    l2.append(input())

print("Table: ", table, "\n")
```

```
Enter the blocks from bottom to top
Enter the number of blocks in l1: 4
D
C
B
A
Enter the number of blocks in l2: 2
F
E
Table:  [['D', 'C', 'B', 'A'], ['F', 'E']]
```

```
print("Enter the goal test's blocks from bottom to top")
g1 = []
g2 = []

n3 = int(input("Enter the number of blocks in l1: "))
for i in range(n3):
    g1.append(input())

n4 = int(input("Enter the number of blocks in l1: "))
for i in range(n4):
    g2.append(input(f""))
goal_table=[g1,g2]
```

```
Enter the goal test's blocks from bottom to top
Enter the number of blocks in l1: 5
D
C
B
E
A
Enter the number of blocks in l1: 1
F
```

```
[14] goal_table
```

```
[['D', 'C', 'B', 'E', 'A'], ['F']]
```

```
3] def h1(l1, l2, table):
       score = 0
       for stack in table:
           for i in range(1, len(stack)):
               if stack[i - 1] == stack[i]:
                   score += 1
               else:
                   score -= 1
       return score
```

```
def h2(current_state, goal_state):
    score = 0
    for i in range(min(len(current_state), len(goal_state))):
        if current_state[i] == goal_state[i]:
            score += 1
        else:
            score -= 1
    return score
```

```
def movegen(i1, i2, table):
    moves = []
    if i1:
        block = i1[-1]
        new_i1 = i1[:-1]
        new_i2 = i2 + [block]
        moves.append((new_i1, new_i2, f"Move {block} from i1 to i2"))

    if i2:
        block = i2[-1]
        new_i1 = i1 + [block]
        new_i2 = i2[:-1]
        moves.append((new_i1, new_i2, f"Move {block} from i2 to i1"))

    if i1:
        block = i1[-1]
        new_i1 = i1[:-1]
        moves.append((new_i1, i2, block ,f"Place {block} from i1 onto the table"))

    if i2:
        block = i2[-1]
        new_i2 = i2[:-1]
        new_table = table + [block]
        moves.append((i1, new_i2, block, f"Place {block} from i2 onto the table"))

    return moves

possible_moves = movegen(l1, l2, table)
j=1
for i in possible_moves:
    new_table = []
    print(f' move', j)
    print(i)
    new_table += list(i[:-1])
    print(new_table)
    print("Cost of h1: ")
    print(h1(l1,l2,new_table))
    print("Cost of h2: ")
    print(h2(new_table, goal_table),"\n")
    j += 1
```

Output:

```
 move 1
(['D', 'C', 'B'], ['F', 'E', 'A'], 'Move A from i1 to i2')
[['D', 'C', 'B'], ['F', 'E', 'A']]
Cost of h1:
-4
Cost of h2:
-2

 move 2
(['D', 'C', 'B', 'A', 'E'], ['F'], 'Move E from i2 to i1')
[['D', 'C', 'B', 'A', 'E'], ['F']]
Cost of h1:
-4
Cost of h2:
0

 move 3
(['D', 'C', 'B'], ['F', 'E'], 'A', 'Place A from i1 onto the table')
[['D', 'C', 'B'], ['F', 'E'], 'A']
Cost of h1:
-3
Cost of h2:
-2

 move 4
(['D', 'C', 'B', 'A'], ['F'], 'E', 'Place E from i2 onto the table')
[['D', 'C', 'B', 'A'], ['F'], 'E']
Cost of h1:
-3
Cost of h2:
0
```

```
def movegen(i1, i2, table):
    moves = []
    if i1:
        block = i1[-1]
        new_i1 = i1[:-1]
        new_i2 = i2 + [block]
        moves.append((new_i1, new_i2))

    if i2:
        block = i2[-1]
        new_i1 = i1 + [block]
        new_i2 = i2[:-1]
        moves.append((new_i1, new_i2))

    if i1:
        block = i1[-1]
        new_i1 = i1[:-1]
        moves.append((new_i1, i2, block))

    if i2:
        block = i2[-1]
        new_i2 = i2[:-1]
        new_table = table + [block]
        moves.append((i1, new_i2, block))

    return moves

def generate_neighbors(state):
  neighbors = []
  for i in range(len(state)):
    for j in range(len(state[i])):
     if state[i][j] != ' ':
      for k in range(len(state)):
        if i != k:
          neighbor_state = [list(row) for row in state]
          block = neighbor_state[i].pop(j)
          neighbor_state[k].append(block)
          neighbors.append(neighbor_state)
          print(neighbors)

  return neighbors
possible_moves=movegen(l1,l2,table)
for i in possible_moves:
  generate_neighbors(i)
```

```
[[['C', 'B'], ['F', 'E', 'A', 'D']]]
[[['C', 'B'], ['F', 'E', 'A', 'D']], [['D', 'B'], ['F', 'E', 'A', 'C']]]
[[['C', 'B'], ['F', 'E', 'A', 'D']], [['D', 'B'], ['F', 'E', 'A', 'C']], [['D', 'C'], ['F', 'E', 'A', 'B']]]
[[['C', 'B'], ['F', 'E', 'A', 'D']], [['D', 'B'], ['F', 'E', 'A', 'C']], [['D', 'C'], ['F', 'E', 'A', 'B']], [['D', 'C', 'B', 'F'], ['E', 'A']]]
[[['C', 'B'], ['F', 'E', 'A', 'D']], [['D', 'B'], ['F', 'E', 'A', 'C']], [['D', 'C'], ['F', 'E', 'A', 'B']], [['D', 'C', 'B', 'F'], ['E', 'A']], [['D', 'C', 'B', 'E'], ['F', 'A']]]
[[['C', 'B'], ['F', 'E', 'A', 'D']], [['D', 'B'], ['F', 'E', 'A', 'C']], [['D', 'C'], ['F', 'E', 'A', 'B']], [['D', 'C', 'B', 'F'], ['E', 'A']], [['D', 'C', 'B', 'E'], ['F', 'A']], [['D', 'C', 'B', 'A'], ['F', 'E']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']], [['D', 'B', 'A', 'E'], ['F', 'C']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']], [['D', 'B', 'A', 'E'], ['F', 'C']], [['D', 'C', 'A', 'E'], ['F', 'B']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']], [['D', 'B', 'A', 'E'], ['F', 'C']], [['D', 'C', 'A', 'E'], ['F', 'B']], [['D', 'C', 'B', 'E'], ['F', 'A']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']], [['D', 'B', 'A', 'E'], ['F', 'C']], [['D', 'C', 'A', 'E'], ['F', 'B']], [['D', 'C', 'B', 'E'], ['F', 'A']], [['D', 'C', 'B', 'A'], ['F', 'E']]]
[[['C', 'B', 'A', 'E'], ['F', 'D']], [['D', 'B', 'A', 'E'], ['F', 'C']], [['D', 'C', 'A', 'E'], ['F', 'B']], [['D', 'C', 'B', 'E'], ['F', 'A']], [['D', 'C', 'B', 'A'], ['F', 'E']], [['D', 'C', 'B', 'A', 'E', 'F'], []]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']]]
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']], [['D',
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']], [['D',
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']], [['D',
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']], [['D',
[[['C', 'B'], ['F', 'E', 'D'], ['A']], [['C', 'B'], ['F', 'E'], ['A', 'D']], [['D', 'B'], ['F', 'E', 'C'], ['A']], [['D', 'B'], ['F', 'E'], ['A', 'C']], [['D', 'C'], ['F', 'E', 'B'], ['A']], [['D', 'C'], ['F', 'E'], ['A', 'B']], [['D', 'C', 'B', 'F'], ['E'], ['A']], [['D',
[[['C', 'B', 'A'], ['F', 'D'], ['E']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']]]
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']], [['D',
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']], [['D',
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']], [['D',
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']], [['D',
[[['C', 'B', 'A'], ['F', 'D'], ['E']], [['C', 'B', 'A'], ['F'], ['E', 'D']], [['D', 'B', 'A'], ['F', 'C'], ['E']], [['D', 'B', 'A'], ['F'], ['E', 'C']], [['D', 'C', 'A'], ['F', 'B'], ['E']], [['D', 'C', 'A'], ['F'], ['E', 'B']], [['D', 'C', 'B'], ['F', 'A'], ['E']], [['D',
```

```python
import heapq

def best_first_search(initial_state, goal_state):
  OPEN = [(initial_state, 0)]
  CLOSED = set()
  while OPEN:
    current_state, cost = heapq.heappop(OPEN)
    if current_state == goal_state:
      return current_state
      CLOSED.add(tuple(map(tuple, current_state)))
      neighbors = generate_neighbors(current_state)
      for neighbor in neighbors:
       if tuple(map(tuple, neighbor)) not in CLOSED:
         if neighbor not in (state for state, _ in OPEN):
             heapq.heappush(OPEN, (neighbor, heuristic(neighbor,goal_state)))


solutionbfs = best_first_search(table,goal_table)
print("Solution:", solutionbfs)
```

```python
def hill_climbing(initial_state):
    current_state = initial_state
    current_score = h2(current_state,goal_table)

    while True:
        neighbors = generate_neighbors(current_state)
        best_neighbor = None
        best_neighbor_score = current_score

        for neighbor in neighbors:
            neighbor_score = h2(current_state,goal_table)
            if neighbor_score > best_neighbor_score:
                best_neighbor = neighbor
                best_neighbor_score = neighbor_score

        if best_neighbor_score <= current_score:
            break

        current_state = best_neighbor
        current_score = best_neighbor_score

    return current_state

final_state = hill_climbing(goal_table)
print("Final State:", final_state)
```

```
Final State: [['D', 'C', 'B', 'E', 'A'], ['F']]
```