Report of the 1st Project of Introduction to Artificial Intelligence - Pacman

Linyang He(15307130240)

April 15, 2018

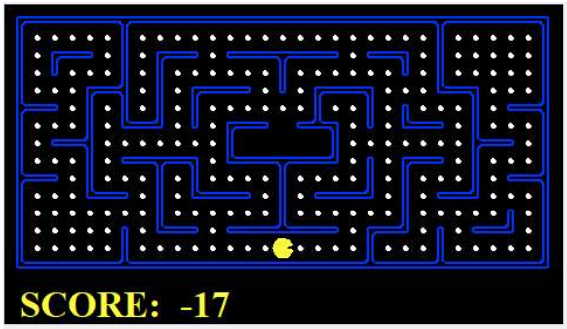

Figure 1: Pacman Game

1 Background

1.1 Pacman

For this project, we will use Python2 to run the Pacman program. For those users who install anaconda but don't have Python2, we can create a virtual environment including Python2 through conda commands. After we build the proper environment, we can run the Pacman as the following code:

```
python pacman.py <options>
```

We should understand some options to help us run the program.

- `-l LAYOUT_FILE`: the `LAYOUT_FILE` from which to load the map layout including testMaze, tinysearch, mediumsearch, etc.
- `-p TYPE`: The agent TYPE in the pacmanAgents module to use including GoWestAgent, SearchAgent, etc..
- `-a AGENTARGS`: Comma separated values sent to agent. In our project, we will use some agent args such as:
  - `fn=bfs`(which means we will select the breadth-first algorithm)
  - `prob=CornersProblem` (which means we are implementing corners problem)
  - `heuristic=cornersHeuristic` (which means the heuristic function for the A* algorithm we will use is defined on cornersHeuristic).

1.2 Search algorithms

1.2.1 Deep First Search (DFS)

The DFS algorithm uses backtracking with the help of stack, the idea is use recursive algorithm and search all the vertices of a graph or tree data with exhaustive searches and explore as far as possible in the same branch before backtracking. The backtracking means that when the current path contains no other nodes unvisited, the search goes backwards on the same path to find nodes to traverse. Next path will be visited only when all the nodes are all visited along the current path. DFS uses stack to complete tasks.
First, put the source vertex on top of a stack, then repeat the following steps until the stack is empty:

1. Take the top of the stack and add it to the visited list.

2. Create a list of that vertex's adjacent nodes, and add the ones which aren't in the visited list to the top of the stack.

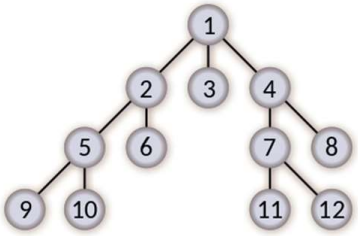

Figure 2: DFS

1.2.2 Breadth First Search (BFS)

Breadth-first search (BFS) is a graph traversing algorithm for searching tree or graph data structures. The principle of BFS is that the root node is expanded first(or some node of a graph with reference of a "search" key), and then the neighbors of the root node are expanded next, then the neighbor's neighbors and so on. BFS algorithm uses queue to complete tasks.
First, put the source vertex in the queue and marked as seen. Then repeat the following steps until the queue is empty:

1. Remove the current node from the queue

2. Search all the unvisited direct vertex of the removed node, and insert the nodes at queue and marked as visited.

BFS visit nodes level by level in graph, and a node is fully explored before any other can begin. The downside of BFS is it usually is slow and require some memory.

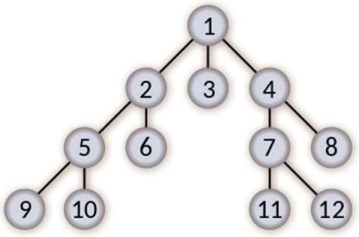

Figure 3: BFS

1.2.3 Uniform Cost Search (UCS)

Uniform cost search is also called Dijkstra's algorithm. It's used to find the shortest path between to vertices of a graph. The difference between Dijkstra's algorithm and DFS or BFS is that the former algorithm might not include all the vertices of the graph. It was initially brought up by Dutch PC researcher Edsger Dijkstra in 1959. Uniform cost searched is based on the fact that subpath B->D of the shortest path A->D is also the shortest between B and D. Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbours to find the shortest subpath to those neighbours. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem

1.2.4 A* Search

A* search approach is widely used in pathfinding and graph traversal with good performance and accuracy, with is an extension of Dijkstra's algorithm. It's an informed search algorithm, the best-first search means that it solves problems by searching among all the possible paths to the goal for the one that incurs the smallest cost, and among these paths it first considers the ones that appear to lead most quickly to the solution. A* search uses an evaluation function f(n) for each node, with the evaluation function providing the estimate for the total cost. A* is a modification of Dijkstra's Algorithm that is optimized for a single destination. Dijkstra's Algorithm can find paths to

all locations; A* finds paths to one location. It prioritizes paths that seem to be leading closer to the goal. Dijkstra's Algorithm works well to find the shortest path, but it wastes time exploring in directions that aren't promising. Greedy Best First Search explores in promising directions but it may not find the shortest path. The A* algorithm uses both the actual distance from the start and the estimated distance to the goal. For the estimate functions (heuristics) in A* search algorithm, they should vary with different problems. We will see more details about the heuristic function in the next section.

1.3 Data Structure

1. Stack. Stack is a kind of linear list where the first input node should output first too. We will use Stack from util.py to implement our deep-first algorithm.

2. Queue. Just like Stack, Queue is also a kind of linear list. The differnt thing for Queue is that the first input node in a Queue should be the last output. It's a wise idea to choose Queue for us to implement the breadth-first search algorithm.

3. PriorityQueue. PriorityQueue is a special Queue, where the output node is the first nor last input node, instead, we will give each node in the PriorityQueue a value representing the priority. Each time we need to pop a node from the PriorityQueue, we should choose the node which has the highest priority. If we take the cost in our pacman search problem as the priority, then we can use the PriorityQueue to implement the Uniform-Cost and A* search algorithm.

2 Project, Results and Analysis

2.1 Question 1

In this question, we need to build a deep-first search agent to find a fixed food dot. Using the Stack data structure, we can easily get a deep first search algorithm. The pseudo-code is as follows:

```
while not Empty(S) do
  u := Pop(S)
    if not visited[u] then
      visted[u] := true
      for all w in Adj[u] do
        if not visited[w] then
          Push(S,w)
        end if
      end for
    end if
end while
```

2.2 Question 2

In this question, we need to build a breath-first search agent to find a fixed food dot. Acutally, just change the Stack in the Question1 to Queue, we can get the answer. And it does work equally well for the eight-puzzle search problem.

2.3 Question 3

As the same to Question2, just change the Stack in the Question1 to PrioriyQueue. We can get the answer. For stayWest, I got cost of 68719479864 and 1 for stay=East.

2.4 Question 4

Just the same as Quetion3, we use PriorityQueue. The difference is the cost. In this question, we denotes our heuristic function as: $f(x) = g(x) + h(x)$ Here the g(x) denotes the cost from the start state to the current state, while the h(x) denotes the estimate distance betweent the current state and the goal state. In this project, we choose manhattanDistance. For the openMaze, path found with total cost of 54 in 0.1 seconds and search nodes expanded is 535.

2.5 Question 5

In this question, we meet a different problem. We should eat all dots in 4 corners. **ATTENTION: Since dict is not hashable, for this particular question, we chang the `visited_postions` in question 5 about BFS from dict to list. That**

means, here you can not use other search algorithm, instead you chang that in corresponding function to.

2.6 Question 6

We will define our own heuristic function to solve the problem. We build 2 methods. Just pay attention to the h(x) part of the A*.

1. We add all the manhattan distance between the pacman and all the unvisited corners together.

2. The second one is choose the minmal manhanttan between the pacman and all the unvisited corners

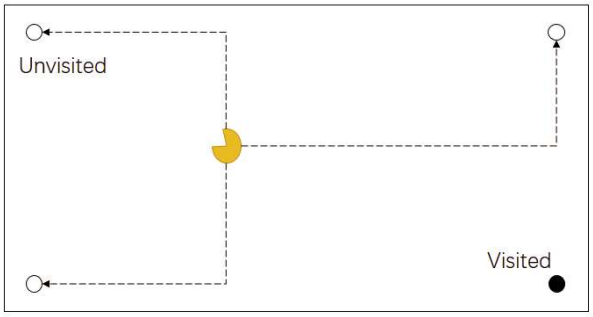

Figure 4: Corner Heuristics

If we choose the first one, the expanded node number is just **502**. And it's 1475 for the second one. So we believe that the first is better, and is default in the code. Moreover, if UCS and A* ever return paths of the same lengths, so the heuristic is indeed consistent

2.7 Question 7

We will build a new heuristic function in this question. Acutally, we don't solve the medium search, but we will describe what we have tried. We need to build. If we want to build a heuristic which expand less nodes possible, we might introduce a "pathDistance". pathDistance is the real distance between the pacman and a node in a maze considering the walls, which is different from both manhattan and eucilean distance. But if we want to build this, we need to use the searchProblem to find the path, which is really time-wasting. Here's a table about this:

Dis	Expanded Nodes	Time Wasting
pathDis	Less	More
manDis	More	Less

Table 1: Food heuristic

So this is quite a balance trick. So if there are different amount of dots unvistied, we might consider use differnt distance function. If there are many unvisited nodes, we don't want to visit too many nodes, so we might consider choose less expanded nodes function, that is pathDistance. While there are relatively litter nodes, we might choose less time-wasting function that is manhattanDistance. As for the trickySearch, the unvisited node is few, so we just choose the manhattanDistance as the heuristic, and we can get the result where it costs just 12.2 senconds and 9551 expanded nodes.

2.8 Question 8

Just use the searchProblem to find the closet dot(with lest path length), we can get the answer. As for the path length, we can use the pathDistance function in the Question7 to find the distance between the pacman and fooddots. The final result:

```
λ python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 323.
Pacman emerges victorious! Score: 2387
Average Score: 2387.0
Scores:        2387.0
Win Rate:      1/1 (1.00)
Record:        Win
```

References

Stuart J. Russell, Peter Norvig (2009) Artificial Intelligence A Modern Approach, 3rd Edition.2009