COMS 4701 Artificial Intelligence

Homework 1: Coding - Search

Due Date: September 29, 2024

Please read carefully:

- You must name your file `maze.py`. Any submission that does not follow this naming will not be graded.
- A skeleton of some essential functions and classes have been provided to you in `maze.py`. Additionally, space has been provided to write other helper functions or statements you may need. DO NOT modify the skeleton function signatures, or any code that is not specified to be modifiable for any reason. This will be run by our autograder, so any unexpected modifications that make it malfunction will likely receive a zero. Test cases have been provided in the main function in the skeleton code.
- Only Python 3.x versions will be graded.
- You may import the following libraries: resource, time, queue. No other library may be imported by you. (Note: You may import libraries imported by the skeleton code)
- To receive points, make sure your code runs. We recommend using Spyder, Pycharm or Google Colab. They all allow you to download .py files. Be aware that if you write your code in some platforms like Codio and copy and paste it in a text file, there may be spurious characters in your code, and your code will not compile. **Always ensure that your .py compiles.**
- **YOU MAY NOT SHARE MATERIALS RELATED TO THIS HOMEWORK INCLUDING BUT NOT LIMITED TO THE QUESTIONS, CODE TEMPLATES OR YOUR SOLUTIONS ONLINE AT ANY TIME. DISCIPLINARY ACTIONS WILL BE TAKEN IF HOMEWORK MATERIALS ARE FOUND ONLINE, WHICH MAY INCLUDE RETROACTIVE GRADE CHANGES.**

In this assignment you will create an agent to traverse a maze. You will implement and compare several search algorithms, and collect some statistics related to their performances. Please read all sections carefully:

I. Introduction

The aim of this homework is to build a search agent for a robotic path planning. You will be implementing and comparing several search algorithms and evaluating their performances. More specifically, we would like to build Mark1, a search agent robot for our AI class. We want Mark1 to be able to navigate in a given map with obstacles. The first important task of the robot is assigned to you. To make crafting this first prototype easier and manageable, the robot leader has allowed for various assumptions to hold in the map and the robot. These are described as follows:

Assumptions:

1. The map is a rectangular arena of size m $\times$ n which is bounded by walls on the four sides.
2. An obstacle in the map is marked by an "o", and an empty positions is marked by " " (blank space) in the map.
3. The robot always starts from one exact position marked "s" in the map.
4. The robot has to reach one and only one goal position marked "g" in the map. The goal position is guaranteed to be reachable from the start position.
5. The robot is allowed to move only in one of the four directions (UP, RIGHT, DOWN and LEFT), one move at a time.
6. The cost of moving from one point to any neighbour is the same and equal to one. Thus, the total cost of path is equal to the number of moves made from start to the goal position.
7. Though the robot is autonomous, it is assumed that the robot is always aware of the complete map and its current location w.r.t to the map at anytime.

Example

m = 7, n = 9
Cost of path = 12 (in the below case), the path is shown with "*"

$$M = \begin{array}{|c|c|c|c|c|c|c|c|}\hline
 & & & & & & & \\ \hline
 & & g & & & & & \\ \hline
 & & o & o & o & o & o & \\ \hline
 & o & & & & & & \\ \hline
 & & o & s & & & o & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
\end{array} \quad \Longrightarrow \quad \begin{array}{|c|c|c|c|c|c|c|c|}\hline
 & & & & & & & \\ \hline
 & & g & * & * & * & * & * \\ \hline
 & & o & o & o & o & o & * \\ \hline
 & o & & & & * & * & * \\ \hline
 & & o & s & * & * & o & \\ \hline
 & & & & & & & \\ \hline
 & & & & & & & \\ \hline
\end{array}$$

NOTE: This example diagram is intended to illustrate the format of the input maze and the expected output maze only and may or may not represent a solution from any of the expected implementations

II. Algorithm Review

Recall from lecture that search begins by visiting the root node of the search tree, given by the initial state. Three main events occur when visiting a node:

- First, we remove a node from the frontier set.
- Second, we check if this node matches the goal state.
- If not, we then expand the node. To expand a node, we generate all of its immediate successors and add them to the frontier, if they (i) are not yet already in the frontier, and (ii) have not been visited yet.
- Nodes **MUST BE** expanded in the order Up, Right, Down, Left as is implemented in the *expand* method of the *MazeState* class of the pseudocode. If this order is not followed, your solution may differ from the expected solution and the autograder will mark your solution as incorrect. **Be careful when implementing any method that uses a stack or LIFO queue.**

This describes the life cycle of a visit, and is the basic order of operations for search agents in this assignment–(1) remove, (2) check, and (3) expand. We will implement the assignment algorithms as described here. Please refer to lecture notes for further details, and review the lecture pseudo-code before you begin.

IMPORTANT: You may encounter implementations that attempt to short-circuit this order by performing the goal-check on successor nodes immediately upon expansion of a parent node. For example, Russell & Norvig's implementation of BFS does precisely this. Doing so may lead to edge-case gains in efficiency, but do not alter the general characteristics of complexity and optimality for each method. For simplicity and grading purposes in this assignment, do not make such modifications to algorithms learned in lecture.

III. What You Need To Submit

Your job in this assignment is to write maze.py, which solves any maze using the graph search method(s) passed in as flags. The program will be executed as follows:

```
$python3 maze.py -m <map_filename.txt> [flags]
```

The flags will be one or more of the following (you will have implemented all of them by the end of the assignment):
-bfs (Breadth-First Search)
-dfs (Depth-First Search)
-astar (A-Star Search)
-ida (Iterative Deepening A-Star Search)
-all (Run all 4 search algorithms)

Some sample execution commands are shown below:

```
$python3 maze.py -m arena1.txt -bfs

$python3 maze.py -m arena2.txt -dfs -astar -ida

$python3 maze.py -m arena3.txt -all
```

Note: Most of the code to format the output has been provided in the skeleton. You simply need to write the *bfs, dfs, astar, ida* functions along with some of the functions in the *MazeState* class. Writing additional helper functions is highly recommended but optional.

IV. What Your Program Outputs

Your program will write the following for each graph search algorithm requested in the flags to the stdout:

path_to_goal: the path taken by the robot shown as "*" on the graph
cost: the number of steps taken to reach the goal
nodes_expanded: the number of nodes that have been expanded
max_nodes_stored: the maximum number of nodes stored in the frontier set during the runtime of the algorithm
max_search_depth: the maximum depth of the search tree in the lifetime of the algorithm
running_time: the total running time of the search instance, reported in seconds
max_ram_usage: the maximum RAM usage in the lifetime of the process as measured by the **ru_maxrss** attribute in the **resource** module, reported in kilobytes.

Your code will only be tested directly on the *bfs, dfs, astar, ida* functions. The input to each of these functions is an array of strings in which each string represents a row of the maze. The expected output is the variables listed above in that order. The path_to_goal is returned as an array of strings in the same format as the input, except with the solution path traced with "*". The other variables are returned as integers or floating point numbers.

As long as these 4 functions return the correct output for any given input, you should receive a full score.

Note on Correctness

All variables, except running_time and max_ram_usage, have **one and only one** correct answer when running BFS and DFS. A* and Iterative Deepening A* nodes_expanded, max_nodes_stored, and max_search_depth might vary depending on implementation details. You'll be fine as long as your algorithm follows all specifications listed in these instructions.

As running_time and max_ram_usage values vary greatly depending on your machine and implementation details, there is no "correct" value to look for. They are for you to monitor time and space complexity of your code, which we highly recommend. **running_time and max_ram_usage MUST be returned even if we do not grade them. If you do not return these values for a test case, you may receive a 0 for that test case.**

A good way to check the correctness of your program is to walk through small examples by hand, like the ones above. Use the following piece of code to calculate max_ram_usage:

```
import resource
dfs_start_ram = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
dfs_ram = (resource.getrusage(resource.RUSAGE_SELF).ru_maxrss - dfs_start_ram)/(2**10)
```

Our grading script is working on a linux environment. For windows users, you may change max_ram_usage calculation code so it is linux compatible during submission (You can test you code on linux platforrm using services such as Google Colab). However, since the value of max_ram_usage will not be tested, this step is not necessary.

V. Implementation and Testing

For your first programming project, we are providing hints and explicit instructions. Before posting a question on the discussion board, make sure your question is not already answered here or in the FAQs.

1. Implementation

You will implement the following four algorithms as demonstrated in lecture. In particular:

- **Breadth-First Search**. Use an explicit queue, as shown in lecture.
- **Depth-First Search**. Use an explicit stack, as shown in lecture.
- **A-Star Search**. Use a priority queue, as shown in lecture. For the choice of heuristic, use the Manhattan distance; that is, the l-1 distance between the current position and the goal positions.
  Note: It may be helpful here to modify the PuzzleState object to allow for comparison between 2 states. You can explore pythons __eq__, __hash__ and __lt__ methods or other such methods for this purpose. Implementations of the __eq__ and __hash__ methods have been provided for you, but you may modify them.
- **Iterative Deepening A-Star Search**. It is recommended that you use a recursive Depth Limited Search helper function. For the choice of heuristic, use the Manhattan distance. As an additional minor optimization, you may think about what the shortest possible solution for a given maze could be and start there instead of starting with a depth limit of 1.

2. Order of Visits

In this assignment, where an arbitrary choice must be made, we always **visit** child nodes in the **"URDL"** order; that is, ['Up', 'Right', 'Down', 'Left'] in that exact order. Specifically:

- **Breadth-First Search**. Enqueue in URDL order; de-queuing results in URDL order.
- **Depth-First Search**. Push onto the stack in reverse-URDL order; popping off results in URDL order.
- **A-Star Search**. Since you are using a priority queue, what happens with duplicate keys? How do you ensure nodes are retrieved from the priority queue in the desired order?
- **Iterative Deepening A-Star Search**. When performing regular A-Star Search, we do not revisit nodes. Is that the case with Iterative Deepening A-Star Search as well? Or do we need to revisit nodes to ensure an optimal solution?

Ensure your algorithm starts at the start node and ends at the goal to avoid alternate solutions which will be marked as incorrect by the autograder.

3. Submission Test Cases

We have provided 3 test arenas for you to test your code. You may test each of your graph search algorithms on the given test arenas. The arenas are small enough that you should be able to validate your code by hand. While all your algorithms should run in a reasonable amount of time ($< 15s$) for arena1.txt and arena2.txt, we do not expect Iterative Deepening A-Star Search to run quickly on arena3.txt. **Make sure your code passes at least these test cases and follows our formatting exactly.**

Additionally, we highly encourage you to use your imagination to create a cool and exciting maze of your own to test your code. If you've created a maze you're especially proud of, you may post it publicly on Ed as a reply to the Ed post announcing that coding HW1 has been released. Ensure you embed your maze as a .txt file. **We may use some mazes from Ed as grading test cases!**

4. Grading and Stress Tests

We will grade your project by running **additional test cases** on your code. We will only run the 4 functions which run your search algorithms. Don't worry about checking for malformed input mazes, including non-rectangular mazes or unsolvable mazes. **You will not be graded on the absolute values of your running time or RAM usage statistics.** The values of these statistics can vary widely depending on the machine. **However, we recommend that you take advantage of them in testing your code.** Try batch-running your algorithms on various inputs, and plotting your results on a graph to learn more about the space and time complexity characteristics of your code. Just because an algorithm provides the correct path to goal does not mean it has been implemented correctly.

VI. Before You Finish

- **Make sure** your code passes at least the submission test cases.
- **Make sure** your algorithms generate the correct solution for an arbitrary solvable mazes.
- **Make sure** your program always terminates without error, and in a reasonable amount of time. **You will receive zero points from the grader if your program fails to terminate. Running times of more than a minute or two may indicate a problem with your implementation.** If your implementation exceeds the time limit allocated, your grade may be incomplete.
- **Make sure** your program output follows the specified format exactly. You will not receive proper credit from the grader if your format differs from the provided examples above.

COMS 4701 Maze FAQs

Q. My search algorithm seems correct but is too slow. How can I reduce its running time?

A. Search algorithm is perhaps one of the best learning materials for computational complexity and Python's idiosyncrasies. There are three dos and don'ts:

1. Don't store possibly large data member such as solution path in search tree node class. Instead, rethink what operation should be fast.

   Explanation: Storing a path from the root node in each node class achieves O(1) lookup time at the expense of O(n) creation time. For example, if the current state is visited after 60,000 intermediate states, the current state has to allocate a list of 60,000 elements, and each of the children states have to allocate a list of 60,001 elements. This would soon use up physical memory, and typically your machine's Operating System kills the search process.

   A key observation is that in the case of search algorithm, path lookup operation is executed just once after search finishes. Thus, the lookup operation is fine to be slow. You might consider another data structure having O(n) lookup time for solution path but requiring O(1) operations during search.

2. Don't use a simple list to represent the frontier. Instead, design your Frontier class which works faster.

   Explanation: one major bottleneck of list, dequeue, or queue class in Python is that their membership testing operation is slow. The membership testing speed is critical for search algorithm because that operation is executed for every child state. Using such list-like data structures is a good first step, But more work might be required to improve runtimes.

   Note that pseudocode in lecture slides does not necessarily reflect implementation details. Rather, it conceptually shows the algorithm's inputs, processing orders, and outputs. One of your missions in this assignment is to make the "frontier" thing into a reality by using Python's "low-level" primitives.

3. Don't use O(n) operations when you have faster implementations.

   Explanation: If you set one O(n) operation under your "`for neighbor in neighbors`" loop, your code will likely exceed the grading time limit. In other words, your could code execute drastically faster if you fix one line of your code.

   For example, merging two sets and checking if an element is in the merged set is an expensive operation, while checking if an element is in one of two sets is a O(1) operation.

Q. Do I need to optimize my search algorithm as much as possible?

A. You don't need to squeeze your code's performance by fancy optimization techniques such as bit shifting or reducing the number of function calls (putting every operation in one function for reducing overhead of function calls). Most of your design choices are about choosing best data structures in terms of time/space complexity.