**Department of Computing**

**CS370: Artificial Intelligence**

**Class: BSCS-10AB**

**Lab 02: Problem-Solving by Searching**

**Group Members**

***AliAmirKhawaja\_344200\_Ateeb Ahmed\_334030\_10-B***

**Date: 09-02-2023**

**Instructor: Dr. Hashir Kiani**

**Lab Engineer: Ms Shakeela Bibi**

**Lab 2: Problem-Solving by Searching**

**Implement Questions 1,2 and 4 from** [**http://ai.berkeley.edu/search.html**](http://ai.berkeley.edu/search.html)**, which are also reproduced below.**

**Note:** You can work in groups of 2 but not more than 2.

**Deliverables:** Submit search.py

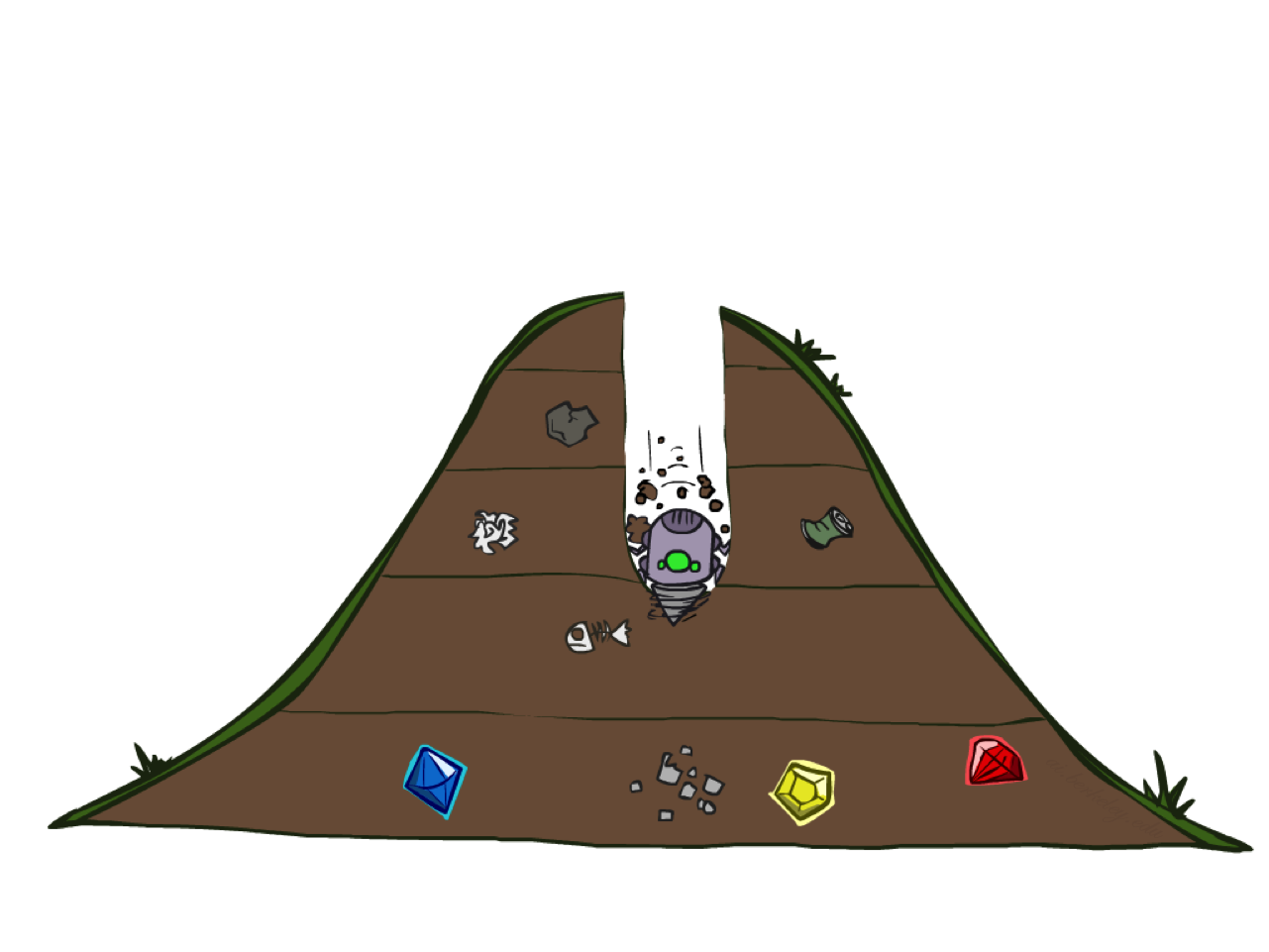
**Deadline:** Before the next lab.

**Table of Contents**

* [Introduction](#_heading=h.3znysh7)
* [Welcome](#_heading=h.2et92p0)
* [Q1: Depth First Search](#_heading=h.tyjcwt)
* [Q2: Breadth First Search](#_heading=h.3dy6vkm)
* [Q4: A\* Search](#_heading=h.1t3h5sf)



**Question 1: Finding a Fixed Food Dot using Depth First Search**

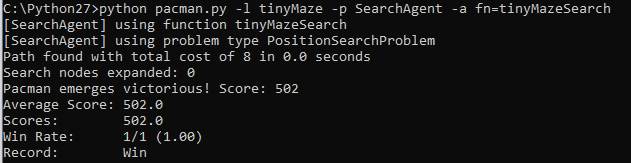


**DFS**

In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

First, test that the SearchAgent is working correctly by running:

**python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch**



The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

***Important note:*** All of your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

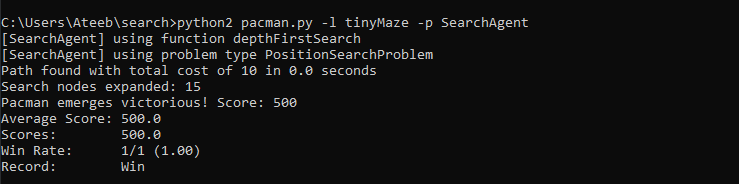
***Important note:*** Make sure to **use** the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

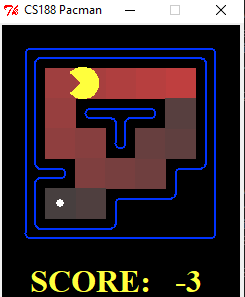
*Hint:* Each algorithm is very similar. Algorithms for DFS, BFS and A\* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need *not* be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states.

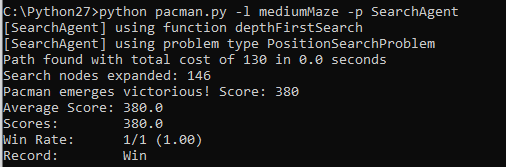
**Your code should quickly find a solution for:**

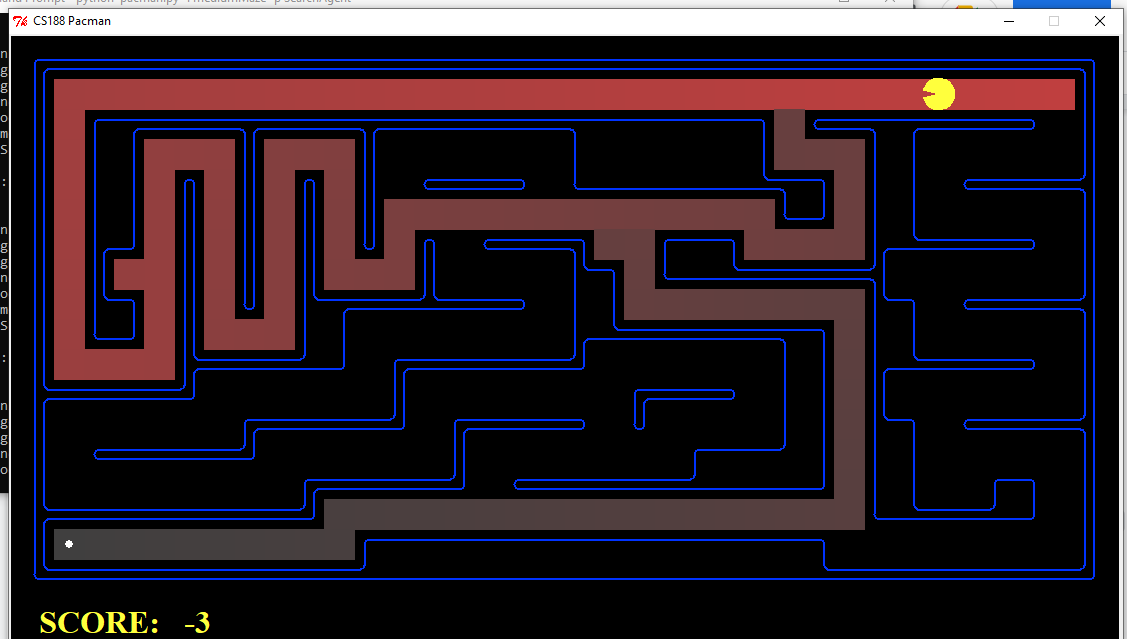
* **python pacman.py -l tinyMaze -p SearchAgent**

****

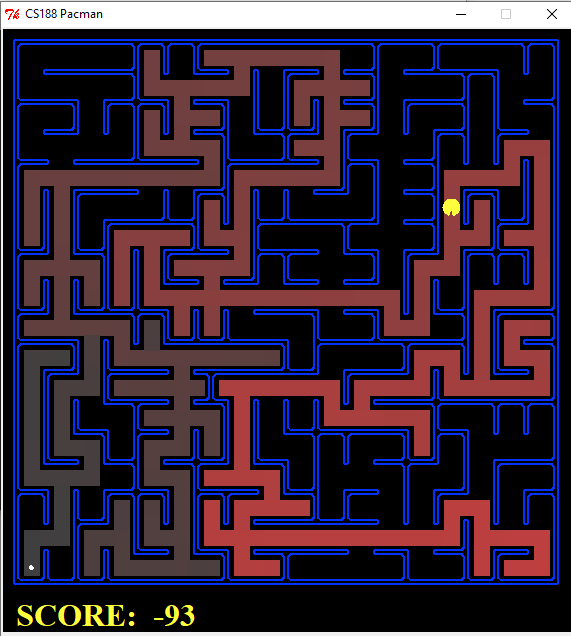
****

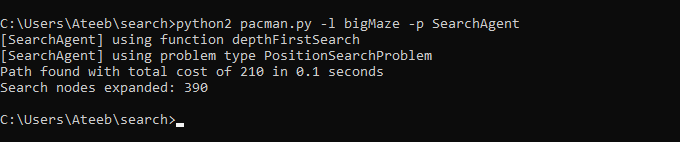
* **python pacman.py -l mediumMaze -p SearchAgent**

****

****

* **python pacman.py -l bigMaze -z .5 -p SearchAgent**

****

****

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration).

* **Is the exploration order what you would have expected?**

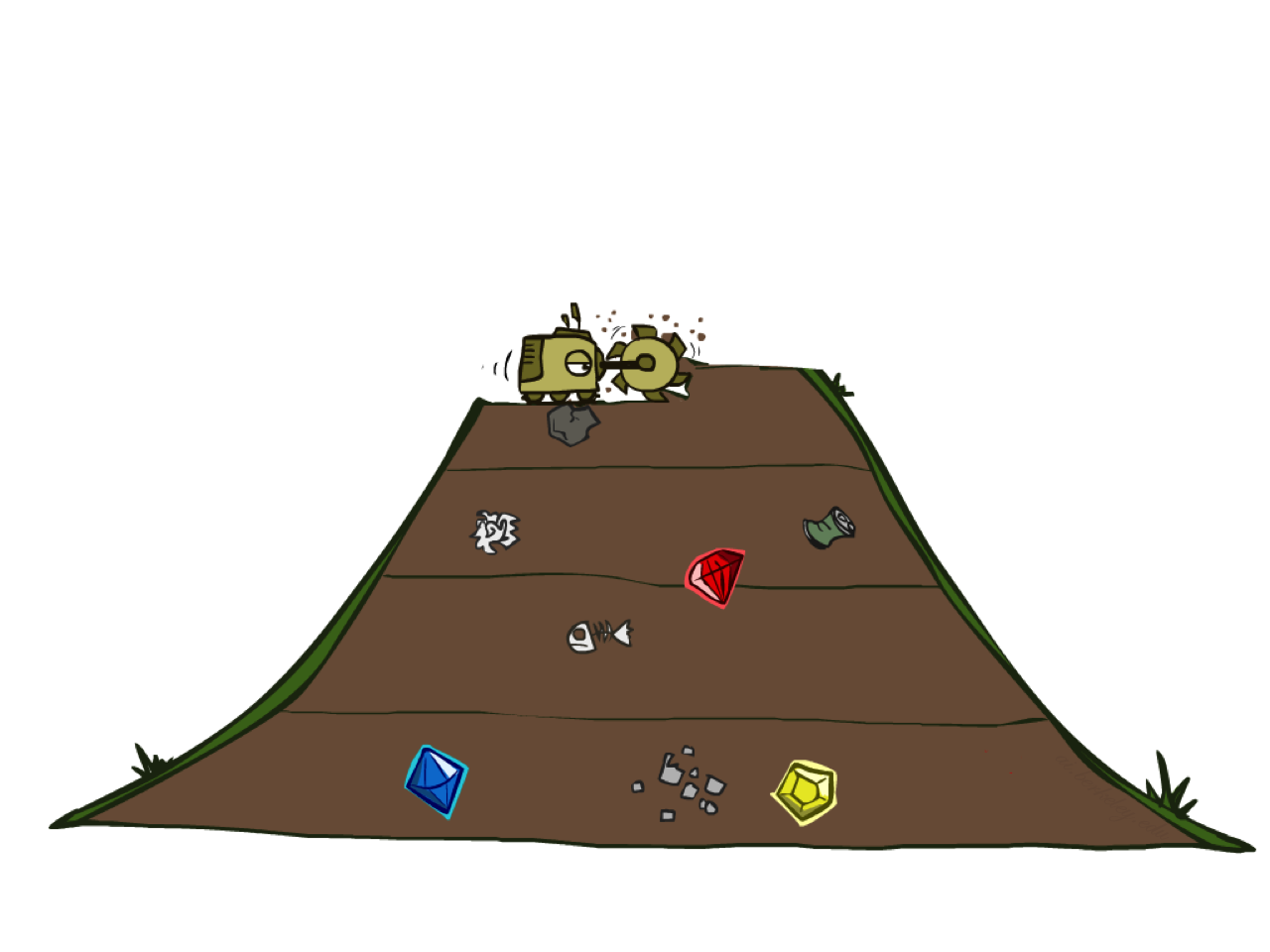
**Ans:** Yes, the exploration order is exactly what was expected from the DFS algorithm. DFS uses a LIFO (last-in, first-out) stack to construct the frontier. DFS adds a successor to the frontier and immediately expands it; in other words, it builds a path by exploring a neighbouring cell, then exploring the cell next to that, then the cell next to that, and so on

* **Does Pacman actually go to all the explored squares on his way to the goal?**

**Ans:** No, Pacman doesn't go to all the explored squares on his way to his goal. It only goes to those explored squares from where a path to food existed.



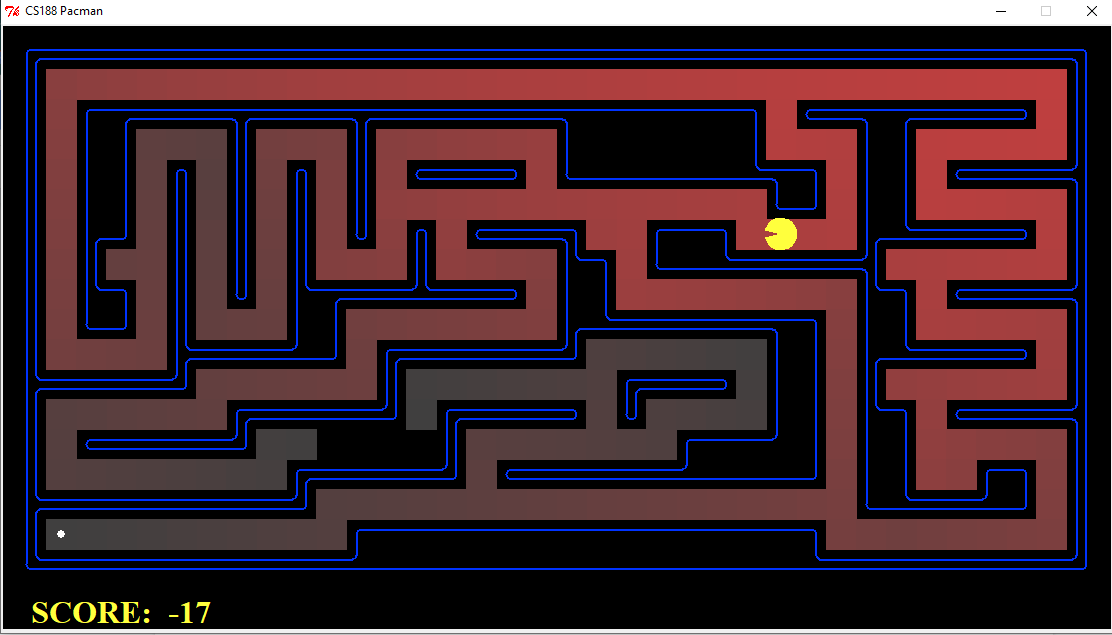
**Question 2: Breadth First Search**

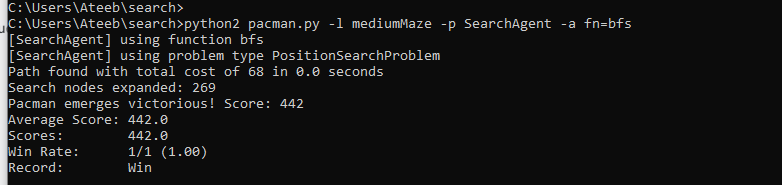


**BFS**

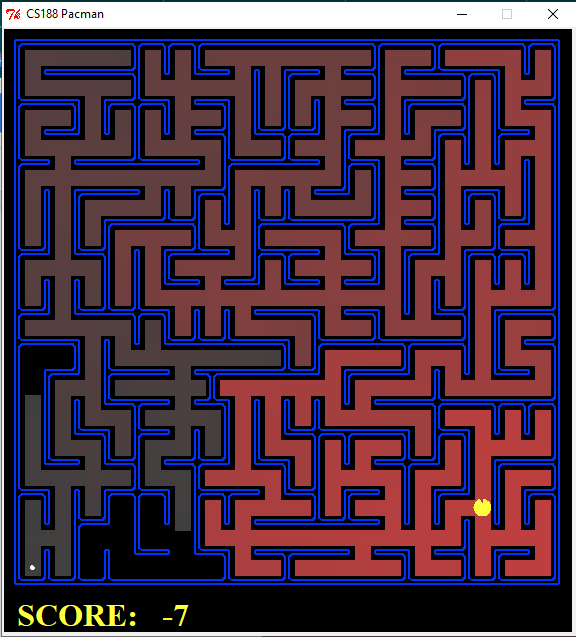
Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

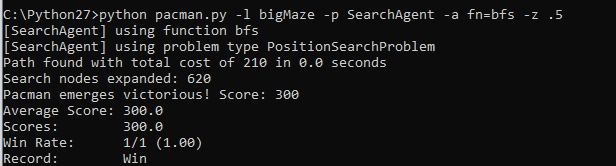
* **python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs**

****

****

* **python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5**

****

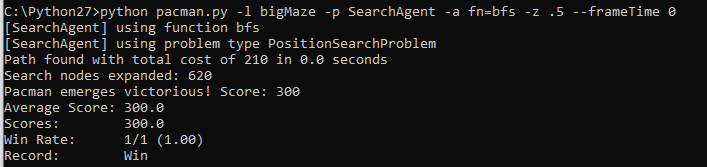
****

* Does BFS find a least-cost solution? If not, check your implementation.

**Ans:** Yes, BFS find a least-cost solution as compared to DFS. If we consider “mediumMaze” where it took 130 steps for DFS to find a path to the goal, the BFS finds a much shorter path in only 68 steps.

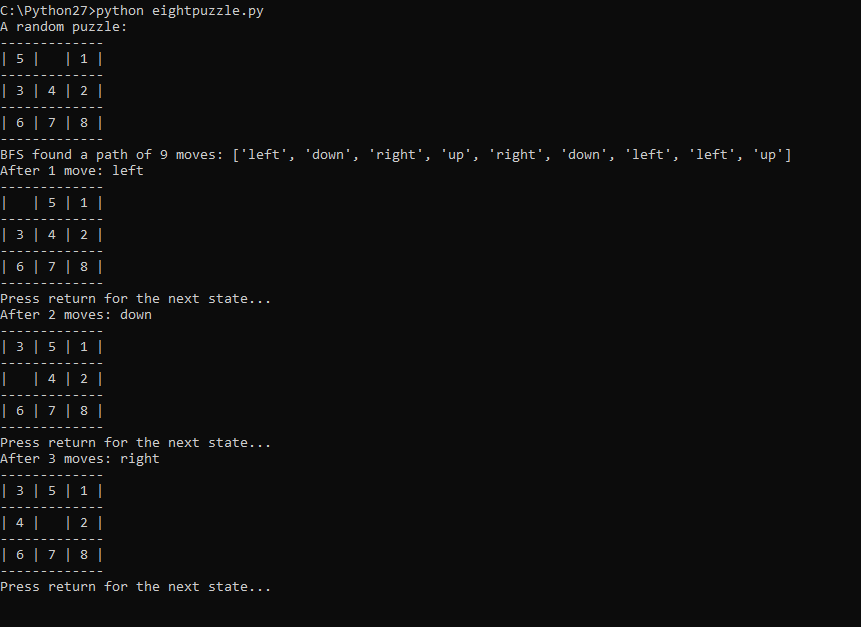
Hence, we can say that BFS is more optimal than DFS in finding a shorter path. However, we also notice that more nodes were expanded in BFS, since BFS works by expanding all nodes and then finding a shorter path. So, we can say that space was compromised.

*Hint:* If Pacman moves too slowly for you, try the option --frameTime 0.



*Note:* If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

* **python eightpuzzle.py**

****

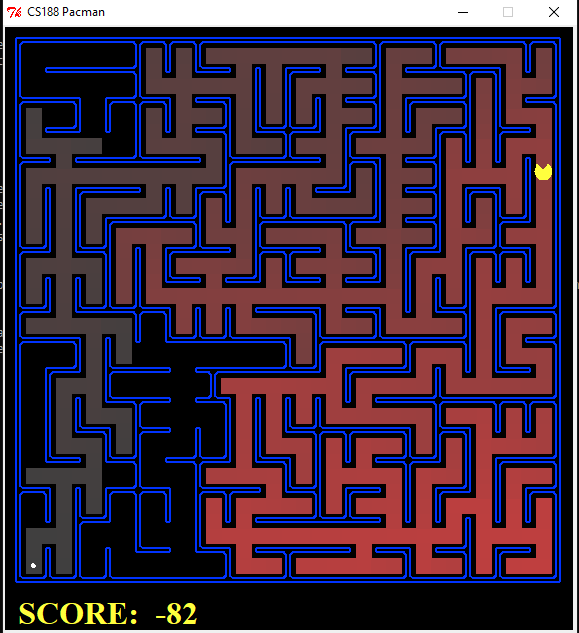


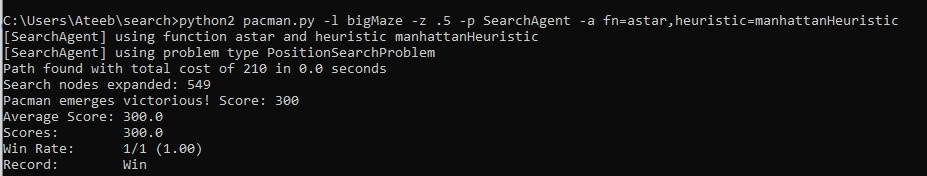
### Question 4: A\* search

Implement A\* graph search in the empty function aStarSearch in search.py. A\* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.

You can test your A\* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (implemented already as manhattanHeuristic in searchAgents.py).

**python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic**

****

****