CS440: Intro to Artificial Intelligence, Fall 2017, Homework 1

Name: Nestor Alejandro Bermudez Sarmiento (nab6)
Credit hours: 4

Collaborated with Edward McEnrue (mcenrue2)

In this assignment I have implemented general-purpose search algorithms to solve well know puzzles. In the first section I'll use such algorithms to 'guide' a 'Pacman' through the maze, finding all the food pellets. In the second section I'll solve the Sokoban puzzle and describe the chosen heuristic and why it is valid.

I've decided to use Python for this assignment. The source code is provided with this report and will also be available under my GitHub account right after the submission deadline had passed. See my GitHub CS440 repository.

To run the programs you should go into the appropriate directory (either part1 or part2) and run **python main.py**. The programs require Python 3.6 to work. The main reason for using Python 3.x was because of the introduction of *lru_cache* which allowed me to cache some of the results of some functions. Note that for it to work the arguments must be hashable, which wasn't the case for all of my functions, so for those I implemented a very dummy caching mechanism using a dictionary and a derived key from the arguments.

Part 1

The following description is valid for both Part 1.1 and Part 1.2.

Located under the **part1** subfolder you will find a main.py file that performs the actual problem solving.
To do so, you will need to set the following variables:

1. *strategy*: defines whether to use BFS, DFS, Greedy BFS or A*.
2. *multipellet*: flag to force the use of Manhattan distance as a heuristic.
3. *maze*: name of the file that needs to be solved.
4. *h*: heuristic to be used. It only takes effect when multipellet is set to **multipellet**

You will also find the maze_problem.py and pacman_problem.py files. The first one corresponds to Part 1.1 of the problem and the second to Part 1.2.

The search_enums.py file defines some handy enumerations for the different search strategies, possible actions and result formatting colors.

The files maze_problem.py and pacman_problem.py follow the guidelines presented in the textbook and implement these functions:

1. *actions*: given the current state. It returns the list of all legal actions the agent can perform from the given state.
2. *goal_test*: determines when the goal has been reached. For both Part 1.1 and Part 1.2 this function checks whether there is any remaining food pellet in the maze.
3. *step_cost*: cost to move from state A to state B. In our case, it is always 1 regardless of the states.
4. *estimated_cost*: returns an approximation of the cost to reach the goal from the given state. This is where the chosen heuristic is used.
5. *result*: it receives the current state and the action to perform, which can be one of four possible values (GO_LEFT, GO_RIGHT, GO_UP, GO_DOWN). Depending on the given action it will update the x or y coordinate of the current node position and generate a new state.

The files also define some auxiliary functions for loading the corresponding file and printing results.

Now, lets talk about design decisions...
First, I decided to use the standard Queue library for Python as my implementation for the *frontier*. The library provides the three types of queues necessary for the exercise:

1. LIFO: for DFS
2. FIFO: for BFS
3. Priority Queue: for Greedy best-first search and A*

I created a separated class called **SearchProblemSolver** that abstracts the details of the graph search and receives a *problem* as argument. As long as the *problem* class defines the functions and methods mentioned above, it will work.

Second, note that I'm storing the path_cost on each node regardless of the search strategy but it is only used for the A* search strategy. Reason for this is to reduce the branching in my code by checking which search strategy is being used.

Third, we know that A* is optimal if the heuristic is admissible AND we don't do repeated state detection. So my first implementation didn't have such detection. Later I realized that Manhattan distance is actually a *consistent heuristic*[1] (see explanation below) so I took advantage of that and added repeated state detection. I did it by just keeping a hashmap where the key is the position in the maze and the value is a boolean.

[1] https://en.wikipedia.org/wiki/Consistent_heuristic

Fourth, I knew since the beginning I'd need to have a representation of the eaten (or remaining) food pellets. I first tried to use a list but the comparison between then summed up fast. Then I decided to use a set since the comparison is more efficient. After additional profiling I realized a big percentage of the time was spent there. After talking with the TAs we thought a bitmap could help the performance. That implied a bunch of other changes and some extra data structures but it definitely paid off. For the bitmap representation I use the *bitarray* package.

Fifth, I added some memoization to some of the most expensive functions. In some cases using the *functools* package from Python's standard library and in some other cases by explicitly defining a hash table. The reason for this was that some of the function arguments were not hashable so *functools* wouldn't work.

Sixth, have a separate data structure called *frontier_set*. It is implemented as a **set** and it holds the same elements that are in the *frontier*. I tried a different frontier implementation that would allow me to check if a value was in it without having to dequeue and requeue the entire thing (unlike **PriorityQueue**) but having a separate structure was better for performance.
Seventh, for tie breaking I implemented __lt__ (less than) for both my state representation and the nodes. That is **PacmanProblemState**, **MazeProblemState** and **Node**. I also had to implement __ne__ (not equal) and __eq__ (equal) because it is necessary if I want to store such objects in a set.

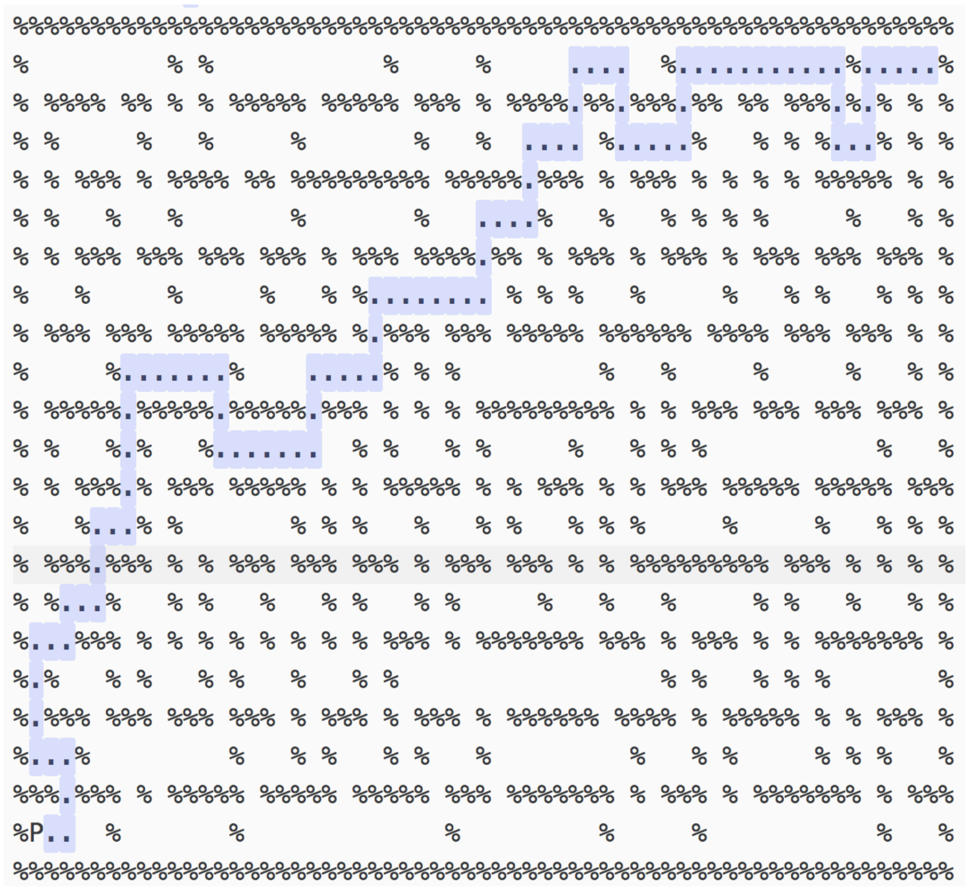
PART 1.1: Basic pathfinding

Under the maze_problem.py file you will find a class called **Heuristic** which defines a single heuristic: Manhattan distance.

PART 1.1: Results

Medium maze

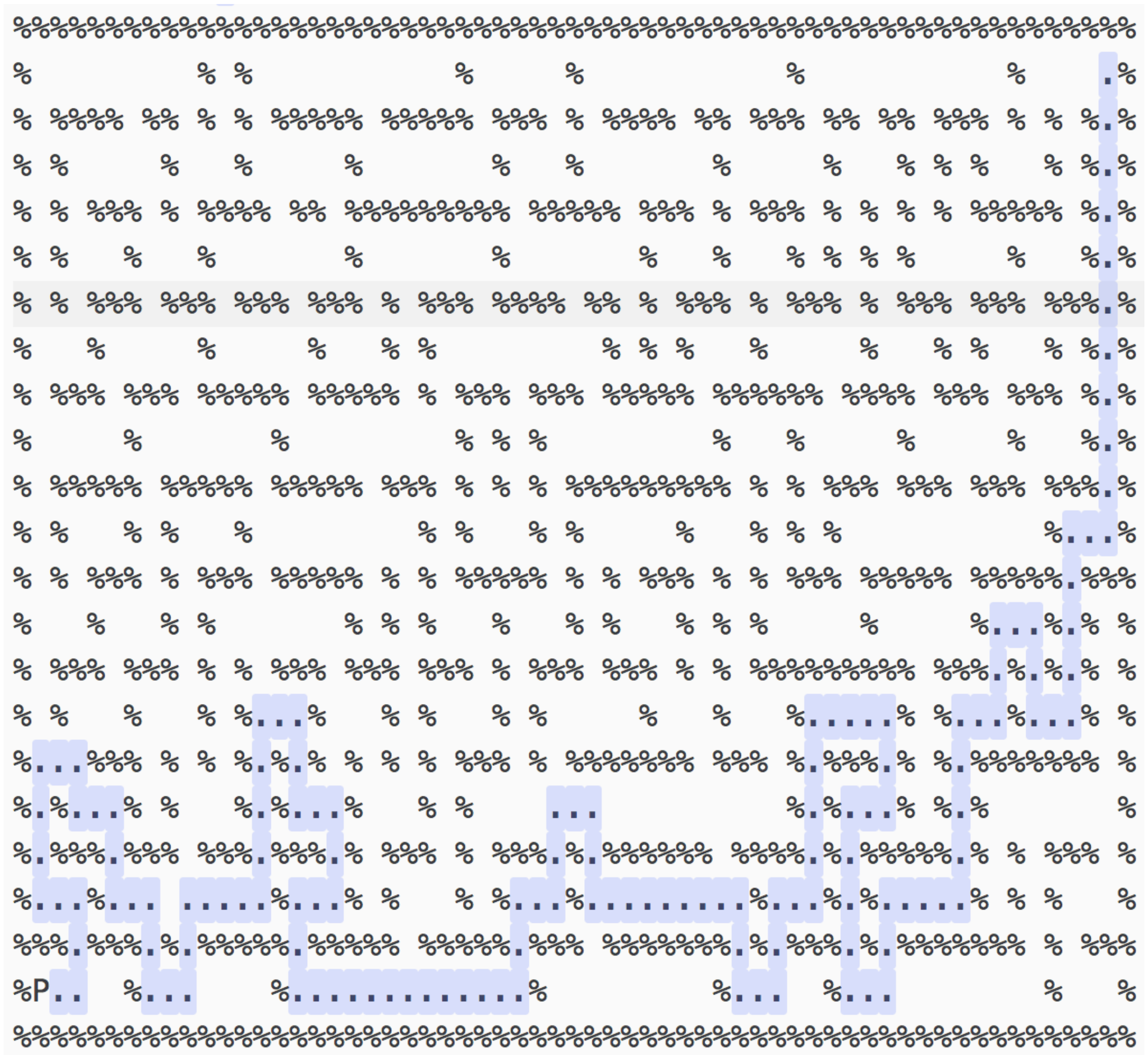
Breadth-first Search:
Path cost: **94**
Expanded nodes: **607**



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

Figure 1: Solution for the medium size maze using BFS

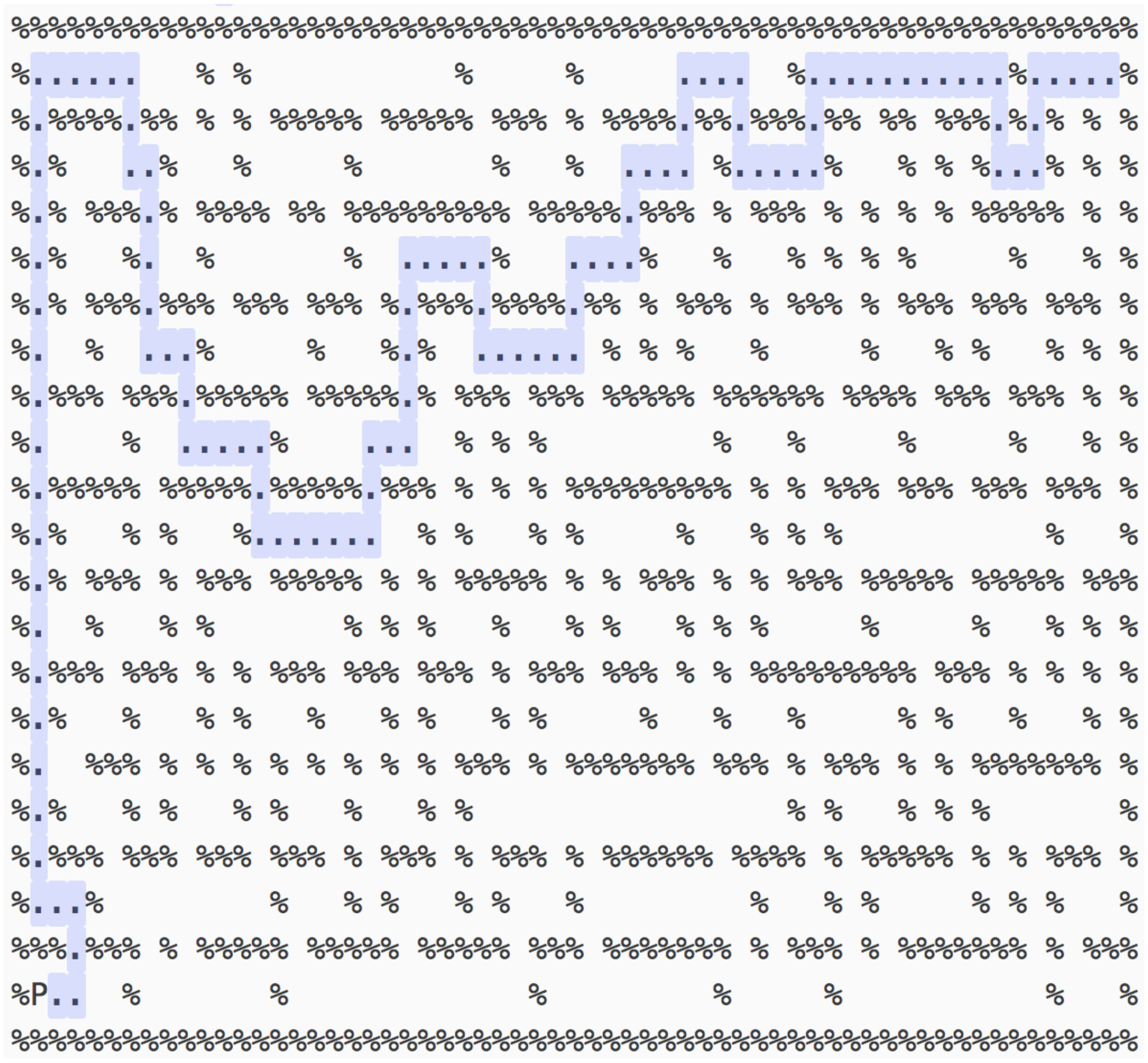
Depth-first Search:
Path cost: **136**
Expanded nodes: **186**



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

Figure 2: Solution for the medium size maze using DFS

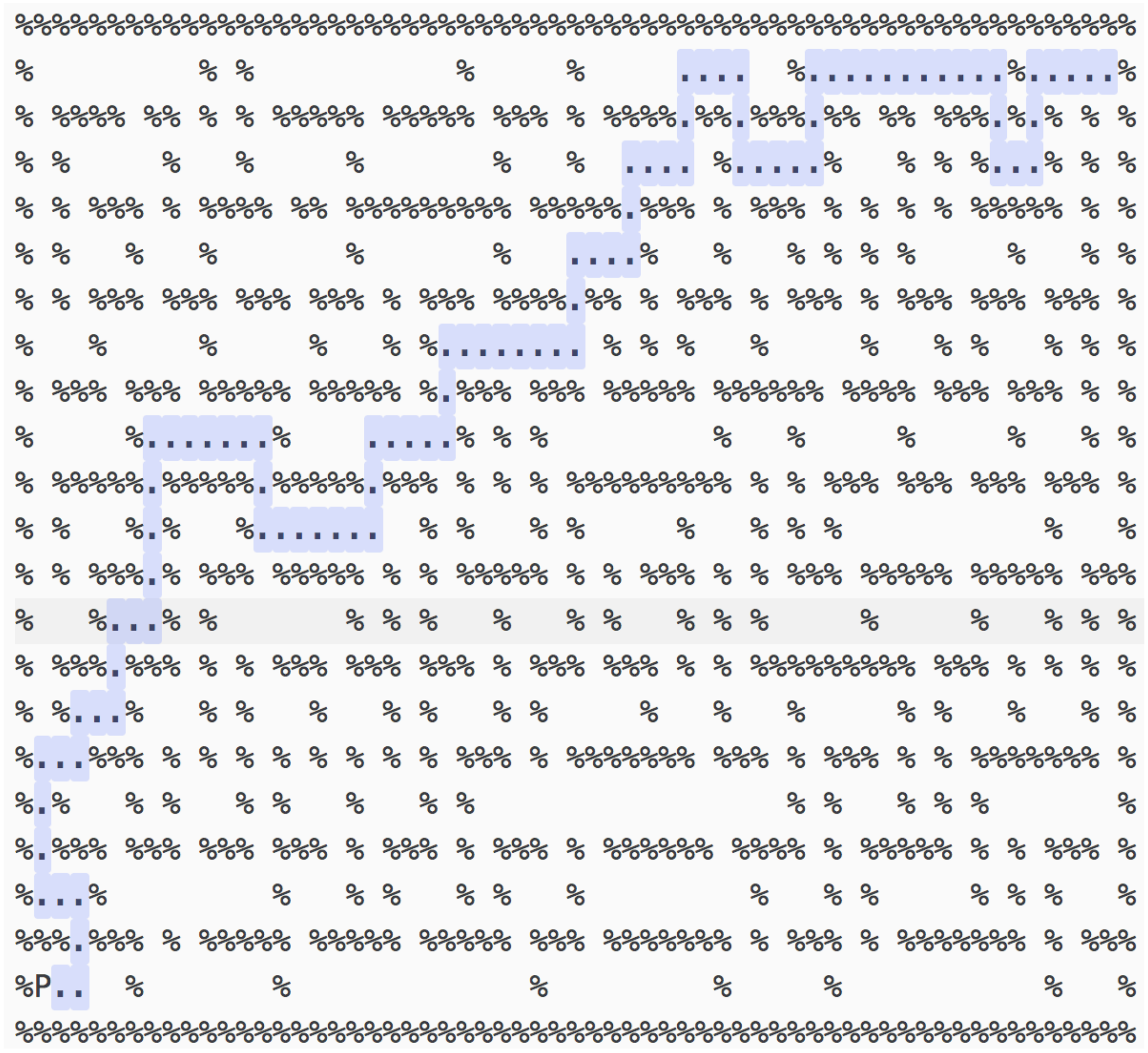
Greedy Best-first Search:
Path cost: **114**
Expanded nodes: **133**



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

Figure 3: Solution for the medium size maze using Greedy BFS

A*:
Path cost: **94**
Expanded nodes: **304**



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

Figure 4: Solution for the medium size maze using A* algorithm

Big maze

Breadth-first Search:
Path cost: **148**
Expanded nodes: **1,258**

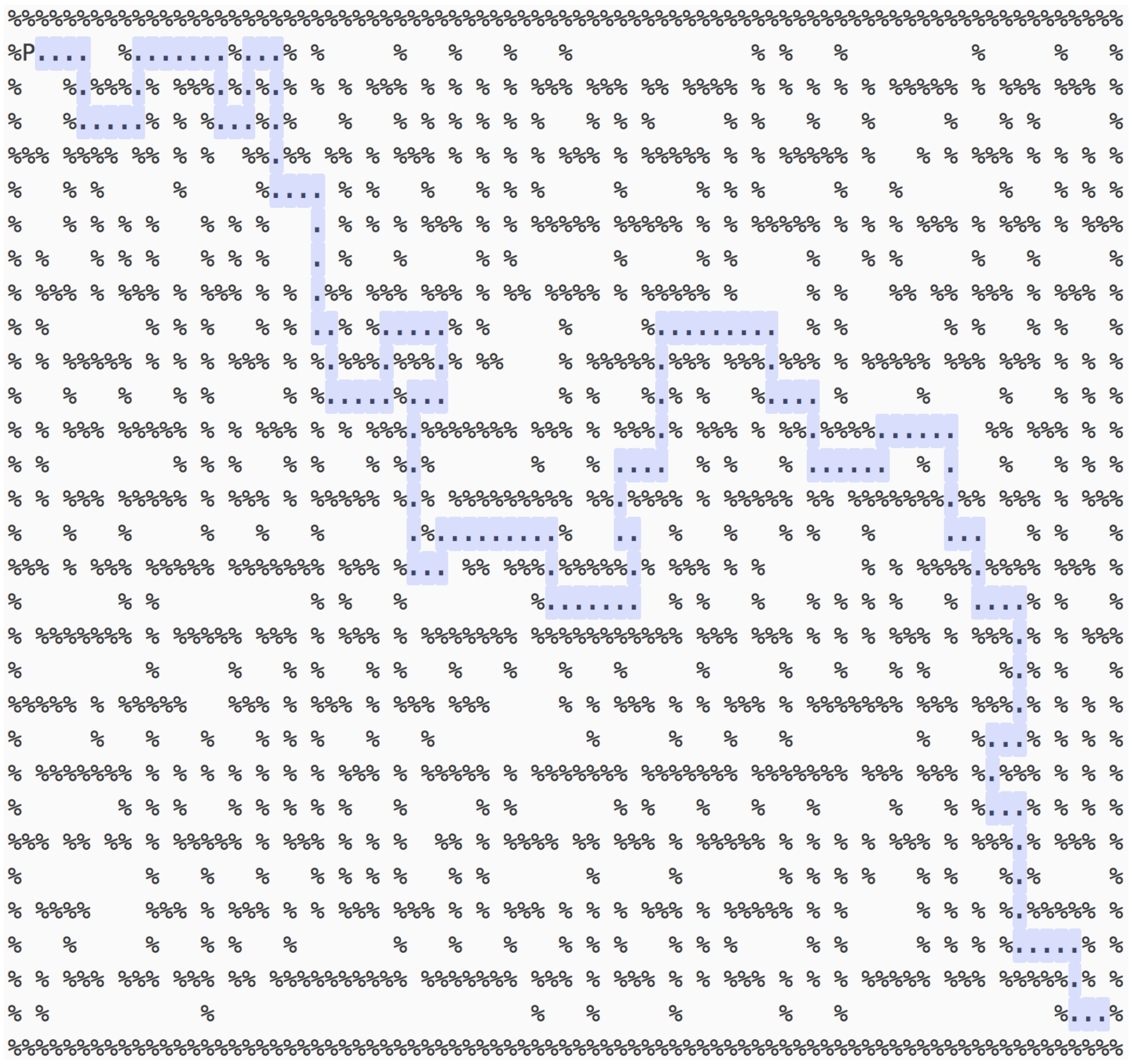


Figure 5: Solution for the large size maze using BFS

Depth-first Search:
Path cost: **234**
Expanded nodes: **251**

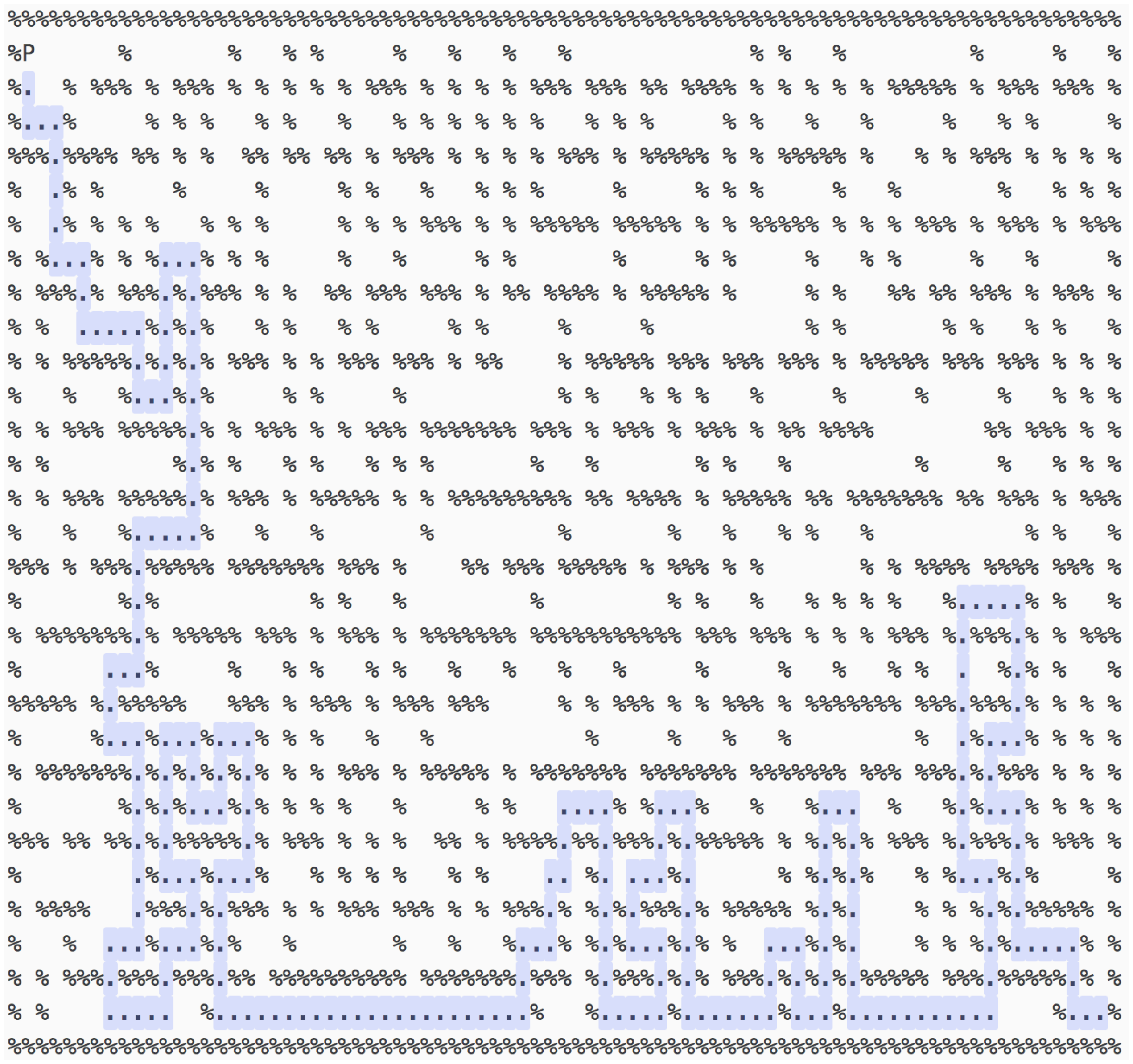
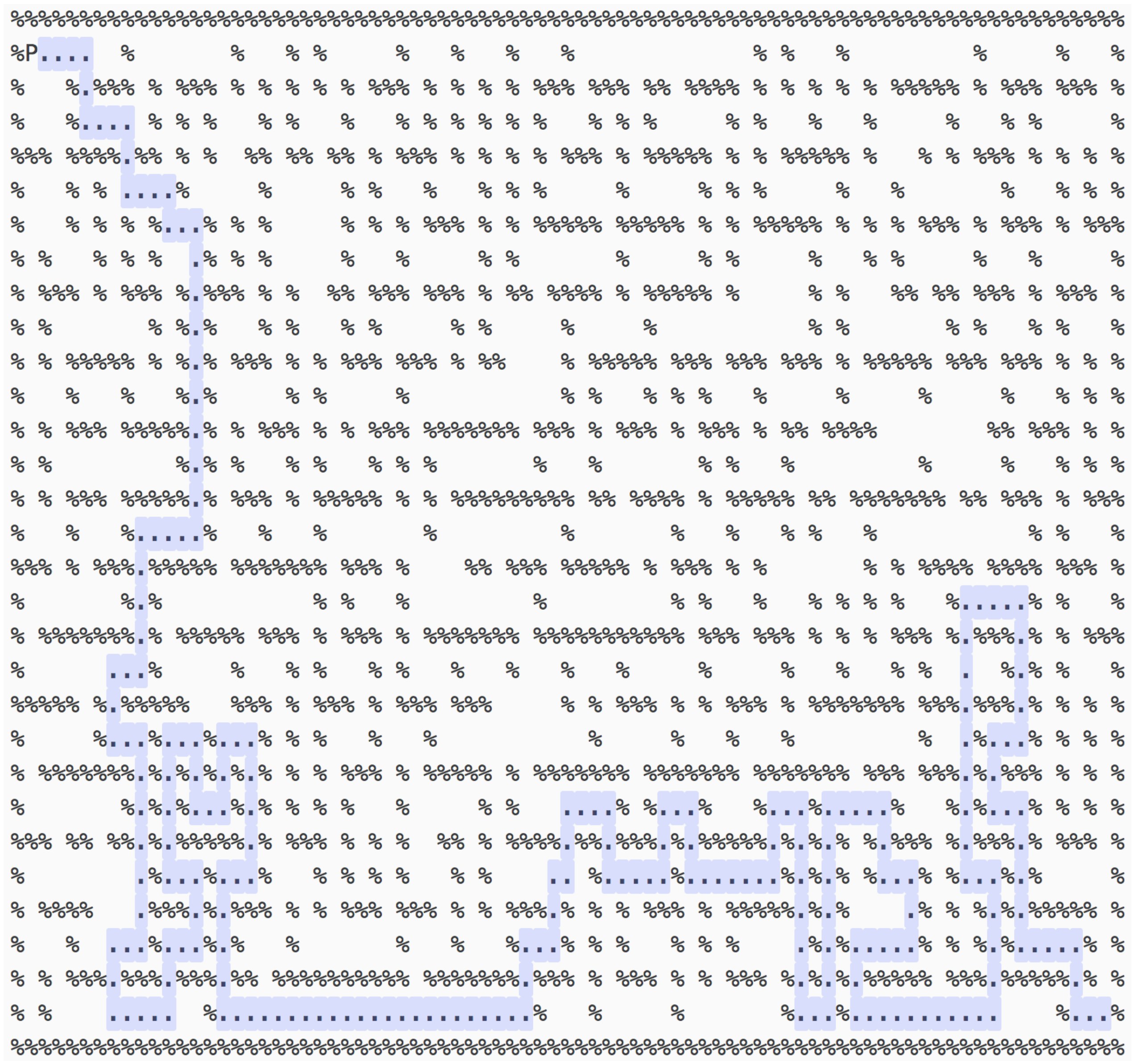


Figure 6: Solution for the large size maze using DFS

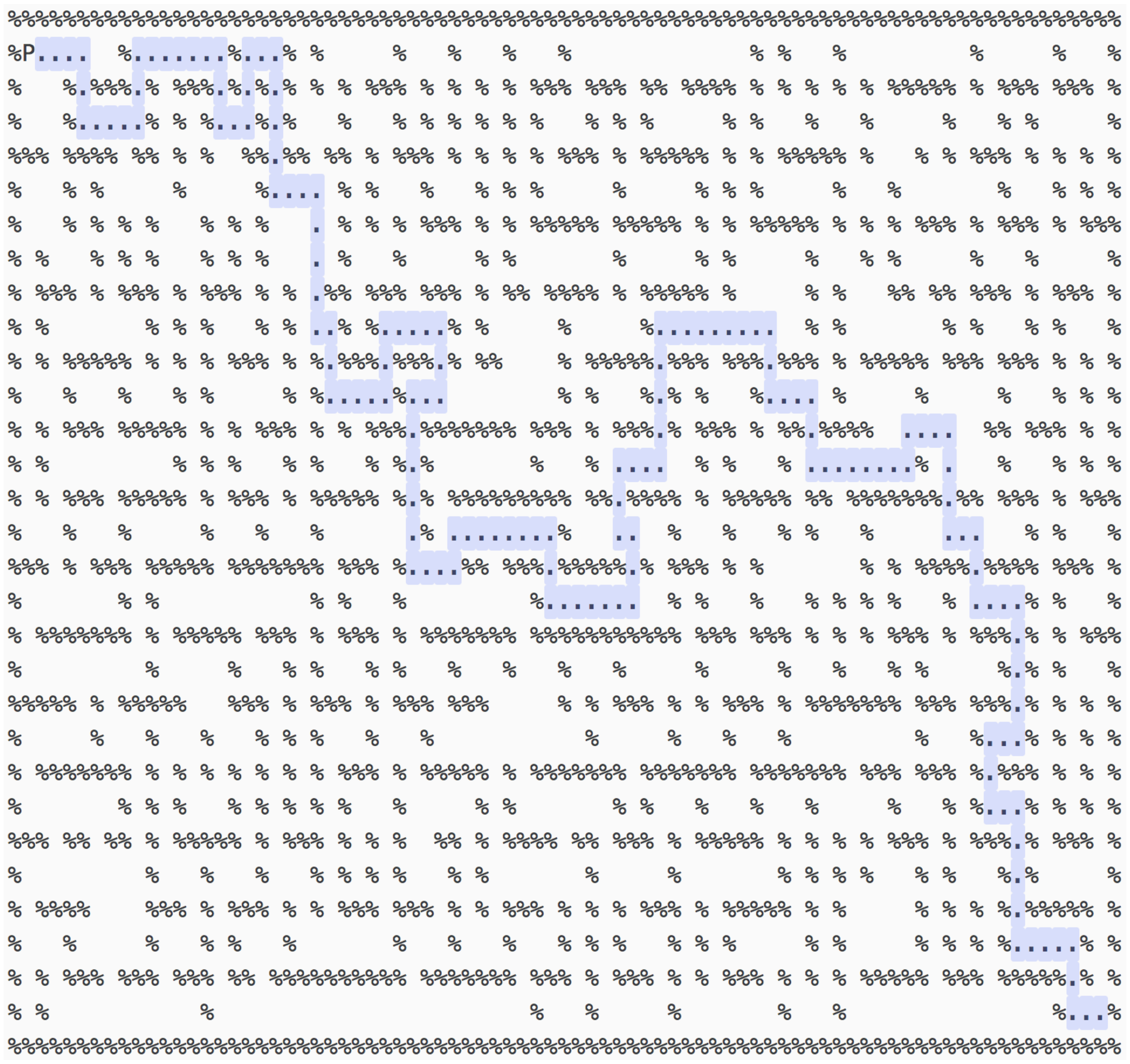
Greedy Best-first Search:
Path cost: **222**
Expanded nodes: **277**



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

Figure 7: Solution for the large size maze using Greedy BFS algorithm

A*:
Path cost: **148**
Expanded nodes: **1,112**

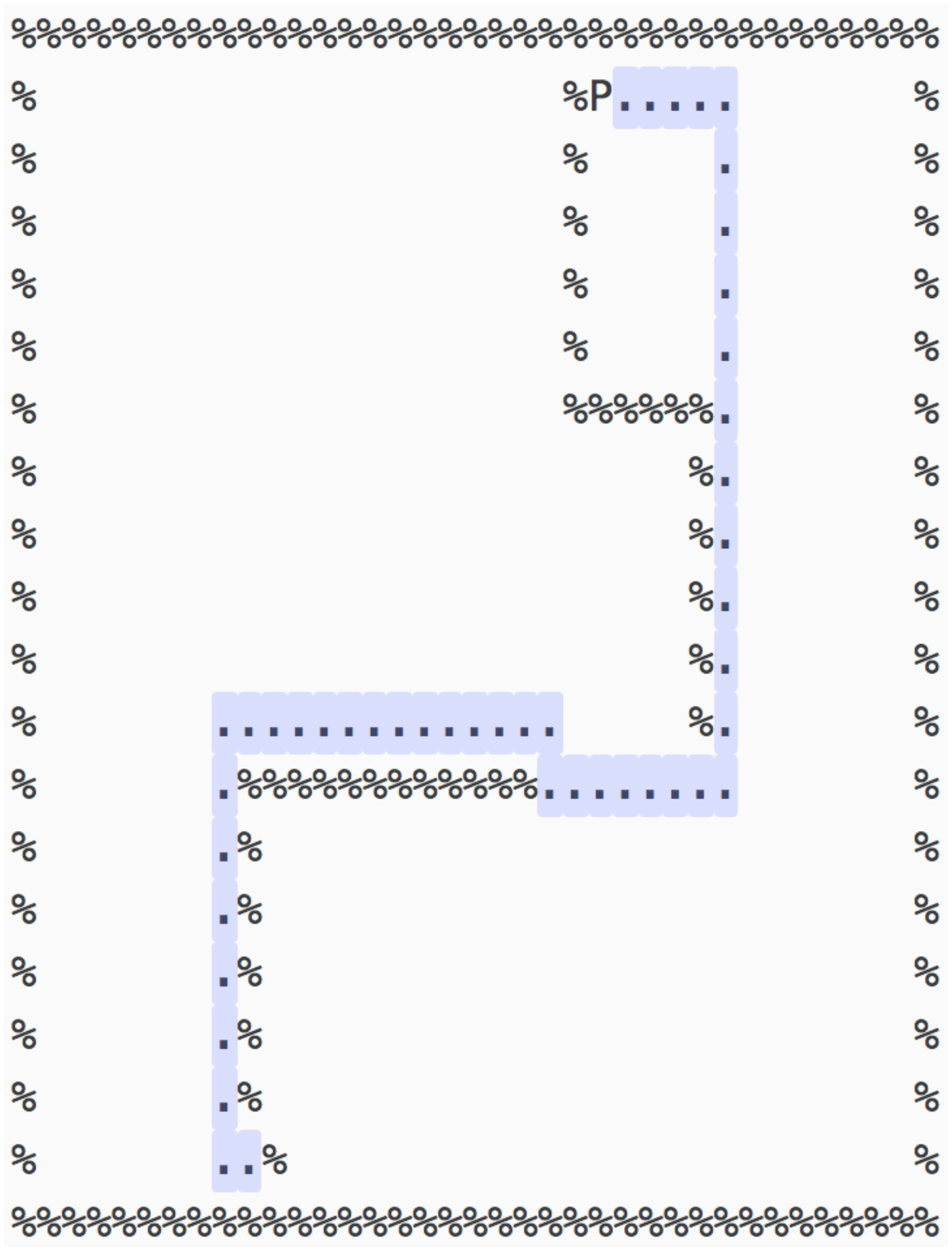


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%P....  %.......%...% %     %   %   %   %             % %   %         %     %   %
%   %.%%%.% %%%.%.%.% % % %%% % % % % %%% %%% %% %%%% % % % % %%%%% % %%% %%% %
%   %.....% % %...%.%   %   % % % % % %   % % %     %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

Figure 8: Solution for the large size maze using A* algorithm

Open maze

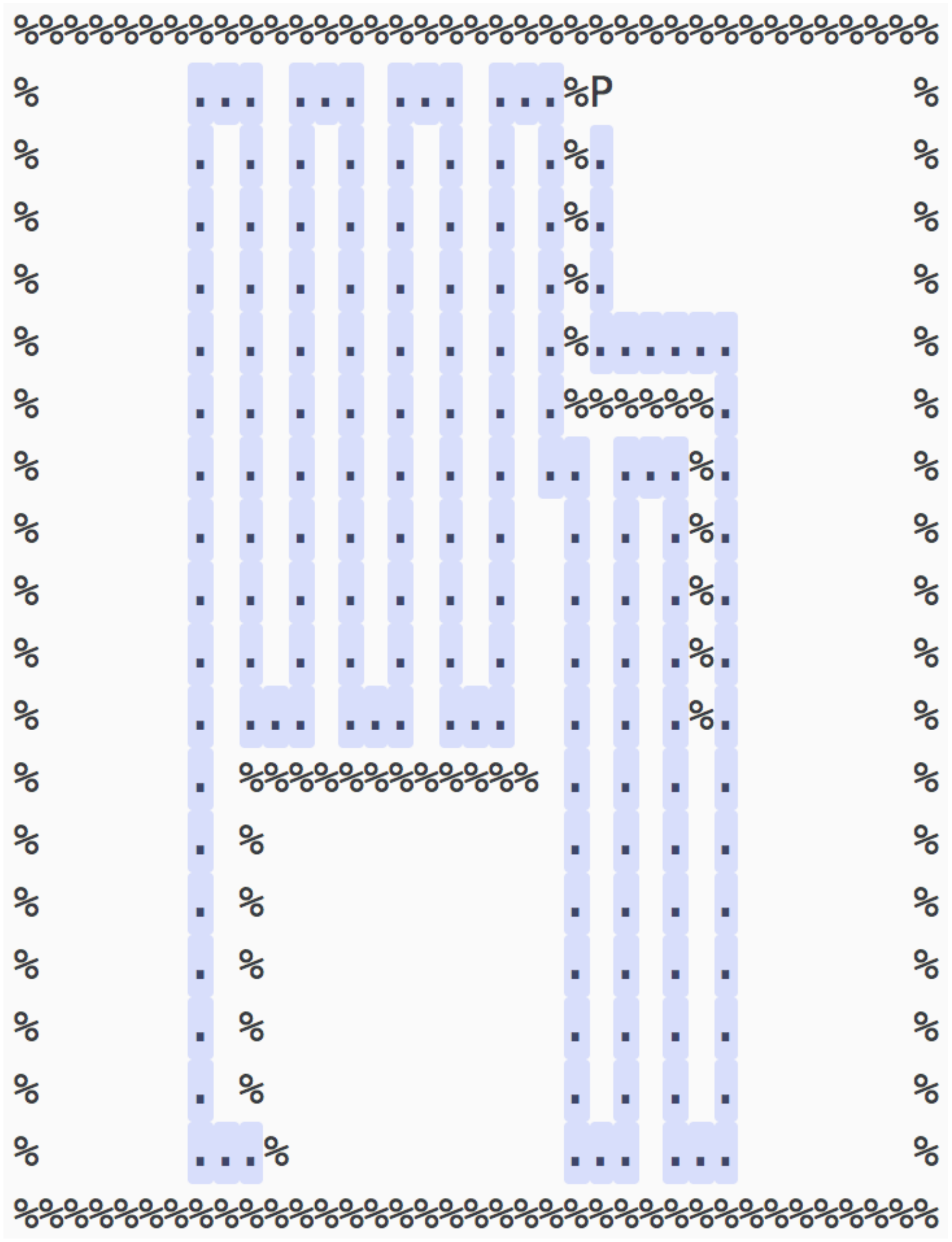
Breadth-first Search:
Path cost: **45**
Expanded nodes: **527**



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

Figure 9: Solution for the open maze using BFS

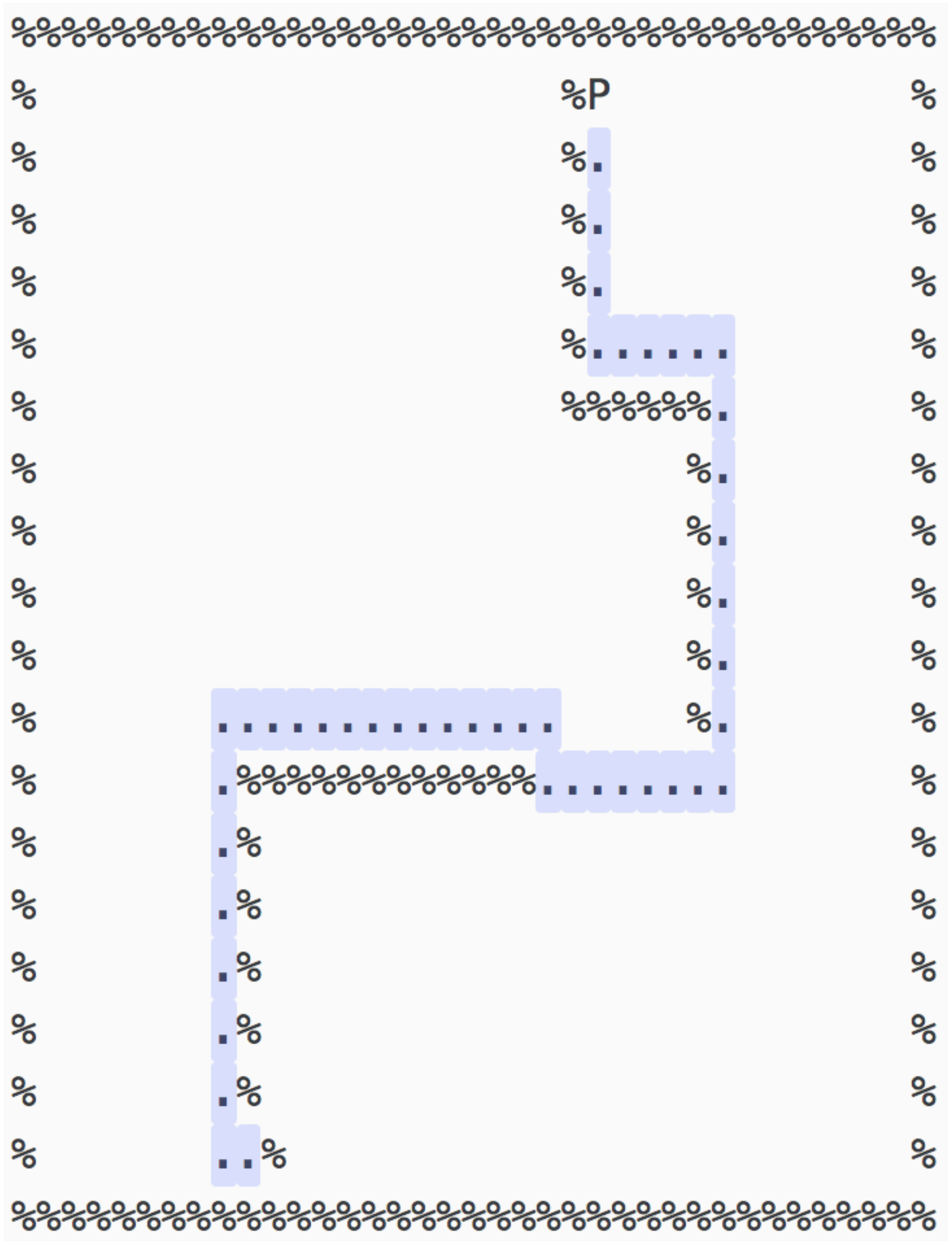
Depth-first Search:
Path cost: **161**
Expanded nodes: **287**



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

Figure 10: Solution for the open maze using DFS

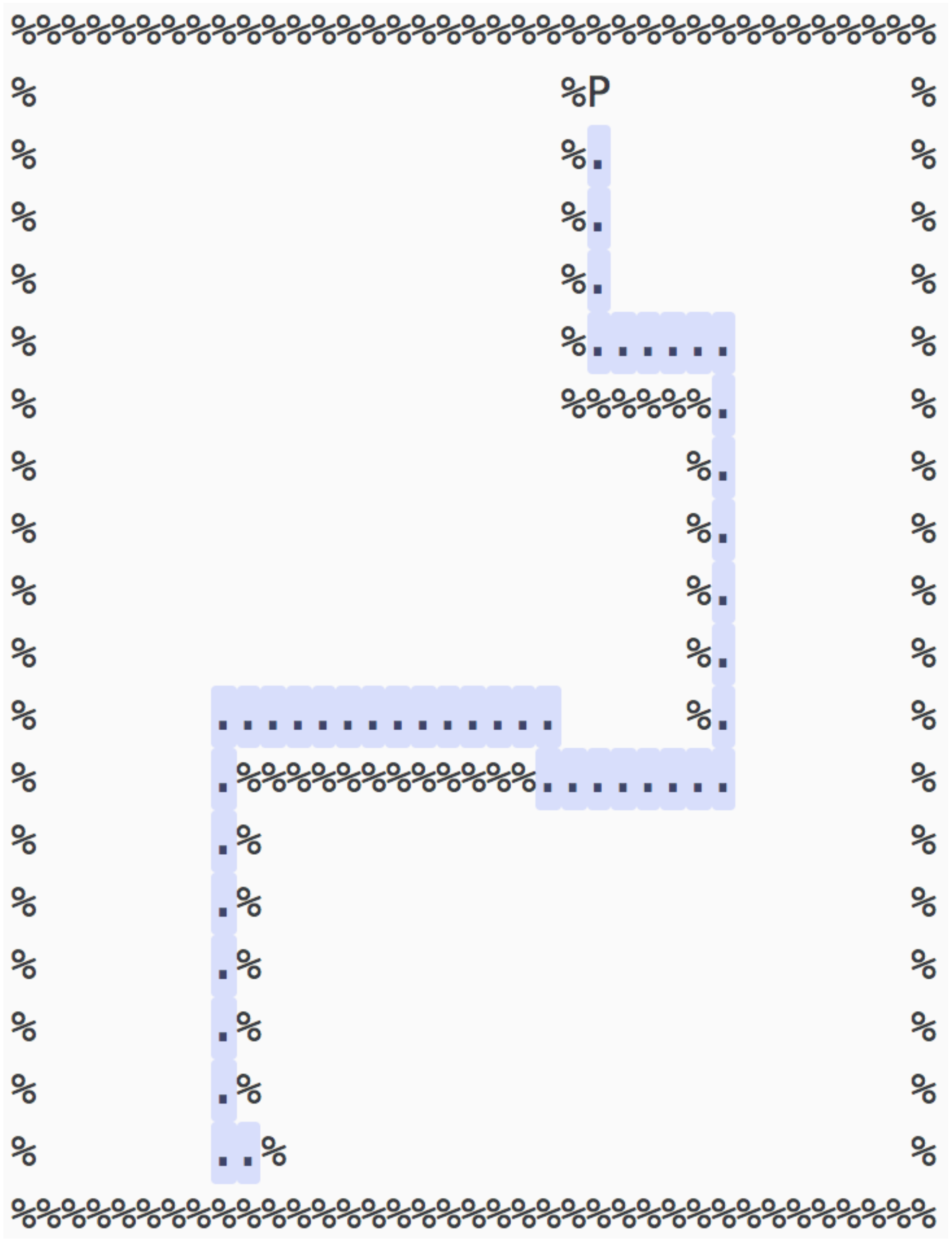
Greedy Best-first Search:
Path cost: **45**
Expanded nodes: **148**



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

Figure 11: Solution for the open maze using Greedy BFS

A*:
Path cost: **45**
Expanded nodes: **233**



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

Figure 12: Solution for the open maze using A* algorithm

PART 1.2: Search with multiple dots

The problem definition for this part is under the **PacmanProblem** class, which lives in pacman_problem.py.

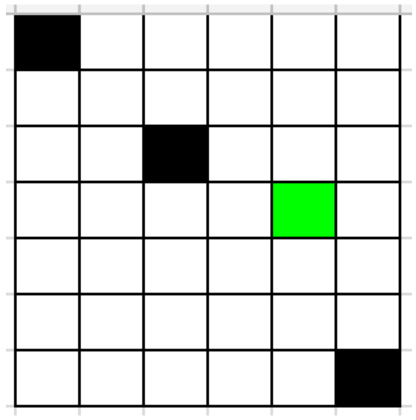
Once part 1.1 was finished I made the following changes to extend the solving algorithm to multiple dots:

1. goal positions: during load time instead of looking for just one occurrence of the food pellet (dot) I look for all of them and, for each, I calculate its (row, column) coordinate.
2. goal reached: whenever the agent encounters a food pellet it is taken out of the maze and its position removed from the remaining goal positions.

Optimizations
As mentioned before, I'm using a bitmap to represent the state of the food pellets. The bitmap starts with all the bits **off** and, as the agent captures food pellets, the respective positions are tuned **on**.

At first I was using the **PriorityQueue** class but, after some profiling, I noticed some of the operations were taking some time. After some quick research I learned that the *heapq* package from Python's standard library has a better performance. The reason for this is that **PriorityQueue** uses *heapq* but it adds thread safety. So I switched to using *heapq*.

Heuristic
I tried a couple of heuristics until I finally settled with one.
First I tried *Manhattan distance*, as one would expect, the performance was awful so I moved on.
Then I tried *Nearest Neighbor Path*. Basically taking the distances to the closest food pellet, move to that position and repeat. Doing research I read that most greedy algorithms fail the admissibility tests and I was aware that this was a greedy algorithm. Finally I could create an example that would prove the inadmissibility of the heuristic.
Consider the following image:



where the green square is the agent and the black squares are the food pellets.
It is easy to see that using the Nearest Neighbor Path will yield a cost of 18 when the optimal solution would be to visit the bottom-right pellet first (path cost of 15). Therefore this heuristic is not admissible and I had to move on again.
Finally I realized that the problem can be "reduced" to the Traveling Salesman Problem (TSP) where the nodes are the food pellets and the weight of the edges are the Manhattan distance between the vertices.
Then I tried to find a relaxed problem of the TSP and I decided to use the *Minimum Spanning Tree*(MST). It seems to be a relaxation because now you are not required to from the last node it was visited. Instead the MST can look like this (where there are multiple edges going out of some nodes).

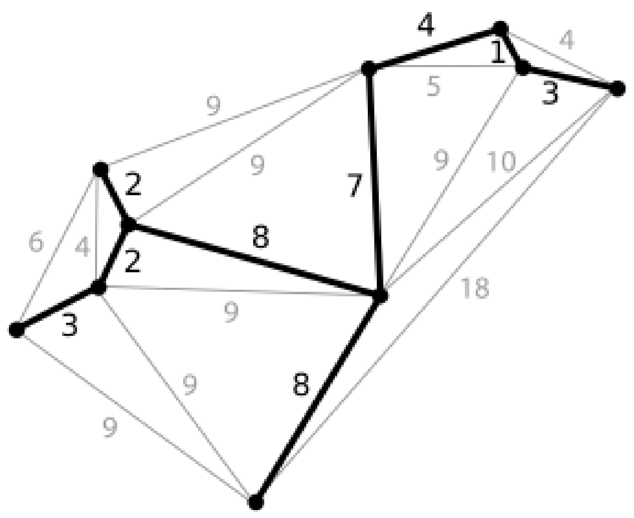


Figure 13: Minimum spanning tree[2]

Based on this I define my heuristic as follows:

$$h(x,y) = \sum_{e}^{E} w(MST(y), e) + L_0(x,y)$$

where:
x: is the agent's position
y: is the list of n available food pellets
MST is a function that finds the MST of the fully connected graph created by the food pellets.
L_0: Manhattan distance between the agent's position and the closest food pellet.
E: the n edges returned by MST
e: each of edges in E
w: the weight of the edge e in the MST.

[2]https://en.wikipedia.org/wiki/Minimum_spanning_tree

Admissibility
By definition we know that the MST has the minimum total weight that covers all the vertices.
Lets assume that the agent did take a shorter path than the total weight of the MST (MST_0). Since the agent had to visit all the food pellets then its path can be seen as an MST (MST_1) for the same vertices. This is a contradiction because we assumed MST_0 had the shortest path.

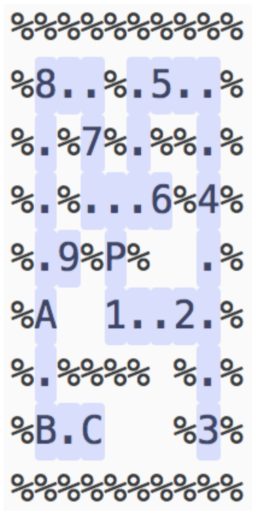
An even better heuristic would be to use Christofides[3] Algorithm. But it does come with extra calculation overhead so for now I'm sticking with the heuristic described above.
Now lets see the results.

[3]`https://en.wikipedia.org/wiki/Christofides_algorithm`

PART 1.2: Results

Tiny maze

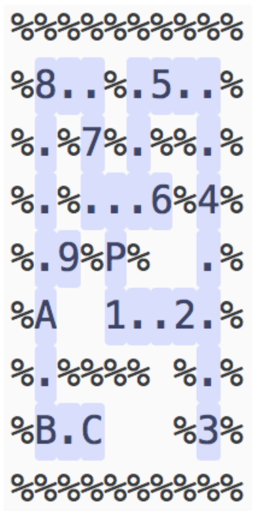
Breadth-first Search:
Path cost: **36**
Expanded nodes: **52,952**



```
%%%%%%%%%%
%8..%.5..%
%.%7%.%%.%
%.%...6%4%
%.9%P%  .%
%A  1..2.%
%.%%%% %.%
%B.C   %3%
%%%%%%%%%%
```

Figure 14: Solution path using BFS

A*:
Path cost: **36**
Expanded nodes: **504**

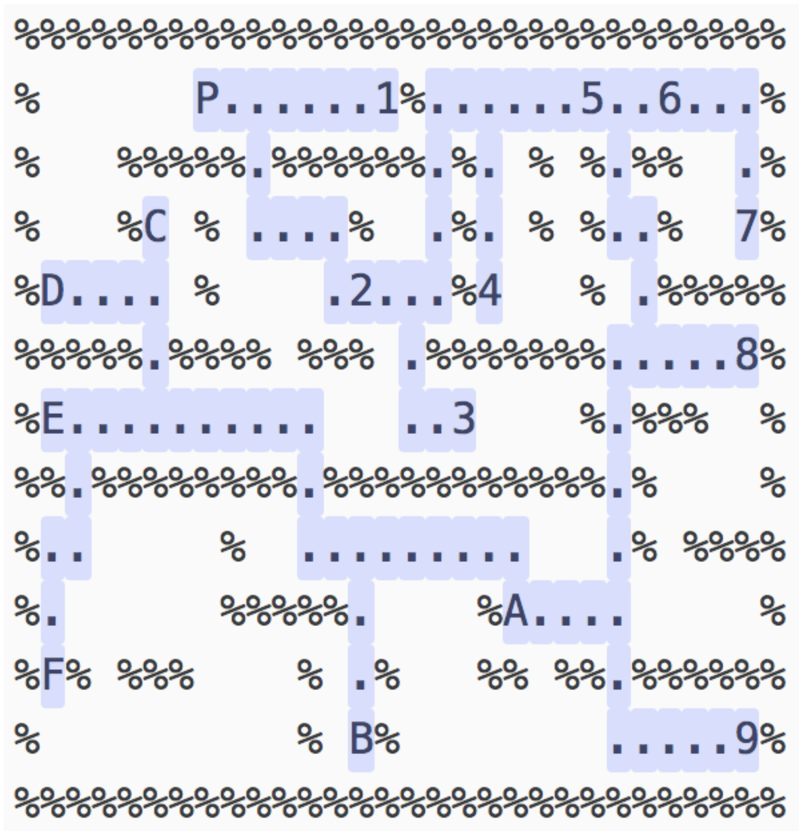


```
%%%%%%%%%%
%8..%.5..%
%.%7%.%%.%
%.%...6%4%
%.9%P%  .%
%A  1..2.%
%.%%%% %.%
%B.C   %3%
%%%%%%%%%%
```

Figure 15: Solution for the tiny maze with 12 food pellets

Small maze

A*:
Path cost: **143**
Expanded nodes: **926,059**

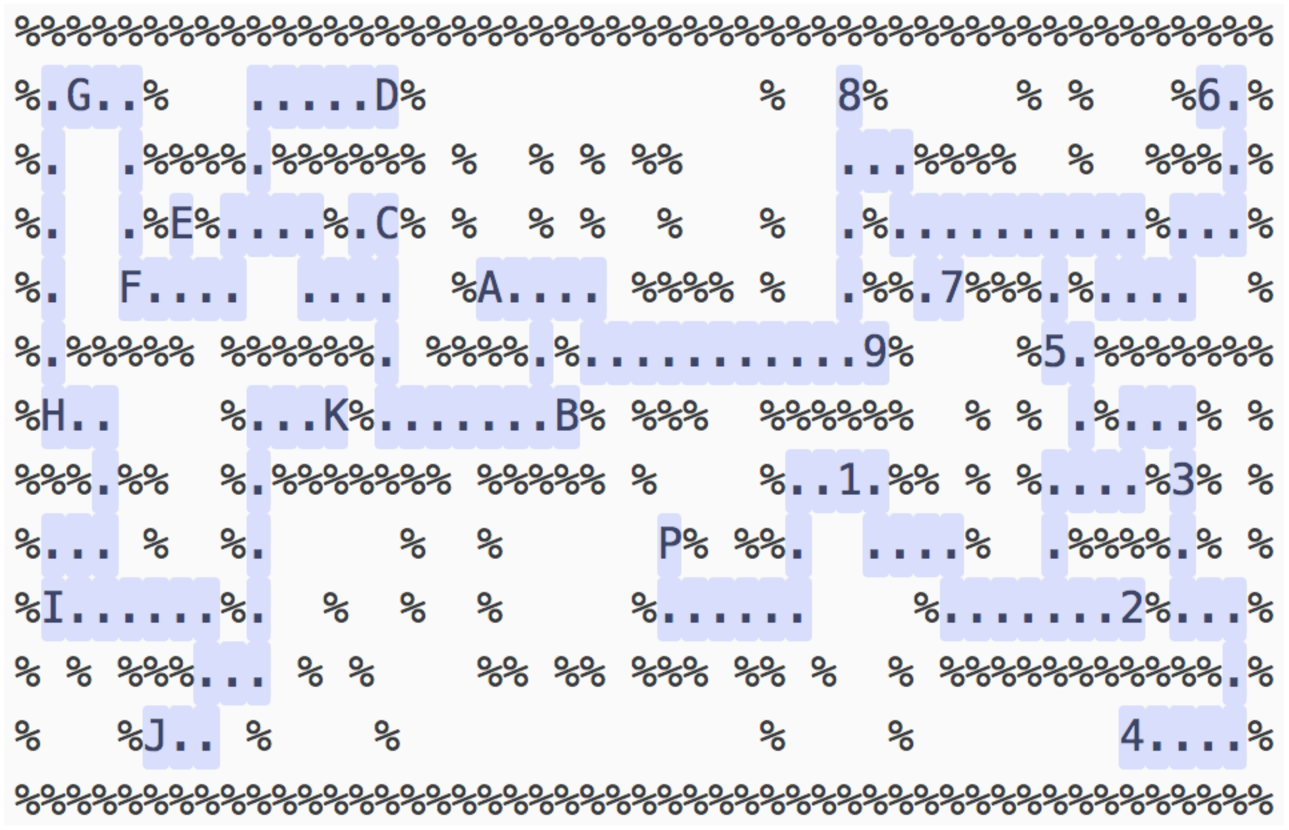


```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      P......1%......5..6...%
%   %%%%%.%%%%%%.%. % %.%%  .%
%   %C % ....%  .%. % %..%  7%
%D.... %    .2...%4   % .%%%%%
%%%%%.%%%% %%% .%%%%%%%.....8%
%E..........   ..3    %.%%%  %
%%.%%%%%%%%.%%%%%%%%%%%.%    %
%..     %  .........   .% %%%%
%.      %%%%%.    %A....     %
%F% %%%    % .%   %% %%.%%%%%%
%          % B%        .....9%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 16: Solution for the small maze with 15 food pellets

Medium maze

A*:
Path cost: **207**
Expanded nodes: **19,978,210**



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%.G..%   .....D%             %  8%     % %   %6.%
%.  .%%%%.%%%%%% %  % % %%      ...%%%%  %  %%%.%
%.  .%E%....%.C% %  % %  %   %  .%..........%...%
%.  F....  ....  %A.... %%%% %  .%%.7%%%.%....  %
%.%%%%% %%%%%%. %%%%.%...........9%    %5.%%%%%%%
%H..    %...K%.......B% %%%  %%%%%%  % % .%...% %
%%%.%%  %.%%%%%%% %%%%% %    %..1.%% % %....%3% %
%... %  %.     %  %      P% %%.  ....%  .%%%%.% %
%I......%.  %  %  %     %......    %.......2%...%
% % %%%... % %    %% %% %%% %% %  % %%%%%%%%%%%.%
%   %J.. %    %              %    %        4....%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 17: Solution for the medium maze with 20 food pellets

Part 2

Lets first define our problem:
Initial state: any position in a maze with X storage locations and that is not a wall and some placement configuration of X boxes.
Goal state: every one of the X boxes is placed on a different storage location.
Cost function: number of steps it took to reach the goal state.
Actions: GO_UP, GO_DOWN, GO_LEFT, GO_RIGHT.
Transition model: for each of the 4 directions:

1. check if the cell ($cell_1$) is empty.
2. if $cell_1$ is empty, going that direction is a valid action.
3. if $cell_1$ isn't and it is a wall, going that direction is an illegal action.
4. if $cell_1$ isn't and it is a box, check the adjacent cell ($cell_2$) to $cell_1$ in the same direction $cell_1$ is with respect to the agent.
5. if $cell_2$ is empty, going that direction is a valid action.
6. otherwise is an illegal action.

For Part 2 I reused the **SearchProblemSolver** class I created for Part 1 and created new classes for the problem definition of Sokoban (sokoban_problem.py).

Lesson learned from Part 1: use good data structures from the beginning. Using bitmaps made a huge difference in Part 1 so I decided to use it as much as possible for Part 2.

Let N, c and r be the number of cells, columns and rows in the maze, respectively, where $N = c \times r$.
First, the *walls* are represented as a bitmap of size N where all the bits are set to 0 except for those positions in which there is a wall in the maze. Similarly, *storage_locations* holds the references to where the storage locations are placed in the maze. It is also a bitmap of size N.
Since neither of them change throughout the game I've kept them as part of the definition problem and not of the state representation. You can think of this as converting the 2D representation of the maze into a linear representation.

Second, for the state representation I created the **SokobanProblemState** class which holds the current position in the maze (absolute position in our linear representation of the maze) and the positions of the boxes in the maze The positions of the boxes, just like the walls and storage locations, are represented as a bitmap of size N where all the position where the boxes are currently in are set to 1 and everything else is set to 0.

It is easy to see that the goal state is reached by performing the following check:

$$\text{box\_state} \;\&\; \text{storage\_locations} = \text{storage\_locations}$$

where & is the bitwise *and* operator.

Heuristic

This problem is clearly a variant of an assignment problem where we want to pair every box with one of the storage locations in an optimal way and, on top of that, consider the distance traveled by the agent.
I decided to use the Hungarian algorithm[4] as my heuristic. Note that I'm completely ignoring the fact that the agent has to move across the maze to reach each of those boxes. I'm only taking the boxes and locations assignment problem so this is definitely a relaxed version of the original problem and, therefore, the heuristic is admissible (although maybe not the smartest).

I considered improving the heuristic by using the Hungarian method for the assignment and MST for the agent traveling but I could come up with an example that makes the heuristic inadmissible so I decided to just use the assignment part of the problem.

To be specific, I'm calculating the assignment of the *misplaced boxes* to the empty storage locations. Again, I used bitwise operations to find these for the sake of performance.

Results

Besides solving the puzzles using A* with Hungarian method as a heuristic I also solved then running BFS. This way I could also check if the costs found by A* were correct.

Lets first have a look at the path cost found.

	BFS	A*
Input 1	8	8
Input 2	144	144
Input 3	34	34
Input 4	72	72

Table 1: Path costs for all puzzles.

So my heuristic is definitely finding the optimal paths for all four input puzzles. Now lets look at the number of nodes expanded by each search strategy.

[4] https://en.wikipedia.org/wiki/Hungarian_algorithm

	BFS	A*	Improvement
Input 1	39	21	46%
Input 2	44,552	44,470	0.2%
Input 3	485,182	34,961	93%
Input 4	565,279	534,247	5.5%

Table 2: Number of expanded nodes for all puzzles.

As one would expect, the number of expanded nodes decreases when using A*. Although the numbers are lower, for Input 2 and 4 the puzzle took longer to be solved than when using BFS. Roughly twice as long (see table below, values are in seconds).

	BFS (s)	A* (s)	Ratio
Input 1	0.011	0.012	0.92
Input 2	3.281	6.401	0.51
Input 3	45.757	8.208	5.57
Input 4	49.964	102.539	0.49

Table 3: Running time for all puzzles.

In case the sequence of actions taken is of interest for you, I have included them with the source code under the *results* folder on both Part 1 and Part 2. For each combination of search strategy and puzzle performed there will be one file.
Each file includes:

1. the output of the profiler to understand where the time is being spent
2. search strategy used
3. running time
4. number of expanded nodes
5. total path cost
6. actual path taken in the form of Action taken =¿ State after action taken. Do note that the state may not be human friendly as it shows the internal (bitmap) representation of the puzzle.