# Programming Assignment 1 - Search – Artificial Intelligence

Teacher: Stephan Schiffel

January 15, 2019

Use the Piazza page, if you have any questions or problems with the assignment. Start early, so you still have time to ask in case of problems!

This assignment is supposed to be done in a group of 2-3 students; up to 4 is allowed. Note that everyone in the group needs to understand the whole solution even though you should distribute the implementation work.

## Time Estimate

13 hours per student in addition to the time spend in the labs, assuming you work in groups of 2-3 students, did lab 1 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. The environment is very similar to the one in the first lab: It is a rectangular grid of cells each of which may contain dirt. Only now the grid may also contain 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.

However, the agent now 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

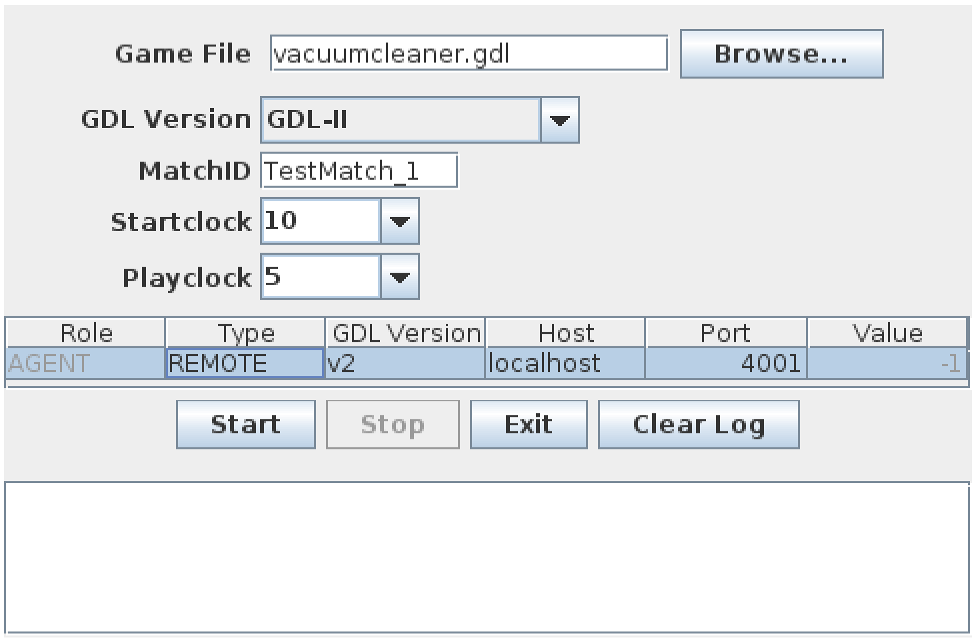
- 100 + 50*D, if you TURN_OFF the robot, but not in the home location and there are D dirty cells left
- 5 for SUCK, if the current location of the robot does not contain dirt
- 1 for SUCK, if the current location of the robot contains dirt
- 1 for all other actions

## Tasks

1. Develop a model of the environment. 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? Maybe you can abstract from certain aspects of the environment to make the state space smaller.
2. Implement the model:
   - Create a data structure for states.
   - Implement methods to compute the legal moves in a state and the successor state resulting of executing an action.
   - Implement the goal test, i.e., a method telling you whether a certain state fulfils all goal conditions.

   After this part you should have an agent that knows the initial state, can generate possible moves in a state as well as think about what the next state would look like. It is a good idea to test whether this state space model is implemented correctly, e.g., by implementing some unit tests.
3. Estimate the size of the state space assuming the environment has width W, length L and D dirty spots. Explain your estimate!
4. Assess the following blind search algorithms wrt. their completeness, optimality, space and time complexity in the given environment: Depth-First Search, Breadth-First Search, Uniform-Cost Search. If one of the algorithms is not complete, how could you fix it? Note: Do this step before you implement the algorithms, so you know what to expect when you run the algorithms. Otherwise you might be very surprised.
5. Implement the three algorithms and make sure to keep track of the number of state expansions and the maximum size of the frontier. Run all the given environments with the three algorithms. Report on the results and compare the results of the different algorithms wrt.
   - number of state expansions (time complexity)
   - maximum size of the frontier (space complexity)
   - quality (cost) of the found solution
   - computation time (in seconds) for finding a solution.

   Do these results support your previous assessment of the algorithms? Explain!



| | |
|---|---|
| Game File | vacuumcleaner.gdl |
| GDL Version | GDL-II |
| MatchID | TestMatch_1 |
| Startclock | 10 |
| Playclock | 5 |

| Role | Type | GDL Version | Host | Port | Value |
|---|---|---|---|---|---|
| AGENT | REMOTE | v2 | localhost | 4001 | -1 |

**Figure 1:** *Game Controller Settings*

6. Think of a good (admissible) heuristic function for estimating the remaining cost given an arbitrary state of the environment. Make sure, that your heuristics is really admissible! Shortly describe your heuristics and explain why it is admissible.

7. Implement A*-Search with an evaluation function that uses your heuristics. Run your agent with all the given environments, report on the results and compare with the results of the blind search methods.

8. Post the results (number of state expansions, cost of the found solution, search time) on Piazza, to see how well you are doing compared to the other students.

9. Optionally (up to 10% bonus points): Implement detection of revisited states in A*-Search. Re-run the experiments and comment on the difference in the results.

10. Try to improve your heuristics while keeping it admissible, but make sure that it does not increase the overall runtime of the search algorithm. You can easily create a heuristics that tells you the optimal costs but is as expensive as a blind search. Try to avoid this.

## Material

The Java project 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 (gamecontroller-gui.jar). To test your agent:

1. 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. Start the simulator (execute gamecontroller-gui.jar with either double-click or using the command `java -jar gamecontroller-gui.jar` on the command line).
3. Setup the simulator as shown in Figure 1. 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.
4. 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.
5. If the output of the simulator contains any line starting with “SEVERE”, something is wrong. The two most common problems are the network connection (e.g., due to a firewall) between the simulator and the agent or the agent sending illegal moves.

Alternatively to the graphical version of the simulator, there is a command line version you can use. Run it as follows:

```
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

For implementing your agent:

- Add a new class that implements the “Agent” interface. Look at RandomAgent.java to see how this is done.
- You have to implement the methods “init” and “nextAction”. “init” will be called once at the start and should be used to store the information about the environment and run the search. “nextAction” gets a collection of percepts as input (which you do not actually need here) and has to return the next action the agent is supposed to execute. So, if you found a solution in “init” you have to store it somewhere and then execute it step by step in “nextAction”.
- Implement a class “State” that contains all the information about a state that you need to keep track of. The State class should have a method that returns a list of all moves that are legal in the state and a method that takes a move and returns the state that results from executing the move.

- After implementing the state space model, you should have an agent that knows the initial state, can generate possible moves in a state as well as think about what the next state would look like. It is a good idea to test whether this model is implemented correctly at this point. E.g., you could write a few lines of code that generates a short sequence of legal moves and prints some information about the resulting states to check whether all the state transitions are correct, all the moves are legal, no legal moves are missing, etc. Finding problems in your state space model when you are working on the search algorithms is difficult and time-consuming.
- Distinguish between nodes of the search tree and states! Each node has an associated state, but there may be several nodes that have the same state associated. In addition to a state, a node contains a reference to its parent (unless it is the root node), the move that was taken to get to the node and (if necessary) the path cost associated for getting from the root node to the node.
- Keep the code for all the algorithms and make it easy to switch between them (e.g., with a command line parameter or by changing a single line of code).

For developing an 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 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.

## 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:

```
prog1
prog1/build.xml
prog1/src
prog1/src/Agent.java
prog1/src/GamePlayer.java
prog1/src/Main.java
prog1/src/NanoHTTPD.java
prog1/src/RandomAgent.java
```

Additional source files (like your agent class) should be added under the `prog1/src`.

## Grading

- 70% - implementation (correct implementation of the model, algorithms and heuristics, quality/readability of the code)

- 30% - report (answers to questions and report on results)

Bonus points:

- up to 5% for the agents that find the optimal solution for the most environments fastest
- up to 10% for (correctly) implementing detection of revisited states and reporting on those results (task 9)