5.1 - Identify and apply review criteria to selected parts of the code and identify issues in the code

A surface level code review was conducted a key part of the system, focusing on core pathfinding logic. The aim of the review was to assess code quality attributes relevant to correctness, readability, robustness, and testability, maintainability.

I chose to review the `AStarIntegerPosition.java` file, as this contains the core of my pathfinding logic for the ILP coursework.

Correctness

As both all tests I've written for pathfinding pass, and I was awarded 100% of the marks in the pathfinding section of the ILP coursework, I can safely say that at least on a surface level, this code is functionally correct, producing at least *a* correctly formed path to the destination (path optimality was not scored in the ILP coursework).

Robustness

As all tests use randomly generated points within the expected input range for the coursework, I would suggest that this pathfinding code is robust as it has passed all of those tests every time they've ran so far. Another argument for robustness is that this is a classic implementation of AStar, changing only the coordinate system to avoid dealing with floating point error (the use of my `IntegerPosition` class). The use of an established algorithm with minimal changes as a starting point makes comparison with similar implementations easy, and gives me more confidence my own implementation holds up.

Code Quality - *Readability, Testability, Maintainablity*

The bulk of the logic is within the only public function - `AStarPathWithCost`, which implements the core logic of the AStar algorithm.

Here is the javadoc comment above:

```
/**
 * Given a start point, an end point, and a list of restricted areas,
returns a list of LngLat as the path, and a Double as the cost * @param
from Start point
 * @param to End point
 * @param maxCost The maximum cost before giving up
 * @param regionLines The lines the path can't cross
 * @return A list of positions as the path
 */
 public static Pair<ArrayList<LngLat>, Double> AStarPathWithCost(LngLat
from, LngLat to, double maxCost, List<Raycasting.Line> regionLines) {
```

```
    ...
    }
```

The javadoc clearly explains not only what this function does and returns, but also what you need to pass into it and why. This makes the class itself usable and serves as a readable entry-point.

However, the `AStarPathWithCost` function itself is 97 lines long. There are articles that indicate that 100 lines is slightly too long for a function [1], with users on stackexchange seeming to agree that a method should usually be short, 5-15 lines![2]. Regardless of metric, this function is slightly too long for the usual code quality standards, especially given the complexity of the code.

Out of those 97 lines, 22 are comments. This indicates that for roughly every 3 lines, there is a comment to help make clear what is being done. 16 of the remaining lines are blank, simply separating different sections of the function. This improves the code/comment ratio even further, bringing it close to one comment every 2.5 lines of code. This indicates that despite the length of the function, it is well commented, making things clear to another user what is going on, improving both the readability and the maintainability of the code.

Along with this main function, there is one helper function and two helper classes. Whilst the function is javadoced, neither of the classes is, indicating that there could be some confusion or misunderstanding over what they're responsible for. This could lead to mistakes in maintenance, and undoubtedly reduces the readability.

However, this splitting of responsibility into some other components should increase the testability, as this makes isolating responsibility easier.

As we were not marked on code style for this section of the ILP coursework, no special effort was made to make everything understandable outside of my own need to know what everything was doing. Given this, the code for especially the main function is both more readable and maintainable than expected.

Overall, some effort could be made to further modularise the code, splitting larger sections up into smaller functions to increase the readability, maintainability, and testability of the implementation. This review indicates that the code is functionally correct and well-aligned with the testing strategy, however highlights places to improve readability, modularity, and testability. These issues do not invalidate the current implementation but would be crucial to address in a larger or longer-lived system.

---

1. https://medium.com/@yegor-sychev/how-long-should-a-method-or-function-be-3c18973cf115 ↩

2. https://softwareengineering.stackexchange.com/questions/133404/what-is-the-ideal-length-of-a-method-for-you ↩