CSC 665 - Pacman Search

Khanh Nguyen

This project implements different search algorithms for Pacman agent to find its path to the goal.

Overview

- Link to the project's original specs
- By utilizing a general graph search and providing it the correspoding search strategy, different search algorithms will be able to reuse that logic to find the solution. In addition, some search problems require the use of A* search, of which the heuristics functions are implemented to achieve both admissibility and consistency.
- The search strategies can be found in `search/search.py`
- The Node class can be found in `search/node.py`
- The search agents can be found in `search/searchAgents.py`
- The utilities class can be found in `search/util.py`

Problem Statements

Part 1

- In order to help Pacman find path to reach its end goal, the search problem requires implementation of the four search strategies including Depth First Search, Breadth First Search, Uniform Cost Search, and A* Search, each of which will return a list of actions for Pacman to reach its goal state. The search problem defines a basic goal state which only consists of 1 goal position on the grid map.

Part 2

- The search problems no longer consist of just 1 end goal position: finding all corners or finding all dots (food pieces). By using the same A* search strategy, each of these problems must come up with different heuristics or state representation to reflect the nature of the problem as well to find the shortest path for Pacman.

How to run

- Download or git clone this repo if not already
- In terminal, go to directory:
    - `cd path/to/CSC665-pacman-search/search`
- To test **Depth First Search (DFS)**:

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

- To test **Breadth First Search (BFS)**:
  - `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
- To test **Uniform Cost Search (UCS)**:
  - `python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`
- To test **A* search**:
  - `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`
- To test **Finding All the Corners**:
  - `python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`
- To test **Corners Problem: Heuristic**:
  - `python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5`
- To test **Eating All The Dots**:
  - `python pacman.py -l trickySearch -p AStarFoodSearchAgent`

Solution Design

General graph search

```
def graphSearch(problem, fringe):
```

- Given the search problem and fringe as arguments, the general graph search function will search through the successors of a problem to find a goal. The argument fringe should be an empty queue. Also, it will return a sequence of actions (directions) from the start to goal state.
- The function starts with the start state of the problem and pushes it to the fringe (partial plan for the search strategy to explore next).
- By looping through the fringe and check until it's empty, or when there's no more candidate to explore, the function will do either 2 things:
  - Checking if the current state is already the goal state. If so, return the list of actions that gets to this state as a result.
  - Choose the corresponding state and node to expand corresponding to the strategy. In addition, it makes sure to check if the current and the to-be-expanded states are not yet added to the list of visited states in order to avoid checking the same states again (which will result in infinite loop!)

Search strategies

1. DFS

```
def depthFirstSearch(problem):
```

   - Search the deepest nodes in the search tree first, using Stack as the strategy.

2. BFS

```
def breadthFirstSearch(problem):
```

   - Search the shallowest nodes in the search tree first, using Queue as the strategy.

3. UCS

```
def uniformCostSearch(problem):
```

   - Search the node of least total cost first, using PriorityQueueWithFunction as the strategy. The lambda function that helps achieve this strategy takes in consideration of the current node's path cost.

4. A* Search

```
def aStarSearch(problem):
```

   - Search the node that has the lowest combined cost and heuristic first, using PriorityQueueWithFunction as the strategy. The lambda function that helps achieve this strategy takes in consideration of the current node's path cost.

State Representation

1. State = position + corners visited

```
class CornersProblem(search.SearchProblem):
    def getStartState(self):
        # implementation ...

    def isGoalState(self, state):
        # implementation ...

    def getSuccessors(self, state):
        # implementation ...
```

   - Since the new search problem's goal is to visit all corners/eat all dots, the state representation will need to change as well to encapsulate the new state definition.

Heuristics

1. For the corners problem:

```
def cornersHeuristic(state, problem):
```

- Calculate the admissible and consistent heuristics for a state to find the sum of distances between its next closest corner and minimum distance between that corner to the rest of them.

2. For the eat all dots problem:

```
def foodHeuristic(state, problem):
```

- Calculate the admissible and consistent heuristics for a state to find the sum of distances between its next closest dot and minimum distance between all the remaining dots.

Solution Implementation

Part 1

- The relevant files for this part are `search.py`, `node.py`, and `util.py`.
- Essentially, the key to the implementation for all of the 4 search strategies rely on the general logic of graph search and the appropriate data structures for each policy.
- DFS: since this search algorithm would explore the deepest nodes first and consider those as part of the **fringe** (partial plan to be potentially expanded in graph search), using Stack would allow this logic to work because the nodes are inserted in the First-In-Last-Out order, or exploring the deepest nodes first and backtrack to root.
- BFS: since this search algorithm would explore the shallowest nodes first and consider those as part of the **fringe** (partial plan to be potentially expanded in graph search), using Queue would allow this logic to work because the nodes are inserted in the First-In-First-Out order, or exploring the neighboring nodes first and continue on to the next layer of children.
- UCS: since this search algorithm uses priority queue with consideration of least accumulated path cost, the order of visiting and expanding nodes will yield better shorter path as well as less number of expanded nodes.
- A* search: this search algorithm is essentially UCS + heuristics. In other words, besides just ordering the order of visiting and expanding nodes based on the current node's accumlated path cost, it will take into account the "intelligence" look-ahead to best approximate the distance to reach goal state.

Part 2

- The relevant files for this part are `search.py`, `node.py`, and `util.py`, and `searchAgents.py`.
- Find all corners: Given that the search problem has a more complex goal compared to those in Part 1, the implementation of the corners problem will need to reflect this change. In essence, each state now composes of the current position and the number of corners it visited. Therefore, the formulation of start state would simply the starting position of pacman and an empty list of visited corners. The goal state would simply when the list of visited corners is filled (length = 4). In addition, the key to make this problem works is in the `getSuccessors` function. With the new definition for the state,

the successors' state will also need to update their list of visited corners; otherwise, just append the same list from the current state.

- Find all corners + heuristics: This is an extension and optimization on top of the "Find all corners" problem above. The heuristics will take into account the next closest corner from the current state. Then from that corner, finding the minimum distance between that corner to the rest of the other corners. The heuristic function would be the sum of the distances of: current_state -> closest_corner + closest_corner -> other_corners

- Eat all dots + heuristics: This problem is very similar to the find all corners in a sense that the corners will now become dots, with more random positions. Therefore, the heuristics function is trying to optimize the node expansion by minimizing distances among the remaining dots on the board relative to the current state.

Solution Result/Evaluation

- For the problems that require heuristics, apparently the number of expanded nodes play an important of evaluating the efficiency of the algorithm.
    - Example: the "Eat all dots" solution can be measured by 4 thresholds including: [15000, 12000, 9000, 7000]. In my implementation, I was able to expand only 5921 nodes.

- Results from running `python autograder.py`:

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*bfsvs*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

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*bfsvs*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

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*bfsvs*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*2problemE.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: testcases/q3/ucs4testSearch.test
*** pacman layout: testSearch
*** solution length: 7
*** nodes expanded: 14
*** PASS: testcases/q3/ucs5goalAtDequeue.test
*** solution: [‘1:A->B’, ‘0:B->C’, ‘0:C->G’]
*** expandedstates: [‘A’, ‘B’, ‘C’]

Question q3: 3/3

Question q4

*** PASS: testcases/q4/astar0.test
*** solution: [‘Right’, ‘Down’, ‘Down’]
*** expandedstates: [‘A’, ‘B’, ’D', ‘C’, ‘G’]
*** PASS: testcases/q4/astar1graphheuristic.test
*** solution: [‘0’, ‘0’, ‘2’]
*** expandedstates: [’S', ‘A’, ’D', ‘C’]
*** PASS: testcases/q4/astar2manhattan.test
*** pacman layout: mediumMaze
*** solution length: 68
*** nodes expanded: 221
*** PASS: testcases/q4/astar3goalAtDequeue.test
*** solution: [‘1:A->B’, ‘0:B->C’, ‘0:C->G’]
*** expandedstates: [‘A’, ‘B’, ‘C’]
*** PASS: testcases/q4/graphbacktrack.test
*** solution: [‘1:A->C’, ‘0:C->G’]
*** expandedstates: [‘A’, ‘B’, ‘C’, ’D']
*** PASS: testcases/q4/graphmanypaths.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

Question q5

*** PASS: testcases/q5/cornertiny_corner.test
*** pacman layout: tinyCorner
*** solution length: 28

Question q5: 3/3

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', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', '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 901 nodes

Question q6: 3/3

Question q7

*** PASS: test*cases/q7/food*heuristic*1.test*
*** PASS: test*cases/q7/food*heuristic10.test
*** PASS: test*cases/q7/food*heuristic*11.test*
*** PASS: test*cases/q7/food*heuristic12.test
*** PASS: test*cases/q7/food*heuristic*13.test*
*** PASS: test*cases/q7/food*heuristic14.test
*** PASS: test*cases/q7/food*heuristic*16.test*
*** PASS: test*cases/q7/food*heuristic17.test
*** PASS: test*cases/q7/food*heuristic*2.test*
*** PASS: test*cases/q7/food*heuristic3.test
*** PASS: test*cases/q7/food*heuristic*4.test*
*** PASS: test*cases/q7/food*heuristic5.test
*** PASS: test*cases/q7/food*heuristic*6.test*
*** PASS: test*cases/q7/food*heuristic7.test
*** PASS: test*cases/q7/food*heuristic*8.test*
*** PASS: test*cases/q7/food*heuristic9.test
*** PASS: test*cases/q7/food*heuristic*grade*tricky.test
*** expanded nodes: 5921
*** thresholds: [15000, 12000, 9000, 7000]

Question q7: 5/4

Question q8

[SearchAgent] using function depthFirstSearch

[SearchAgent] using problem type PositionSearchProblem
*** Method not implemented: findPathToClosestDot at line 614 of searchAgents.py
*** FAIL: Terminated with a string exception.

Question q8: 0/3

Finished at 9:28:22

Provisional grades

Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 0/3

Total: 23/25

NOTE: Question 8 was not part of the assignment for CSC 665 at SFSU.

Conclusion

- Among the search strategies, A* search will ensure Pacman to find the best path with the condition that its heuristics must be admissible and consistent.
- The essence of AI partially still comes from human's intelligence/intuition. Essentially, these search strategies and heuristics are emulation or reflection of how our brain could be processing such information but with much faster speed.
- In the future, multi-agent games would involve more complicated set up and strategies. However, this game project plays an important role in helping me as well as other people get started in AI and game development.