CS 301 **Project Assignment 4** Wednesday, October 18, 2017

To Deliver: 1. Release 4.0 in your SVN repository

To Download: —
Hard & final submission deadline: Monday, October 30, at 11.59 PM

Learning Objectives:
- Experience in test-driven design and limitations of testing when it comes to randomized algorithms and graphics.
- Using inheritance to share code across classes that solve same problems in different ways
- Implementing different versions of algorithms in an object oriented manner
- Learn to work with exceptions

Motivation/Background information

There are many different ways to escape from a maze. In this project, we want to implement such algorithms for the robot whose functionality is specified by the given Robot.java interface which you implemented in Project 3. The interface follows what a Lego robot may provide for as sensors to recognize distant objects (here walls) and the ability to move forward and to turn on the spot, as in this video for example (link).
We also want to compare algorithms in terms of their energy efficiency to operate the robot. So there is a some energy consumption being involved in any movement and sensor operation and the robot draws energy from its battery on its journey.

Algorithms: We want to consider three different Robot driver algorithms:

1. **Wizard**: as there is little magic in algorithms, the wizard is in fact a cheater who uses the information of the distance object (handed to it via setDistance method) to find the exit by looking for a neighbor that is closer to the exit. (Hint: you find similar functionality implemented in the MazeContainer.getNeighborCloserToExit() to support drawing the yellow line that shows the path to the exit).The wizard is intended to work as a baseline algorithm to see how the most efficient algorithm can perform in terms of energy consumption and path length. It is also a good candidate for testing a maze, a robot implementation …
2. **WallFollower**: the wall follower is a classic solution technique. The robot needs a distance sensor at the front and at one side (here: pick left) to perform. It follows the wall on its left hand side. Warning: Think how the driver's focus on its environment may have an impact on properties of this classic solution algorithm (termination, correctness). This is why we have the isAtExit methods.
3. **Pledge**'s algorithm: this is a refined wall follower that is able to run around and leave an obstacle. It picks a random direction as its main direction and applies wall following when it hits an obstacle. It counts left (-1) and right (+1) turns and when the total becomes zero, it is able to leave an obstacle following its main direction again. I found this animation helpful (link) as well as a number of youtube videos (link and link). I found the wikipedia page only useful once I understood the algorithm. Understanding the wall follower algorithm is a prerequisite.
4. **Optional** for BONUS points: **Explorer**: the explorer is inspired by Robert Frost's poem and takes the road less traveled (http://www.poetryfoundation.org/poem/173536). It is a simple

random search algorithm that retains a memory where it has been before, i.e., it counts how often it has been on a position (x,y). It operates by 3 rules:

4.1. If outside of a room, it 1) checks its options by asking its available distance sensors, 2) picks a direction to the adjacent cell least traveled (ties are resolved by throwing the dice, i.e., randomization) 3) it rotates if necessary and moves a step into the chosen direction.
4.2. If the explorer gets into a room, it scans it to find all possible doors, picks the door least used (ties are resolved by randomization) and then runs for the door to leave the room.
4.3. Of course, if it is at the exit or it can see the exit, it goes for it.

For a nice summary of Maze Solving Algorithms click here. This Wikipedia page is also a good start. There are also helpful Youtube videos to illustrate algorithms.

Graphical User Interface (GUI): We also need to adjust the maze application class to provide us with a user interface such that

1. we can select a robot driver algorithm (plus one option for a manual operation).
2. we can select a maze generation algorithm (DFS, Prim, Eller)
3. we can select the skill level (0-15)
4. and we can start the generation once all selections have been made.

Hint: check for Buttons and Combo Boxes (http://docs.oracle.com/javase/tutorial/uiswing/components/combobox.html). The RobotDriver will basically do the same thing as the manual driver implemented in Project 3 but this time in a fully automated way (uses drive2exit).

Same as before: The robot configuration we will consider at this point is the following:

- It has a room sensor and has 4 distance sensors.
- It can move forward and can rotate.
- The initial battery level is 3000.
- The energy costs are:
  - Distance sensing in one direction: 1
  - Rotating by 90 degrees (left or right): 3
  - Moving forward one cell: 5

Design decision

To retain a reasonably uniform design, I would like to fix the following naming conventions for classes:

- Classes that implement the RobotDriver interface for the different algorithms are named: Wizard.java, WallFollower.java, Pledge.java, (optional: Explorer.java).
- Provide each class with at least one constructor that has no parameters and operates a robot with the default configuration we consider (missing settings for fields rely on call appropriate set methods).
- Classes that implement test cases for each class have the class name followed by suffix Test, e.g. WizardTest.java for the Wizard.java class.

Task list:

1. **Design:**
   1. Perform a class design with CRC cards.
   2. Note: there are several reasonable designs!

    1. Consider the use of a software design pattern (we will look into these the coming week, so this is optional). Hint: for the RobotDriver consider the use of an abstract driver following the Template Pattern.
    3. Put your crc design in writing, simply list each class, its responsibilities and collaborators. Add this text to your code base under a folder reports as P4.pdf.
2. **Implement the Robot Driver algorithms.**
    1. Use the pseudocode programming process, start with adding the content of the CRC cards as comments to your class. Sketch out the algorithms in pseudocode first.
    2. Review your pseudocode, care for special situations like rooms.
    3. You can use as many additional other classes, methods as you like.
    4. For testing: ask yourself what a correct robot driver implementation distinguishes from a faulty one. Write test cases first!
    5. Challenges
        1. Complexity of search algorithm
        2. Integration of automated driver into overall maze codebase.
3. **Implement GUI extension**
    1. Add functionality to the UI on the title page to allow for a user
        1. to select a robot driver algorithm (suggestion: combo box)
        2. to select the size of the maze, the skill level (suggestion: combo box)
        3. to select the maze generation algorithm (suggestion: combo box)
        4. and to start. (suggestion: start button)
4. **SVN**
    1. Update your subversion repository whenever you made progress and got a particular feature or method going. As before, the expectation is to have at least 10 revisions with comments in your svn repository for this project.
    2. When you have finished the project assignment, create a tag with a Release 4.0 before the due date. Include your junit test cases in your release. Keep source code and test code in separate directories (src and test).
    3. Have the CRC design description in your code base and release 4.0 under reports/P4.pdf.

Grading:
You will receive points for:

1) **Your SVN Repository**:
    1) your subversion repository is set up correctly and the grader can download your release4.0 from its tags folder.
    2) your project as at least new 10 revisions with comments since P3 that shows the progress you in several steps for this assignment
2) **Your Design:**
    1) You will receive points for your CRC class design (file: reports/P4.pdf).
3) **Your implementation:**
    1) You will receive points for implementing the various robot driver classes. Code is expected to have comments, at least a statement on top of each method (using the javadoc format with /** …. */).
    2) You will receive points for implementing test cases for the robot driver classes. Test cases are expected to come with a comment that specifies what is tested, what is the correct behavior and (if not obvious) why that behavior is correct.

3) You will receive points if test cases perform and your code passes our own tests. We will run and evaluate your tests to see if they perform a rigid test and if they achieve a reasonable coverage on the tested classes.
4) You will receive points if the overall MazeApplication runs with the manual operation and the automated driver algorithms. This requires some additional GUI feature to select and start a driver algorithm. This includes correct operation of the maze graphics that is displayed in the first person perspective as well as the map perspective with the current location and the suggested path to the exit (based on the distance matrix).
5) BONUS points: if you implement code, Junit test code and integrate the Explorer algorithm into the maze code base.