LAB 4

PRELAB

1. What are different types of informed searches? Explain about them brief?

Ans:

There are 2 types of informed searches are present, they are:-

1. Greedy best first Search Algorithm
2. A* Algorithm

Greedy Best first search algorithm:

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n).

A* Algorithm:

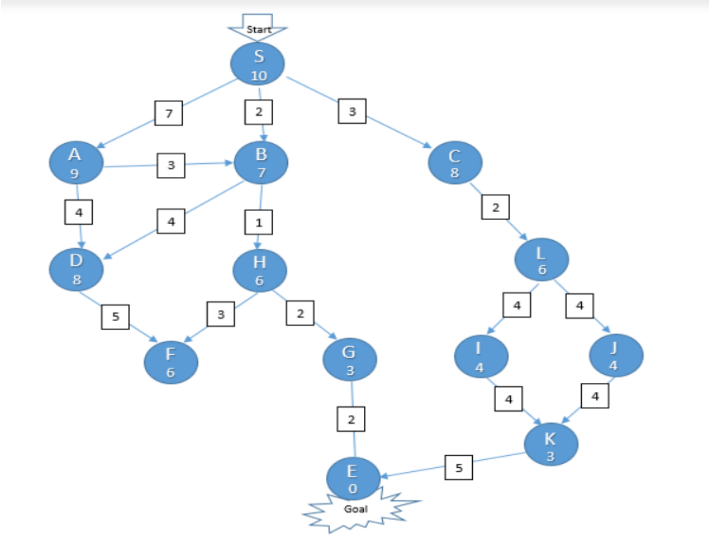
The most widely known form of best-first search is called A∗ A search (pronounced "A-star ∗ SEARCH search"). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n) .

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n .

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal.

The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

2. Differentiate the approach of Greedy BFS and A* BFS on the given tree and compare their time and space complexities.



Ans.

For GBFS Search it takes the heuristic function f(n) = h(n), h(n) is the estimated cost of path in the search tree, From the above graph, following steps are occurred to reach goal state:

1. Reads start and goal state and start's for searching

2. then form the neighbourhood nodes it checks the cost of the paths, and chooses the Least cost path

3.since from the above figure starting node is starting Node is S and goal node is E, so it starts searching for the least cost and selects the best node.

4. if the child node path is lower than parent it chooses child node. Path travelled: S -> B -> H -> G-> E

For A* search algorithm it takes the heuristic function f(n) = h(n) + g(n), h(n) is the estimated cost of path and g(n) is the cost of path it travelled throughout the nodes

From the above graph, following steps are occurred to reach goal state:

1. Reads start and goal state and start's for searching

2. Then checks the path cost and node cost it travelled among them, hen it chooses the least cost.

3. Then it takes the least cost path and moves from the one node to the next node it taken Path travelled: S -> B -> H -> G-> E

Time complexity for Greedy BFS is O(b^m) and for a* is O(b^m),

Space complexity for Greedy BFS is a linear polynomial and for a* is O(b^m)

INLAB

1. Write a python program to find the path in the following maze using A* algorithm,considering 1 as the wall and 0 as the movable region.

```
maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
start = (0, 0)
end = (7, 6)
```

CODE

```
class Node:
    def __init__(self,f,h,g,curr_position,parent):
        self.parent = parent
        self.position = curr_position

        self.g = g
        self.h = h
        self.f = g+h
maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = [0, 0]
end = [7 , 6]

h = [0.0]*10

for i in range(10):
    h[i] = [0]*10
def generate_h_value(h):
    for i in range(0,10):
        for j in range(0,10):
            h[i][j]=abs(end[0]-i)+abs(end[1]-j)
```

```
opened = []
node = Node(h[start[0]][start[1]]+0,h[start[0]][start[1]],0,start,'NA');
opened.append(node)
```

Loading...

```
closed = []

neighbors = []
def isSafe(maze,x,y):
    if x >= 0 and x < 10 and y >=0 and y < 10 and maze[x][y] == 0:
        return True
    return False

def next_state(l):
    x = l[0]
    y = l[1]
    if isSafe(maze,x + 1,y) == True:
        neighbors.append([x+1,y])
    if isSafe(maze,x,y + 1) == True:
        neighbors.append([x,y+1])
    if isSafe(maze,x - 1,y) == True:
        neighbors.append([x - 1,y])
    if isSafe(maze,x,y-1) == True:
        neighbors.append([x,y-1])
    if isSafe(maze,x-1,y-1) == True:
        neighbors.append([x-1,y-1])
    if isSafe(maze,x+1,y-1) == True:
        neighbors.append([x+1,y-1])
```

```
    if isSafe(maze,x-1,y+1) == True:
        neighbors.append([x-1,y+1])
    if isSafe(maze,x+1,y+1) == True:
        neighbors.append([x+1, y+1])


generate_h_value(h)

while opened:
    best_node = opened[0]
    best_node_index = 0
    for index,item in enumerate(opened):
        if item.f < best_node.f:
            best_node = item
            best_node_index = index

    opened.pop(best_node_index)
    closed.append(best_node)

    if best_node.position == end:
        break
    next_state(best_node.position)
    for node in neighbors:
        flag = 0
        child = Node(h[node[0]][node[1]]+best_node.g+1,h[node[0]][node[1]],best_node.g+1,node,best_node)
        for index,i in enumerate(closed):
            if child.position == i.position and child.g <= i.g:
                i.f = child.f
                i.h = child.h
                i.g = child.g
                i.position = child.position
                i.parent = child.parent
                flag = 1
                break
        if flag == 1:
            continue
        for index,i in enumerate(opened):
            if child.position == i.position and child.g <= i.g:
                i.f = child.f
                i.h = child.h
                i.g = child.g
                i.position = child.position
                i.parent = child.parent
                flag = 1
                break
        if flag == 0:
            opened.append(child)

for index,i in enumerate(closed):
        print(i.position)
```

OUTPUT

```
[0, 0]
[1, 1]
[2, 2]
[3, 3]
[4, 3]
[5, 4]
[6, 5]
[7, 6]
```

POST LAB

1. Write a python program in which the system tries to reach and generate the sentence “Hello, World!”. You have to print the difference of the sentence generated in that step and the actual sentence at each step.

   Sample Output:

```
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 1 Solution Iello, World!
Best score so far 0 Solution Hello, World!


...Program finished with exit code 0
Press ENTER to exit console.
```

CODE

```
import random
import string
def solution(length=13):
  return [random.choice(string.printable) for _ in range(length)]
def hello(sol):
  target = list("Hello, World!")
  diff = 0
  for i in range(len(target)):
    s = sol[i]
    g = target[i]
    diff += abs(ord(s) - ord(g))
  return diff
def mutate(sol):
  index = random.randint(0, len(sol) - 1)
  sol[index] = random.choice(string.printable)
best = solution()
best_score = hello(best)
while True:
  print('Best score so far', best_score, 'Solution', "".join(best))
  if best_score == 0:
    break
  new = list(best)
  mutate(new)
  score = hello(new)
  if hello(new) < best_score:
    best = new
    best_score = score
```

OUTPUT

```
Streaming output truncated to the last 5000 lines.
Best score so far 388 Solution q\u_;8djT_~;]
Best score so far 388 Solution q\u_;8djT_~;]
Best score so far 379 Solution q\u_;8[jT_~;]
Best score so far 379 Solution q\u_;8[jT_~;]
Best score so far 368 Solution q\u_;8[jT_~F]
Best score so far 357 Solution q\u_;8[_T_~F]
Best score so far 357 Solution q\u_;8[_T_~F]
Best score so far 357 Solution q\u_;8[_T_~F]
Best score so far 357 Solution q\u_;8[_T_~F]
Best score so far 333 Solution q\u_;8[_T_~j]
Best score so far 333 Solution q\u_;8[_T_~j]
Best score so far 333 Solution q\u_;8[_T_~j]
Best score so far 305 Solution q\u_;8?_T_~j]
Best score so far 305 Solution q\u_;8?_T_~j]
Best score so far 300 Solution $\u_;8?_T_~j]
Best score so far 299 Solution $mu_;8?_T_~j]
Best score so far 299 Solution $mu_;8?_T_~j]
Best score so far 299 Solution $mu_;8?_T_~j]
Best score so far 299 Solution $mu_;8?_T_~j]
Best score so far 276 Solution $mu_;8(_T_~j]
Best score so far 269 Solution $du_;8(_T_~j]
Best score so far 269 Solution $du_;8(_T_~j]
Best score so far 261 Solution $dm_;8(_T_~j]
Best score so far 261 Solution $dm_;8(_T_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
Best score so far 239 Solution $dm_;8(_j_~j]
```