CMPE 321 Artificial Intelligence

Fall 2018-19

Homework 2

Due date: 21/11/2018 23:59

Modern warehouses - especially the ones mentioned in the Industry 4.0 trend - utilize mobile robots to store and deliver goods efficiently. You may also see these warehouses as automated factories if you want. In this assignment, you are going to develop a program that controls these warehouse robots using A* search. Similar to your first homework, you will use the OpenAI Gym (`http://gym.openai.com/`) for simulating the warehouse environment.

1 Single robot warehouse

The warehouse is composed of grid cells. There can be obstacles for the robot in the warehouse. You may think obstacles as separators forming compartments in the warehouse or other machines like conveyor belts etc. You have one robot occupying a cell in the warehouse and it needs to go to a target cell to fetch a product or process a good in a machine. The robot can move up, down, left, and right (no direct diagonal movement). Additionally, the robot may wait at a time step. The robot must not hit either the walls of the warehouse or the obstacles in the warehouse. It should navigate to its target as quickly as possible.

The types of grid cells will be represented by characters.

- `a` robot's location
- `A` target cell for the robot
- `.` empty cell
- `*` obstacle

Here is an example warehouse map of size 8x8.

```
. . . . A . . .
. . . . . . . .
. . . . . . . .
. . * * * * . .
. . . . . * . .
. . . . a * . .
. . . . . . . .
. . . . . . . .
```

You will use the `gym-warehouse` environment package for developing your agent. You can find this package in the course Moodle page.

For actions you can use the following members of the environment (which are basically numbers). The Python object **env** is the Gym environment you need to create (in this case it is the `SingleRobotWarehouseEnv` environment from `gym-warehouse` package).

env.ACTION_LEFT 0
env.ACTION_DOWN 1
env.ACTION_RIGHT 2
env.ACTION_UP 3
env.ACTION_WAIT 4

For sensory information, you can use the following method of the environment. **env.look()** method returns the warehouse the agent sees as a 2-dimensional array of characters representing the cells. For example, you can get the following output when you call **env.look()** for the mentioned example warehouse.

```
>>> env.look()
[['.', '.', '.', '.', 'A', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '*', '*', '*', '*', '.', '.'],
 ['.', '.', '.', '.', '.', '*', '.', '.'],
 ['.', '.', '.', '.', 'a', '*', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.'],
 ['.', '.', '.', '.', '.', '.', '.', '.']]
>>>
```

Your robot can perform an action using the **env.step()** method which takes the action as an argument. Here is an example usage of the method.

You can find a program for a dummy agent that you can build on.

Tasks:

1. Develop a program that controls the warehouse robot using **A* search**.

The following tasks are related to the report that you must submit.

2. Which heuristic function have you used for the search? Discuss whether it is admissible or not.
3. What is the branching factor (b) of this task?
4. Describe your program concisely and clearly. Do not put all the code to the report as a description. You can list code segments from your agent program (or some respective pseudocode) and explain what it does.

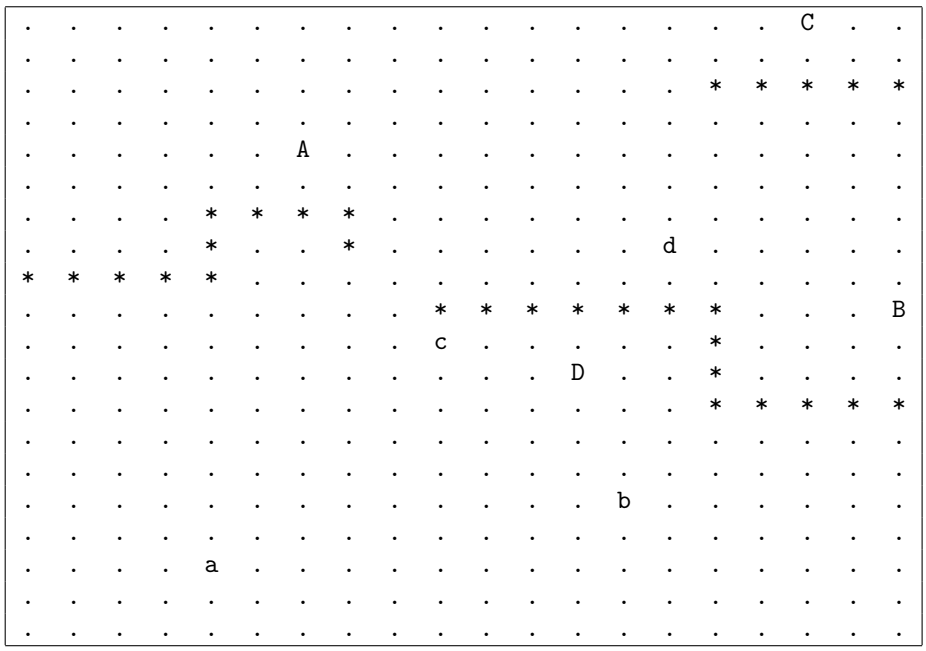
2 Multiple robot warehouse

You have the similar warehouse as mentioned in the previous part. However, this time you have multiple warehouse robots that can work simulatenously. Each robot has a unique target cell that it needs to arrive to. A robot must not bump into another robot. Additionally, two robots must not change their cells at the same time step (i.e., for example, if robots a and b are in cells (x, y)

and $(x, y+1)$*, respectively at time* t*, they cannot pass through each other and find themselves in cells* $(x, y+1)$ *and* (x, y)*, respectively at time* $t+1$*).*

In addition to characters used in the previous part, robots are represented by small letters (e.g., `a, b, c, ...`) and target of each robot is represented by the respective capital letter (e.g., `A, B, C, ...`).

Here is an example warehouse map of size 20x20.



You will use the `MultiRobotWarehouseEnv` environment from same package mentioned in the first part.

The **env.step()** method is obviously different than the single robot warehouse environment. Instead of one action, it expects a list of actions composed of exactly one action for each robot. For example, when you call env.step([env.ACTION_UP, env.ACTION_UP, env.ACTION_WAIT, env.ACTION_LEFT]) the robots `a,b,c,` and `d` performs up, up, wait, and left actions (robot `c` does not move), respectively.

Although the original A* search algorithm is designed for single agent, we will use an algorihtm based on a modified version of A* for solving the problem in a multiple robot warehouse domain. The following pseudocode depicts this algorithm. Basically, the algorithm iterates over each robot one by one (i.e., r below) in the order given in the instance (i.e., alphabetical order starting from a) to find a path for the robot that takes it to its target. To this end, the algorithm uses modified version of A* that considers the paths found so far for the previously iterated robots (*allpaths*) in order to avoid collisions. When there is a valid path ($path_r$) for the robot that does not contradict with the paths of previous robots, the resulting path is added to *allpaths* structure and the iteration continues with the next robot. If there is no collision free path for the current robot, the algorithm stops with failure to find a global path for all the robots.

$allpaths \leftarrow \emptyset$;
for $r \in \{a, b, \ldots\}$ **do**
    $path_r \leftarrow findPathWithModifiedA^*(r, allpaths)$;
    if $path_r$ *is valid* **then**
        add $path_r$ to $allpaths$;
    else
        return false;

The following is a small warehouse instance that explains the behaviour of the above algorithm. The algorithm first finds a path for the robot a, which can be shown as $(2, 0, 0)$, $(1, 0, 1)$, $(0, 0, 2)$, $(0, 1, 3)$, $(0, 2, 4)$. The third argument in each tuple is the time step. For instance, at time step 1 robot a is in cell $(1, 0)$. Next, the modified A* search finds the following path for the robot b: $(1, 2, 0)$, $(2, 2, 1)$, $(2, 1, 2)$, $(2, 0, 3)$, $(1, 0, 4)$, $(0, 0, 5)$. Note that it does not generate the path using the top portion of the warehouse, since it would cause a collision considering the already found path for robot a.

```
B  .  A
.  *  b
a  .  .
```

Tasks:

1. Develop a program that implements the above algorithm to control the warehouse robots.

The following tasks are related to the report that you must submit.

2. Describe your program concisely and clearly. Do not put all the code to the report as a description. You can list code segments from your agent program (or some respective pseudocode) and explain what it does.
3. Is this algorithm complete? Discuss.
4. Is this algorithm optimal? Discuss.
5. Consider that you have solved the problem by using the classical A* search algorithm. You see the problem has one meta-agent (or meta-robot) and actions are composite actions that have components for each agent. Compare the properties of this algorithm w.r.t. the one you have implemented in terms of time complexity, completeness, and optimality?

Implementation Language

You must use either Python or Java as the implementation language. Python is the suggested one due to its simplicity.

In case you choose Python, you must submit `singlerobot.py` and `multirobot.py` files. Templates for these files will be included in the homework package in Moodle. Your program will be tested by using the following command:

```
$ python singlerobot.py
...
$ python multirobot.py
```

In case you choose Java, you have to use XMLRPC protocol. Download the Apache XMLRPC Java package version 3.1.3 from `http://archive.apache.org/dist/ws/xmlrpc/binaries/`. You must use the `SingleRobot.java` and `MultiRobot.java` template files to implement the homework.

You need to first start the XMLRPC server using the command:

```
$ python singlerobot_xmlrpc_server.py
..OR..
$ python multirobot_xmlrpc_server.py
```

Then, in another terminal run your respective Java program.

Submission

Each person must submit **his or her own work**.

For a Python implementation, you need to submit your Python files `singlerobot.py` and `multirobot.py` using the course Moodle page. If there are source files that these agent program files depend on, you should also submit them.

For a Java implementation, you need to submit your Java source files `SingleRobot.java` and `MultiRobot.java` using the course Moodle page. Additionally, you must compile your Java programs and form Jar files `SingleRobot.jar` and `MultiRobot.jar`. For evaluation, your Jar files will be used.

You are **not allowed** to use special packages (e.g., algorithmic packages for searching etc.). You can only used modules from the standard Python and Java libraries.

Your program will be evaluated under Linux.

You may be asked for a demo session.

The report must be one document in **PDF** format. It should cover report related tasks of all 2 parts with a clear structure.

The final submission file must be either a **tarball** or a **Zip** archive. Do not submit in other archive formats like rar, 7zip, etc.

Your submission will be graded w.r.t. the maximum points calculated according to the following formula: $100 - (2^{NumOfLateDays} \times 5)$.