Lab1 - Search

Naman Goyal (2015CSB1021)

January 27, 2017

1 Introduction

The Lab implements search in Pacman world in Python. The reports presents the heuristic function along with the question specific observations make during implementation of the lab.

2 Heuristic

Objective To device an admissible consistent heuristic function for A* Search for 2 search problems - namely visiting all corners, eating all food dots. Visiting all corners is similar to eating all food dots where the unvisited corners in the search space are replaced by food dots.

Inspiration The problem can be modeled as finding a lower bound for Travelling Salesman Problem where the Pacman position is the start city and all the remaining food dots are cities to be visited. There is only one difference with TSP i.e. the saleman - Pacman - only visits start city once.

Strategy The problem was reduced to finding a lower bound for this search space. A minimum spanning tree was constructed using Prim's Algorithm for all the food dots. The heuristic function value is the sum of edges weights in the MST and the shortest edge weight connecting Pacman position to any node in MST.

$$h(n) = \sum_{Edges} Edges\,Weights\,MST + minDistance(Pacman\,Position, MST) \leq h^*(n) \tag{1}$$

2.1 Rationale

Admissible The heuristic is admissible because any path has to pass through all the food dots. Hence minimum path cost is lower bounded by sum of edges in the MST and shortest distance between Pacman and MST.

Consistency Since the parameter used to calculated edge weights - mazeDistance, manhattanDistance - follow triangular inequality. Any heuristic build upon them is consistent.

2.2 Statistics

The table populates the search nodes expanded for trickySearch in Question 7.

Heuristics	**Search Nodes expanded**	**Improvement over Null**
Null	16688	0 %
manhattanDistance	7137	57.23 %
mazeDistance	255	98.47 %

```python
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.
    """
    position, foodGrid = state

    foodList = foodGrid.asList()

    if problem.isGoalState(state): # If goal heurisitic returns 0
        return 0

    value = 0 # Intialise the heuristic value = 0
    closestFood = min(foodList, key= lambda x: mazeDistance(x, position, problem.startingGameState))
    closestFoodDist = mazeDistance(closestFood, position, problem.startingGameState)
    value += closestFoodDist # Add the distance from closed food to pacman

    """Get Minimum Spanning Tree of unvisited food dots using Prim's"""
    mstDict = {x: False for x in foodList}
    Queue = util.PriorityQueue()

    keyDict = {x:float("inf") for x in foodList} # Intialize infinity weights
    keyDict[closestFood] = 0  # Make closed food dot distance 0

    for x in keyDict: # Add all food dots to Queue
        Queue.push(x, keyDict[x])

    while sum(mstDict.values()) < len(mstDict) :
        u = Queue.pop()
        mstDict[u] = True
        value += keyDict[u] # Add edge weight to path cost
        for v in foodList:
            if mstDict[v] == False and mazeDistance(u, v, problem.startingGameState) < keyDict[v]:
                keyDict[v] = mazeDistance(u, v, problem.startingGameState) # Update edge weight for adjacent food dots to currently selected food dot
                Queue.update(v, keyDict[v])

    return value
```

Code 1: foodHeuristic

Hence an improvement of 98.47 % was observed over an A* Star with Null Heuristics i.e. expanding just 255 nodes rather than 16688 nodes for trickySearch.

3 Other Observations

3.1 Depth First Search

The Depth first Search could be implemented both iteratively and recursively.

```python
1 def depthFirstSearchIterative(problem):
2         """Generate all successors"""
3         for successor, action, stepCost in  problem.getSuccessors(state):
4             if successor not in explored: # Add to frontier if not in explored
5                 frontier.push(successor)
6                 parent[successor] = (action, state)
```

Code 2: Snippet from Depth First Search Iterative

```python
1 def breadthFirstSearch(problem):
2         """Generate all successors"""
3         for successor, action, stepCost in  problem.getSuccessors(state):
4             if successor not in explored + frontier.list: # Add to frontier if not in
      explored or frontier
5                 frontier.push(successor)
6                 parent[successor] = (action, state)
```

Code 3: Snippet from Breadth First Search

It was observed that the path found by the iterative DFS version is correct if and only if the same node is allowed to be added multiple times in frontier. This is unusual from normal graph search - compare line 4 in DFS Code 2 and BFS Code 3. This can be explained by following graph.

We assume both iterative and recursive version visit the nodes from left to right on same level.

$$\begin{tikzcd}
 & A \arrow[dl, "0.3"'] \arrow[dr, "0.1"] & \\
B \arrow[rr, "0.4"] & & C
\end{tikzcd}$$

Then if C is the goal node then solution resulted by DFS should be (A -> B -> C) with a cost of 0.7

```
Step 1: Add A to frontier. | Old Frontier = [] | New frontier = [A]
Step 2: Expand A. | Old Frontier = [A] | New frontier = [B, C[1]]
Step 3: Expand B. | Old Frontier = [B, C[1] | New frontier = [C[2], C[1]]
Step 4: Expand C[2]. | Return solution (A -> B -> C) cost = 0.7
```

DFS : Allow same node to be added multiple times in frontier

```
Step 1: Add A to frontier. | Old Frontier = [] | New frontier = [A]
Step 2: Expand A. | Old Frontier = [A] | New frontier = [B, C[1]]
Step 3: Expand B. | Old Frontier = [B, C[1] | New frontier = [C[1]]
Step 4: Expand C[1]. | Return solution (A -> C) cost = 0.1
```

DFS : Don't allow same node to be added twice in frontier

[1]Instance of node C generated as successor of A in frontier
[2]Instance of node C generated as successor of B in frontier

References

[1] Estimating the Held-Karp lower bound for the geometric TSP
`http://www.sciencedirect.com/science/article/pii/S0377221796002147`