THU AI Assignment I: Berkeley Pacman Project

Rui Wang*
2015010445

1 General Structure of Pacman Project

The Pacman Project consists of several files. It is difficult for newbies to understand the relationship of these files at first so it is worthwhile to record some of my personal understanding of this python project.

For project 1 "Search", there are several useful files(main branch) and several auxiliary files. Auxiliary files are for displaying and keyboard interaction mainly and could be ignored, while files listed below are critical for the understanding of the program structure:

- *util.py*, the most fundamental file, defining basic data structures such as stack, queue, priority queue and priority queue with special function.
- *search.py*, where general algorithms of searching are implemented and functions are defined. All search functions receive *problem* as a parameter and returns a list of actions(as defined in class *Directions* in *game.py*. Apart from that, an abstract class *SearchProblem* is defined(just the basic structure, no content) at the beginning of the file.
- *searchAgents.py*, where agents and problems are defined. Problems are inherited from *SearchProblem* in *search.py*. *Problem* stores the whole pacman map and the current searching status, i.e., where pacman is. Agents are strategies that use search algorithms implemented in *search.py*. Also, this is where the *problem* is implemented.
- *packman.py*, where an important class *GameState* is defined. *GameState* stores the parameters that will be passed to (and stored in) a *problem* class.
- *game.py*. This file can also be ignored for the time being.

In a word, the main thread of the project follows this: Search agents in *searchAgents.py* calls a specific search function(e.g.depthFirstSearch) in *search.py* and gets the planned path. The graphical display will only show pacman running with the final path while the searching process is carried out too fast to be displayed.

2 Search Functions Performance and Evaluation

2.1 depthFirstSearch

2.1.1 Performance

The depthFirstSearch here uses a stack to realize recursive search. For big maze, DFS expands 390 nodes before finding the answer.

*wangrui15@mails.tsinghua.edu.cn

2.1.2 How to Print a Path while Saving Time & Space

In DFS, we need to print a path. One simple solution is just to use a list to store a whole path for each node. However, this wastes a lot of space. In fact in a searching tree, what we need to do is to memorize each node's parent so that we can trace all the way from goal back to the start point.

Another optimization method(to save both time and space) is to keep a bool list marking whether a node has been visited yet instead of keeping a list of visited nodes for each node.

2.1.3 Analysis and Discussion

Here is a little(or not?) problem regarding the implementation of DFS. On lecture slides, it is stated clearly that DFS only needs to expand nodes that are neither in *frontier* list nor *expanded* list and, stop once one of such paths is found. This is true when our only interest is to find one **POSSIBLE** path from start point to goal. However, this cannot guarantee our passing through the *autograder.py*, which requires the same path and expanding order as the standard answer. In order to pass, we need to make several adaptations.

- A node in *frontier* should be allowed to be visited and added into *frontier* again.
- DFS is terminated only when the next node to be expanded is the goal. DFS will not stop if goal is added into *frontier* and is not the next one to be expanded.

We can generalize from above that the autograder actually prefers a path found through normal recursive process-namely, the process where a function calls itself. Such a process does not have a *frontier*. In a recursive function, *frontier* defined previously is non-existent, thus there is no ban visiting nodes which would otherwise be stored in *frontier*.

2.2 breadthFirstSearch

2.2.1 Performance

BFS runs relatively slower than DFS. This is natural since BFS will expand all nodes before going to the next step. For big maze again, 620 nodes are expanded.

2.3 Analysis and Discussion

For BFS, there is no longer the "little problem" in DFS. Just skipping all nodes visited(whether in *frontier* or not) will do.

2.4 Uniform Cost Search

2.4.1 Performance

UCS uses a priority queue to pick out the shortest path always. It will not stop until both of the two requirements are met:

- The goal has been reached.
- The goal happens to be the next node to be expanded.

For big maze, UCS runs as poor as BFS, expanding 620 nodes before success. Since all roads are equal in cost, UCS virtually does the same thing as BFS, hence the same result.

For medium maze, medium dotted maze and medium scary maze, UCS expands 269, 186 and 108 nodes respectively, which is quite reasonable.

2.4.2 Analysis and Discussion

For StayEastAgent and StayWestAgent, we see a fairly high cost, as a result of punishment on going the other direction. UCS gives an agent the "tendency" to take a certain action. If all actions cost the same, UCS will do the same thing as BFS.

2.5 A* Search

2.5.1 Performance

The heuristic here is the Manhattan Distance between the current state and the goal. A* expands 549 nodes compared to 620 by BFS. It is a bit better but not more, due to the many walls that completely distorts the actual path from a Manhattan path.

2.6 Finding All the Corners

There is not much to say, except that the problem description might well be misunderstood. We are not asked to really define a state in the object *problem* in our search function. Rather, the state passed in is *GameState* and can not be altered. We are only asked to design a state that is maintained during search.

2.7 Corners Heuristic

2.7.1 Heuristic Function

The heuristic used here is manhattan distance. The length of the shortest path from the current state to all of the corners not yet reached. Note that the path does NOT necessarily go to the closest corner first, so we need to list all the possibilities and calculate the shortest of them.

The admissibility and consistency of such a function could be proved. This is not difficult so we will save energy for the next heuristic.

2.7.2 Performance

For CornersProblem map, BFS expands 1966 nodes while A* expands 741. However, the time consumed by A* is 0.1s, exceeding that of BFS. This is due to the time consumed at Heuristic calculation.

2.8 Eating All the Dots

2.8.1 Heuristic

The heuristic function used here is the total length of a minimum spanning tree(MST) plus the distance from current location to the nearest dot. The MST algorithm chosen here is Prim since this should be a dense graph, with every dot linked to each other with a cost of their Manhattan distance.

Proof that this function is admissible and consistent is as follows:

Proof. **Admissibility**: If there is a path shorter than MST + shortest distance, MST should adopt it(if there is a loop, cut the longest edge) instead because a path that links all dots must satisfy the condition of an MST. Thus we prove MST + shortest distance must be smaller than any other path.

Consistency: First we prove that the shortest edge that any dot is linked with other dots must be included in an MST. In other words, in MST, a dot is definitely linked with its closest neighbor

via the shortest path in the graph. This is easy with proof by contradictory.
Now let's consider removing one dot from the MST and see if it will lead to a decrease more than the Manhattan distance. Let Mw be the total weight of edges in the current MST, h_{new} be the new h value, l be the shortest edge of the last eaten dot, d_m be the Manhattan distance from pacman to the last eaten dot.

- If the dot removed is not previously the closest food. Other parts of the MST will stay unchanged. Thus the new MST will weigh $h - l$. Therefore, the new h value will be

  $$h_{new} = (h - l) + l - d_m$$

  namely $h_{new} = h - d_m$. d_m is no greater than the actual path length that pacman took to come here.
- If the dot removed is the closest food. Now pacman should be at the removed dot's position. The h value is decreased by no greater than the actual distance.

□

2.8.2 Performance

This heuristic expands 7137 nodes for trickySearch and uses 4 seconds to calculate the path. With Kruscal Algorithm, it is around 4.7 seconds.

3 Further Comparison and Discussion

Further tests for DFS, BFS, UCS and A*S are also carried out on openMaze. Since DFS cannot guarantee optimality, it is making stupid moves on openMaze.

The general qualities of these search functions are tabulated:

A General Comparison				
Searching Algorithm	Optimality	Nodes Expanded	Calculation Time	Special Use
DFS	No	Relatively fewer	Relatively quick	N/A
BFS	Yes	Many	Slow	N/A
UCS	Yes	Many	Slow	Varying cost, e.g. mediumScaryMap
A*	Yes	Fewer	Quick	Large-scale problem, e.g. trickySearch