Assignment 1: Vacuum Robot Planner

CS 730/830, Spring 2025

Due at **11pm on Mon, Jan 27**

Overview

You will write a program to solve robot planning problems in Vacuum World, a simple grid environment where some cells are contaminated with dirt. This problem is fundamentally similar to game AI path-finding and NASA rover surface operations. Given a world description, your program will return a sequence of actions that will move the robot around and vacuum up all the dirt. The vacuum robot can only move in the four cardinal directions. All actions have the same cost and the objective is to achieve a clean world with a plan of minimum cost.

Input

Your program should read a world description from standard input, such as:

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

The format is:

1. The number of columns and the number of rows, on separate lines. Worlds might be hundreds or thousands of cells in each dimension.
2. Rows of characters, one character per cell and one world row per line of input. An underscore (`_`) means a blank cell, a hash (`#`) means a blocked cell that the robot cannot enter, an asterisk (`*`) means a dirty cell. An at-sign (`@`) is the starting position of the robot.

You can assume that the world is correctly formatted and solvable.

You should implement your planner as a single search over the state space of the system (robot and environment), from the initial state to a goal state. Your program should accept a command-line argument specifying the search algorithm to run, either `uniform-cost` or `depth-first`. Your depth-first search should use only linear memory (ie, no closed list), but should be complete in finite domains like vacuum world (ie, do cycle checking). Your uniform-cost search should use a closed list.

Output

Your program should write only the following to standard output: a list of actions (`N`, `S`, `E`, `W`, or `V` (vacuum), one per line) followed by the number of search nodes expanded and generated (one per line, in that order). For example:

```
katsura(1153)> ./run.sh uniform-cost < tiny-2.vw
E
V
W
N
N
V
E
E
S
V
85 nodes generated
39 nodes expanded
```

Execution

Read the submission instructions on the course web page for information on the required `make.sh` and `run.sh` scripts. Basically, if you are using a compiled language, you should supply a `make.sh` script that builds your code and the final executable should be named `run.sh`. If you are using an interpreted or JIT compiled language, your `make.sh` script can do nothing and your `run.sh` should load and run your program with the arguments supplied to the script.

On the course web page, we supply some utility programs and input files. Most of the programs we distribute in this class will tell you their command-line arguments if you run them with the `-help` option.

`*.vw` a few example benchmark problems.

`make-vw` takes four command line arguments (the number of rows, number of columns, percentage of blocked cells (eg, `0.15`), and number of dirty cells) and prints a new random world to standard output. There are some optional arguments too — try `-help`.

`vacuum-plan-reference` a sample solution.

`vw-validator` runs your program, validates its output, and displays the solution plan (using digits to show progression from start to end). For example:

```
vw-validator -time 5 -- ./run.sh uniform-cost < tiny-1.vw
```

will run uniform-cost search on `tiny-1.vw` and a 5 second time limit (note the `./` at the start of the executable name).

As the example illustrates, the validator takes two groups of arguments, separated by `--`. The first group are optional flags to control the validator's behavior: `-novis` turns off display of the solution plan (useful for large problems), `-time` *float* specifies a time bound in seconds (default `60`), and `-mem` *float* specifies a memory bound in gigabytes (default `1`). The second group of arguments are the program to run and the arguments to pass to it. The validator expects a world on standard input and will pass it to your program.

Submission

Electronically submit your solution using the instructions on the course web page, including your source code as well as a transcript of your program running with the validator, and a PDF file called `written.pdf` with your brief written answers to the following questions:

1. Describe any implementation choices you made that you felt were important. Clearly explain any aspects of your program that aren't working. Mention anything else that we should know when evaluating your work.
2. What is the size of the state space for this problem?
3. What is the time and space complexity of each algorithm you implemented? Which algorithms are admissible?
4. Provide empirical results supporting your answers to the previous question.
5. What suggestions do you have for improving this assignment in the future?

Graduate Extensions

Those in 830 must extend the base assignment to accept as an optional last command-line argument the flag `-battery` that indicates, when present, that you must model the battery usage of the vacuum robot. Special cells (marked `:`) are recharging stations and a special action (`R`) completely recharges the battery when in those cells. Each action (except `R`) uses one unit of power. The robot starts with a full charge, which is defined to be exactly enough to move from one corner of the world to the opposite one, vacuum the cell, and move back to the original location, assuming no obstacles.

Evaluation

Grading is done with an automated script, so be sure you have one algorithm working before starting the next, so that we can give you some credit even if you don't implement both.

Rough guide to grading:

0 nothing

1 something but basically nothing

2 write up is correct but no code works

5 one of the two algorithms works and half of the writeup is clear and correct.

8 Very slow on large problems, but code looks nice. Write-up has an error in reasoning.

10 Everything runs smoothly and correctly. The implementation roughly on par with the reference solution even for large problems. Write-up is clear and convincing, with no errors.

Design Suggestions

Make the state representation (stuff like current location, remaining dirt) separate from the node representation (stuff like g value, parent pointer). To minimize copying time and memory use, you probably want to make both of those separate from the static problem information (eg, which cells are blocked) that doesn't change from node to node. Ideally, these should all be independent from the search algorithms (we will be using the vacuum world again with different search algorithms in assignment 2, and the search algorithms with a different state representation in assignment 8). One way to do this is to have each search algorithm take an initial state, state expansion function, and goal test function and do all its work in terms of those functions, with no other knowledge about the problem being solved.

Start by making sure you can read in and print out a world state. Use very small examples until you are sure your basic functions work correctly.

Detecting and handling duplicate states is very important in this domain.

Use appropriate data structures for the state representation, the open list, and the closed list. Consider such options as: linked list, array, hash table, binary heap.