Rush Hour Solver

By Devon Burnham and Alexander Mot

Design decisions

Classes

In full Java style, our code is separated into classes, each with their own purpose. Below is a short description of why each one is necessary to our project.

> **Car**
> The `Car` class makes everyone's lives easier by providing an interface which we can use to access individual cars, rather than searching the char array for cars each time we need to work with them. They hold their position, direction, length, and colour, and have methods to check for victory, generate a hashCode, and compare equality.

> **RushHour**
> This class is based on the `RushHour` class from the previous assignment. It contains some constants, a 6x6 `char` array representation of the board, and an `ArrayList` of the cars of which it is composed. It has a couple of constructors, one from a file (for initial construction) and another from a list of `Cars` (for use during move generation).

> **Solver**
> The `Solver` class, as one would assume, handles everything to do with actually solving the boards. The main function, `solve()`, uses a basic BFS algorithm to move through a graph of valid board states to find a solution, recording the steps along the way. The function `writeToFile()` then handles delivering the solution to the user, both through the console and in solution files.

Data structures

The primary data structures used in our project have been `Queues`, `PriorityQueues`, `Stacks`, and `HashMaps`. `HashMaps` were mostly used to keep track of the visited boards, which required us to implement `hashCode()` and equality comparison functions on the `RushHour` class.

Algorithms

Initially, a BFS algorithm was used to solve the boards while we worked out the rest of the program. Then, we implemented an A* heuristic-based algorithm (tested with a first-order blocking heuristic), but found that despite the theoretically improved efficiency it actually ran slower than our BFS algorithm, due to the extra time required to calculate the heuristic, and so we moved back to BFS. This automatically provides the shortest possible path to the solution.

Project History and Division of Labour

We collaborated heavily on this project, utilizing features such as IntelliJ's "Code With Me" and Visual Studio Code's "Live Share" in order to work synchronously on the same file. For more asynchronous work, we used a shared private GitHub repository and merged our changes when needed. This workflow obfuscates who exactly wrote each line of code, but we feel that the workload was shared equitably between us.

Planning

Before starting work on this project, each of the classes and functions needed were planned out, with the exception of `boardDiff()` function, which was added in a later refactor. Fortunately for us, much of this code is based on the earlier `RushHour` assignment, so we reused the structure of the `RushHour` and `Car` classes, and copied methods when possible. There were also some snippets taken from the previous graphs assigmnent, particularly in regards to keeping track of the steps required to solve the board. The only major change made was switching to using HashMaps rather than a simple array to keep track of steps.

Once we began coding, we tried to split the tasks up and each work concurrently on different functions, but we found that that made it difficult to have a full understanding of the entire program, which we wanted to have. Thus, we eventually we moved to collaborating on the same code using the aforementioned features of Intellij and VS Code.

Determinism

Initially, we used `HashSets` to store the list of cars for each `RushHour` board. This worked well, and was performant, but it meant that the order of the moves generated were not consistent from run to run, and thus debugging was much more difficult. We solved this problem by switching to an ArrayList, which has constant order. This also helped with our refactor, as the `boardDiff()` function depends on the order of cars being constant.

Refactoring the Solution Format

Near the end of the project, we became aware that our previous solution files, which consisted of the final, solved position of the boards, were not as desired: they should have been a list of all the moves necessary to reach the solved state. This refactor meant that we now had to keep track of the steps throughout the program's execution. We found that the easiest way to integrate this with our current flow was to use `HashMaps`, linking a board state with an `ArrayList` of the steps required to reach that state. The `ArrayList` is created by taking the parent's steps, and adding the steps to reach the current state from the parent. To save space, once all children have their `ArrayLists` finalized, the parents are removed from the `HashMap`.

Performance

Our `multiTest()` benchmarking function spawns an `Executor` threadpool and runs each individual puzzle concurrently in a lambda function. Running it on a Ryzen 5 3600, this finishes all 35 provided puzzles in less than a second from program start to all files successfully written. This is extremely fast, enough so that we questioned our algorithm, but we verified solution correctness in multiple ways, including checking solved states for sanity and comparing solution steps against the provided solutions. The Rust version of this project is optimized to a higher degree, so if you want flamegraphs, unstable hashers, and callgrind profiling, check that one out.

Rust

We also wrote a version of this project in the Rust programming language, which is included (with its own documentation) in its own folder. This side project was an excellent way to learn Rust, and much unnecessary (yet enjoyable) optimization occurred there.

Conclusions and Lessons Learned

- Concurrency is powerful
- Challenge your assumptions. A* should have been better, but the heuristic calculations offset its theoretical performance.
- Remote collaboration is non-trivial but very useful
- While Intellij is undoubtedly the best IDE for Java (other than Eclipse, of course), its "Code with Me" feature is still too early for us to use effectively
- VS Code's plugin ecosystem makes it a serious rival to even the most well-featured IDE
- Rust is extremely fast