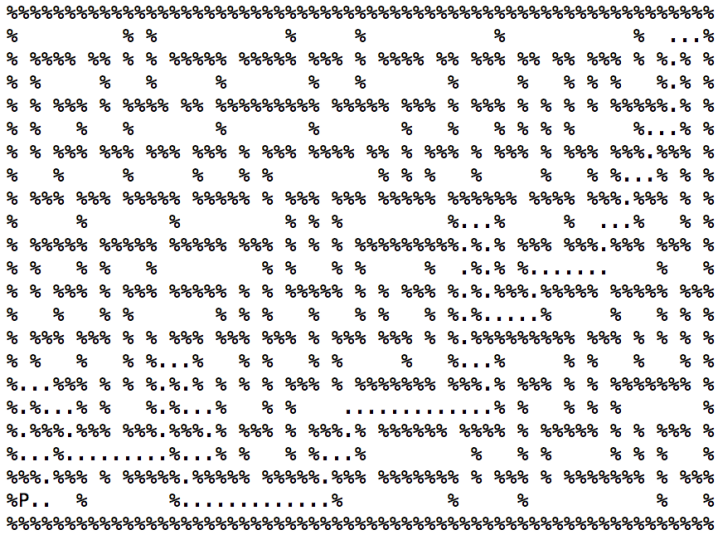
ECE 448 Assignment 1 Report (3 credits)

Renjie Fan, rfan8
Jiangtian Feng, jfeng18
Anchu Zhu, azhu8

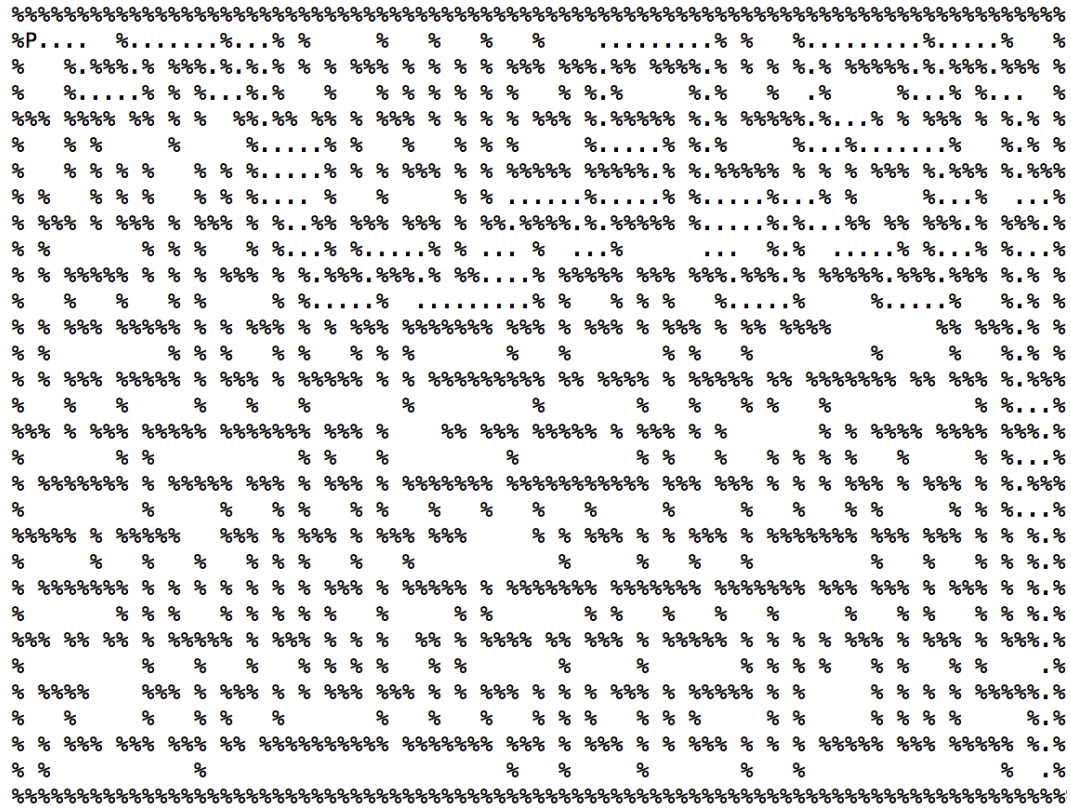
We use python to write the algorithms, all helper functions are in data_structure.py.
To run the codes, type **python part.py filename.txt** in terminal.

Part 1.1

For DFS and BFS, we choose to use the deque data structure rather than traditional queue and stack as the search queue, because if allows popping from both left and right, which provides more straightforward thinking and convenient coding. For Greedy Best and A*, we choose to use heap queue, the python version priority queue, because heappop() pops out the item with smallest key, which greatly reduce lines of code.
For all the four algorithms, we have 2 lists, path[] and visited[]. Visited records nodes expanded and path records the path solution. In the search loop, we pop an element out from the search queue as the current point, append it to visited, exam if it's already visited or matched the goal, append unvisited nodes to path and push new possible nodes in to the search queue.
As we import the maze from txt files and store it in a 2-D list, maze[]. If we print the medium maze line by line, its data structure looks like:
For better search performance, we implement a maze_to_graph() function, to turn the 2-D list to dictionary data structure. Briefly, each location is a key of the dictionary, and every reachable location of the key is appended after the key.
DFS: deque.pop() pops out the right-most element, which is same as the stack.pop().
Medium maze: solution cost: 116 nodes expanded: 261



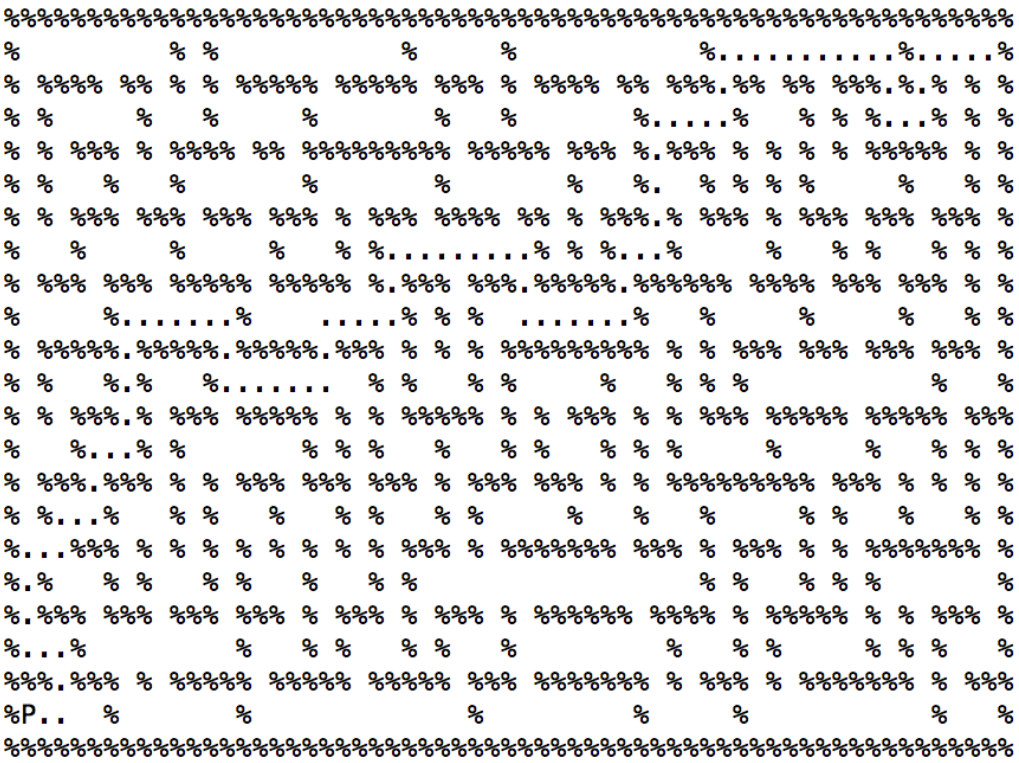
Big maze: solution cost: 240 nodes expanded: 491



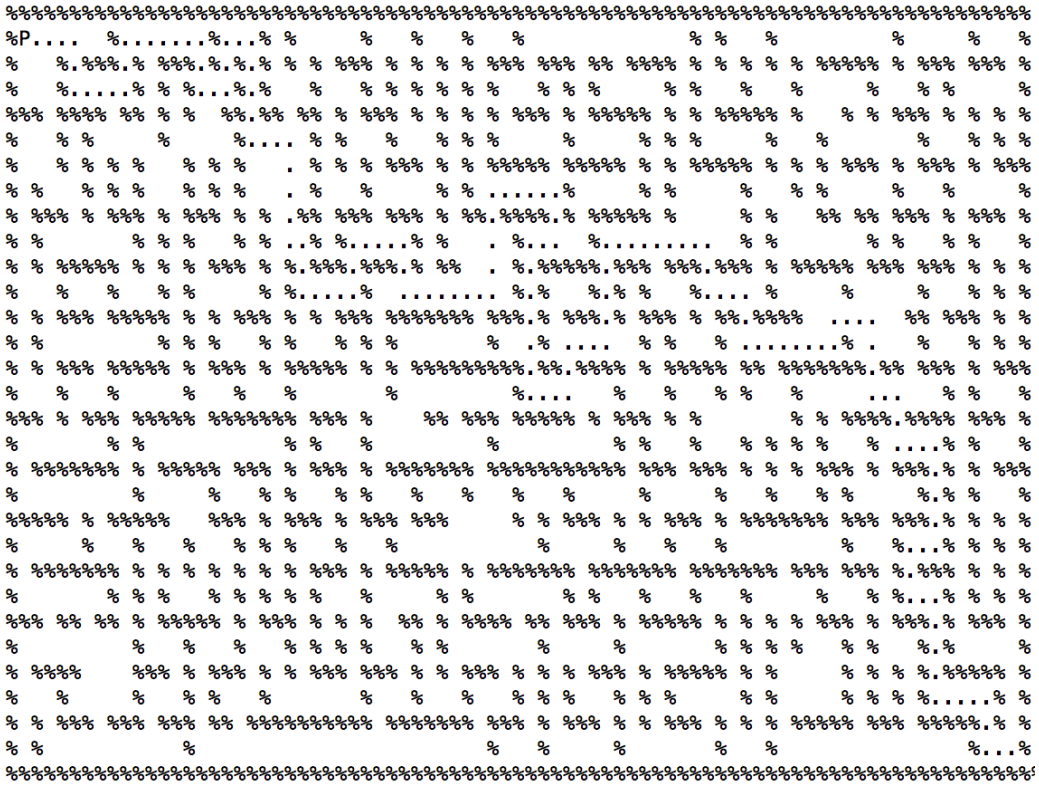
Open maze: solution cost: 217 nodes expanded: 944

My DFS solution result on the open maze seems encounter some problems, but I believe my algorithm is correct. The path_to_solution() is a function that traces back from goal to start point to find the solution path of the maze. It probably has some limits such as the graph below.

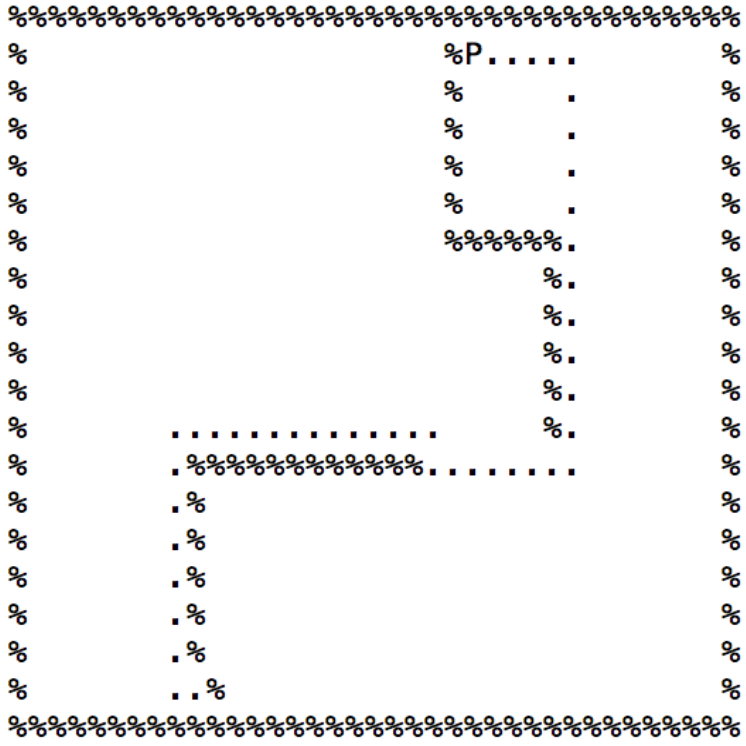
BFS: deque.popleft() pops out the left-most element, which is same as the queue.pop().
Medium maze: solution cost: 94 nodes expanded: 1258



Big maze: solution cost: 148 nodes expanded: 2785

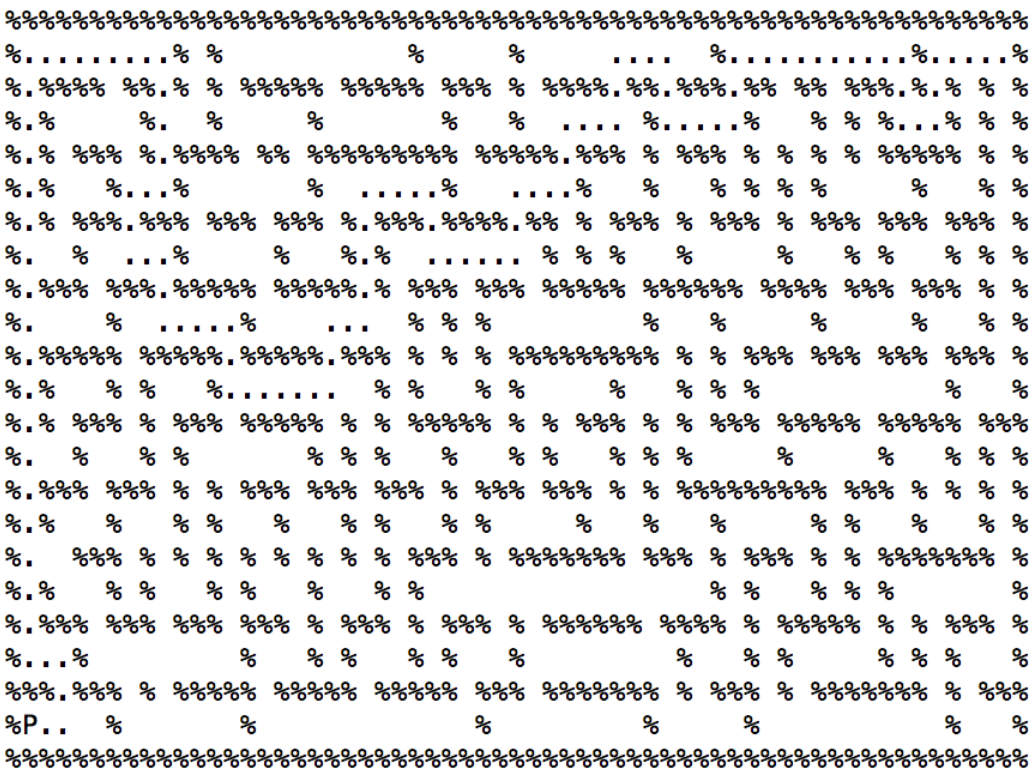


Open maze: solution cost: 45 nodes expanded: 1883

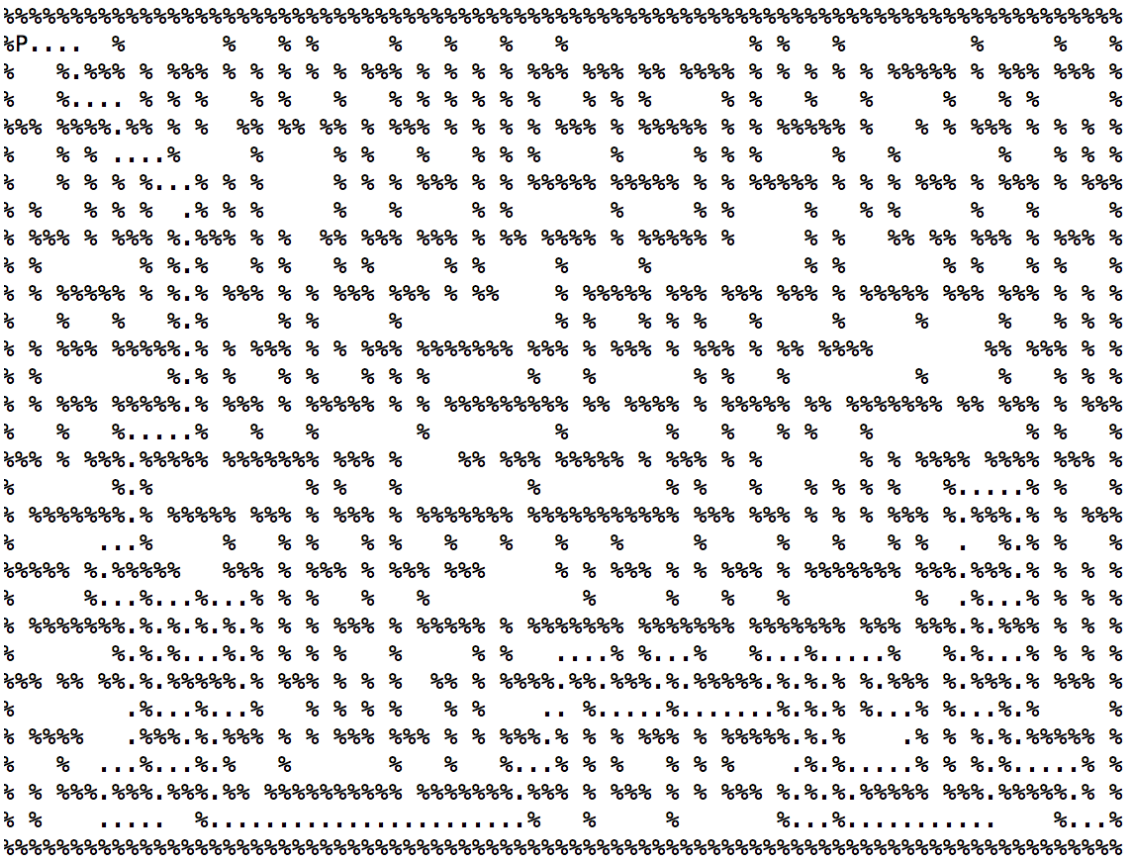


Greedy Best-First Search: data structure in the heap queue is (distance, location), the heappop() pops out the element with smallest distance. Element[1] is the point.

Medium maze: solution cost: 118 nodes expanded: 188



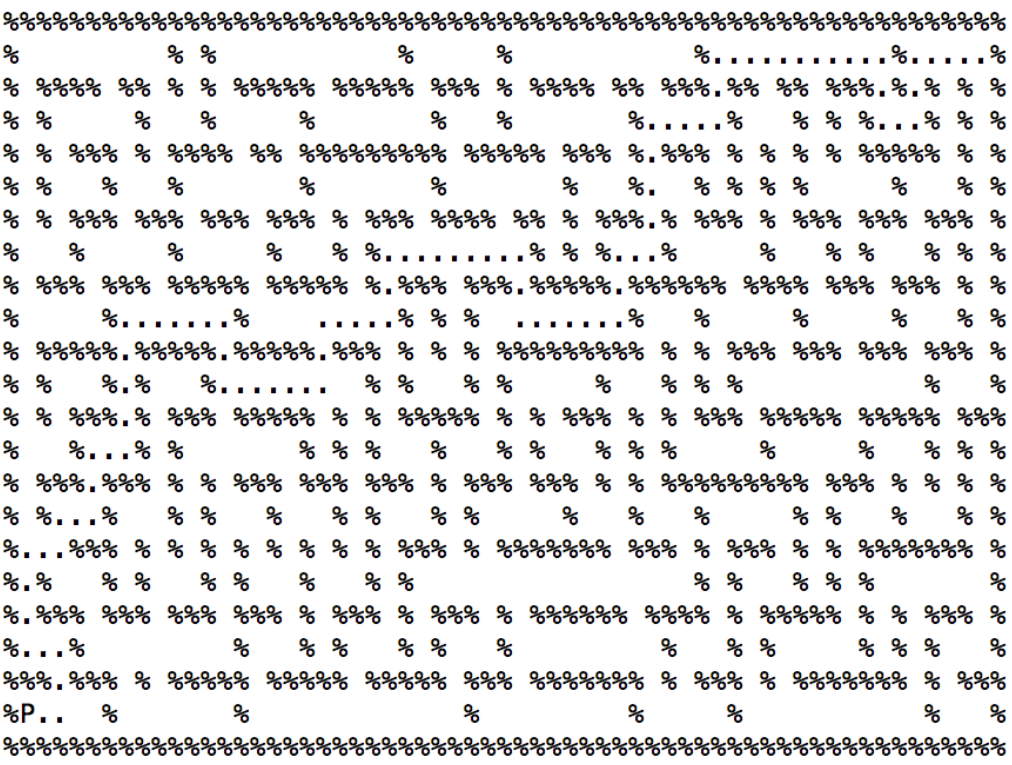
Big maze: solution cost: 222 nodes expanded: 450



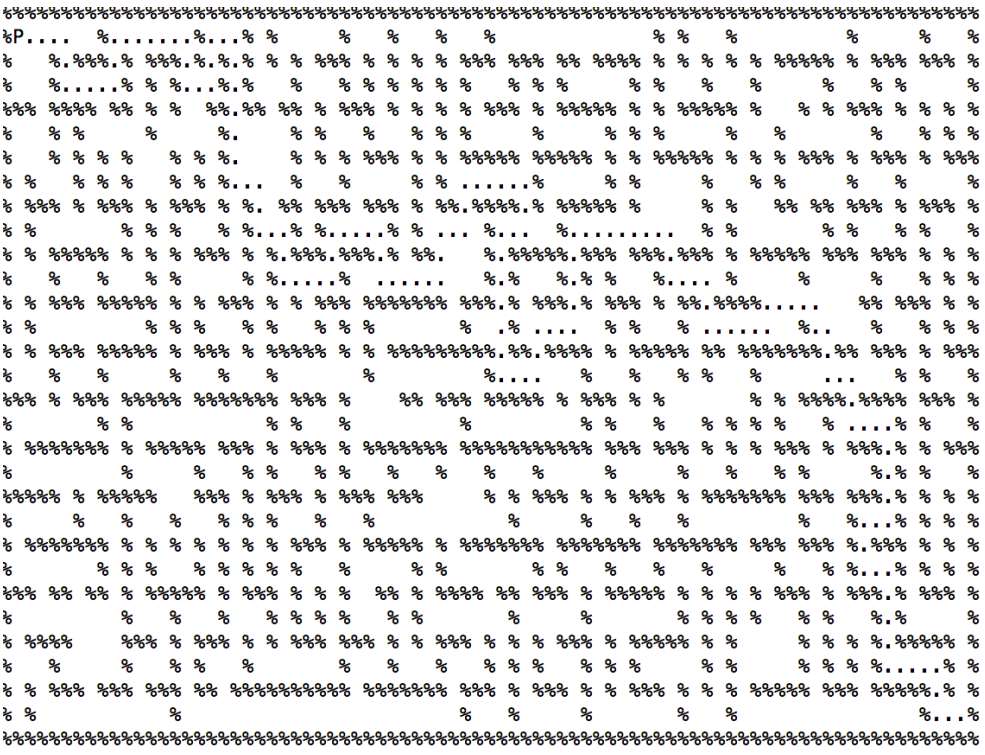
Open maze: solution cost: 45 nodes expanded: 421

A* Search: data structure in the heap queue is (distance+cost, cost, location), the heappop() pops out the element with smallest distance+cost. Element[2] is the point.

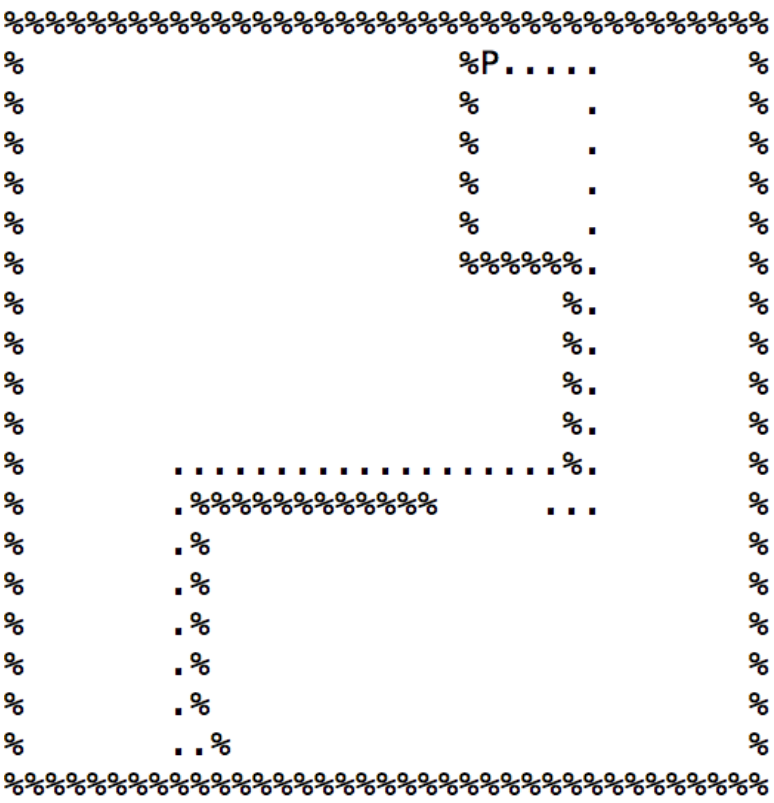
Medium maze: solution cost: 94 nodes expanded: 665



Big maze: solution cost: 148 nodes expanded: 2457



Open maze: solution cost: 45 nodes expanded: 777



-All solution cost is got by counted length of each algorithm's solution.
-All #nodes expanded is got by counted length of each algorithm's visited list.
-For greedy best-first search and A* search, the heuristic function is Manhattan distance.
-To get solution from path, we write a function, path_to_solution(). It reverses the path list, find adjacent node of each current node, delete unwanted nodes and reverse the list back to return the correct solution.

Part 1.2

At first, for multiple targets, the initial state is the starting point marked by P in the input; and the goal state is find a path that starts from P, which passes through all dots. For the optimal solution, it should generate a path with lowest cost. Thus, I choose A* search for part1.2 because it search for the minimum solution cost.

I use two kinds of A* search(different heuristic functions):

AStar1(maze, P, Goal):

maze: the maze input

P: starting point

Goal: the goal state: the dot we need to search

g(n): the cost to reach the next node, for AStar1() the cost to expand every next node is same.

h(n): the cost to get from the next node to the goal; we use Manhattan_distance to judge the cost from node to goal.

AStar2(Dots, P, maze)

Dots: remaining dots that need to search next; if this list is empty, the search achieves to the goal state that finds a path that go over all the dots.

P: starting position

maze: maze input

g(n): the cost to reach the next dot, but g(n) is different for every remaining dots, because their Manhattan_distance to current starting point(P) is different. Thus, for A* search, when expanding next dot, we should make sure it will expand the dot with least cost; I utilize AStar1 function as helper function for g(n): specifically, by calling AStar1 function several times, I save the solution cost for going from current staring point to the next dot in 'Dots' list one by one. Then, I find which dot(s) I should go next that the cost is minimal, which means g(n) is minimal.

h(n): AStar2() that includes g(n) and h(n) is actually just a recursion function:

base case: 'Dots' list is empty

If utilizing g(n) could generate a single dot which has lowest cost, then recursively call AStar2() with updated starting point(one point in the Dots that has lowest cost generated by g(n)) and Dots(remove the one with lowest cost generated by g(n)). However, if g(n) generates several dot in Dots that they have same step-cost. Then, we randomly expand one of them by recursively call test() function(test() is same as AStar2(), except test() doesn't print the solution path) until it reaches the base case;(in the process, if g(n) generates several points again in next stage, just recursively use test() to test one more time to make sure that we get minimum cost indeed) then we acquire the cost for searching from this dot to goal state, store it in a variable named 'leastcost'. Now, we still have to test other points generated by g(n) and acquire their cost to achieve the goal state. Finally, compare them to get the minimum, and call AStar2()with updated starting point and Dots.(in fact, we can just return the minimum cost we acquire now then the search is done; but by doing so, we cannot print the solution path)

In conclusion, my heuristic function should be admissible due to the fact that every step of the search process has lowest cost, which means it never overestimate the cost to reach the goal.

Tiny Search:

```
solution cost: 36
nodes_expanded: 24477
solution path:
%%%%%%%%%%
%7..%.4..%
%.%6%.%%.%
%.%...5%3%
%.8%P%  .%
%9. 0..1.%
%.%%%% %.%
%a.b   %2%
%%%%%%%%%%
```

Small Search:

```
solution cost: 153
nodes_expanded: 16533
solution path:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      P......0%...  .7..8...%
%   %%%%%.%%%%%%.%. %.%.%%. .%
%   %b % .   %  .%. %.%. %..9%
%c.... % ....1...%a...%. %%%%%
%%%%%.%%%% %%%..%%%%%%%.....6%
%d...............2    % %%% .%
%%.%%%%%%%%.%%%%%%%%%%% %....%
% .     %  .........    %.%%%%
%..     %%%%%.    %4.  ...   %
%e% %%%    % .%   %%.%%.%%%%%%
%          % 3%     ........5%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Medium Search:

```
solution cost: 240
nodes_expanded: 58126
solution path:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% e  %   .....j%             %  2%     % %   %8.%
% .  %%%%.%%%%%% %  % % %%      ...%%%%  %  %%%.%
% .  %c% .  % b% %  % %  %   %  .%..........%  .%
%...d..........  %9.... %%%% %  .%%.1%%%.%......%
%.%%%%%.%%%%%%. %%%%.%...........3% .  %4 %%%%%%%
%f..   .%...i%.......a% %%%  %%%%%% .% %. %...% %
%%%.%% .%.%%%%%%% %%%%% %    %  0 %%.% %....%6% %
%  . % .%.     %  %      P% %%  .....%  .%%%%.% %
%g......%.  %  %  %     %........  %    ...5%...%
% % %%%... % %    %% %% %%% %% %  % %%%%%%%%%%%.%
%   %h...%    %              %    %        7....%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Part 1 Extra Credit

In part1.extra.py:

Because of the big maze's high dot density, we implement two different sub-optimal search algorithms based on dot density and wall density. In part1.extra.py, beside basic helper functions you can find in part1.py, we implement next_best(), dot_density() and wall_density(). To test the 2 density evaluation functions, change the last line of next_best() function, and don't forget wall_density has maze as parameter.
Next_best() function is called in main function every time the pacman looks for a reasonably good next choice, it finds out all nearest dots to current position by Manhattan distance heuristic and pass the nearest dots list to dot_density() or wall_density().
Dot_density() function calculates number of "." within 4 units of Manhattan distance. In such a dots densely distributed maze, pacman could eat all dots more quickly if it chose to eat a dot with fewer dots around it.
After choose the next best dot by using dot density algorithm or wall density algorithm, run A* search algorithm to eat the dot. In conclusion, it's a sub-optimal algorithm using A* search and non-admissible heuristic.
Wall_density() function calculates number of "%" within 2 units of Manhattan distance. Pacman could eat all dots more quickly if it chose to eat a dot with fewer walls around it.
Dot density algorithm:
Solution cost: 318 Nodes expanded: 406
Wall density algorithm:
Solution cost: 326 Nodes expanded: 456

In part1_ec.py:

This file contains the third sub-optimal search algorithm, which is the brief of the one we implemented for part1.2, because the complete version of the part1.2 algorithm takes unbelievably long time to run. Basically, the heuristic to choose next dot to eat is replaced as A* search, which means it runs A* search for every left dot on the maze and choose the one with lowest cost as next sub-goal.
Solution cost: 287 Nodes expanded: 850796
When we try to merge either wall density method or dot density method with this algorithm, the solution cost unexpectedly jumps to over 1000. We predict that it caused by the difference between Manhattan distance heuristic and A* search heuristic.

Individual Contribution:

Renjie Fan is responsible for part1.1.py, data_structure.py
Jiangtian Feng is responsible for part1.2.py, part1.ec.py
Anchu Zhu is responsible for writing the report, part1.extra.py