CSE 537 - Artificial Intelligence

Report: Project 1

Khan Mostafa 109365509
Abhijit Betigeri 109229784

`{khan.mostafa, abhijit.betigeri}@stonybrook.edu`
Department of Computer Science
Stony Brook University

Finding Fixed Food dots using Search Algorithms

Q1. DFS

Methodology Used: A generic search function using stack as the fringe. DFS starts by expanding from the root and goes all the way down to leaf nodes for search.

Execution Details

Set 1:

```
python pacman.py -l tinyMaze -p SearchAgent
```

Function: `depthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `10`
Search Nodes Expanded: `15`

Set 2:

```
python pacman.py -l mediumMaze -p SearchAgent
```

Function: `depthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `130`
Search Nodes Expanded: `146`
Time: `0.1s`

Set 3:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Function: `depthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `210`
Search Nodes Expanded: `390`
Time: `0.1s`

Q2. BFS

Methodology Used: A generic search function with queue as fringe. BFS works by expanding each level one by one and return a successful result when a goal state is found in one level. It returns optimal, in term of length, path to result.

Execution Details
<u>Set 1</u>:
```
python pacman.py -l tinyMaze -p SearchAgent –a fn=bfs
```

Function: `breadthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `8`
Search Nodes Expanded: `15`
Time: `0.1s`

<u>Set 2</u>**:**
```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

Function: `breadthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `68`
Search Nodes Expanded: `269`
Time: `0.1s`

<u>Set 3</u>**:**
```
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

Function: `breadthFirstSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `210`
Search Nodes Expanded: `620`
Time: `0.3s`

<u>Note</u>: `python eightpuzzle.py` works correctly with same generic function.

Varying the Cost Function

`MediumDottedMaze` & `MediumScaryMazes` are used – here the cost function is varied taking into consideration dangerous steps & food rich areas.

Q3. Uniform Cost Search

Methodology Used: Same generic function as BFS and DFS is also used for UCS with a PriorityQueue as the fringe. For each expandable node in fringe, an associated path cost is also added as priority key in the fringe. Lower cost nodes are expanded first.

Execution Details:
<u>Set 1</u>:
```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

Function: UniformCostSearch (ucs)
Problem Class: PositionSearchProblem
Total Cost for the Path Found: 68
Search Nodes Expanded: 269
Time: `0.1s`

<u>Set 2</u>:
```
python pacman.py –l mediumDottedMaze -p StayEastSearchAgent
```

Total Cost for the Path Found: `1`
Search Nodes Expanded: `186`
Time: `0.1s`

Set 3:

```
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Total Cost for the Path Found: `17183894840`
Search Nodes Expanded: `169`
Time: `0.1s`

A* Search:

Q4. A* Search

Methodology Used: A generic search function with priority queue as the fringe is used for search. As the priority key, a cost function `f(n) = g(n) + h(n)` associated with the state (node) is used. Here, `g(n)` is the actual cost to reach to that node, and `h(n)` is an estimate found using a heuristic function.

Execution Details:

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

Function: `aStarSearch`
Problem Class: `PositionSearchProblem`
Total Cost for the Path Found: `210`
Search Nodes Expanded: `549`
Time: `0.1s`

Compared to UCS – A* gives an optimal solution in better time.

E.g. `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=ucs`
Total Cost for the Path Found: `210`
Search Nodes Expanded: `620`
Time: `0.1s`
Here search nodes expanded is 620 compared to 549 in A*.

Finding All the Corners

Finding whether all the four corners have been reached.

Q5. Detection of all four corners reached

Methodology Used: We Use the bitmap structure to keep track of all the corners visited. While expanding a node to get the set of successors, the corner vectors are also updated accordingly to reflect whether that successor would visit a new corner or not.

Execution Details:
Set 1:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Function: `breadthFirstSearch`

Problem Class: `CornersProblem`
Total Cost for the Path Found: `29`
Search Nodes Expanded: `277`
Time: `0.1s`

Set 2:
```
python pacman.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

Function: `breadthFirstSearch`
Problem Class: `CornersProblem`
Total Cost for the Path Found: `107`
Search Nodes Expanded: `2006`
Time: `0.1s`

Q6. Finding all Corners with A* Search

Heuristic description: We are computing the cost of the nearest corner from current location plus the cost to visit the rest of the remaining corners. Manhattan distance is used a measure of distance.

Execution Details:
```
python pacman.py -l mediumCorners -p SearchAgent –a
fn=aStar,prob=CornersProblem,heuristic=cornersHeuristic
```

Function: `aStarSearch`
Heuristic: `cornersHeuristic`
Problem Class: `CornersProblem`
Total Cost for the Path Found: `107`
Search Nodes Expanded: `709`
Time: `0.1s`

Eating All the Dots

Checking of the code with `testSearch` for `FoodSearchProblem`.

```
python pacman.py -l testSearch -p AStarFoodSearchAgen
```

Or,
```
python pacman.py -l testSearch -p SearchAgent -a
fn=astar,prob=FoodSearchProblem,heuristic=nullHeuristic
```

Function: `aStarSearch`
Heuristic: `nullHeuristic`
Problem Class: `FoodSearchProblem`
Total Cost for the Path Found: `7`
Search Nodes Expanded: `14`

UCS algorithm with `tinySearch` Layout – very Slow:
```
python pacman.py -l tinySearch -p SearchAgent -a
fn=ucs,prob=FoodSearchProblem
```

Function: `uniformCostSearch`
Problem Class: `FoodSearchProblem`

Total Cost for the Path Found: `27`
Search Nodes Expanded: `5057`
Time: `0.4s`

Q7: Optimal way to eat all dots with A* search

Methodology used to solve the problem:
The foods are visualized to be on the circumference of the convex hull. The convex hull is constructed taking into consideration several foods and such that no food falls outside convex hull but there might be food inside convex hull. Then nearest food (in terms of the location) present on the convex hull from the Pac man is considered – that food is traversed and so on.

Food locations can be considered as points on a plane. Based on this model two heuristics are used. For the heuristic, we relax the problem to have no internal walls and try to estimate a lower bound of shortest path to visit all these points. An actual estimate will be as hard as traveling salesman problem. So, two safe and faster estimates are used as heuristic function to solve the problem:

`convexArchLenPlusDistance`: It finds a convex hull of all points. If the largest arm of the hull is taken out, we get an arch, what we can call the convex arch. This arch includes all points on the hull. The length of the arch is the cost to visit all points on the convex hull. This is a lower bound on visiting all points. To better fit out heuristic to actual cost, we also consider the cost of Pac man to move onto the arch. It is apparent that, the cost for Pac man to reach out to the arch will be no lesser than the minimum distance between Pac man and any point on the arch. For computing convex hull of points, an openly available python source by Tom Switzer is used. (Switzer, n.d.)

`quadrantExtremesDistance`: The grid can be divided into four natural quadrants with respect to Paceman's location. We can find farthest point in each quadrant. It is apparent that, reaching all these extremes is a lower bound to actual cost. We estimate the cost to reach to the closest of the extremes plus the rest of the extremes thereon.

Max of the two heuristics is used for better estimate. Which works by expanding just under 7000 nodes for `trickySearch` problem.

Execution Detail:
```
python pacman.py -l trickySearch -p SearchAgent -a
fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

Function: `aStarSearch`
Heuristic: `nullHeuristic`
Problem Class: `FoodSearchProblem`
Total Cost for the Path Found: `60`
Search Nodes Expanded: `6945`
Time: `2.6s`

We just expand 6945 nodes to find optimal path. Exhaustive search with null heuristic/BFS would expand 16688 node in 2.9 seconds.

A sanity check of the heuristic can be done with `easySearch`, which is done on same grid as `trickySearch`, with all internal walls removed. In which a BFS finds the optimal path of cost 29.

Greedy search non-optimal solution for `MediumMaze`

Solving `MediumMaze` optimally within reasonable time with A* search seems to be a hard problem. However, greedy search (where `f(n)=h(n)`) can find a (non-optimal) path. A greedy search with the same `foodHeuristic` finds a path of cost 210 by expanding over thirty-five thousand nodes. However, a more naïve and greedier estimate `pointsLeftEstimate` (which considers only number of foods left) yields a path of cost 184 by expanding a little over 300 nodes.

To execute the greedy search type:

```
python pacman.py -l mediumSearch -p SearchAgent –a
fn=greedy,prob=FoodSearchProblem,heuristic=pointsLeftHeuristic
```

Analysis

Usage of BFS vs DFS depends heavily on the structure of the tree, number of solutions etc. If the solution is not far from the root of the tree, then BFS might be better. If the tree is very deep, solutions are rare, then DFS might take long time, and BFS could be faster. If the tree is wide, then BFS might need a lot of memory. And if the solutions are frequent but located deep in the tree, BFS could be impractical. Deeper search its better off using DFS. DFS is space efficient compared to BFS.

A* search algorithm can be tweaked efficiently with other path finding algorithms by analyzing how it evaluates and what type of heuristics it uses. With an admissible and consistent heuristic, A* can find the optimal solutions to complex problems considerably faster than exhaustive BFS. Uniform cost search expands the node with lowest path cost and is optimal for general step costs but whereas BFS expands the shallowest nodes first, it is complete, optimal for unit step costs but has exponential space complexity.
There can be cases, where search space cannot be reduced by using any admissible heuristic. In those cases, greedy search can be useful. Greedy Best First search does not however guarantee optimality, but it can be used to find a feasible solution, when finding an optimal solution is reasonable time is hard.

References

Switzer, T. (n.d.). *2D Convex Hulls: Graham Scan.* Retrieved from http://tomswitzer.net/2010/03/graham-scan/