ILP Tutorial 4 - Testing

2023-2024

In the previous tutorial, we covered the A* algorithm. It this tutorial, we will revisit it and consider some possible tests for it. But first, we'll cover a simple JUnit test example.

You may download the relevant code from `https://github.com/bikonomo/ILP23-24_tutorial_4`. This includes both the finished A* code (in `main.java`) and a `Calculator.java` class used for the first exercise. These are technically unrelated, but they were put together for ease of access.

1 Simple JUnit Example

This is a (hopefully) short exercise to familiarise you with writing JUnit tests. If you're already familiar with JUnit tests, you may feel free to skip this part.

TASK: The `Calculator.java` class contains a partial implementation of a Calculator that does several standard mathematical manipulations. Open it and do the following tasks:

- Copy the code into a new Java project and see what the three methods in it are doing. (You may notice that Lecture 8 showed tests for a similar method.)
- Then, create a new test class for the Calculator class. In IntelliJ, this can be easily done by right-click on the name of the class (not on the left-hand Project tab, but in the Code tab) and selecting "Create Test" from the context options. In this test class, write several JUnit tests for the methods in Calculator. Focus on common cases first, and then consider edge cases (if any). Make sure your tests are clear.
- You may encounter issues with the methods in Calculator, either by writing the unit tests, or by simply looking at the methods. Fix these issues, but follow the guidance of "Before you fix a bug, write a test for it."
- Briefly discuss within your groups how many tests people would write for each method. Is there any disagreement?

You may think that writing tests for such simple methods may be boring or unnecessary. Well, I won't disagree much with the former, but in practice even simple methods should often get tested – coding mistakes often happen in "easy" sections of the code precisely because people are often less careful with them.

2 Revisiting A*

Before we can properly test a piece of software, we need to be well aware of how it works. In this first part, we will take a look at a finalised full implementation of the A* algorithm and attempt to understand it well.

If you feel like you got a good understanding of it from last tutorial, then feel free to not spend too much time on this part and head to the testing part sooner rather than later.

TASK: Make sure you've downloaded and opened the code (link is at the start of the pdf), and consider / discuss the following questions within your groups:

- Check the final output for the default maze given in the code. Then make some changes to the maze that you think might be interesting. Does the final path the algorithm gives make sense?
- How is the path stored as the algorithm is running, and how is the optimal path reconstructed at the end?
- What is the purpose of the `findNeighbor()` method?
- Consider how the algorithm explores the maze. What nodes does it explore first? What happens when it reaches a wall?
- The code only uses orthogonal movement. Change it such that diagonal movement would also be allowed (keep the cost of a diagonal movement the same as an orthogonal movement), and check what the final path is. Can you comment on why it behaves the way it does? What would you do to fix it? Does it need fixing?
- Discuss the code quality. Do you think the code is clear? Are there enough / too many comments? Is there anything you'd do differently?

3 Testing A*

Imagine that you (along with everyone else on your table) are part of a software development team that has just been assigned to a project that contains the above implementation of A*. Your project lead has now tasked you with writing tests for the code. The main goal of this is to ensure that the algorithm is correct (even if parts of it get changed later).

TASK: Think about and discuss how you would test the implementation of the A* algorithm. Do not write any tests, but think about how you might implement them.

IMPORTANT! Before you read the upcoming bullet points (which are more specific and guided), please try to think about how to do the testing on your own. This task is very vague, but this is on purpose - in real life, you will likely not have a list of guiding bullet points to help you, and you will have to figure almost everything out by yourselves. If you get stuck, ideally ask your tutor first. Even when you're reading the bullet points, you should read them one by one and discuss each before moving on.

Once you've thought about the testing of the A* algorithm on your own, try to answer the following questions:

- First, let's consider the algorithm at a high level (i.e. not looking at the exact implementation, but more just at a conceptual level). What quantities do you want to compare in your tests?
  - You could make some tests which take as input an example maze (of varying sizes), plug the maze into the algorithm, and check that the final printed output of the algorithm (i.e. the thing stored in the `maze` variable at the end of the main method) is what it's supposed to be. Is this reasonable?
  - What about only comparing the final path itself (ignoring which cells got explored)?
  - What about comparing simpler values, such as the length / cost of the final path? If the goal is to find the shortest path, then you would expect the cost of it to be the same regardless of the exact implementation – is this reasonable?
- Other than the final result, what intermediate steps would you test?
  - There is actually a lot you can cover here. Try to list several examples, including with what exactly the relevant unit test might compare.
- What are some "edge cases" (i.e. weird or unlikely situations) that you might want to test for?
- While most of the algorithm is in `AStarMazeSolver`, there is also the shorter class `Cell`. What kinds of tests would you run for it?
- Now let's look at the exact implementation of `AStarMazeSolver`. You might have noticed already that it might be difficult to write proper JUnit tests for much of its functionality. You can easily test some parts like the `findNeighbor()` and `heuristic()` methods, but much of it is making use of global variables and the main method.
  - Is it clear why implementing the tests would be hard in this case?
  - How would you go about with implementing tests for these harder bits?

- Consider how you might refactor some of the code to make it easier to write tests. Specifically, what are some parts of the code that could be written into their own stand-alone (and thus easily-testable) functions.

If you've gotten to here and have time left in the tutorial, try to **a)** write some of the tests that you proposed above; and/or **b)** implement as much as possible from the refactoring ideas you suggested above, and then write tests for the refactored bits.