[CSE 537 : ARTIFICIAL INTELLIGENCE]

PROJECT REPORT v

Project 1 : The Searchin' Pacman

By:
Robin Manhas (111463105)
Harsh Gupta (111470974)

Table of Contents

Introduction

The aim of this project report is to show a critical analysis of our implementations of the following searching algorithms in the Pacman project

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- A* search

In this project, we were provided with a nice implementation of Pacman game (Refer : http://www3.cs.stonybrook.edu/~cse537/project01.html)

Pac-Man agent had to find path through its maze world, both to reach a particular location and to collect food efficiently. Our first job was to build general search algorithms and apply them to Pac-Man scenarios.

The project was sub divided into 7 different problem statements, which shall be individually explored later.

Adhering to the hint provided for implementing search algorithms in the project page, (http://www3.cs.stonybrook.edu/~cse537/project01.html), we have created a unified implementation for DFS, BFS, UCS and A* algorithms, which shall be discussed next. The only changes are in the details of how the fringe is maintained. The generic search method takes into account the fringe as well as heuristic function (if any) and performs the search using algorithm specific queuing strategy.

Generic search method

We implemented a common search method called “solveTheTraversalProblem” which takes in the following arguments:

1. Problem: The search problem object
2. Type: To differentiate between DFS, BFS, UCS and A*
3. Fringe: The initialized data structure to be used for queuing
4. Heuristic: Heuristic function (if any)

```
def solveTheTraversalProblem(problem,type,fringe,heuristic=nullHeuristic):
visitedList = []
returnList = []
while(fringe.isEmpty() == False):
    parent = fringe.pop()
    #print "popped: ",parent
    if(problem.isGoalState(parent[0]) == True):
        returnList = parent[1]
        break
    if parent[0] not in visitedList:
        childList = problem.getSuccessors(parent[0])
        for child in childList:
            if child[0] not in visitedList:
                path = parent[1] + [child[1]]
                #print "adding child: , ", childNode
                cost = 0
                if(type == 3):
                    cost = parent[2]+child[2]+heuristic(child[0],problem)-heuristic(parent[0],problem)
                    childNode = (child[0], path, cost)
                    fringe.update(childNode,cost)
                elif(type == 2):
                    cost = parent[2] + child[2]
                    childNode = (child[0], path, cost)
                    fringe.update(childNode,cost)
                else:
                    childNode = (child[0], path, child[2])
                    fringe.push(childNode)
        visitedList.append(parent[0])

return returnList
```

Every node (parent or child node) are a tuple composed of 3 values: Location on grid, List of directions taken since source to reach the node and the Cost incurred on the path. On reaching the goal state, the list of directions is copied in th returnList and returned to the caller method.

Problem 1: Depth First Search Algorithm

Implementation

As explained in the previous section, the generic search algorithm requires problem, type and fringe as the arguments. In our depthFirstSearch method, we initialize the fringe as a stack data structure using the provided "util.py".

```
def depthFirstSearch(problem):
fringe = util.Stack()
initList = []
root = (problem.getStartState(),initList,0)
fringe.push(root)
return solveTheTraversalProblem(problem,0,fringe)
```

Initially, the root node has no direction, hence the tuple is created with the direction list as null. The same is passed to generic search algorithm, which uses the stack fringe to explore the tree as far as possible in the same branch until the end is reached, before backtracking.

Results

The results collected for the commands mentioned in project page have been summarized below.

Command	Time taken (in seconds)	Search nodes expanded	Total cost
`python pacman.py -l tinyMaze -p SearchAgent`	**0.00086**	**15**	**10**
`python pacman.py -l mediumMaze -p SearchAgent`	**0.00830**	**146**	**130**
`python pacman.py -l bigMaze -p SearchAgent`	**0.02492**	**390**	**210**

Based on the visualization, we could clearly see (especially evident in mediumMaze) that the path traversed by pacman wasn't optimal. It kept exploring the straight path which was a longer route to the destination. Clearly there existed a shorter route to destination (had it taken a left rather than going straight after starting from the start state).

Problem 2: Breadth First Search Algorithm

Implementation

In our breadthFirstSearch method, we initialize the fringe as a queue data structure using the provided "util.py"

```
def breadthFirstSearch(problem):
fringe = util.Queue()
initList = []
root = (problem.getStartState(), initList, 0)
fringe.push(root)
return solveTheTraversalProblem(problem, 1, fringe)
```

Initially, the root node has no direction, hence the tuple is created with the direction list as null. The same is passed to generic search algorithm, which uses the stack fringe to explore the tree as far as possible in the same branch until the end is reached, before backtracking.

Results

The results collected for the commands mentioned in project page have been summarized below.

Command	Time taken (in seconds)	Search nodes expanded	Total cost
`tinyMaze -p SearchAgent -a fn=bfs`	**0.00101**	**15**	**8**
`mediumMaze -p SearchAgent -a fn=bfs`	**0.01474**	**269**	**68**
`bigMaze -p SearchAgent -a fn=bfs`	**0.04349**	**620**	**210**

Comparing the performance of BFS with DFS, we can clearly observe that although BFS takes a little longer to complete the operations and at times expands more nodes, the results are more optimal than the DFS, as evident in case of mediumMaze. The DFS took the longer path whereas the BFS was able to discover the shorter path to goal, as it searched level by level instead of going for shallow nodes exploration first.

Problem 3: Uniform Cost Search

Implementation

We then implemented UCS using priority queue data structure implemented in util.py. The data structure gave us the functionality to update the priority (cost incurred to reach node) for a node in the queue.

```
def uniformCostSearch(problem):

fringe = util.PriorityQueue()
initList = []
root = (problem.getStartState(), initList, 0)
fringe.push(root,0)

return solveTheTraversalProblem(problem, 2, fringe)
```

Results

Command	Time taken (in seconds)	Search nodes expanded	Total cost
`tinyMaze -p SearchAgent -a fn=ucs`	**0.00093**	**15**	**8**
`mediumMaze -p SearchAgent -a fn=ucs`	**0.01555**	**269**	**68**
`bigMaze -p SearchAgent -a fn=ucs`	**0.04523**	**620**	**210**
`mediumDottedMaze -p StayEastSearchAgent`	**0.01019**	**186**	**1**
`mediumDottedMaze -p StayWestSearchAgent`	**0.01049**	**169**	**17183894840**

Comparing the results of UCS with BFS, we do not observe much difference, as the cost of traversal between the nodes for both the algorithms is same (i.e., 1). Therefore, the cost to reach a certain node will always be incremented by 1 in case of UCS, which makes it similar to performing BFS, as priority queue will be acting similar to a normal queue.

Also taking a closer look at StayEast and StayWest search agent results, we see that the total cost is very high for West and very low for East (1) because of the exponential cost functions. The results for the same have been appended in the table above.

Problem 4: A* Algorithm

Implementation

The A* algorithm requires 2 parameters, the problem and the heuristic function. We tweaked the generic search method to incorporate the differences in calculating the cost for UCS and A*. Assuming that the heuristic function gives the cost as ***h(n)***, the cost ***f(n)*** for each node is given as:

$$f(n) = g(n) + h(n)$$

Where ***g(n)*** is the cost of traversing from start to current node (similar to cost in UCS).

```
def aStarSearch(problem):
fringe = util.PriorityQueue()
initList = []
root = (problem.getStartState(), initList, heuristic(problem.getStartState(),problem))
fringe.push(root,heuristic(problem.getStartState(),problem))
return solveTheTraversalProblem(problem, 3, fringe, heuristic)
```

Results

Command	Time taken (in seconds)	Search nodes expanded	Total cost
`tinyMaze astar manhattanHeuristic`	**0.000827**	**14**	**8**
`mediumMaze astar manhattanHeuristic`	**0.012446**	**221**	**68**
`bigMaze astar manhattanHeuristic`	**0.035675**	**549**	**210**

Comparing the performances of A* using Manhattan distance as heuristic vs UCS, we observe that A* is a far better choice in terms of nodes expanded or time taken by the algorithms in all the given 3 maze types. Major difference can be seen in bigmaze, where the UCS expands 620 nodes, the A* algorithm expands just 549.

Same improvement can also be seen for mediumMaze, where UCS expands 269 nodes, astar expands just 221. Hence it can be safely concluded that A* with manhattan distance as heuristic functions is a better search algorithm for the current problem set.

Problem 5: Traversing all 4 corners of the game

Implementation

Two different approaches were tried for the state representation that encoded all the necessary information. Firstly, we created a tuple inside the getStartState function, that returned a tuple of format: (self.startingPosition, corners). Hence the information about the 4 corners was appended. However, this approach seemed to be too transparent, in that it exposed the co-ordinates of 4 corners explicitly.

Our current implementation:

```
def getStartState(self):
   "*** YOUR CODE HERE ***"
   corners = [0,0,0,0]
   return (self.startingPosition, corners)
```

To avoid this, we instead passed an array of length 4, where the array index 0,1,2,3 represent the North, South, East and West directions respectively. 0 value means the corner has NOT been visited yet. 1 value means the corner has already been visited by Pacman.

The updation of these values are done inside the getSuccessor method (as this method always gets called whenever a new position is reached).

```
def getSuccessors(self, state):
    ********** Some code before *************************
     # here we update the corner array based on which corner position was reached (if any)
          if nextState == self.corner1:
             corner = [1 ,cornersList[1], cornersList[2], cornersList[3]]
          elif nextState == self.corner2:
             corner = [cornersList[0],1,cornersList[2],cornersList[3]]
          elif nextState == self.corner3:
             corner = [cornersList[0],cornersList[1],2,cornersList[3]]
          elif nextState == self.corner4:
             corner = [cornersList[0],cornersList[1],cornersList[2],3]
```

Results:

Command	Time taken (in seconds)	Search nodes expanded	Total cost
`tinyCorners SearchAgent bfs CornersProblem`	**0.00683**	**252**	**28**
`mediumCorners SearchAgent bfs CornersProblem`	**0.24333**	**1966**	**106**

As described in the project page, bfs on corner problem is able to expand just under 2000 nodes for medum corner problem, which shows that the corners state representation was implemented successfully.

Problem 6: Corner's Heuristic

Implementation

The thought process behind our implementation of corner's heuristic was that from any given location of in the grid, we must first identify which quadrant of the grid is pacman present and if there is any unvisited corner in that quadrant. If not, find any other nearest corner, calculate its distance and then add the distance to next nearest unvisited corner from this corner, and so on until all corners are visited.

In order to make the heuristic admissible and consistent, we used manhattan distance between pacman position and corners, which is a sure way of getting the lower bound on maximum distance needed to be covered for visiting all corners.

```
def cornersHeuristic(state, problem):
******** initialization and declaration code here **********
for i in range(0,4):
    if visitedCorner[i] == 0:
        toVisit += [cornerList[i]]. # add all corners to to visit list

while len(toVisit) != 0: # while there are corners to be visited
    curDist = 99999
    curNode = -curDist
    for i in range(len(toVisit)): # find an unvisited corner with min distance from current position
        dist = getManhattanDistance(position, toVisit[i])
        if dist < curDist and dist >= 0:
            curDist = dist
            curNode = i

    # modify heuristic and change position
    totalDist += curDist # append current distance to the total distance (heuristic cost)
    position = toVisit[curNode] # update current position as the position of corner
    toVisit.remove(toVisit[curNode])

return totalDist
```

Result

The results for mediumCorners problem using AStar is as follows:

```
Pacman robinmanhas$ python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
start: (5, 1)
corners: ((1, 1), (1, 12), (28, 1), (28, 12))
Time taken:  0.044162
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 692
```

As observed, the nodes expanded are just **692**.

Problem 7: Food Heurstic

Implementation

To come up with an admissible and consistent heuristic, we came up with 3 different approaches. The first one was just an extension to corner's heuristic as described above, just that we had added all the nodes returned by foodGrid.asList() into the toVisit array. However, because the size of food items list was too huge, it led to producing a very slow and inadmissible heuristic.

The next approach (commented code) was to first find nearest food from pacman and then nearest food from this location until all food items are finished, using the mazeDistance utility function. This approach, although gave the results quickly (Search nodes expanded as **95** with a cost of **60** in **5** seconds for tricky Search), was failing the "test_cases/q7/food_heuristic_15.test" admissibility test.

We read the current admissible approaches used to tackle such problems (refer: http://lucieackley.com/heuristic.pdf) and came up with our own improved heuristic function that takes the actual distance between the 2 food points (considering the walls and other conditions), instead of manhattan distance, to find out accurately the distance of 2 points in maze. We used this distances between each node to create a form of minimum spanning tree (using priority queue to pick up least weighted edges in each iteration). Finally, when all the food nodes were connected, we found the nearest food item from pacman position and hence added the pacman to the tree.

The total distance of this minimum spanning tree acted as an admissible and consistent heuristic because minimum spanning tree for the food grid gave us the minimum distance the pacman MUST cover, in order to eat (visit) all the food items, and did not overestimate the distance at any times.

To further refine our heuristic and make it faster, we created a dictionary mapping of the mazeDistance between 2 nodes and saved the calculated distances in the dictionary. Our source and destination index being key (as tuples), the distance being value. Both source,dest and dest,source key were inserted once the distance was calculated. This helped us in saving close to 2 seconds of recalculation time.

Core Logic:

```
def foodHeuristic(state, problem): **********Initializations removed **********
# take arbitraty food point from unvisited list, add it to visted and find nearest neighbours
for vertex in toVisit:
    node = (firstNode,vertex)
    nodeRev = (vertex,firstNode)
    if node in distanceList:
        fringe.push(node, distanceList[node])
    elif nodeRev in distanceList:
        fringe.push(node,distanceList[nodeRev])
    else:
        dist = mazeDistance(firstNode, vertex,gameState) # push vertex in priority queue based on distance
        fringe.push(node,dist)
        distanceList[node] = dist
        distanceList[nodeRev] = dist

while(fringe.isEmpty() == False):   # pop from priority queue based on least distance
    item = fringe.pop()
    if item[1] not in visited:
        visited.add(item[1])
        itemRev = (item[1],item[0])
        if item in distanceList:
            total += distanceList[item]
        elif itemRev in distanceList:
            total += distanceList[itemRev]
        else:
            dist = mazeDistance(item[0], item[1], gameState) # calculate maze distance source and dest
            total += dist
            distanceList[item] = dist
            distanceList[itemRev] = dist
        for n in toVisit:       # calculate dist between dest and all adjoining unvisited food items, add to
                                pqueue
            if n not in visited:
                remaining = (item[1],n)
                remainingRev = (n,item[1])
                if remaining in distanceList:
                    fringe.update(node, distanceList[remaining])
                elif remainingRev in distanceList:
                    fringe.update(node, distanceList[remainingRev])
                else:
                    dist = mazeDistance(item[1], n, gameState)
                    fringe.update(remaining, dist)
                    distanceList[remaining] = dist
                    distanceList[remainingRev] = dist
```

```
for vertex in visited:  # finally once all food items are visited, find the one closest to pacman and connect tree
    node = (position, vertex)
    nodeRev = (vertex,position)
    if node in distanceList:
        dist = distanceList[node]
    elif nodeRev in distanceList:
        dist = distanceList[nodeRev]
    else:
        dist = mazeDistance(position, vertex, gameState)
    if (dist < curMin):
        curMin = dist
return total + curMin      #  return minimum spanning tree distance
```

Result

Running command " python pacman.py -l trickySearch -p AStarFoodSearchAgent" yielded the following result :

Command	Time taken (in seconds)	Search nodes expanded	Total cost
trickySearch AStarFoodSearchAgent	16.02	255	**60**

As can be seen from the results above, the heuristic implemented expands quite minimal nodes and completes the search in reasonable time.

References:

1. Project reference page: http://www3.cs.stonybrook.edu/~cse537/project01.html
2. Python tutorials: http://www3.cs.stonybrook.edu/~cse537/Python-Tutorial.html
3. Proof of Admissibility and Consistency of Food Problem Heuristic http://lucieackley.com/heuristic.pdf
4. Prims algorithm for minimum spanning tree: http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/