Lab 3 - Single-Agent Search – Artificial Intelligence

Teacher: Stephan Schiffel

January 22, 2022

You can work in groups of up to 3 students to do this assignment (2 is ideal). Use Piazza or Discord, if you have any questions or problems with the assignment.

Time Estimate

3 hours per student in addition to the time spend in the labs, assuming you did labs 1 and 2 and attended the lectures or worked through chapters 2 and 3 in the book.

Problem Description

Find a good plan for the vacuum cleaner agent. It is a rectangular grid of cells each of which may contain dirt or an obstacles. The agent is located in this grid and facing in one of the four directions: north, south, east or west.

Here is a link to an example of what the environment might look like.

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.
- SUCK: suck the dirt in the current cell
- TURN_OFF: turns the robot off. Once turned off, it can only be turned on again after emptying the dust-container manually.

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

Your 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

1. (5 points) The material for this lab contains code for a model of the environment and states. Read and understand that code. What constitutes a state of the environment? Which information about the environment is not in the state and why?
2. (10 points) Assuming the environment has width W, length L, D dirty spots and relatively few obstacles compared to accessible spaces. Estimate the size of the state space (number of possible states) in terms of W, L and D. Explain your estimate!
3. (15 points) Assess the following search algorithms wrt. their completeness, optimality, space and time complexity in the given environment: Depth-First Search, Breadth-First Search, Uniform-Cost Search, A* Search. If one of the algorithms is not complete, how could you fix it?
4. **(5 bonus points)** Compare the estimates for the size of the search tree and size of the state space. Where does this difference come from? How could you exploit this difference to improve the efficiency of the search?
5. **(10 bonus points)** The code contains a heuristic function that estimates the number of steps needed to finish the task from a given state. Is the heuristic admissible? Is it consistent? Explain.
6. (50 points) Implement A* Search to solve the problem (essentially, complete the function in `AStarSearch.java` or `search.py` at the point with the `TODO` comment). You can use the provided heuristic, or implement your own. Keep track of
   - the number of state expansions
   - the maximum size of the frontier (measured in number of nodes)
   - the cost of the found solution (path cost)
   - the run-time of the search (in seconds)

   Run the agent on all the given environments and record the results. Report on the results and compare with your estimated complexity.
7. (15 points) Implement detection of revisited states in A* Search. Make sure your heuristic fulfills the necessary requirements for that change in case you use your own heuristic.
8. (5 points) Re-run the experiments on all the environments using detection of revisited states. In addition to size of the frontier, you should now also keep track of the size of the closed list (set of visited states). Compare the results to the ones you got without detection of revisited states.

   How much effort did you save in terms of time and memory? Did you find better solutions than before?

9. **(25 bonus points)** Think of a better (admissible/consistent) heuristic function for estimating the remaining cost given an arbitrary state of the environment. Make sure, that your heuristics is really admissible/consistent. Implement this heuristic and re-run the experiments with all environments. Compare the results!

Material

The code for the lab can be found on Canvas.

The files in the archive are similar to those in the first lab.

The archive contains code for implementing an agent in the src directory. The agent is actually a server process which listens on some port and waits for the real robot or a simulator to send a message. It will then reply with the next action the robot is supposed to execute.

The zip file also contains the description of some example environments (vacuumcleaner*.gdl) and a simulator (vacuumcleanersim.jar) similar to lab1. To test your agent:

1. If you use the Java project, run the "Main" class in the project. If you added your own agent class, make sure that it is used in the main method of Main.java. You can also execute the `ant run` on the command line, if you have Ant installed. The output of the agent should say "NanoHTTPD is listening on port 4001", which indicates that your agent is ready and waiting for messages to arrive on the specified port.

2. If you use the Python code, run `gameplayer.py`. The output should say something along the lines of "VacuumCleanerAgent is listening on port 4001 ...", which indicates that your agent is ready and waiting for messages to arrive on the specified port.

3. Start the simulator (execute vacuumcleanersim.jar with either double-click or using the command `java -jar vacuumcleanersim.jar` on the command line).

4. Setup the simulator as shown in Figure 1, but choose one of the gdl files included in this lab. Change Startclock to some time (measured in seconds) high enough to finish the search (e.g., 3600 would be 1 hour). If your agent is not done with the search before startclock is up, you will probably see timeout errors in the log of the simulator.

5. Now push the "Start" button in the simulator and your agent should get some messages and reply with the actions it wants to execute. At the end, the output of the simulator tells you how many points your agent got: "Game over! results: 0". Those points correspond more or less with negated costs. That is, more points means the solution had lower costs. You can pretty much ignore those points and simply look at the cost your agent prints out when it is done planning.

6. If the output of the simulator contains any line starting with "SEVERE", something is wrong. The three most common problems are the network connection (e.g., due to a firewall) between the simulator and the agent, the agent sending illegal moves or the agent not responding within the set time constraints (e.g., it takes too long to find a plan).

Alternatively to the graphical version of the simulator, there is a command line version you can use. This might be more convenient to run multiple environments in a row or otherwise automate the testing. Run it as follows:

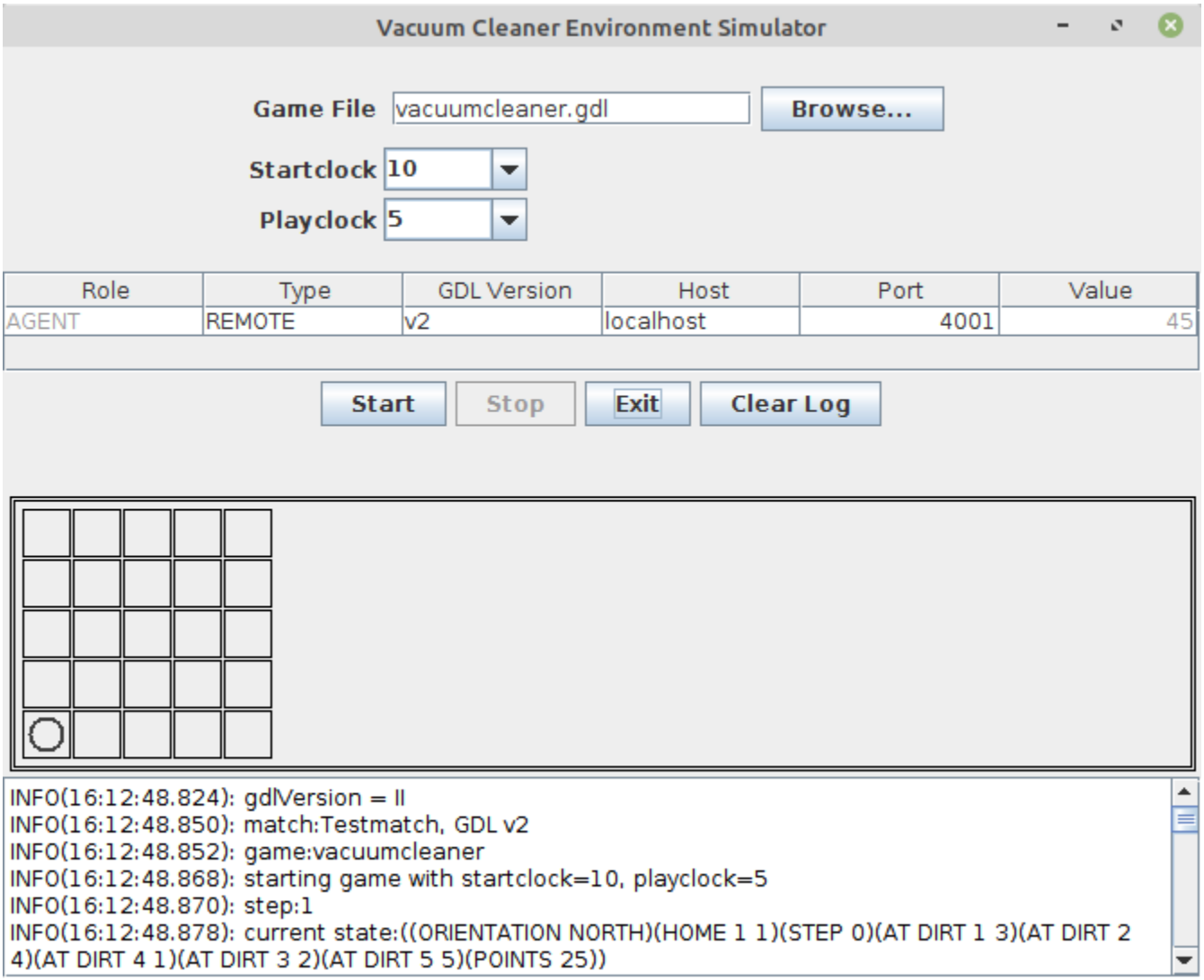

Figure 1: *Game Controller Settings*

```
java -jar simplegamecontroller.jar vacuumcleaner_obstacles_1.gdl 3600 5 localhost 4001
```

For this to work, you need to start the agent first. Change the name of the gdl file and the number for the startclock (3600 in the example) as needed.

Hints

General:

- Do the space and time-complexity estimates and look at the size of each of the environments. What do you expect A* to do with the environments? Which ones can it realistically solve? Start with the easier ones.
- The search might run for a while. It is very helpful to see some signs of progress. I recommend printing out something about what is happening in the search occasionally. For example, you could print out the number of expanded states, evaluation of the current node, size of the frontier, etc. every 100 or 1000 state expansions. This way, you see at least that the search is still running and not stuck in some endless loop somewhere.

For developing a (better) admissible heuristics:

- Remember: The cost of the optimal solution of a relaxation of the problem is an admissible heuristics. How can you relax the problem, such that you instantly (or quickly) know the cost of the optimal solution?
- The maximum of two admissible heuristics is again admissible. The same holds for consistent heuristics. So, if you have two admissible heuristics you can simply take the maximum value of both and you will get an admissible heuristics that is at least as good.
- Does this problem remind you of some other well-known problem? Maybe someone already figured out good heuristics for that.
- You may use some third-party library to compute the heuristics, if necessary.

Handing in

Hand in a PDF file with a report that answers all the questions above and a ZIP archive containing your code. The files in the ZIP archive must have the following structure:

```
lab3
lab3/build.xml
lab3/python_src
lab3/python_src/agent.py
...
lab3/src
lab3/src/Agent.java
lab3/src/GamePlayer.java
lab3/src/Main.java
lab3/src/NanoHTTPD.java
```

```
lab3/src/RandomAgent.java
...
```

If you programmed your agent in Python, you can leave out the `src` folder. Similarly, if you used Java, leave out the `python_src` folder.

Additional source files (if any) should be added at the appropriate place.