Assignment 1: Sample World Planner

Due: May 25, 2023 at 11:59PM

1 Assignment 1: Sample World Planner Specification

1.1 Overview

For this assignment, you will write a program that solves robot path planning problems in Sample World, a simple grid environment that has some cells that we want the robot to sample. This problem is similar to tasks that a NASA rover or AUV might perform. Given a description of a world, your program will return a sequence of actions that move the robot around and perform a sampling operation. The robot can only move in the four cardinal directions. All actions have the same cost and the goal is to sample all marked locations (with a minimal cost plan, if possible). Your program will implement several different search algorithms to generate solutions.

Find the starter files and repository at `https://classroom.github.com/a/bS59SjW0`.

1.2 Goals

1. Practice interacting with Github
2. Practice writing Java/Python
3. Implement depth-first and uniform-cost search algorithms

1.3 Input Descriptions

Your program should read a world description from standard input with a form like:

```
4
3
@*__
__#*
__*#
```

Inputs all have the format:

1. The first line is the number of columns
2. The second line is the number of rows
3. The remaining lines are a description of the Sample World with the representation: **@** is the starting location of the robot, **#** is a blocked cell, ***** is a sample location, **_** is a blank cell.

Assume that all inputs are correctly formatted and solvable.

In addition to the world description input, your program should accept one command-line argument:

algorithm: one of **dfs** (depth-first search with cycle detection) or **ucs** (uniform cost search).

Not every world can be solved in a reasonable amount of time with every algorithm.

1.4 Actions

The robot can perform one of five actions from any given state: moving in one of the four cardinal directions or sampling its current location. It cannot move into a blocked cell or move off the board, and it can only perform a sample action if it is on a cell that was marked to sample. It cannot sample the same cell twice.

1.5 Output

Your program should only output the list of actions followed by two algorithm statistics:

Action list: A sequence of movements (U, D, L, R) and sample actions (S)

Statistics: The number of nodes that your algorithm generated and expanded, on separate lines

Example output for `small2` world (your nodes generated/expanded may not match exactly):

```
R
U
U
R
R
D
D
S
24 nodes generated
10 nodes expanded
```

1.6 Execution

You should write your code in one of two languages: Java or Python. You should name your source file that has the program's main method `SampleWorld`. One of the following options should invoke your program from the command line:

```
> python SampleWorld.py dfs < small1.txt
```

(Assuming the code has been compiled)

```
> java SampleWorld dfs < small1.txt
```

The `<` character redirects the file contents to stdin, rather than needing to copy-paste the file contents.

Also included is a utility program that, given a world and a solution, verifies the solution:

```
> java -jar sw-validator.jar -f /path/to/small1.txt
```

This utility takes a filename with the -f flag, and reads in your solution from standard in. You should enter the printed solution without the statistics lines at the end. You might need to hit enter an extra time at the end to finish the validation.

2 Design Ideas

In your state representations, only include information that actions can change (robot location, sample locations). Each node that you create should include one state, and other details important to the node (with details like g values and parent nodes). Represent static parts of the world (grid size, blocked cells) separately, to save memory. There's no need to hold thousands of copies of a blocked cell. A search algorithm only needs an initial state, a state expansion function, and a goal state function in order to solve a problem instance.

Start your implementation with incremental testing: make sure you can read a world file and store it internally before you consider valid actions before you consider generating nodes before you consider search algorithms. Print out each state during testing to better understand how your program is progressing.

Detecting repeated states is very important for efficiency – use good data structures for your state representation, open list, and closed list. Utilize your language's built-in data structures rather than building your own from scratch. Closed lists are often held in hash tables for fast lookups, and open lists are held in heaps for fast insertion and extraction. Make sure your hash table uses a good hash function to detect duplicate states.

3 Submission

Submit the code for your solution along with a writeup that answers the following questions

1. Describe choices you made in your code that you feel are important. Mention any specific aspects of your implementation that might be interesting as I evaluate your program.
2. Which of your implemented algorithms are admissible?
3. How large is the Sample World state space, as a function of width, height, and number of sample locations?
4. What suggestions do you have for improving this assignment?

For submission, commit/push your updates to your Github repository. I automatically have access to the repository, so if you see your updated files on Github, I have access to them too.

4 Evaluation

Here is how each part of the assignment contributes to your grade.

Partial credit will be given for solutions that are partially implemented.

- +15%: Your writeup is clear and descriptive.
- +15%: Your program reads in the start state and stores it in a well-structured way
- +35%: Depth-first search is implemented efficiently and correctly
- +35%: Uniform cost search is implemented efficiently and correctly