CS440/ECE448 Fall 2017

Artificial Intelligence

Assignment 1:
Search

(4 credits)

Team members:

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Oct 2nd, 2017

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Part1 Basic Pathfinding & Search with multiple dots

1.1 Basic Pathfinding

In this part of assignment, we implemented a maze searching program. Given several different size of mazes each with exactly one start point and one goal point, the problem became finding the minimum distance between two nodes in a graph.

Programming environment: python 2.7, pygame 1.9.2, windows 10(1.2 animation environment)

Data Structure

Before each searching strategy, we transfer the text maze file into a bi-directed graph data structure. Each edge represents the available path.

State Representation

In 1.1, each state is simply represented by the current position. Below are the resulting maze solutions using different search strategies and some screenshots from our animation for extra credit.

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Depth-first search

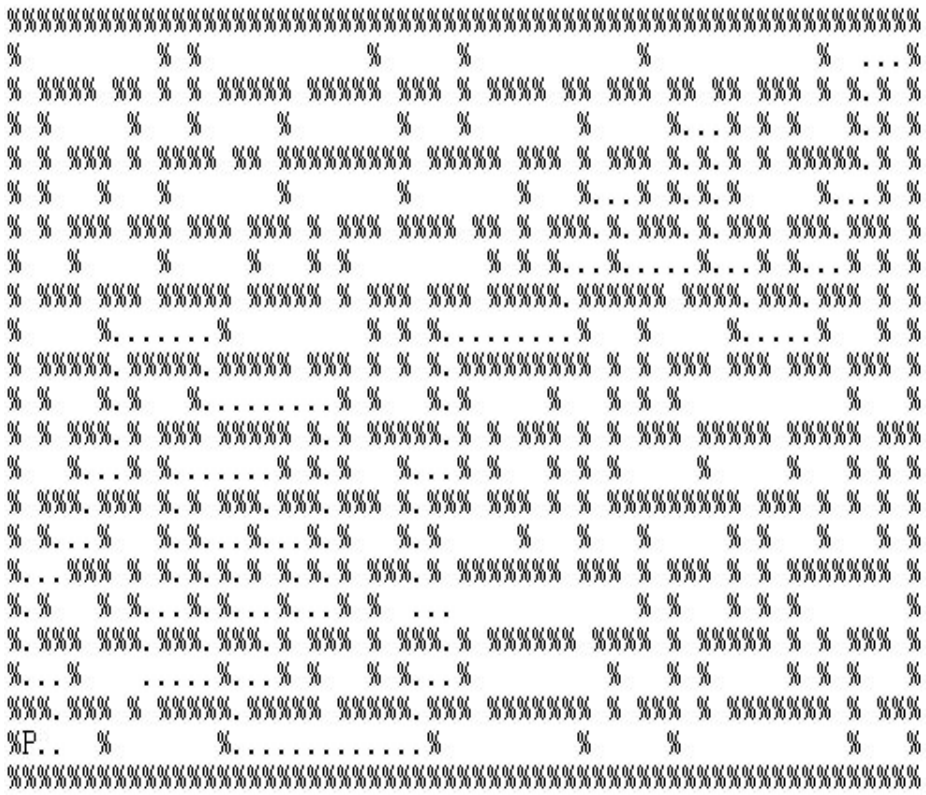
In DFS, we used a visited set for repeated state detection and adopted the stack data structure to store the frontier nodes. The solutions, path costs and numbers of nodes expanded for each maze are as follows:

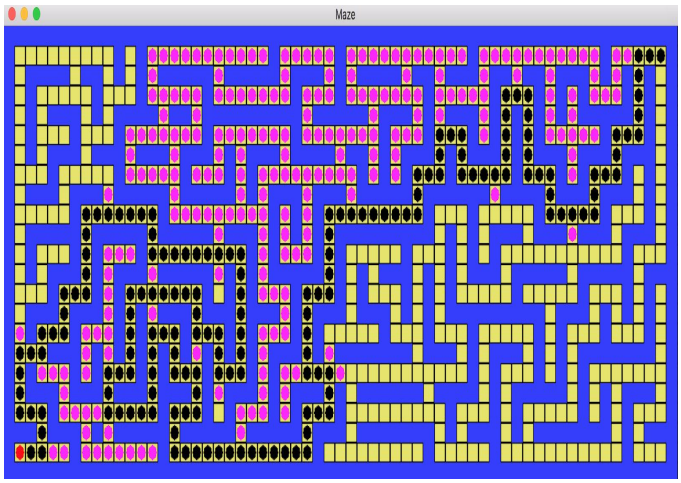
Medium maze

Path cost: 170

Node expanded: 391

Solution:





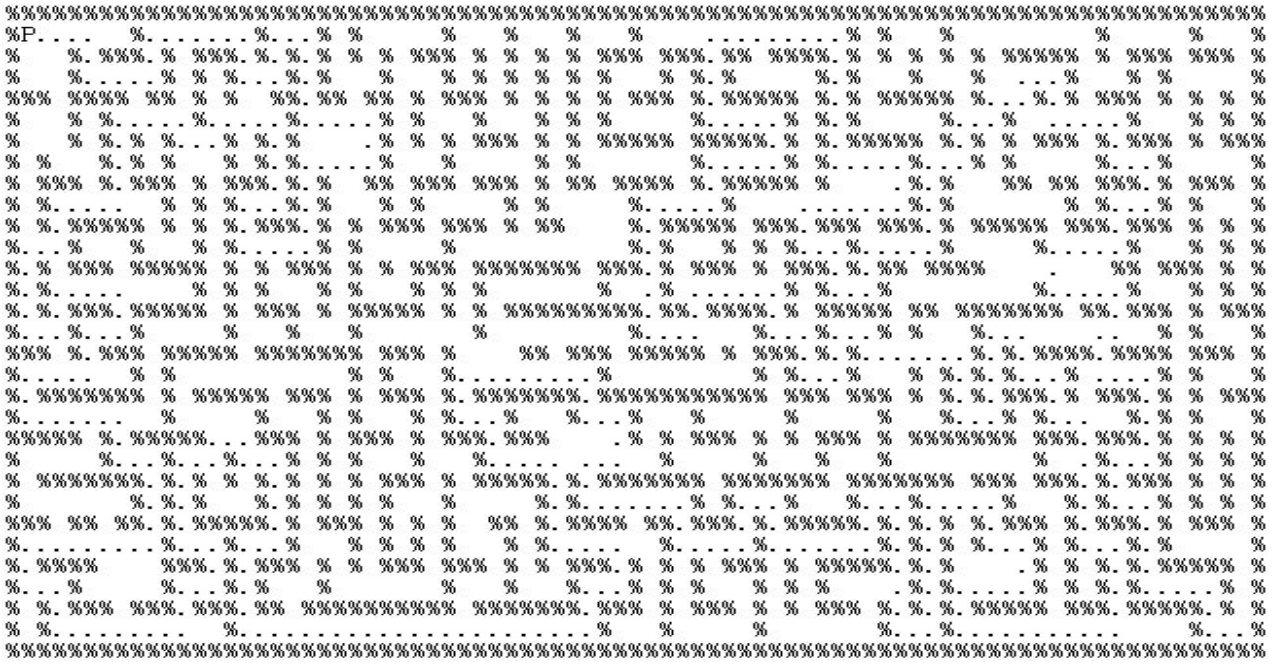
Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Big maze

Path cost: 500

Node expanded: 943

Solution:

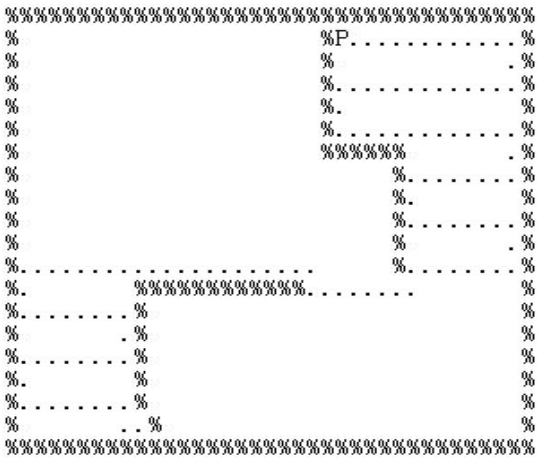


Open maze

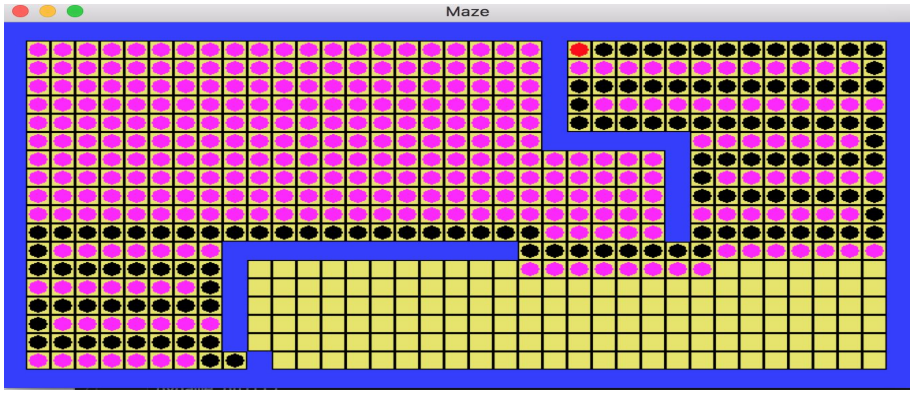
Path cost: 125

Node expanded: 336

Solution:



Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)



Code Snippet:

```
def DFS(self, s):
    '''
    :param s: the node to start from
    :return:
    '''
    # Mark all the vertices as not visited
    visited = {}
    parentMap = {}
    for key in self.vertex:
        visited[str(key)] = False
    stack = []
    visited[s] = True
    stack.append(s)
    while stack:
        s = stack.pop()
        if s == str(self.locationDot):
            self.printPath(s, parentMap)
            break

        print s,
        for i in self.vertex[str(s)]['edge']:
            if visited[i] == False:
                stack.append(i)
                visited[i] = True
                parentMap[str(i)] = str(s)
    print parentMap
```

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Breadth-first search

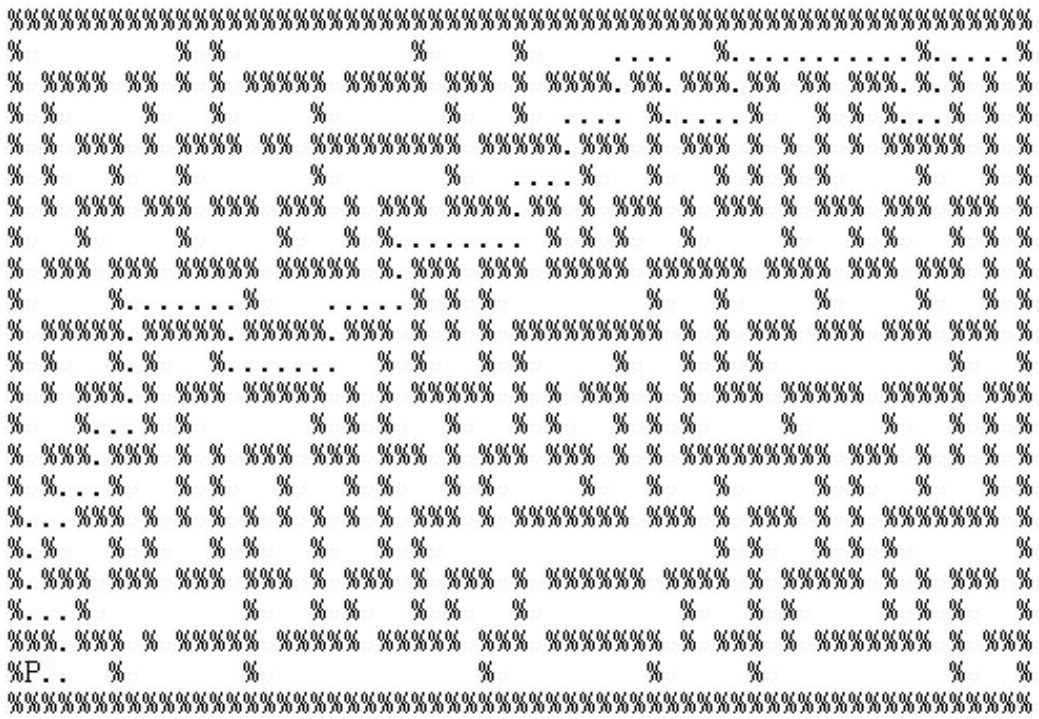
In BFS, we also used a visited set for repeated state detection, the only difference with the DFS is that we use the queue data structure to store the frontier nodes. The solutions, path costs and numbers of nodes expanded for each maze are as follows:

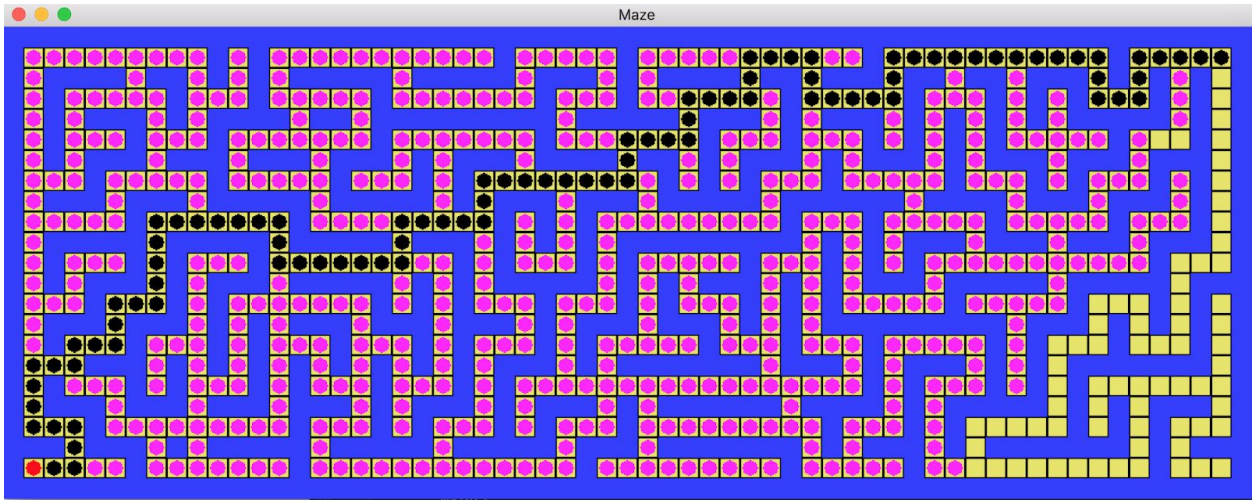
Medium maze

Path cost: 94

Node expanded: 610

Solution:





Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Big maze

Path cost: 148

Node expanded: 1256

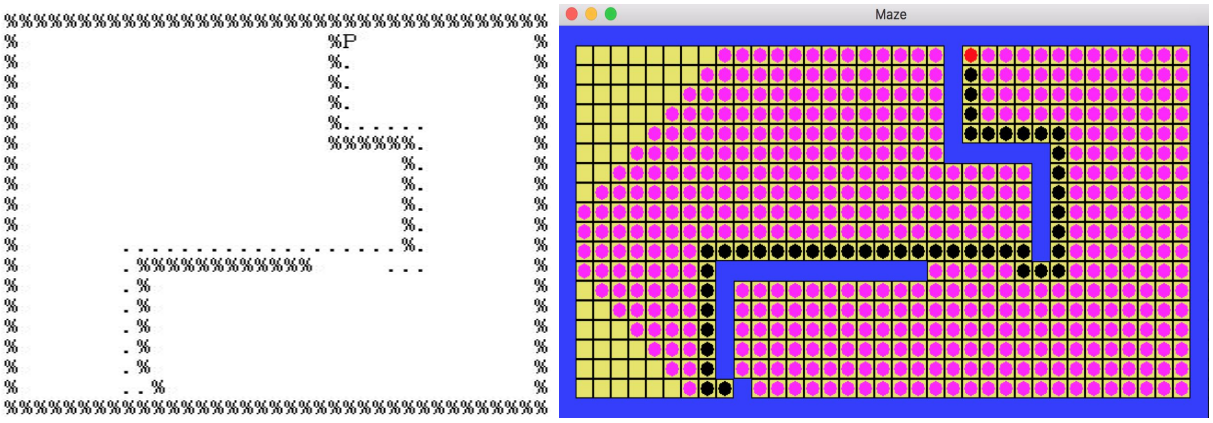
Solution:

Open maze

Path cost: 45

Node expanded: 523

Solution:



Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Greedy best-first search

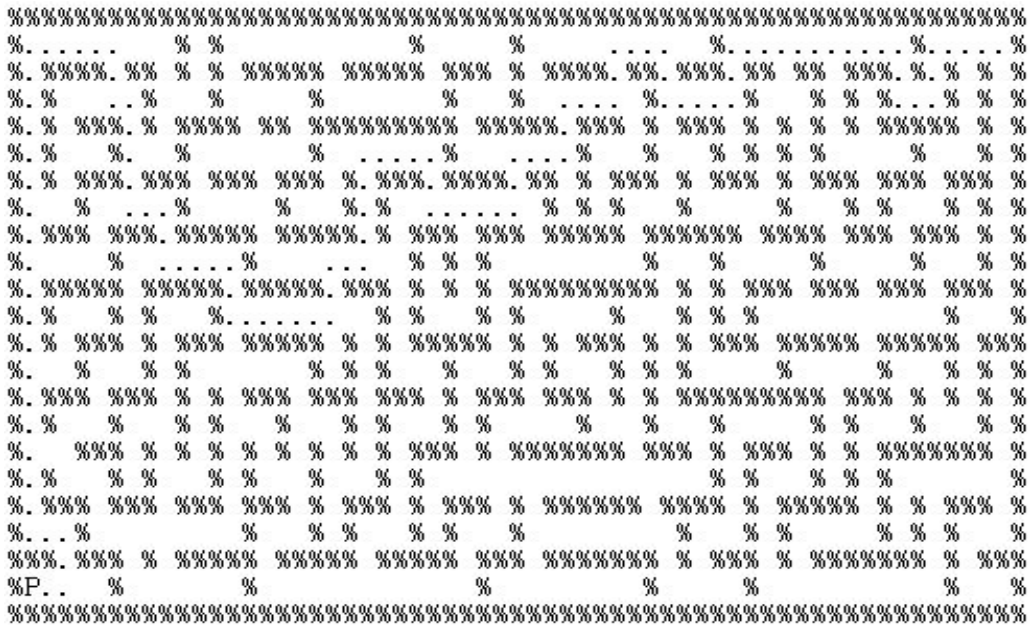
In greedy BFS, we also used a visited set for repeated state detection, the difference between the BFS is that we use the priority queue structure (manhattan distance as the key)to store the frontier nodes. The solutions, path costs and numbers of nodes expanded for each maze are as follows:

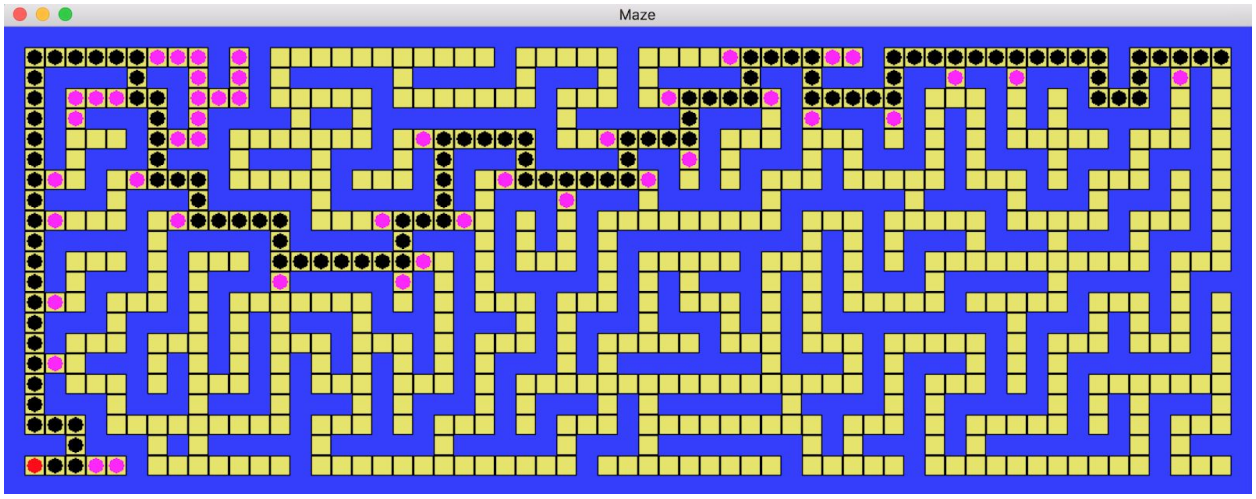
Medium maze

Path cost: 114

Node expanded: 134

Solution:





Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Big maze

Path cost: 222

Node expanded: 278

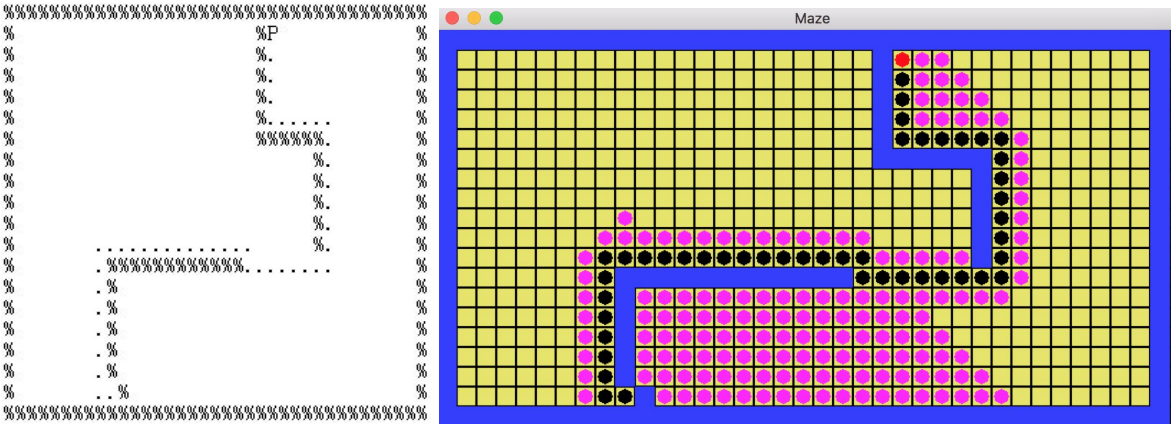
Solution:

Open maze

Path cost: 45

Node expanded: 149

Solution:



Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Code Snippet:

```
def GreedyBFS(self, s):
    visited = {}
    parentMap = {}
    for key in self.vertex:
        visited[str(key)] = False

    frontier = PriorityQueue()
    frontier.put((0, str(s)))
    visited[str(s)] = True

    while frontier:
        s = frontier.get()[1]
        if s == str(self.locationDot):
            self.printPath(s, parentMap)
            break
        print s

        for i in self.vertex[s]['edge']:
            if visited[i] == False:
                priority = self.vertex[i]['manhattanDist']
                frontier.put((priority, i))
                parentMap[str(i)] = s
                visited[i] = True
```

A* search

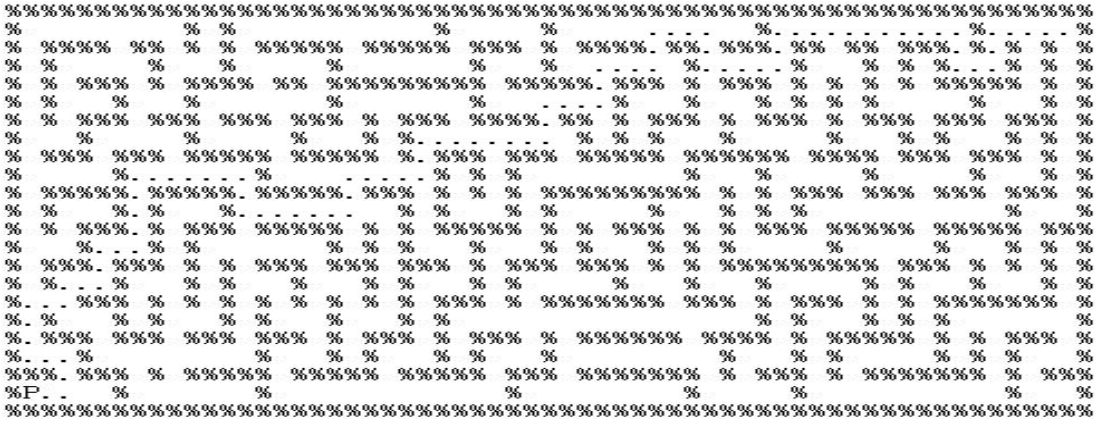
In 1.1, our A* search did not do repeated state detection. The solutions, path costs, and numbers of nodes expanded for each maze are as follows:

Medium maze

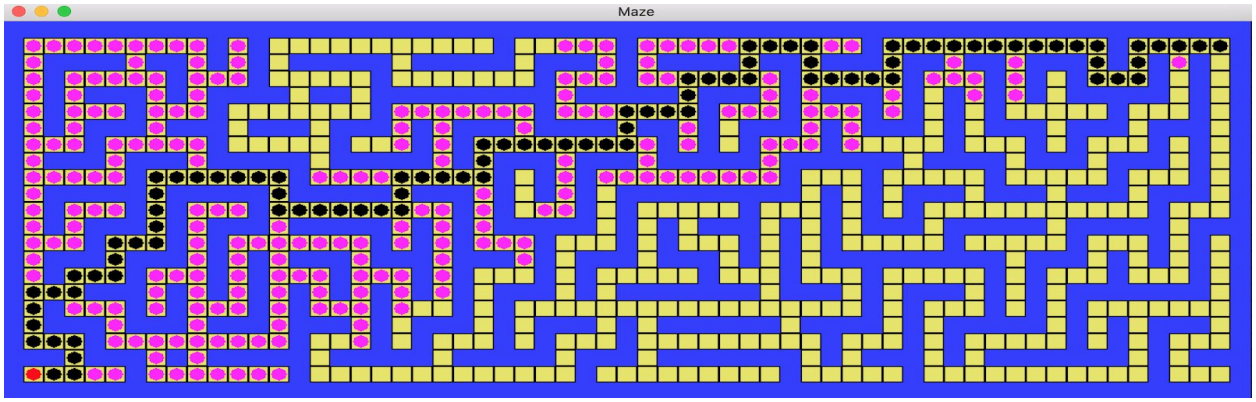
Path cost: 94

Node expanded: 324

Solution:



Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)



Big maze

Path cost: 148

Node expanded: 1118

Solution:

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

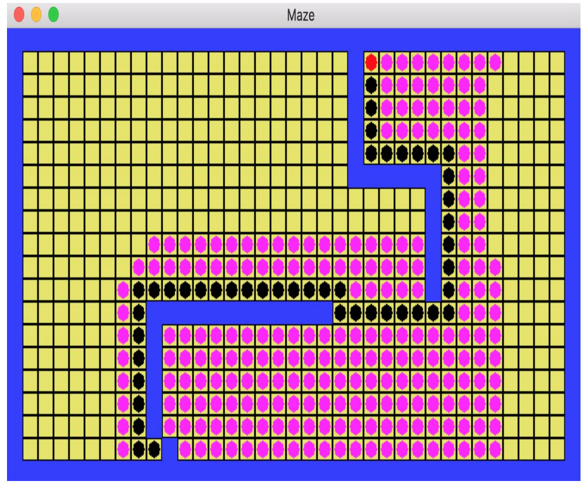
<u>Open maze</u>

Path cost: 45

Node expanded: 230

Solution:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                     %P            %
%                     %.            %
%                     %.            %
%                     %.            %
%                     %......       %
%                     %%%%%%.       %
%                          %.       %
%                          %.       %
%                          %.       %
%                          %.       %
%       ..............     %.       %
%       .%%%%%%%%%%%%........       %
%       .%                          %
%       .%                          %
%       .%                          %
%       .%                          %
%       .%                          %
%       ..%                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```



Code Snippet:

```
def aStar(self, s):
    frontier = PriorityQueue()
    frontier.put((0, str(s)))
    cost_so_far = {}
    cost_so_far[s] = 0
    parentMap = {}

    while not frontier.empty():
        s = frontier.get()[1]

        if s == str(self.locationDot):
            self.printPath(s, parentMap)
            break

        for i in self.vertex[str(s)]['edge']:
            new_cost = cost_so_far[s] + 1
            if i not in cost_so_far or new_cost < cost_so_far[i]:
                cost_so_far[i] = new_cost
                priority = new_cost + self.vertex[i]['manhattanDist']
                frontier.put((priority, i))
                parentMap[str(i)] = str(s)
```

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

1.1 Summary

The path costs and numbers of nodes expanded for each algorithm on each maze is summarized in the table below:

	Path cost			Expanded nodes		
	medium	big	open	medium	big	open
BFS	94	148	45	610	1256	523
DFS	170	500	125	391	943	336
Greedy	114	222	45	134	278	149
A*	94	148	45	323	1118	230

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

1.2 Search with multiple dots

State Representation

In 1.2, each state is the current position and a list of unvisited dots. To avoid inequality of states caused by orders of unvisited dots, we will sort the order of dots when creating a state.

Heuristic

The heuristic we use is the sum of edge lengths in the minimum spanning tree (MST) of the current position and unvisited dots. Each edge length is the distance between two unvisited dots (or an unvisited dot and the current position) calculated by A* search. Actually, A* search only needs to execute at the beginning of the whole algorithm (we will see later). This heuristic is definitely admissible because A* search returns the optimal (shortest) path between two nodes. The minimum distance a state needs to reach the goal state will never be less than the MST, as it needs to visit all unvisited nodes. The actual distance will include some overhead when some positions are visited more than once. Thus, our heuristic leads to an optimal solution.

Algorithm

The algorithm is somewhat tricky. Instead of searching positions in maze, we search states. A state transition is picking a dot from unvisited dot list and directly jump to the dot. Some dots may appear on the A* path between other nodes, thus we should not jump to each unvisited dot. Our method is choosing all unvisited dots “reachable” (without other dots blocking on the A* path). The blocking dots set between any two dots (plus the start position) are calculated in the very beginning. When we expand a state node, we check for each unvisited dot, if all blocking dots are already visited, then it is a reachable dot, we add it to frontier. Meanwhile, because our position can only be the positions of dots, we can compute the path cost using A* search between any two nodes in the very beginning. We did repeated state detection in 1.2 by using an open set and a closed set.

1.2 Solutions

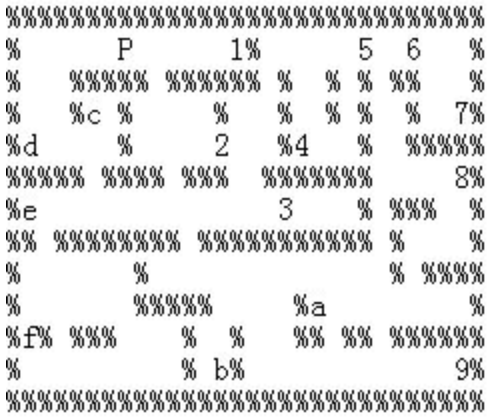
The solutions of three mazed are as below:

tinySearch:

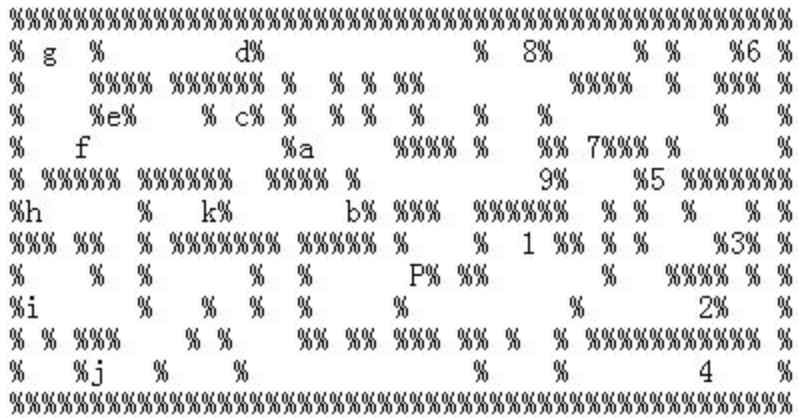
```
%%%%%%%%%%
%8  % 5  %
% %7% %% %
% %   6%4%
% 9%P%   %
%a  1  2 %
% %%%% % %
%b c   %3%
%%%%%%%%%%
```

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Small search:



Medium search:



The path costs, expanded nodes and average running time is listed below:

Maze	Path cost	Expanded nodes	Average Time
tiny	36	49	0.1 s
Small	143	408	3 s
medium	207	2712	3 min

The implementation for part 1.2 can be found in **pacman.py**.

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Extra credit: suboptimal search

For the big maze, the only change we make to our heuristic is multiply the original heuristic with a constant, which will make the heuristic no longer admissible. The larger the constant is, the less nodes will be expanded, thus the less time it takes, but results in longer path.

We decrease the constant from 2 to 1.29, and find that it takes unreasonable time starting from 1.28. The path costs, expanded nodes and average running time with regard to the constant is listed below:

Constant	Path cost	Expanded nodes	Average Time
2	490	1047	30 s
1.5	273	757	1 min
1.4	265	1762	1.5 min
1.3	262	3092	2.5 min
1.29	262	3610	3 min

Extra credit: maze searching animation

Using pygame 1.9.2 under python 2.7 environment, we have implemented an animation to animate part 1.2's searching process.

The blue and yellow squares refer to the "%" (wall) and " " (available paths). As shown in the basic path finding process, we marked the expanded nodes in purple circles and at last marked the optimal path in black circles.

We have screen-recorded our animations and they can be found in the code package (smallSearch.mp4 and mediumSearch.mp4).

In smallSearch.mp4, we show the complete algorithm process. At first, we show the optimal path between each two nodes (marked with black circle). After the whole dots searching sequence is confirmed, we print the best solution in that order marked with the former single goal A* path between each pair.

In mediumSearch.mp4, we only show the last whole final optimal dots searching sequences.

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Part 2: Sokoban

Programming environment: python 2.7
The implementation for part 2 can be found in the files sokoban_helper.py and sokoban.py.

First, we read the text file of a given Sokoban. Then we implemented a function to find all the deadlock positions on a Sokoban board and mark them with an 'x'. A deadlock position is defined as a spot where no boxes are allowed to go, otherwise it will not be possible for the box to reach any goal (dot). This includes locations that are in the corners without goals and the path connecting the corners given that there is no goal on any of the spaces.

We have implemented a Sokoban class that generates all possible configurations of a Sokoban board from a given configuration (i.e. available moves from the current location). To find the list of possible configurations we check for the following in the function "isAvailable" that takes the new player location as a parameter:

- If the new move ends up off the board or on a wall, it is invalid
- If the new move ends up pushing a box, make sure that the new location of the box is not another box or a deadlock location
- Make sure that the move made does not result in two adjacent boxes both next to a wall, three boxes circled around one wall, or four boxes grouped together (as a square). (Unless they are all on a goal state.)
  → These configurations will create a deadlock.

The result of the available moves will be printed as a list of possible game configurations of type Sokoban in the "getMovesList" function, and will then be used in our A* search. For this assignment we decided to implement A* search with two different heuristics. Details of the two heuristics are explained below.

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

A* with heuristics 1 (h1):

The minimum sum of all the manhattan distances from a box to a corresponding dot. The requirement is that no box can be paired with the same dot as another box. This is admissible, because in reality each box must correspond to one and only one dot, and this distance cannot be less than our heuristics, as our heuristics is already the minimum value it can be.

	Path Cost	**Expanded Nodes**	**Running Time (s)**
Input 1	8	22	0.00804
Input 2	144	11261	364.056
Input 3	34	6307	167.165
Input 4	72	2814	25.289

A* with heuristics 2 (h2):

The sum of the manhattan distances of each box to its closest dot. This is admissible, because it is the minimum distance each box must go in order to reach its goal. In most cases it is even less than the minimum distance each box must go, as the manhattan distance does not take into account any walls in between.

	Path Cost	**Expanded Nodes**	**Running Time (s)**
Input 1	8	22	0.00717
Input 2	144	11276	392.208
Input 3	34	9658	337.314
Input 4	72	2858	33.531

Extra Credit:

For the input “Extra 2”, we multiplied our heuristics from h1 by 4 and got the following results:

Path cost: 39

Expanded nodes: 632

Running time: 40.667s

Path taken (x,y coordinates of the board):

[(3, 4), (4, 4), (5, 4), (5, 3), (4, 3), (4, 4), (3, 4), (3, 5), (3, 6), (2, 6), (2, 5), (1, 5), (1, 4), (2, 4), (1, 4), (1, 3), (1, 2), (2, 2), (2, 1), (3, 1), (4, 1), (4, 2), (5, 2), (5, 1), (6, 1), (7, 1), (7, 2), (8, 2), (8, 3), (7, 3), (8, 3), (8, 4), (8, 5), (7, 5), (7, 6), (6, 6), (5, 6), (5, 5), (4, 5)]

Menglin Tian (mtian6)
Renxuan Wang (renxuan2)
Yisi Liu (yisil2)

Individual Contributions:

Menglin Tian - Implement of part 1.1's DFS, BFS searching strategies, part 2 & part 2's extra credit

Renxuan Wang - Implement of part 1.1's A* searching strategy, part 1.2 & part 1's extra credit

Yisi Liu - Implement of part 1.1's Greedy BFS searching strategy & part 1's searching animation