Java Maze Solver with Route Memory

I built a Java program for a simulated robot navigating procedurally generated mazes. On its first run, the robot explored using local observations and backtracked when a route led nowhere. The final controller could remember a successful route and use it to reach the target more directly on subsequent runs. I completed the project for CS118 at the University of Warwick, developing the navigation logic within a supplied simulator and adapting it to different maze structures and restrictions on the information it could store.

The problem

This project sought to analyse the differences between strategies in Java to solve classes of mazes with different types of generated structures. This includes loops, voids where no walls are generated, and mazes where the end point is at the center of one of these voids. We were then asked to look at the efficiency of the solutions with respect to the number of steps the program would take on average to complete the maze. The project was then extended to ask students to provide solvers for the existing classes of mazes with some restrictions, such as: not being allowed to store coordinates of cells in the maze.

Design decisions

A central design challenge was choosing what information to store. I initially saw the application of a hashmap to store positions of junctions efficiently using coordinates as keys. It importantly provided constant time lookups and retrievals for exploring and backtracking through the maze. For an exercise that prohibited recording junction locations, I adapted the controller to use a stack to implicitly store the order the agent found junctions.

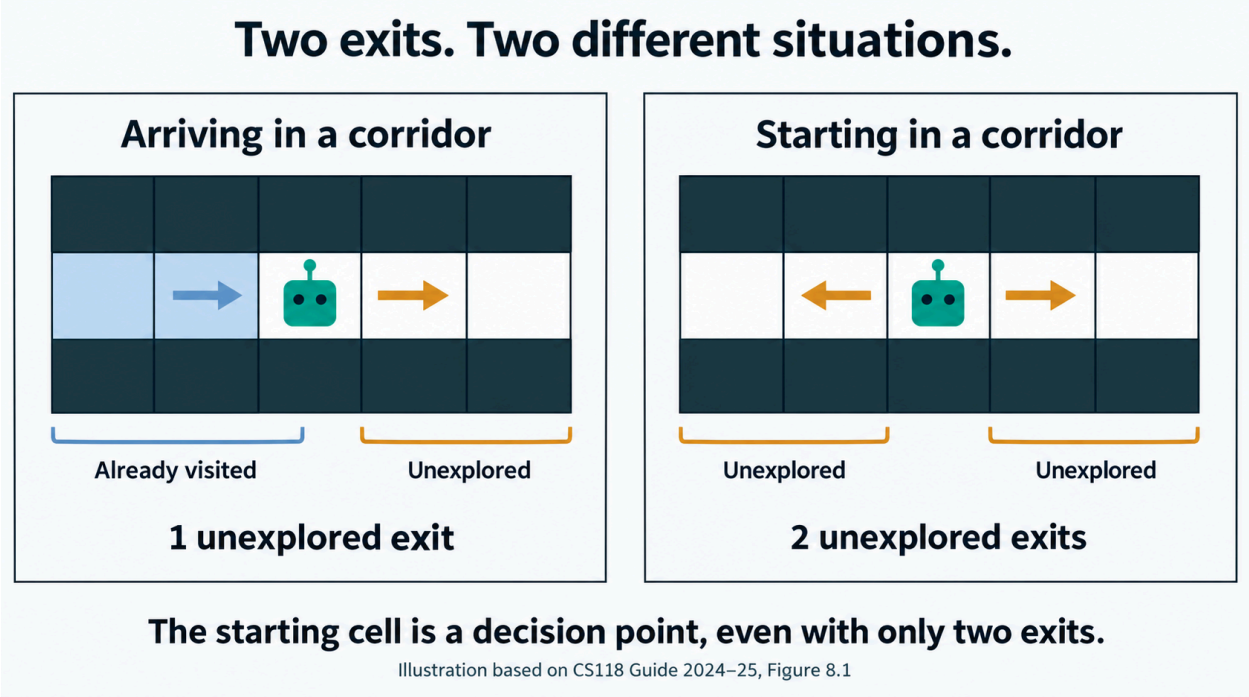
I reused the exploration stack to remember the route to the target. During the first run, the stack stored the information needed to backtrack through earlier junctions. As the robot abandoned unsuccessful branches, their entries were removed, leaving a record of the successful route. On subsequent runs, the controller read the remaining information in start-to-finish order and used it to choose the appropriate exit at each junction. This allowed the robot to follow the known route without repeating the exploration required on its first attempt or storing a separate data structure.

Testing and edge cases

Some mazes were generated as test edge cases designed to trick algorithms. For instance, a blank maze that contained no walls was supposed to be navigated by an agent, as well as mazes that contained “voids” of empty spaces rather than narrow corridors.

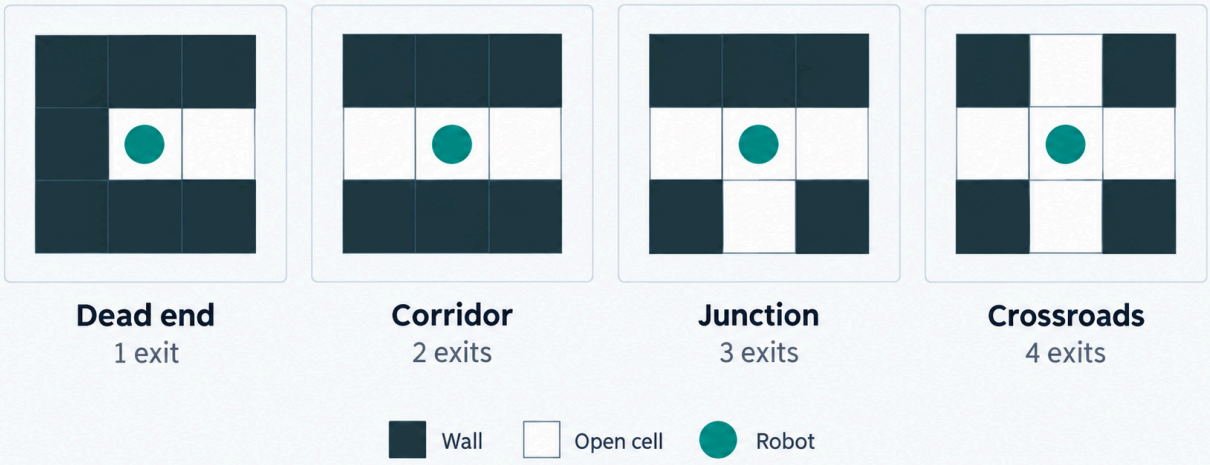
A more subtle edge case arose when the robot started in a cell with two exits. Most generated mazes allowed only one possible first move, which made it easy to overlook the start as a place where a decision had to be recorded. When the robot reaches an

ordinary corridor, one exit is the route it came from. At the starting cell, neither exit has that role: both are unexplored alternatives. A controller that records decisions only at three- or four-way junctions can miss this distinction and leave the first choice out of its traversal history. After exploring the chosen branch, its backtracking logic may then have no record that another route remains available at the start. The starting cell therefore needs to be accounted for as a decision point, even though its layout looks like a corridor. This was a small difference in the initial conditions with a significant consequence: a controller could navigate many complex mazes successfully while still leaving part of a much simpler maze unexplored.

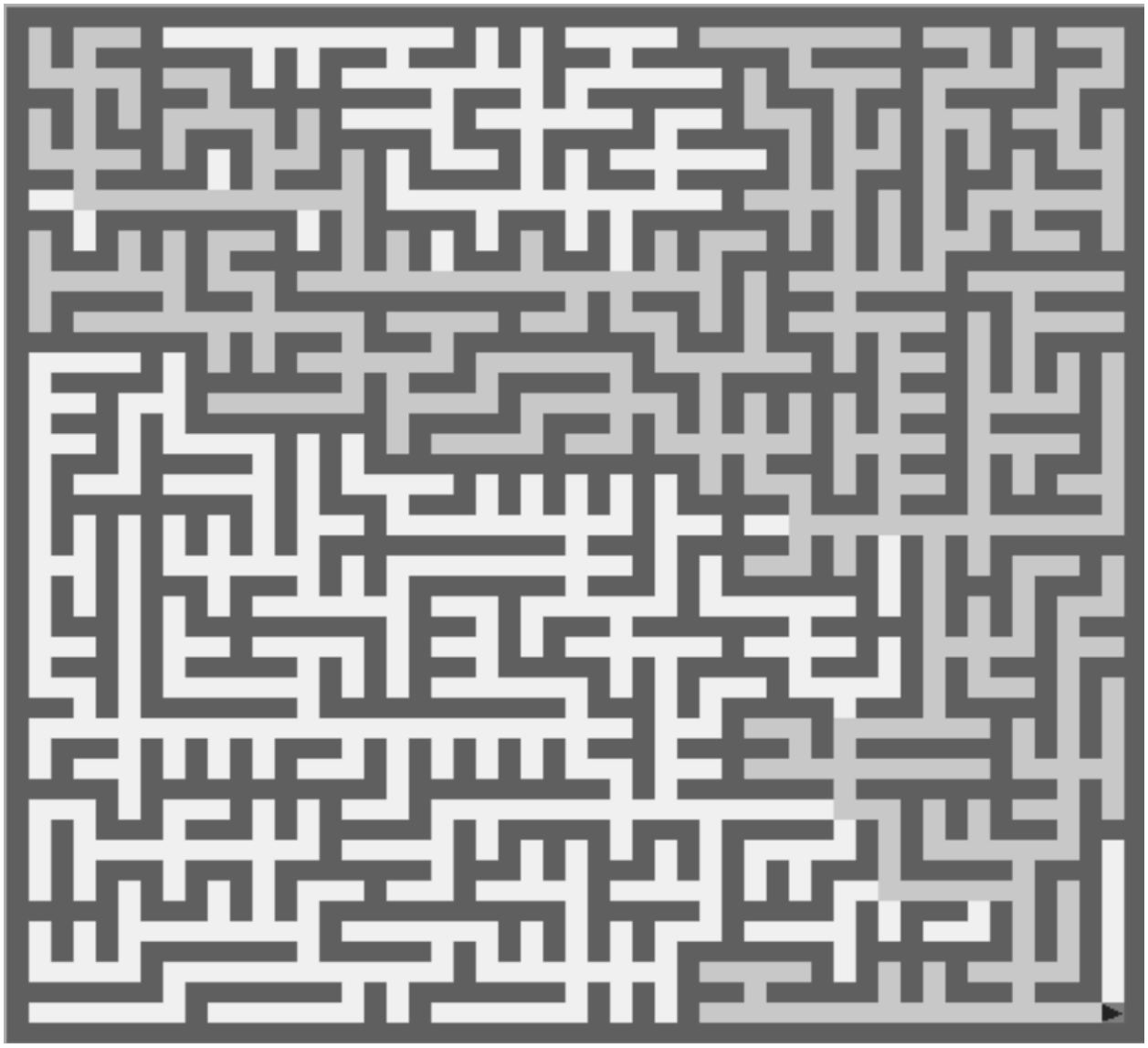


Reading the maze, one cell at a time

Open neighbours determine the local structure.



Adapted from CS118 Guide 2024–25, Figure 8.1



First run and later run — place side by side under “Route memory”. This shows the difference between the first run and the later run for the bot that learns the correct route after exploring on the first run through the maze.