Lab 2 - State Space Model – Artificial Intelligence

Teacher: Stefán Ólafsson

January 16, 2023

Note: For this lab, you can work together in teams of up to 3 students, although ideal is probably 2. You can use Piazza or Discord to find team mates and discuss problems. You will need Python (>= 3.8) and a text editor or IDE.

Time Estimate

4 hours in addition to the time spend in the lab class assuming you attended/watched the lectures or read the relevant parts of book chapters 2 and 3. In addition, you should understand how a hash map works.

Problem Description

Develop a state space model for the vacuum cleaner agent.

The environment is very similar to the one in the first lab, with the difference that *the agent now has complete information about the state of the environment.* That is, the agent knows where it is initially, how big the environment is and which cells are dirty. The goal of the agent is to clean all dirty cells, return to the initial location and turn the robot off while minimizing the cost of all actions that were executed.

The environment is a rectangular grid of cells each of which may contain dirt. The agent is located in this grid and facing in one of the four directions: north, south, east or west.

The agent can execute the following actions:

- TURN_ON: This action initialises the robot and has to be executed first.
- TURN_RIGHT, TURN_LEFT: lets the robot rotate 90° clockwise/counter-clockwise
- GO: lets the agent attempt to move to the next cell in the direction it is currently facing. If the agent is facing a wall, the action has no effect.
- SUCK: suck the dirt in the current cell. If the current cell is already clean, the action has no effect.
- TURN_OFF: turns the robot off. Once turned off, it can only be turned on again after emptying the dust-container manually.

Initially, TURN_ON is the only possible action. Once the agent is turned on, all actions except TURN_ON are possible. However, some actions may not be useful in all situations.

The agent's actions have the following costs:

- 1 + 50*D, if you TURN_OFF the robot in the home location and there are D dirty cells left
- 100 + 50*D, if you TURN_OFF the robot, but not in the home location and there are D dirty cells left
- 5 for SUCK, if the current location of the robot does not contain dirt
- 1 for SUCK, if the current location of the robot contains dirt
- 1 for all other actions

Tasks

Ultimately, the goal is to implement an agent that can act in this environment optimally. You will solve this problem using some search algorithm in lab3. For this assignment your task consists of implementing the state space model for the agent, such that you can use search later to find a solution to the problem.

1. (10 points) Many environments have the property that there are several paths that lead to the same state. Which of the search algorithms that we covered in class (breadth-first, depth-first and uniform-cost search) benefit from detecting repeated states? For each algorithm give a short explanation how detecting repeated states can make it better (is it faster, use less memory, etc?).

2. (10 points) Which data structure is typically used for detecting repeated states and why?

3. (10 points) Develop a model of the environment (on paper). What constitutes a state of the environment? What is a successor state resulting of executing an action in a certain state? Which action is legal under which conditions? What is the cost of the actions?

Maybe you can abstract from certain aspects of the environment to make the state space in your model smaller. Think about, which properties of the environment can change and which stay constant for a given environment. Only the properties that change should be part the state. Also think about which actions should (not) be possible in which situation.

4. (10 points) Assume the environment has width W, height H, D dirty spots. Estimate an upper bound on the size of the state space (number of possible states) in terms of W, H and D. Explain your estimate!

Implement the `expected_number_of_states` function in `environment.py` to return your estimate.

5. (50 points) The material for this lab contains code stubs for a model of the environment and states as well as some code that generates all reachable states from that model. Implement your model by implement the missing parts in `environment.py`.

You can test your model by executing `env_tester.py -s N` (where N is an integer). This will run a simulation of the environment starting at the initial state and executing a sequence of N random actions. It will print out the intermediate states and all legal actions in each state. Check whether the printed information is what you'd expect!

If course, you should feel free to write your own code in addition, to test your model, e.g., by writing unit tests.

Points are awarded mainly for correct implementation of the state space model and somewhat for efficiency (small states, fast state generation), and readability.

6. (10 points) Because the state space contains cycles, we need to be able to detect repeated states. Therefore, the `State` class needs to be hashable, that is, implement suitable __hash__() and __eq__() functions. Implement those functions.

You can run `env_tester.py` without the `-s` argument and it will try to generate all reachable states in the environment. For the given problem, this should be a finite number. The program will also check whether states hash correctly, that is, whether they can be used as keys into a dictionary. You should test this for different environment sizes. Width, height and number of dirt spots can be provided as additional arguments to `env_tester.py`.

Hints:

- To use a user defined object as key in a dictionary it needs to be hashable. The main requirements are that is has a `__hash__()` method that returns a value (integer) that **does not change during the lifetime of the object** and that two objects that are equal (`o1 == o2` is `True`) must have the same hash value.
- Technically, defining `__hash__()` to return 1 for all states satisfies this condition, but that is not a good hash function. (Why?)
- It is important that the hash value does not change during the lifetime of the object (Why?), which usually requires that the object itself is not changed. That is, the attributes of a State object that influence the hash value must not change, because that would change a state's hash value.
- Doing this in Python is fairly easy, if you use the hash function of builtin types as recommended by the documentation for object.__hash__(). Also, Stackoverflow has some more examples and discussion on the topic.

7. (**10 bonus points**)

Report and comment on the results you get when testing the hash function of your State class. How many hash collisions and hash index collisions do you get? Is this good or bad?

8. (**5 bonus points**)

Why is it important to have few hash collisions if hashing is used to detect repeated states?

Material

The main file of the program is called `env_tester.py`. This is the program you should execute. Running it without any parameters will try to create a 5x5 environment with 5 dirt spots, generate all reachable states and report on their number and how well they work as keys in a hash map. Run `env_tester.py -h` to learn about other command line options! There should be no reason to change that file unless you want to add your own tests.

The file you need to edit is called `environment.py`. It contains:

- a class `State`, meant to represent a state of the vacuum cleaner environment. You will have to add some attributes to that class and implement reasonable `__str__()`, `__hash__()`, and `__eq__()` methods.

- a class `Environment`, meant to represent one particular instance of a vacuum cleaner environment. You may have to add some attributes here. Also, this class has methods that implement the state space model: `get_initial_state()`, `get_legal_actions()`, `get_next_state()`, `get_cost()`, and `is_goal_state()`.
- a function `expected_number_of_states()`, that is supposed to return a reasonable estimate of the number of possible states.

All places that might need changes are marked with `# TODO:`

1 Handing In

Hand in a PDF file with the answers to the questions and a zip file with your code on Canvas. The contents of the zip file should have the following structure:

```
lab2
lab2/environment.py
lab2/env_tester.py
lab2/pysize.py
```