CSL 303 : Artificial Intelligence

TUTORIAL ASSIGNMENTS 4 and 5

Uninformed and Informed Search Techniques

Date Assigned: 3rd September, 2021 **Date Submitted :** 19th September, 2021

Submitted by:

Name: S. Bhavyesh

Roll Number: 191210045

Branch: Computer Science and Engineering

Semester: 5th

Submitted to: Dr. Chandra Prakash

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

A-7, Institutional Area, near Satyawadi Raja Harish Chandra Hospital, New Delhi, Delhi 110040

PART A : Introductory Problem [25 Marks]

Vacuum World [5 Marks]

1. Consider the vacuum world scenario discussed in class.

There are four rooms (as shown below), one vacuum cleaner and a room may or may not contain dirt.

Room 1	**Room 2**
Room 3	**Room 4**

Inputs:
Write python/C program that takes four inputs through a single input file. First line of input file specifies the room number in which vacuum cleaner is located, second line specifies the presence or absence 1 or 0 of dirt in rooms 1, 2, 3 and 4, respectively, each separated by comma and third line in input file specifies the algorithm to be used :

dfs: Depth First Search
bfs: Breadth First Search

Assume valid actions as L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down,
S=suck_dirt and N=no_op.
eg. In example in input file below (input.txt) vacuum cleaner is in Room 1, dirt is only in Rooms 1 and 4 and DFS is to be used in reaching to the goal state.

input.txt
1
1,0,0,1
dfs
python TA_4_5_P1_vacuum_world.py "input.txt" "output1.txt"

Hints:
a) Your program should create an appropriate data structure (node) that can capture problem states.
b) Construct state space graph, identify nodes and vertices of this graph.
c) Based on the type of search algorithm, expand from initial state to next state and so on.
d) Once the goal is reached (i.e. no dirt in any of the rooms), program should terminate.

Outputs: Sequence of <current_room,action> pair until goal state (all rooms clean) is reached, output each sequence on a new line and save it into output file specified in fourth argument (say "out.txt" above).
eg.
1,S
1,R
.... and so on until goal is reached
Above output should be stored in output file, say "output.txt"

CODE:

In [57]:
```
"""
    Class below is a representation of the states. State is a quintuple (V,1,2,3,4) where
    V is room number of vacuum cleaner[1,4] and 1,2,3,4 represent the status
    of the respective rooms(clean=0 and dirty=1).
    Also, action denotes the action by which we arrived at the current state,
    from the parent node.
"""
class Node:

    def __init__(self,state=None,action=None,parent=None):
        self.state=state
        self.action=action
        self.parent=parent
```

In [58]:
```
"""
    Utility function that prints the actual shortest
    path claimed by the algorithm ~ 'effective route'
"""
def path(g):
    if(g.parent.parent!=None):
        path(g.parent);
    output.write("%s , %s\n" %(g.parent.state[0],g.action))
```

In [59]:
```
action={
    -2: 'D',
    -1: 'R',
     0: 'S',
     1: 'L',
     2: 'U',
}
```

In [60]:
```
"""
    Generation of ENTIRE state space representation in the form of adjacency lists. All possible
    states are first generated then by recognizing a pattern, we can generate adjacency list for each state
"""
import numpy as np

state=np.array([[v,r1,r2,r3,r4] for v in range(1,5) for r1 in range(0,2)
      for r2 in range(0,2) for r3 in range(0,2) for r4 in range(0,2)]).astype(int)

# change in rooms, one means left-right, two means up-down
one=np.array([1,0,0,0,0]).astype(int)
two=np.array([2,0,0,0,0]).astype(int)

# if room can be cleaned then update the state accordingly by decrementing the repsective 1 to zero
clean1=np.array([0,-1,0,0,0]).astype(int)
clean2=np.array([0,0,-1,0,0]).astype(int)
clean3=np.array([0,0,0,-1,0]).astype(int)
clean4=np.array([0,0,0,0,-1]).astype(int)

# generate state space
space={}
for i in range(len(state)):
    if(state[i][0]==1):
        if(state[i][1]==1):space[tuple(state[i])]=[tuple(state[i]+one),tuple(state[i]+two),tuple(state[i]+clean1)]
        else:space[tuple(state[i])]=[tuple(state[i]+one),tuple(state[i]+two)]

    elif(state[i][0]==2):
        if(state[i][2]==1):space[tuple(state[i])]=[tuple(state[i]-one),tuple(state[i]+two),tuple(state[i]+clean2)]
        else:space[tuple(state[i])]=[tuple(state[i]-one),tuple(state[i]+two)]

    elif(state[i][0]==3):
        if(state[i][3]==1):space[tuple(state[i])]=[tuple(state[i]-two),tuple(state[i]+one),tuple(state[i]+clean3)]
        else:space[tuple(state[i])]=[tuple(state[i]-two),tuple(state[i]+one)]

    elif(state[i][0]==4):
        if(state[i][4]==1):space[tuple(state[i])]=[tuple(state[i]-two),tuple(state[i]-one),tuple(state[i]+clean4)]
        else:space[tuple(state[i])]=[tuple(state[i]-two),tuple(state[i]-one)]
```

In [61]:
```
# check if node is not in visited and within bounds

def valid(node,visited=[None]):
    if(node[0]<=1 and node[1]<=1 and node[0]>=1 and node[1]>=1
       and node not in visited):
        return True
    return False
```

In [36]:
```
# Depth-First Search

def dfs(start,output):

    # initialize the fringe and visited lists
    stack=[start]
    visited=[]
    cost=0
    while True:
        curr_node=stack.pop()
        visited.append(curr_node.state)
        if(sum(list(curr_node.state[1:]))==0): # all rooms are clean

            # find the effective path chosen
            # and store it in a file

            path(curr_node)
            output.write("%s , %s\n" %(curr_node.state[0],'N'))
            output.close()
            print("Total nodes visited:",cost)
            break
        else:
            # fetch the neighbors from the graph
            # and push into fringe accordingly

            nbrs=space[curr_node.state]
            for i in range(len(nbrs)):
                if(nbrs[i] not in visited):
                    stack.append(Node(nbrs[i],action[curr_node.state[0]-nbrs[i][0]],curr_node))
            cost+=1
```

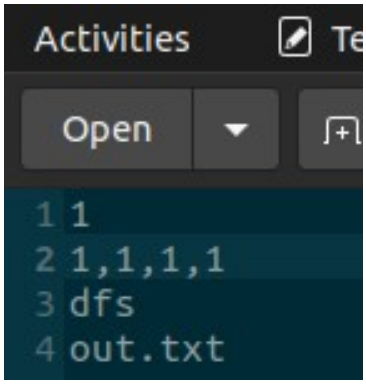
```
def bfs(start,output):

    # initialize the fringe and visited lists
    queue=[start]
    visited=[]
    cost=0
    while True:

        curr_node=queue.pop(0)
        visited.append(curr_node)

        if(sum(list(curr_node.state[1:]))==0): # all rooms are clean

            # find the effective path chosen
            # and store it in a file

            path(curr_node)
            output.write("%s , %s\n" %(curr_node.state[0],'N'))
            output.close()
            print("Total nodes visited:",cost)
            break
        else:

            # fetch the neighbors from the graph
            # and push into fringe accordingly

            nbrs=space[curr_node.state]
            for i in range(len(nbrs)):
                if(nbrs[i] not in visited):
                    queue.append(Node(nbrs[i],action[curr_node.state[0]-nbrs[i][0]],curr_node))
                    visited.append(nbrs[i])
            cost+=1
```

In [64]:
```
# Execution Block

f=open('input1.txt',encoding='utf8')

# extract initial state of problem
start=[int(f.readline())]
start=start+[int(y.strip()) for y in f.readline().split(',')]
start=tuple(start)
algorithm=f.readline()[:-1]
output=open(f.readline()[:-1],'w')

# do the execution
eval(algorithm+"(Node(start),output)")
```

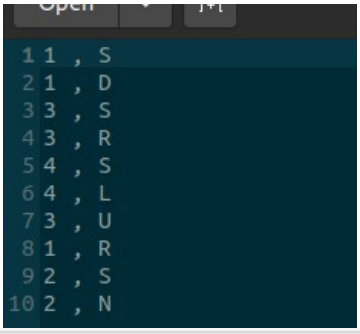
OUTPUTS:

For DFS:

INPUT



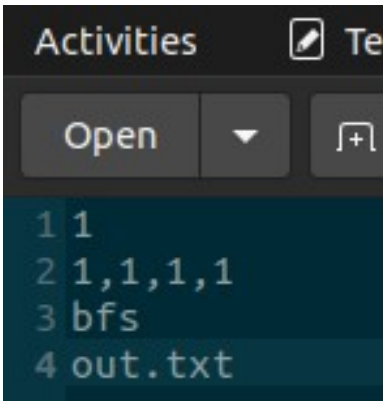
OUTPUT



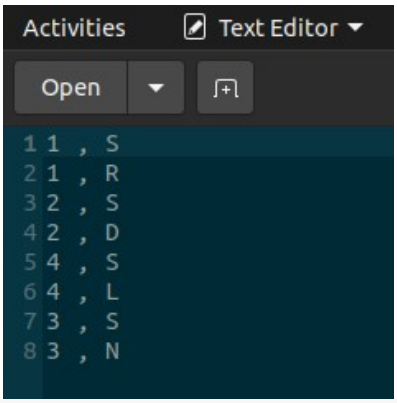
Total nodes visited: 9

For BFS:

INPUT



OUTPUT



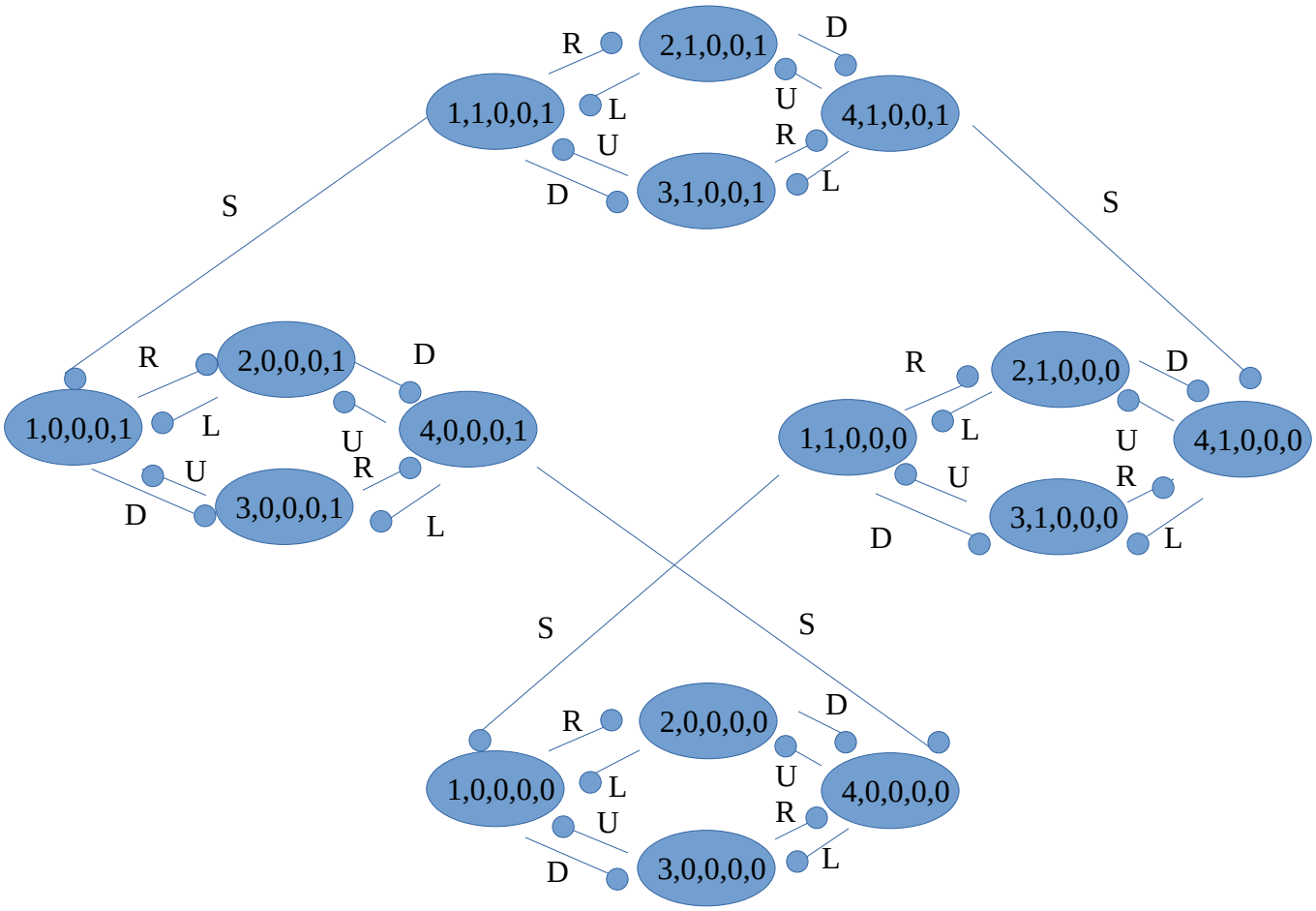
Total nodes visited: 61

NOTE: The actual nodes traversed are not mentioned, only the nodes that lie on the effective path to the goal are mentioned, because there are too many nodes explored and it is hard to print each of them on a new line. Nonetheless, the algorithms work fine.

NOTE: For 1 vacuum cleaner, and n rooms, the total number of possible states are n * 2^n. Here, n=4 implies that there are 4*2^4=64 states.

OBSERVATION/COMMENTS

1. The state space representation for the input (1,1,0,0,1) is:



Where nodes are of the form (V,1,2,3,4), V represents room number of vacuum cleaner, and 1,2,3,4 represent the status of rooms (clean – 0, dirty – 1).

From this diagram, we can see that the diamond-shaped entity is the repeating element in the graph, and the 'sucking' action links one diamond-shaped graph to another. Using this fact, and some intuition, we can construct the entire state space graph.

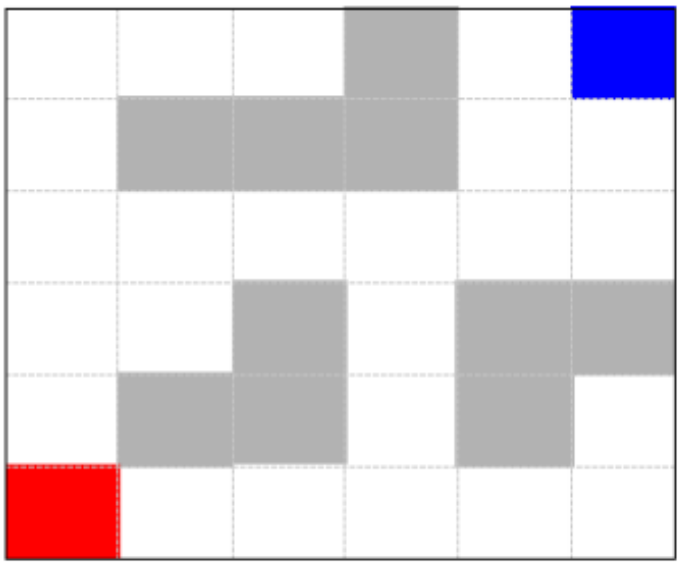
2. Although the cost of effective path is same, the number of nodes traversed is different in both cases is different.

3. Here I could have dynamically generated nodes in state space, but I thought of doing something different. Of course, storing the entire state space graph is not space-efficient but it definitely makes the algorithms faster.

Maze Problem [10 Marks]

2. Consider a maze comprising of square blocks in which intelligent agent can move either vertically or horizontally. Diagonal movement is not allowed. Cost of each move is 1.

Red block: initial position, Blue block: goal position and Grey block: obstacle



Apply following Blind/Uninformed and Informed algorithms :

(a) dfs: Depth first search
(b) bfs: Breadth first search
(c) dls: Depth limited search, use 3 as default depth
(d) ucs: Uniform cost Search
(e) gbfs: Greedy Best First Search
(f) astar: A Algorithm*

Inputs:
Write a python program that takes input number of square blocks as input (i.e. 6 x 6) in first line. Second line contains the initial position of intelligent agent which is (1,1) and the goal square block which is (6,6) in above example. Third line contains the coordinates of the obstacles. Fourth line contains the search strategy.

eg. input file: input.txt
6,6
1,1;6,6
2,1;2,5;3,2;3,3;3,5;4,5;4,6;5,2;5,3;6,3
dfs
python TA_4_5_P2_maze_world.py "input 2 .txtı output 2 .txtı

Outputs: Sequence of blocks that are explored (each on separate line) as per search algorithm so as to reach goal position. Last line should contain the total search cost.

CODE:

In [1]:
```
# Check if node is within the bounds of the maze,
# has not been visited before, and is not an obstacle

def valid(node,visited=[None]):
    if(node[0]<=6 and node[1]<=6 and node[0]>=1 and node[1]>=1
       and node not in visited and node not in obstacles):
        return True
    return False
```

In [2]:
```
# Depth-First Search

def dfs(start,goal):

    # initialize variables
    stack=[start]
    visited=[]
    cost=0

    while True:
        curr_node=stack.pop()
        visited.append(curr_node)

        print(curr_node)
        i,j=curr_node

        # goal found!
        if(curr_node==goal):
            print("| Cost=",cost)
            break

        # entertain all possibilities (up,down,left,right)
        else:
            if(valid((i-1,j),visited)):stack.append((i-1,j))
            if(valid((i,j-1),visited)):stack.append((i,j-1))
            if(valid((i,j+1),visited)):stack.append((i,j+1))
            if(valid((i+1,j),visited)):stack.append((i+1,j))
        cost+=1
```

In [3]:
```
# Breadth-First Search

def bfs(start,goal):

    # initialize variables
    queue=[start]
    visited=[]
    cost=0

    while True:
        curr_node=queue.pop(0)
        visited.append(curr_node)

        print(curr_node)
        i,j=curr_node

        # goal found!
        if(curr_node==goal):
            print("| Cost=",cost)
            break

        # entertain all possibilities (up,down,left,right)
        else:
            nbrs=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)]
            for k in range(4):
                if(valid(nbrs[k],visited)):
                    queue.append(nbrs[k])
                    visited.append(nbrs[k])
        cost+=1
```

In [4]:
```
def dls(start,goal):      # Depth-Limited Search
    for l in range(3,13): # increase depth iteratively
        stack=[(0,start)] #initialize stack
        visited=[]
        cost=0

        # terminates the iterative deepening, when goal is found
        goal_found=False

        while True:
            if(len(stack)==0):
                print("Goal not found")
                break
            curr_node=stack.pop()
            visited.append(curr_node[1])

            if(curr_node[0]>l): # ignore if depth exceeds limit
                continue
            else:
                print(curr_node[1],end=' ') # do the usual dfs algo
                i,j=curr_node[1]
                if(curr_node[1]==goal):
                    print("| Cost=",cost," | Depth = ",l)
                    goal_found=True
                    break
                else: # entertain all possibilites(up,down,left,right)
                    if(valid((i-1,j),visited)):stack.append(((curr_node[0]+1),(i-1,j)))
                    if(valid((i,j-1),visited)):stack.append(((curr_node[0]+1),(i,j-1)))
                    if(valid((i,j+1),visited)):stack.append(((curr_node[0]+1),(i,j+1)))
                    if(valid((i+1,j),visited)):stack.append(((curr_node[0]+1),(i+1,j)))
                cost+=1
        if(goal_found): break
```

In [9]:
```
def ucs(start,goal): # Uniform-cost Search
    q=[(0,start)] # tuple of the form (dist from goal,coordinates)
    q_states=[start]
    visited=[]
    cost=0

    while True:
        curr_node=q.pop(0) # eject from fringe
        q_states.pop(0)

        visited.append(curr_node[1])
        print(curr_node[1])
        i,j=curr_node[1]

        if(curr_node[1]==goal): # goal found
            print("| Cost=",cost)
            break
        else:
            nbrs=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)] # generate possible neighbors

            for i in range(4):
                # if the nbr does not exist in visited+fringe, insert in fringe
                if(valid(nbrs[i],visited) and nbrs[i] not in q_states):
                    q.append(((curr_node[0]+1),nbrs[i]))
                    q_states.append(nbrs[i])

                # better path found! -> update the node
                elif(nbrs[i] in q_states and q[q_states.index(nbrs[i])][0]>(curr_node[0]+1)):
                    q[q_states.index(nbrs[i])][0]=curr_node[0]+1

            # prioritize least cost path
            q.sort()
            cost+=1
```

In [10]:
```
import numpy as np

def gbfs(start,goal): # Greedy Best-First Search
    maze=[(x,y) for x in range(1,7) for y in range(1,7)]     # generate Manhattan distances
    maze=list(set(maze)-set(obstacles))
    dist=np.sum(np.array([6,6])-maze,axis=1)

    q=[(10,0,start)]     #(h(n),g(n),node)
    r_state=maze         # lookup table for nodes
    r_data=[0]*len(maze) # lookup table for the nodes' corresponding (h(n),g(n))
    cost=0

    while True:
        curr_node=q.pop(0)
        print(curr_node[2])
        i,j=curr_node[2]

        if(curr_node[2]==goal):  # goal found!-> terminate
            print("| Cost=",cost)
            break
        else:
            nbrs=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)]  # generate all neighbors
            for i in range(4):
                # update lookup table if node visited for the first time,
                # or if a better path exists to the node
                if(valid(nbrs[i])):
                    if (r_data[r_state.index(nbrs[i])]==0
                    or (curr_node[1]+1)<r_data[r_state.index(nbrs[i])][1]):
                        r_data[r_state.index(nbrs[i])]=(dist[maze.index(nbrs[i])],(curr_node[1]+1))
                        q.append((dist[maze.index(nbrs[i])],(curr_node[1]+1),nbrs[i]))
            q.sort()
            cost+=1
```

In [11]:
```
import numpy as np

def astar(start,goal): # A* Search
    maze=[(x,y) for x in range(1,7) for y in range(1,7)]     # generate Manhattan distances
    maze=list(set(maze)-set(obstacles))
    dist=np.sum(np.array([6,6])-maze,axis=1)

    q=[(10,10,0,start)]  #(h(n)+g(n),h(n),g(n),node)
    r_state=maze         # lookup table for nodes
    r_data=[0]*len(maze) # lookup table for the nodes' corresponding (h(n),g(n))
    cost=0

    while True:
        curr_node=q.pop(0)
        print(curr_node[3])
        i,j=curr_node[3]

        if(curr_node[3]==goal):  # goal found!-> terminate
            print("| Cost=",cost)
            break
        else:
            nbrs=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)]  # generate all neighbors
            for i in range(4):
                if(valid(nbrs[i])):
                    # update lookup table if node visited for the first time,
                    # or if a better path exists to the node
                    if (r_data[r_state.index(nbrs[i])]==0
                    or (curr_node[2]+1)<r_data[r_state.index(nbrs[i])][1]):
                        r_data[r_state.index(nbrs[i])]=(dist[maze.index(nbrs[i])]+curr_node[2]+1,(curr_node[2]+1))
                        q.append((dist[maze.index(nbrs[i])]+curr_node[2]+1,dist[maze.index(nbrs[i])],(curr_node[2]+
            q.sort()
            cost+=1
```

In [12]:
```
# Execution Block

f=open('input.txt',encoding='utf8')

dims=[int(y.strip()) for y in f.readline().split(',')]

initial=[y.strip() for y in f.readline().split(';')]
start=int(initial[0][0]),int(initial[0][2])
goal=int(initial[1][0]),int(initial[1][2])

obstacles=[y.strip() for y in f.readline().split(';')]
for i in range(len(obstacles)):
    obstacles[i]=int(obstacles[i][0]),int(obstacles[i][2])

global obstacles

algo=['dfs','bfs','dls','ucs','gbfs','astar']
for i in range(6):
    print("Algo used:",algo[i])
    eval(algo[i]+"(start,goal)")
    print("")
```

In [1]:
```
import numpy as np

# below code snippet is for showing the initial state of
# the maze diagrammatically
maze=np.zeros(36)
maze[3]=maze[7]=maze[8]=maze[9]=maze[20]=1
maze[28]=maze[22]=maze[23]=maze[25]=maze[26]=1
maze[30]=2
maze[5]=3

maze=maze.reshape((6,6))
print(maze)
print("")
```

```
[[0. 0. 0. 1. 0. 3.]
 [0. 1. 1. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 1.]
 [0. 1. 1. 0. 1. 0.]
 [2. 0. 0. 0. 0. 0.]]
```

OUTPUTS:

```
Algo used: dfs
(1, 1)
(2, 1)
(3, 1)
(4, 1)
(5, 1)
(6, 1)
(6, 2)
(4, 2)
(4, 3)
(4, 4)
(5, 4)
(6, 4)
(6, 5)
(6, 6)
| Cost= 13
```

```
Algo used: bfs
(1, 1) (1, 2) (2, 1) (1, 3) (3, 1) (1, 4) (2, 3) (4, 1) (1, 5) (2, 4) (4, 2) (5, 1) (1, 6) (3, 4) (4, 3) (6, 1)
(2, 6) (4, 4) (6, 2) (3, 6) (5, 4) (5, 5) (6, 4) (5, 6) (6, 5) (6, 6) | Cost= 25

Algo used: dls
(1, 1) (2, 1) (3, 1) (4, 1) (1, 2) (1, 3) (2, 3) (1, 4) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (4, 2) (1, 2) (1, 3) (2, 3) (2, 4) (1, 4) (1, 5) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (4, 2) (4, 3) (1, 2) (1, 3) (2, 3) (2, 4) (3, 4) (1, 4) (1, 4) Goal no
t found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (4, 2) (4, 3) (4, 4) (1, 2) (1, 3) (2, 3) (2, 4) (1, 4) (1, 5)
(1, 4) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (4, 2) (4, 3) (4, 4) (5, 4) (3, 4) (1, 2) (1, 3) (2, 3) (1, 4)
(1, 5) (1, 6) (2, 6) (3, 6) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (4, 2) (4, 3) (4, 4) (5, 4) (6, 4) (5, 5) (3, 4) (2, 4) (1, 2)
(1, 3) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (4, 2) (4, 3) (4, 4) (5, 4) (6, 4) (6, 5) (5, 5) (5, 6) (3, 4)
(2, 4) (2, 3) (1, 4) (1, 2) Goal not found
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (4, 2) (4, 3) (4, 4) (5, 4) (6, 4) (6, 5) (6, 6) | Cost= 13  |
Depth =  10
```

```
Algo used: ucs
(1, 1) (1, 2) (2, 1) (1, 3) (3, 1) (1, 4) (2, 3) (4, 1) (1, 5) (2, 4) (4, 2) (5, 1) (1, 6) (3, 4) (4, 3) (6, 1)
(2, 6) (4, 4) (6, 2) (3, 6) (5, 4) (5, 5) (6, 4) (5, 6) (6, 5) (6, 6) | Cost= 25
```

```
Algo used: gbfs
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 6)
(3, 6)
(2, 4)
(3, 4)
(4, 4)
(5, 4)
(5, 5)
(5, 6)
(6, 6)
| Cost= 14
```

```
Algo used: astar
(1, 1)
(1, 2)
(1, 3)
(1, 4)
(1, 5)
(1, 6)
(2, 6)
(3, 6)
(2, 4)
(3, 4)
(4, 4)
(5, 4)
(5, 5)
(5, 6)
(6, 6)
| Cost= 14
```

OBSERVATION/COMMENTS

1. UCS is giving the same cost as BFS. This is because UCS in the case of same edge costs in the entire graph will act as BFS. Also, if the edge cost is same everywhere, then BFS inherits the property of UCS, that is, optimality.

2. Depth-limited search resolves the problem of infinite depth search by DFS.

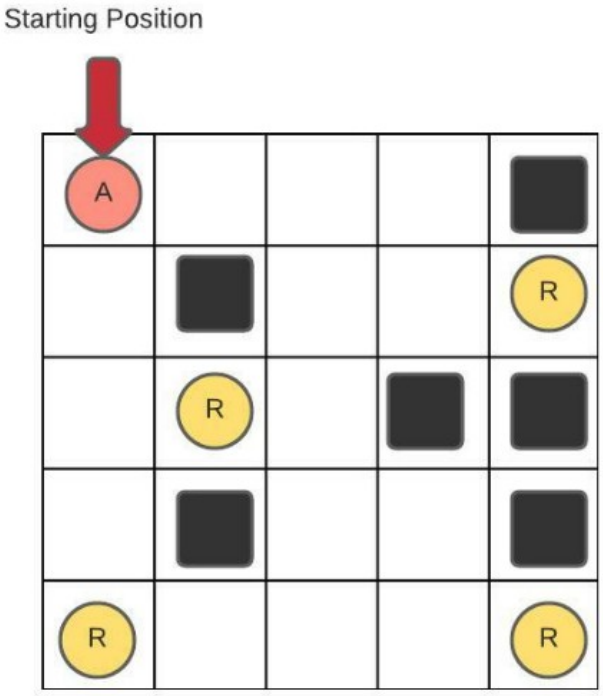
3. UCS finds the optimal path at the cost of visiting too many nodes.

4. GBFS and A-star give the same total cost. This is because, the h(n)+g(n) value is same for all nodes, so in A*, the tiebreak is done by the smaller h(n) value, which essentially makes it behave like GBFS. Note that the algorithms have no flaw in them, it is just the design of problem that causes this. Infact, tie breaking strategy can greatly affect the effectiveness of A*-search.

3. Maze problem with multi-goal [10 Marks]

Consider the maze given in the figure below. The walled tiles are marked in black and your agent A cannot move to or through those positions.

Inputs:
Write python/C program that takes the maze as a 5x5 matrix input where 0 denotes an empty tile, 1 denotes an obstruction/wall, 2 denotes the start state and 3 denotes the reward. Assume valid actions as L,R,U,D,S,N where L=move_left, R=move_right, U=move_up, D=move_down.



Use the A* algorithms as (astar) on the resultant maze for your agent to reach all the rewards, and keep a record of the tiles visited on the way.

Hints:
a) Your program should create the appropriate data structure that can capture problem states, as mentioned in the problem.
b) Once the goal is reached (i.e. Reward position), program should terminate.

Outputs: The output should have the sequence of the tiles visited by each algorithm to reach the termination state stored in an output file labeled as "out_astar.txt" and so on. Print the number of steps required to reach the goal.

CODE:

In [1]:
```
# check if node is valid: within bounds
# and not already visited

def valid(node):
    if(node[0]<=5 and node[1]<=5 and node[0]>=1 and node[1]>=1 and node not in obstacles):
        return True
    return False
```

```
: import numpy as np
  def astar(start,goal):

      maze=[(x,y) for x in range(1,6) for y in range(1,6)] # calculate Mahnattan distances for every valid location
      maze=list(set(maze)-set(obstacles))
      dist=np.sum(np.abs(np.array(goal)-maze),axis=1)

      q=[(dist[maze.index(start)],0,start)]     # initialize variables
      r_state=maze
      r_data=[0]*len(maze)
      r_data[r_state.index(start)]=(dist[maze.index(start)],0)
      cost=0
      while True:

          curr_node=q.pop(0) # remove node from fringe and add it to visited list
          print(curr_node[2],end=' ')

          i,j=curr_node[2] # extract co-ordinates for future use

          if(curr_node[2]==goal): # goal found!
              print("| Total Cost=",cost,"| Effective Cost=",curr_node[1])
              return cost, curr_node[1]
              break
          else: # entertain all possibilities (up,down,left,right)
              nbrs=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)]
              for i in range(4):
                  if(valid(nbrs[i])):
                      # check if node is new or it exists in queue if new node, then simply insert
                      # else check if new path is better than old path
                      if (r_data[r_state.index(nbrs[i])]==0
                      or (curr_node[1]+1)<r_data[r_state.index(nbrs[i])][1]):
                          r_data[r_state.index(nbrs[i])]=((dist[maze.index(nbrs[i])]+curr_node[1]+1),(curr_node[1]+1))
                          q.append(((dist[maze.index(nbrs[i])]+curr_node[1]+1),(curr_node[1]+1),nbrs[i]))
              q.sort()
              cost+=1
```

```
In [3]: import numpy as np

        # below code snippet is for showing the initial state of
        # the maze diagrammatically
        maze=np.zeros(25)
        maze[4]=maze[6]=maze[13]=maze[14]=maze[16]=maze[19]=1
        maze[0]=2
        maze[9]=maze[11]=maze[20]=maze[24]=3

        maze=maze.reshape((5,5))
        print(maze)
        print("")
        # tuples defining the respective entities start state, goal state, and obstacles
        start=(1,1)
        goals=[(3,2),(5,1),(2,5),(5,5)]
        obstacles=[(1,5),(2,2),(3,4),(3,5),(4,2),(4,5)]
        global obstacles

        cost_tot=0
        cost_eff=0
        while(len(goals)>0):

            # find the closest goal using Manhattan Distance
            min_ind=np.argmin(np.sum(np.abs(np.array(goals)-start),axis=1))

            # use a-star on the current state and nearest goal found
            scores=astar(start,goals[min_ind])
            cost_tot+=scores[0]
            cost_eff+=scores[1]

            # repeat the problem with goal state as new start state
            start=goals[min_ind]
            goals.pop(min_ind)

        print("")
        print("Total cost:",cost_tot)
        print("Effective cost:",cost_eff)
```

OUTPUTS:

```
[[2. 0. 0. 0. 1.]
 [0. 1. 0. 0. 3.]
 [0. 3. 0. 1. 1.]
 [0. 1. 0. 0. 1.]
 [3. 0. 0. 0. 3.]]

(1, 1) (1, 2) (2, 1) (3, 1) (3, 2) | Total Cost= 4 | Effective Cost= 3
(3, 2) (3, 1) (4, 1) (5, 1) | Total Cost= 3 | Effective Cost= 3
(5, 1) (5, 2) (5, 3) (5, 4) (5, 5) | Total Cost= 4 | Effective Cost= 4
(5, 5) (5, 4) (4, 4) (5, 3) (4, 3) (3, 3) (2, 3) (2, 4) (2, 5) | Total Cost= 8 | Effective Cost= 7

Total cost: 19
Effective cost: 17
```

OBSERVATION/COMMENTS

1. The algorithm used here is to first find the closest reward by Manhattan Distance. Then use A*-search to move towards the reward. Then repeat the same with the current reward as new start state.

2. Although the above algorithm does not guarantee optimality, it is much faster than standard A*-search. This fact is useful when speed is the priority and sub-optimal solutions are allowed. Although here the effective path is 17 units longs, versus the optimal path length of 16.

Q4. PacMan Game Implementation

Question 1 (3 points) : Finding a Fixed Food Dot using Depth First Search

```
# I initially wanted to create a class to store the parent data
# of nodes, since storing a list for every node is not space-
# efficient. But the question said to not touch codes outside what
# is to be filled, and it needs only the list.

stack=util.Stack()    # fringe
visited=[]            # record of visited nodes

# start state is in fringe with no node history
stack.push((problem.getStartState(),[]))

while(True):
    if(stack.isEmpty()):
        return []

    curr_node=stack.pop()
    visited.append(curr_node[0])

    if(problem.isGoalState(curr_node[0])):   # goal reached, return the list of nodes and actions
        return curr_node[1]

    else:
        # get possible neighbors
        nbrs=problem.getSuccessors(curr_node[0])

        for i in range(len(nbrs)):
            if(nbrs[i][0] not in visited):
                # push unvisited neighbor into the fringe
                stack.push((nbrs[i][0],curr_node[1]+[nbrs[i][1]]))
```

```
bridges@bridges-CF-C2AHCCZC7:~/Downloads/TA_4_5_search$ python3.6 autograder.py
Starting on 9-18 at 14:20:53

Question q1
===========
*** PASS: test_cases/q1/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases/q1/graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases/q1/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q1/graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases/q1/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###
```

OBSERVATION/COMMENTS

Although the method of assigning a list of historical actions to each node is obviously space-inefficient, it makes the problem statement much simpler. Otherwise I would have to create a class for storing parent data, and code the backtracking function.

Question 2 (3 points): Breadth First Search

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    queue=util.Queue()    # fringe
    visited=[]            # record of visited nodes

    # start state is in fringe with no node history
    queue.push((problem.getStartState(),[]))

    while(True):
        if(queue.isEmpty()):
            return []

        curr_node=queue.pop()
        visited.append(curr_node[0])

        if(problem.isGoalState(curr_node[0])):    # goal reached, return the list of nodes and actions
            return curr_node[1]

        else:
            # get possible neighbors
            nbrs=problem.getSuccessors(curr_node[0])

            for i in range(len(nbrs)):
                if(nbrs[i][0] not in visited):
                    # push unvisited neighbor into the fringe
                    # and declare it visited to nullify duplicate entries
                    visited.append(nbrs[i][0])
                    queue.push((nbrs[i][0],curr_node[1]+[nbrs[i][1]]))
```

```
Question q2
===========
*** PASS: test_cases/q2/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###
```

OBSERVATION/COMMENTS

Breadth first search expands more nodes, but ensures completeness. Whereas, depth first search ensures speed at the cost of completeness.

Question 3 (3 points): Varying the Cost Function

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    queue=util.PriorityQueue()    # fringe
    visited=[]                    # record of visited nodes
    queue.push((problem.getStartState(),[]),0) # start state is in fringe with no node history

    while(True):
        if(queue.isEmpty()):
            return []
        curr_node=queue.pop()
        visited.append(curr_node[0])
        if(problem.isGoalState(curr_node[0])):    # goal reached, return the list of nodes and actions
            return curr_node[1]

        else:
            q_states=[]
            for i in queue.heap: # below code extracts nodes that are currently in priority queue
                q_states.append(i[2][0])

            nbrs=problem.getSuccessors(curr_node[0])              # get possible neighbors
            # two cases: 1. if node is neither in visited, nor in queue means it is a new node
            #            2. node is in the queue
            for i in range(len(nbrs)):
                if(nbrs[i][0] not in visited and nbrs[i][0] not in q_states): # if node is new, then push it in the queue
                    queue.push((nbrs[i][0],curr_node[1]+[nbrs[i][1]]),problem.getCostOfActions(curr_node[1]+[nbrs[i][1]]))
                # if node exists then check if current path is cheaper than what currently exists
                elif(nbrs[i][0] in q_states):
                    curr_g_n=problem.getCostOfActions(queue.heap[q_states.index(nbrs[i][0])][2][1])
                    new_g_n=problem.getCostOfActions(curr_node[1]+[nbrs[i][1]])
                    if(new_g_n<curr_g_n): # if new path is cheaper, then update its path + priority
                        queue.update((nbrs[i][0],curr_node[1]+[nbrs[i][1]]),new_g_n)
```

```
Activities   Terminal ▾                                   Sep 18  2:25
                                                  bridges@bridges-CF-C2AHCCZC7: ~
Question q3
===========
*** PASS: test_cases/q3/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q3/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q3/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q3/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q3/ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases/q3/ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases/q3/ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

OBSERVATION/COMMENTS

Sometimes, UCS behaves the same as BFS, as we can see in test case 1. This is because the cost of movement is same throughout the maze.

Question 4 (3 points): A search*

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    queue=util.PriorityQueue()    # fringe
    visited=[]                    # record of visited nodes
    queue.push((problem.getStartState(),[]),heuristic(problem.getStartState(),problem)) # start state is in fringe with no node
history
    while(True):
        if(queue.isEmpty()): return []
        curr_node=queue.pop()
        visited.append(curr_node[0])
        if(problem.isGoalState(curr_node[0])):    # goal reached, return the list of nodes and actions
            return curr_node[1]

        else:
            q_states=[]    # below code extracts nodes that are currently in priority queue
            for i in queue.heap:
                q_states.append(i[2][0])
            nbrs=problem.getSuccessors(curr_node[0])             # get possible neighbors

            # two cases: 1. if node is neither in visited, nor in queue means it is a new node
            #            2. node is in the queue
            for i in range(len(nbrs)):
                # if node is new, then push it in the queue
                if(nbrs[i][0] not in visited and nbrs[i][0] not in q_states):
                    queue.push((nbrs[i][0],curr_node[1]+[nbrs[i][1]]),problem.getCostOfActions(curr_node[1]+[nbrs[i]
[1]])+heuristic(nbrs[i][0],problem))
                elif(nbrs[i][0] in q_states): # if node exists then check if current path is cheaper than what currently exists
                    curr_g_n=problem.getCostOfActions(queue.heap[q_states.index(nbrs[i][0])][2][1])
                    new_g_n=problem.getCostOfActions(curr_node[1]+[nbrs[i][1]])
                    if(new_g_n<curr_g_n):    # if new path is cheaper, then update its path + priority
                        queue.update((nbrs[i][0],curr_node[1]+[nbrs[i][1]]),new_g_n+heuristic(nbrs[i][0],problem))
```

```
Question q4
===========
*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         222
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

OBSERVATION/COMMENTS

For Pacman layout ‘mediumMaze’ A* performs better than UCS, this is because A* uses extra information that guides its search.

Question 5 (3 points): Finding All the Corners

```
def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1,1), (1,top), (right, 1), (right, top))
    for corner in self.corners:
        if not startingGameState.hasFood(*corner):
            print('Warning: no food in corner ' + str(corner))
    self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
    # Please add any code here which you would like to use
    # in initializing the problem


    "*** YOUR CODE HERE ***"
    # copy of corner co-ordinates
    # I won't use the actual self.corners tuple,
    # to preserve its data
    self.corners_unvisited=list(self.corners)

def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    # return starting position and unvisited corners
    return self.startingPosition, self.corners_unvisited
```

```
def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    # goal state is when all corners
    # are visited, which means the
    # unvisited list will have 0 elements
    return len(state[1])==0
```

```
successors = []
for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
    # Add a successor state to the successor list if the action is legal
    # Here's a code snippet for figuring out whether a new position hits a wall:
    #   x,y = currentPosition
    #   dx, dy = Actions.directionToVector(action)
    #   nextx, nexty = int(x + dx), int(y + dy)
    #   hitsWall = self.walls[nextx][nexty]

    "*** YOUR CODE HERE ***"
    # different from: new_corners_status=state[1],
    # this will only give the reference to the list,
    # which can be tampered with.

    new_corners_status=state[1][:]
    #  Here's a code snippet for figuring out whether a new position hits a wall
    x,y = state[0]
    dx, dy = Actions.directionToVector(action)
    nextx, nexty = int(x + dx), int(y + dy)
    hitsWall = self.walls[nextx][nexty]

    # if valid state,
    if(hitsWall==False):
        # check if valid state is one of the unvisited corners,
        # update the unvisited list status accordingly
        if((nextx,nexty) in new_corners_status):
            new_corners_status.pop(new_corners_status.index((nextx,nexty)))
        # add state as successor anyway
        successors.append((((nextx,nexty),new_corners_status),action,1))

self._expanded += 1 # DO NOT CHANGE
return successors
```

```
Question q5
===========
*** PASS: test_cases/q5/corner_tiny_corner.test
***      pacman layout:           tinyCorner
***      solution length:                 28

### Question q5: 3/3 ###
```

OBSERVATION/COMMENTS

I am maintaining the record of unvisited corners as a list, then I'm returning the start state as well as the list of unvisited corners. Changing the list of univisited corners, by popping corners that are visited, allows for straightforward algorithm for successors.

Question 6 (3 points): Corners Problem: Heuristic

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    # distance from current node to
    # farthest unvisited corner is my heuristic
    max_corner_dist=0

    if(problem.isGoalState(state)==False):
        # update maximum distance from an unvisited corner
        for i in range(len(state[1])):
            if(util.manhattanDistance(state[0],state[1][i])>max_corner_dist):
                max_corner_dist=util.manhattanDistance(state[0],state[1][i])
        return max_corner_dist

    # if goal state, then heuristic should be zero
    return 0 # Default to trivial solution
```

```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'West', 'West', 'South', 'South', 'South', 'West', '
West', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'Wes
t', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'East', 'East', 'East', 'E
ast', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'N
orth', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'North', 'North', 'East', 'East', 'South', 'South'
, 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North',
 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 1140 nodes

### Question q6: 3/3 ###
```

OBSERVATION/COMMENTS

Earlier my heuristic was the average Manhattan distance from each corner, but then I noticed that the maximum distance from any corner is a better heuristic, because the agent will have to move atleast those number of times. From this, I learnt that a tighter lower bound on heuristic is generally better.

Question 7 (4 points): Eating All The Dots

```
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    # maze distance from current node to
    # farthest food pellet is my heuristic
    max_food_dist=0

    for i in range(len(foodGrid.asList())):

        # check distance from current node to farthest food pellet
        if(mazeDistance(position, foodGrid.asList()[i], problem.startingGameState)>max_food_dist):
            max_food_dist=mazeDistance(position, foodGrid.asList()[i], problem.startingGameState)

    return max_food_dist
```

```
Question q7
===========
*** PASS: test_cases/q7/food_heuristic_1.test
*** PASS: test_cases/q7/food_heuristic_10.test
*** PASS: test_cases/q7/food_heuristic_11.test
*** PASS: test_cases/q7/food_heuristic_12.test
*** PASS: test_cases/q7/food_heuristic_13.test
*** PASS: test_cases/q7/food_heuristic_14.test
*** PASS: test_cases/q7/food_heuristic_15.test
*** PASS: test_cases/q7/food_heuristic_16.test
*** PASS: test_cases/q7/food_heuristic_17.test
*** PASS: test_cases/q7/food_heuristic_2.test
*** PASS: test_cases/q7/food_heuristic_3.test
*** PASS: test_cases/q7/food_heuristic_4.test
*** PASS: test_cases/q7/food_heuristic_5.test
*** PASS: test_cases/q7/food_heuristic_6.test
*** PASS: test_cases/q7/food_heuristic_7.test
*** PASS: test_cases/q7/food_heuristic_8.test
*** PASS: test_cases/q7/food_heuristic_9.test
*** PASS: test_cases/q7/food_heuristic_grade_tricky.test
***     expanded nodes: 4239
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###
```

OBSERVATION/COMMENTS

Earlier I was using Manhattan distance for my heuristic. But then I noticed that they had already implemented mazeDistance, which is a more accurate measure for distance to a food pellet. Replacing Manhattan distance, with maze Distance worked for me, but honestly, I did not expect I would get extra credit for that (5/4!).

Question 8 (3 points): Suboptimal Search

```python
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    "*** YOUR CODE HERE ***"
    # use uniform-cost search to find nearest food pellet
    # UCS was chosen as it is always optimal.
    return search.uniformCostSearch(problem)

def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state

    "*** YOUR CODE HERE ***"
    # check if current state belongs to
    # the list of food co-ordinates
    if(state in self.food.asList()):
        return True
    return False
```

```
Question q8
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_1.test
***     pacman layout:          Test 1
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_10.test
***     pacman layout:          Test 10
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_11.test
***     pacman layout:          Test 11
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_12.test
***     pacman layout:          Test 12
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_13.test
***     pacman layout:          Test 13
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_2.test
***     pacman layout:          Test 2
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
```

```
*** PASS: test_cases/q8/closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_4.test
***     pacman layout:          Test 4
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_5.test
***     pacman layout:          Test 5
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_6.test
***     pacman layout:          Test 6
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_7.test
***     pacman layout:          Test 7
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_8.test
***     pacman layout:          Test 8
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases/q8/closest_dot_9.test
***     pacman layout:          Test 9
***     solution length:                1

### Question q8: 3/3 ###


Finished at 14:21:29
```

OBSERVATION/COMMENTS

I chose UCS as my search algorithm, as it gurantees optimality and completeness, although A* is faster.

AUTOGRADER RESULTS