COMPSCI 270 **Instructor:** Kris Hauser

Homework 1 **Due:** 02/06/2017

1 Instructions

Read the assignment carefully, complete the programming tasks, and answer all of the written questions. Place all of your search code as indicated in *search.py*. Add your written answers as indicated in *hw1_answers.py* based on the question number listed for each question. Submit both files on Sakai.

2 Minecraft Maze

In this homework, you will implement search algorithms for solving a maze navigation problem in Minecraft.

3 Programming

The *hw1.zip* archive contains this file and three Python files: *hw1.py*, *search.py*, and *hw1_answers.py*.

3.1 Running the program

In order to run the search program, you must extract the provided Python files into a working directory of your choosing and copy the *MalmoPython.lib* and *MalmoPython.pyd* files from the Project Malmo *Python_Examples* directory into your working directory.

Additionally, you must ensure that the Project Malmo Minecraft client is already running, as you did in HW0. Once the client is running, you can start the program with one of the following commands:

- `python hw1.py bfs`
- `python hw1.py gs`

These commands run the search program using either BFS or Greedy Search, respectively, but they won't work until you have implemented these algorithms in Problems 2 and 3 below.

4 Problems

4.1 Define the search problem

Programming For this question, you will complete the definition of the search problem for the maze navigation task. In order to do so, you will need to complete the implementation of the *get_successors* method of the *MazeProblem* class in *search.py*, which returns a list of valid states which can be reached from the given state by taking a single action and a list of which action must

be taken to reach each corresponding successor. Fix the method so it returns the correct successors and actions.

The MazeProblem class has multiple methods which are described below.

1. *__init__*
    - Called when creating an instance of the class. Receives a grid which represents the maze. The grid is implemented as a list of lists of strings, where each element is a single character which indicates the type of block at the given location (*grid[i][j]* is the block at coordinates (i, j)). The types of blocks are:
        - "a" (air) - open space where the character will fall and thus fail the mission immediately
        - "d" (diamond) - regular maze blocks upon which the character can walk
        - "E" (emerald) - the block upon which the character starts
        - "R" (redstone) - the goal block
2. *get_start_state*
    - Returns the starting state.
3. *is_goal_state*
    - Accepts a state and returns True if the state is a goal state, otherwise returns False.
4. *get_successors*
    - Accepts a state and returns a list of its successor states and a list of the actions required to reach each successor.
5. *eval_heuristic*
    - Accepts a state and returns the value of a heuristic for that state.

The function *pretty_print_grid* in this file will print the grid to the console to help you visualize it. Note that the view in Minecraft is from the perspective of the character, which starts on the start state and faces down in this grid visualization.

The valid actions in this environment are:

- "n" - move one block north
- "s" - move one block south
- "e" - move one block east
- "w" - move one block west

Written Questions

4.1.1 Keeping in mind the permitted actions and block types, what are the requirements for a given state B to be a valid successor to a given state A?

4.1.2 What is the worst case branching factor for this environment? What is the size of the state space?

4.2 Implement Breadth-First Search

Programming In this problem, you will complete the implementation of BFS in the *breadth_first_search* function in *search.py*. This function receives a problem specification (like the MazeProblem you modified in question 1) as an argument and returns a "plan" which consists of a list of actions to take in order to move from the start state to the goal state.

As provided, BFS can take a very long time because it revisits states. How can you detect states which have already been visited and avoid them? Modify the *breadth_first_search* function so that states are not revisited.

Written Questions

4.2.1 How many nodes does BFS without VSD need to visit in the worst case, compared to BFS+VSD? Your answer should be a big-O expression that applies to all state spaces for this problem.

4.2.2 Compare two of the following possible methods of visited state detection: 1) a list of visited states, 2) a grid of Boolean values indicating whether a state has been visited or not, and 3) checking whether a new state is on any node on the path from the root to the node being expanded. Give the time and space complexity for adding a new state to the structure, as well as checking whether a state has been visited. Which method do you prefer, and why?

4.3 Implement Greedy Search

Programming For this problem, you will implement a heuristic and greedy best-first search (greedy search). Greedy search is a best-first search which evaluates nodes using a heuristic that estimates the cheapest path from a state to the goal.

Implement an appropriate heuristic for greedy search in the *eval_heuristic* method of the *MazeProblem* class in *search.py*. This method takes a state and a problem specification as arguments and returns an integer value of the heuristic evaluated for that state.

Implement greedy search by finishing the *greedy_search* function in *search.py*. Much like the BFS function, this function receives a problem specification as an argument and returns a list of actions to take in order to move from the start state to the goal state. You should use visited state detection in your implementation. If a path can't be found, it should return "NOPATH" as a string.

Written Questions

4.3.1 Is greedy search guaranteed to always find a path to the goal, if one exists? When it finds a path, is the path guaranteed to be optimal?

4.3.2 What function did you choose as your heuristic? Is it admissible? Is it consistent? Justify your answers.