## Preface

To compile the `maze-solver` binary follow the instructions in Appendix A. Precompiled 64-bit binaries have also been included.

If it's challenging to read the Rust source code, a summary of the necessary Rust-specific syntax has been included in Appendix B.

All commands in this document assume that the maze `txt` files are in the same directory as the binary. For full instructions on how to use the executable use the `-h` flag.

# 1 Maze Solver

## 1.1 Maze as a Search Problem

For a problem to be considered a search problem, it must have multiple states, where one of these states is a start state and another is an end state. By solving a search problem, we are trying to go from the start state to the end state by exploring the intermediate states. The search problem is solved once we find an ordered sequence of nodes which takes us from the start state to the end state. A maze can be considered a search problem to try to find the maze's exit from the entrance. The solution will be in the form of a path, which is used as a guide from the start of the maze to the end of the maze via the different "tiles" (represented by the character '-'). Search problems and mazes can have multiple valid solutions, which is why we often want to not just find a solution but instead the shortest possible solution.

## 1.2 Solving the Maze with Depth-FirstSearch

### 1.2.1 Outline of Depth-First Search

The depth-first search (DFS) algorithm is an algorithm that, as the name suggests, explores as deep as it can in a graph until it reaches a dead end. The graph reaches a dead end when the current node has no neighbours which have not already been visited. DFS searches along the branch of a graph as far as possible before it backtracks to the most recently visited node with unvisited neighbours. This is often accomplished with a stack, which unvisited nodes are pushed to. DFS, therefore, does not find the shortest path through the maze but will find a path through the maze given that one exists.

### 1.2.2 Solving "maze-Easy.txt" with Depth-First Search

Implementing the depth-first search algorithm (which explores Left, Up, Down then Right) results in the following route through the maze.

$$[\ (1,0),(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1),(8,1),(9,1),(10,1),(11,1),(12,1),(13,1),(14,1),(15,1),(15,2),(15,3),(15,4),(15,5),(15,6),(16,6),(17,6),(17,7),(17,8),(18,8),(18,9)\ ]$$

This list is an ordered sequence of zero-indexed ($x$,$y$) coordinates that represent the depth-first solution to the maze.

### 1.2.3 Performance of Depth-First Search

An implementation of DFS written in the Rust programming language solves the mazes with the following performances.

```
$ maze-solver -r --dfs maze-Easy.txt maze-Medium.txt maze-Large.txt maze-VLarge.txt
```

| | Easy | Medium | Lage | VLarge |
|---|---|---|---|---|
| Route Length | 27 | 321 | 1028 | 3737 |
| Nodes Explored | 35 | 549 | 61689 | 380665 |
| Time | 5.63 µs | 55.38 µs | 4.93 ms | 26.99 ms |

Table 1: DFS Performance on the Mazes

Note that the full routes have not been included in the report for the sake of brevity but can be obtained by removing the `-r` flag from the above command.

Analysing the results, we can see that as the size of the maze increases, the number of nodes explored scales at a much faster rate. On the easy maze, DFS explores 8 nodes which were not in the final route. Whereas on the very large maze, 99.01% of the nodes explored were not in the final route. We can see that DFS explores a lot of unnecessary nodes, which is not an issue for smaller mazes but quickly scales with a larger input. The larger examples highlight the flaws in DFS: it does not find an optimal path and it visits many unnecessary nodes along the way.

### 1.3 Improved Algorithm

#### 1.3.1 Outline of A*

Given that DFS solves the maze in a matter of microseconds, it is unlikely that much improvement can be made in terms of the time performance of the algorithm. However, DFS does not always calculate the shortest route. Therefore, we can use an algorithm which performs better in terms of producing the shortest path. One such algorithm is the A* search algorithm. This algorithm searches for nodes with the smallest $f(n)$, where

$$f(n) = g(n) + h(n)$$

Here $g(n)$ is the cost from the start node to the current node $n$. The heuristic, $h(n)$, is the estimated cost from $n$ to the end. As long as the heuristic is admissible, such that it never overestimates the actual cost to get to the end node, then the A* algorithm will find the shortest path. This is because A* is an informed search. In other words, it uses additional information related to the goal state, namely the heuristic function $h$, to search more accurately.

#### 1.3.2 Performance of A*

An implementation of A* solves the mazes with the following performances. Note that in the following table, % refers to the percentage change between the result from DFS and A*.

```
$ maze-solver -r maze-Easy.txt maze-Medium.txt maze-Large.txt maze-VLarge.txt
```

| | **Easy** | | | **Medium** | | | **Large** | | | **VLarge** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | DFS | A* | % | DFS | A* | % | DFS | A* | % | DFS | A* | % |
| Route Length | 27 | 27 | 0 | 321 | 321 | 0 | 1028 | 974 | -5 | 3737 | 3691 | -1 |
| Nodes Explored | 35 | 61 | 74 | 549 | 2082 | 279 | 61689 | 42055 | -32 | 380665 | 273650 | -28 |
| Time | 5.63 µs | 16.20 µs | 188 | 55.38 µs | 367.51 µs | 563 | 4.93 ms | 9.96 ms | 102 | 26.99 ms | 49.72 ms | 84 |

Table 2: DFS, A* and the percentage change between them

Again, the full routes have been removed but can be obtained by removing the `-r` flag.

From the data, we can determine that A* is certainly more performant in finding the optimal path. DFS only found the optimal path for the smallest maze and provided suboptimal paths for every other maze. Given that the A* heuristic is admissible, it managed to find the shortest path for all given mazes. In terms of nodes explored, A* does explore more than DFS for both the easy and medium mazes. However, for the large and very large mazes A* explored around 30% fewer nodes than DFS. I hypothesise that given even larger mazes, A* would continue to outperform DFS as the depths of each branch of the search space get larger. The advantage DFS has for smaller mazes is based mostly on the fact that the ratio of correct paths to incorrect paths is much smaller, therefore a more simple and brute force approach works well. The results also fail to account for the fact that the bias DFS has for exploring certain directions first likely gives it an advantage for the small and medium mazes. To get a better understanding of the performance differences between the two algorithms one could extend this exploration to calculate the average performance over many randomly generated mazes of different sizes.

## 1.4 Conclusion

In the search problem of determining a route through a maze, both DFS and A* can be used to find a valid solution. Depth-first search is advantageous in smaller mazes where the optimal solution is needed. However, A* is an improvement upon this algorithm which gives an optimal solution and explores fewer nodes for large mazes.

## Appendix A Compiling the Maze Solver

1. Install Rust - visit `https://www.rust-lang.org/tools/install` and follow the instructions to install rust.
2. Change to the root of the project directory (Where `Cargo.toml` is).
3. Compile the code with `cargo build --release` (if the download is slow, try cancelling and re-running the command).
4. Run the executable title `maze-solver` in `./target/release/`.

**Note.** the results above were gained from compiling the code with slightly different flags. The program was compiled with `RUSTFLAGS="-C target-cpu=native" cargo build --release`. These flags make the program slightly more optimised at the cost of the binary being less compatible between different systems.

# Appendix B Overview of Basic Rust Syntax

| | |
|---|---|
| `usize` | A pointer-sized (36/64 bit) unsigned integer. |
| `mut` | Defines a variable to be mutable, by default all variable are constant. |
| `pub` | Makes something publicly accessible. |
| `Vec<T>` | A vector (linked list) for some type T. |
| `Option<T>` | An optional wrapper around some type T. It either has some value x `Some(x)` or `None`. The closest thing Rust has to null. |
| `.unwrap()` | Unwraps an optional type to its value, ignoring the possibility of `None`. Crashes if it encounters a `None`. Useful when we assume something is true. |
| `.unwrap_or(x)` | Like `unwrap()` except if it encounters `None` it defaults to the value `x`. |
| `.expect("Err!")` | Like `unwrap()` except it prints the provided error message if it encounters `None` |
| `FxHashMap<K, V>` | A hashmap (dictionary) with a key of type `K` and a value of type `V`. |
| `\|x\| {x + 1}` | A closure. An inline function that take input value(s) and runs some code. Like a lambda function in Python. |
| & | A reference. Like a pointer to an object, except it cannot be null and cannot point to bad memory. `let y = &x; //y is a reference to x` |
| * | The dereference operator. Opposite of a reference, get the original object. `*y == x; // dereferences y to get value of x` |
| `impl S` | Implements the given function(s) for a struct S. The closest thing Rust has to Object Oriented programming. |