Project 4: Maze navigation, part 2

OVERVIEW
In this lab, you will continue your work from Project 3 in order to successfully navigate your robot through a maze.

TASKS
1. Grab the repository and copy your code from Project 3 over. Update your git repository to get the starter code:

   cd ~/catkin_ws/src/e28-labs
git clone git@github.swarthmore.edu:e28-fall2016/project4-TEAM.git project4

   Once again, replace TEAM with your team name. This command will populate your project4 directory. You should also copy your final code from the project 3 scripts directory into project 4. Don’t forget to change the roslib.load_manifest line at the top to reflect the new package for the code.

2. Check out maze.py. I added a module named maze.py that helps represent 2D mazes with borders along grid lines. It allows you to conveniently query each cell in the maze to see whether each neighbor is reachable, or get a list of reachable neighbors for a given \((x, y)\) location in the maze.

   Maze cells are indexed by column, and then row. Columns increase to the right, and rows increase towards the top of the maze. For instance, on the left below is a maze set up in the lab, with the corresponding ASCII text representation of the maze on the right.
In both images above, (0, 0) is the bottom-left location in the maze. Read through the maze.py module, and run it as a standalone script to execute the testing code embedded in it. Make sure you understand the basics of how to use the Maze class before you go on. You can call functions in this module from your ROS node script by adding the line

```python
import maze
```

near the top of your file by the other import lines.

3. **Implement Dijkstra’s algorithm.** Add a function to the maze.py module with the signature

```python
def find_path(maze, x0, y0, x1, y1):
```

that takes as input a Maze instance and two \((x, y)\) pairs corresponding to the start and end of the path. The function should return a list of \((x, y)\) tuples for all cells along the path from the start to the end, or return the empty list if no path could be found.

Make sure you test your Dijkstra’s algorithm implementation thoroughly by adding any necessary code to the tests in maze.py.

4. **Navigate through the maze.** Extend your Project 3 code to have the robot travel through the maze when commanded to by a user. After starting up, your robot should wait for a command (published to the /maze_command topic) telling it a starting position and orientation, and an ending position in the maze. It should then call your Dijkstra’s algorithm implementation to find a path, and drive along it until it arrives at the goal. The command format should be

```bash
x0 y0 dir x1 y1
```

Hence, running the command

```bash
rostopic pub -1 /maze_command std_msgs/String "1 0 right 3 2"
```

should instruct the robot: “you are in the space just to the right of the bottom-left corner of the maze, facing right; drive to the square two rows rightward and two rows up.”

**Safety:** Be careful not to trip over the walls or supporting blocks getting in, out, or around the maze. The maze panel close to the lab door is not anchored to its neighbors, so you can use it as a “gate” to enter and exit the maze. It’s best to walk around the walls rather than step over them.

Please make sure the robot transitions to an idle state immediately upon a bumper hit, as the starter code from Project 3 was configured to do. In general, do your best to avoid damaging the maze or the robots!
**Consideration:** Please share the maze with your classmates. It should be large enough to test multiple robots on short runs. If you are working in the lab at the same time as other groups, please communicate with each other to coordinate access to the shared resource.

**Troubleshooting:** If you find your robot is good at short paths, but less successful at long paths, you may find that you need to occasionally re-center the robot in a cell in order to prevent odometry error from accumulating to harmful levels. One possible strategy is to keep a counter of the number of cells traversed and to perform some proactive error-correcting behavior after the counter exceeds some predetermined value.

**Generality:** I reserve the right to change the maze layout as the lab progresses. Please make sure you don’t overfit your code to any particular maze layout!

5. **Going further.** Once you’ve finished the minimum functionality (using Dijkstra’s algorithm to move around in the maze based upon ROS message commands), think about ways to improve your program. Speed, smoothness, and robustness are all good goals to pursue.

**WHAT TO TURN IN**

Please use git to add, commit, and turn in your Project 4 code. Also, please submit a plain text file or PDF addressing these points:

- Who did what in your group?
- At this point in the semester, we are using a large number of software tools, including git, ssh, ROS command line tools, ROS GUI tools (possibly, for visualizing things), text editors, and others I’m sure I’ve left out. Give an example of a time when a tool wasn’t working the way you wanted it to – how did you resolve the problem? What resources did you consult? Have you learned anything in general this semester about getting the hang of new tools?
- How did you know what code to add to the maze.py module to test your Dijkstra’s algorithm? Did writing the tests reveal anything about how your code should work?
- What was your approach to modifying the “state machine” architecture to get the robot to navigate along the paths returned by Dijkstra’s algorithm?
- Provide a short description of the functionality you added beyond the basic skills required, and how you developed it.
- Include in your writeup a link to a YouTube video (accessible by me – this means “unlisted” rather than “private”) which clearly demonstrates the baseline functionality along with any additional functionality you added. The movie should demonstrate your code on two or three start-goal pairs that I will specify closer to the lab deadline.