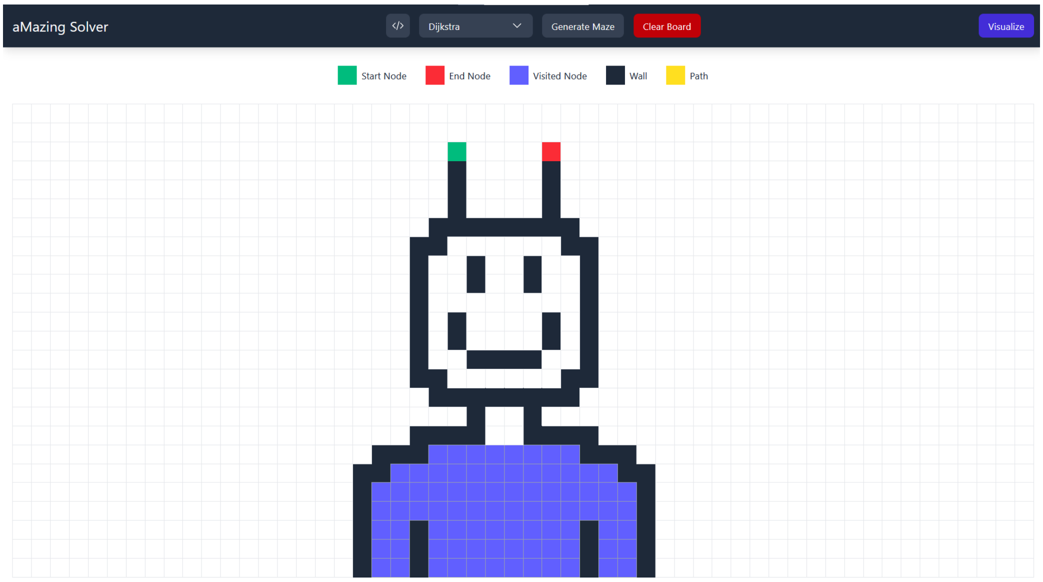

aMazing Solver

Report for project group 30, PkD 2025.

Simon Fikse   Zaidoun Younis-Khalid   Loke Öhrström

Contents

1 Introduction

This project aims to help people visualize and understand different maze solving algorithms. Its purpose is to show clear differences between their approaches, and allow the user to learn coding and get a deeper understanding of algorithms.

The program is hosted on a website. On the site the user can create their own maze or generate one automatically. They can then choose between four predefined solvers or design their own using 'aMazing Language'. Once a solver is selected, the program visualizes how it solves the maze, providing an interactive learning experience.

2 Usage

When entering the website, you are presented with a grid that fills most of the screen. The size of the grid changes when you zoom in or out and refresh the page. On the left side, there is a green node and on the opposite side, on the right, there is a red node. The green node represents the starting position of the maze, while the red node marks the goal. The nodes can be moved by hovering the mouse cursor on top of them and holding down the left mouse button. They will follow the cursor until the mouse button is let go.

Clicking on a blank node on the grid turns it black, which means that it is a wall that cannot be walked through. Clicking on a black node turns it back to blank (white-colored) meaning it is walkable.

At the top of the screen, a dark blue header contains four clickable buttons in its center.

1. </> - opens a modal window with a text editor for writing code. Will show the selected algorithm written in Amazing language. On the top right of the window an icon can be clicked opening a new tab to the projects GitHub page.

2. Algorithm - opens a drop-down menu with all available maze solving algorithms. Clicking on one selects it to be used to solve mazes. 'Custom' uses the code written in the text editor.

3. Generate Maze - generates a maze that fills the grid.

4. Clear Board - Returns the state of the grid to its original position by turning every node except the green and red blank.

On the right side of the header, there is a button that says Visualize. Clicking on it starts an animation of the selected algorithm that attempts to navigate the current maze. First purple is drawn to show nodes that the algorithm has visited. After visiting the end node, a yellow path is drawn from green to red, showing the solution the algorithm found.

Below the header, an explanation of all colors can be found.

3 Documentation

3.1 Mazes

3.1.1 Interface

The core types that different parts of the program interface with are mazes, paths, maze solvers and maze generators. A maze stores its start and end positions, the width and height of the maze and a two-dimensional array of cells describing the layout where each cell can be either a wall or empty. A path is an array of actions where each action can either be checking whether a cell at a position is a wall or to move one step in a cardinal direction. A maze solver is a function that takes a maze as input and returns a path that solves that maze. A maze generator takes a width and a height as input and returns a randomly generated maze as output.

3.1.2 Path Verification

The verify_path function takes a path and a maze as parameters and returns whether that path successfully moves from the start to the end of the maze without ever moving into a wall cell.

3.1.3 Solver Wrapper

To aid the creation of maze solvers, a solver_wrapper function has been created that takes a function as input and returns a maze solver. The wrapper defines variables to keep track of the current position in the maze and the path it has taken to get there. The input function is then passed five arguments: the end position of the maze, a function to get the current position in the maze, a function to check whether a position lies within the maze, a function to check whether a cell at a position is a wall and a function to move one step in a cardinal direction. These functions also update the current position and path accordingly.

3.2 Maze Solving Algorithms

3.2.1 A*

A* [1] is implemented as a maze solver using the solver wrapper. It will always find the shortest path and always halt.

3.2.2 Depth First Search

DFS [2] is implemented as a maze solver using the solver wrapper. It will not always find the shortest path but will always halt.

3.2.3 Dijkstra's Algorithm

Dijkstra's algorithm [3] is implemented as a maze solver using the solver wrapper. It will always find the shortest path and always halt.

3.2.4 The Maze Routing algorithm

The maze routing algorithm [4] is implemented as a maze solver using the solver wrapper. It will not always find the shortest path and may not halt when the maze is not solvable.

3.2.5 aMazing Language Implementations

In addition to typescript, each maze solving algorithm used is also implemented as strings executable by the aMazing Language in maze mode.

3.3 Maze Generation

3.3.1 Recursive Division

Recursive division is a maze generation algorithm that works by recursively dividing the board into smaller sections, adding horizontal and vertical walls with passage openings, and continues dividing until the sections reach a minimum size [5].

3.4 Component Overview

React is a library that lets you build user interfaces with components. In a React app, a component is a reusable piece of UI defined as a function that returns JSX markup [6]. This component overview provides a detailed explanation of the components.

3.4.1 App

The `App` component is the entry point to the React app.

3.4.2 Context

The `GridContext` and `EditorContext` components use React's Context API to create contexts that allow components to share state and stateful logic without the need to explicitly pass props [7].

GridContext and EditorContext

The `Editor` and `Board` components share information with the `Header` component. Contexts establish a single source of truth from which the components can derive their data. GridContext defines a `GridProvider` component that wraps the components that need access to the shared information. In `App`, the `GridProvider` wraps both the `Header` and `Board` components, allowing them to share the `grid` and `disabled` states, as well as the stateful logic, which includes the functions `setGrid` and `setDisabled` to manage them [8].

```
const [grid, setGrid] = useState<Grid>(makeGrid(0, 0));
const [disabled, setDisabled] = useState<boolean>(false);
```

Similarly, `EditorContext` defines an `EditorProvider` component that shares information about the code in the editor, logs and the stateful logic.

To access shared information, we define custom Hooks called `useGrid` and `useEditor` that can be imported into other components.

```
const { grid, setGrid, disabled, setDisabled } = useGrid();
```

3.4.3 Board

The `Board` component is the interactive grid where the maze-solving algorithms are visualized.

Data Types

`Board` uses two essential types:

- `Node`: Represents a single cell in the grid with properties about its row and column indices, and optional properties for the start node, end node, and walls (isStart, isEnd, isWall).
- `Grid`: Represents the entire grid as a 2D array of nodes, references to the start and end nodes, and the number of rows and columns.

Integration with GridContext

The component uses `useGrid`, which provides the grid state and a function `setGrid` to manage the grid state and trigger a new render when the state changes.

The `disabled` state in the `Board` component controls if user interactions are allowed. When the `disabled` state is set to true, the board receives the "disabled" class, which sets pointer events to none in CSS, preventing interactions with the board.

```
<div
    ref={boardRef}
    className={clsx({"disabled": disabled}, "board")}
    onMouseUp={handleMouseUp}
>
```

Grid Initialization

When the `Board` is mounted, a `useEffect` Hook [8] gets dimensions using `boardRef` and calculates the number of rows and columns based on the size of a 24-pixel cell. Refs, created with the `useRef` Hook, remember information without triggering new renders and give access to DOM elements in React [8]. The grid is created using `makeGrid`, and the initial grid is updated with the new grid. The effect will run only once because `setGrid` is stable and does not change between renders, which causes the effect to run only once on mount (when the component first renders). The grid is initially defined in `GridContext`, but because the `Board` component is not rendered at this point and the dimensions are based on the layout defined by CSS, the dimensions cannot be retrieved until after the component has been mounted. This is why `useEffect` is used here.

Interacting with the Grid

The `Board` component enables user interactions through mouse event handlers.

- `handleMouseDown`: This function is called when a mouse button is pressed on a cell and checks if the clicked cell is a start node, an end node, or a regular node. Then, it assigns an action to `nodeRef` or `wallRef`.

- `handleMouseEnter`: This function is called when the mouse moves over a cell while the mouse button is pressed down. It uses helper functions to move the start node or end node and modifies walls.

- `handleMouseUp`: When a mouse button is released, it resets the references `nodeRef` and `wallRef` to null and updates the grid state to sync changes made during interactions.

Rendering

The grid is rendered as an HTML table, where each cell is a node in the grid and has a unique ID that can be retrieved using the `getNodeID` function. Using IDs helps when manipulating cells in the DOM.

The update logic decouples visual feedback from the underlying grid state for performance reasons. When the user interacts with the board, the component uses helper functions to manipulate the DOM by adding or removing CSS classes on the cells to reflect the changes visually. When an interaction is completed, `updateGrid` is called to update the grid state.

Helper Functions

- `editWall`: Modifies the class attributes of a specific cell. If the cell is not a start or end node, it changes the `isWall` property of the node based on whether the current action in `wallRef` is to add or remove a wall. This function adds or removes the corresponding CSS class to visually update the cell.

- `moveNode`: Repositions of the start and end nodes. First, it ensures that the target cell is not a wall and is not occupied by the opposite node. Then, it moves the node by removing the relevant CSS class from the old cell and adding it to the new one, while also updating the `isStart` or `isEnd` properties of the nodes.

- `styles`: Uses the clsx library to conditionally return a string of CSS classes for a given node. This helps in applying different styles for the cells (e.g. walls, start, and end nodes) based on the properties of the node.

Visualizing Algorithms

Algorithms are visualized on the board with an asynchronous function called `visualize`. Because the `Grid` type is not directly compatible with the `Maze` type, which is used by the maze-solving algorithms, a helper function `gridToMaze` is used to convert the `Grid` to a `Maze`. The reason for the asynchronous approach is to see the progression of the algorithm searching and drawing the path, as the DOM updates would otherwise happen instantly. Each step of the visualization is delayed by 10 milliseconds using `setTimeout`. The Promise API provides a `resolve` function, which is used to indicate that an asynchronous operation (the delay) is complete and allow the next step.

3.4.4 Editor

The `Editor` component is a simple code editor for users to implement their own algorithms using aMazing Language.

Interacting with the Editor

The editor consists of two `Textarea` components, one for users to write code and another for displaying error messages that would otherwise appear in the console.

A helper function, `handleKeyDown`, prevents the default tab behavior to let users make tab indentations within the editor. It detects tab key presses using the `KeyboardEvent` type from React and updates the position of the cursor.

3.4.5 Header

The `Header` component interfaces with the `Board` and `Editor` components.

The component inherits states and stateful logic from `GridProvider` and `EditorProvider`. The `disabled` state is used in `Header` to disable buttons and apply different styles. The `code` state and `setCode` and `setLog` functions are used to manage the code editor. When running algorithms from the code editor, `code` is passed to `evaluate_solver`.

The different maze-solving algorithms are displayed in the dropdown using an array of objects. Additionally, a Custom option allows users to write their own algorithm. A placeholder function is used in place of the maze solver, as it is not defined until the code is evaluated.

```
fn: (): never[] => []
```

Implementations of the available algorithms can be displayed in the dropdown when selected. This is handled using a `useEffect` Hook that updates the `code` state using `setCode`.

The code editor will capture errors that would otherwise display in the console. These errors are displayed within the editor itself by updating the `log` state using `setLog`.

Helper Functions

- `clearSearch`: Clears previous searches and paths if there are any.
- `clearBoard`: Resets the entire board and updates the `grid` state with `setGrid`, if there are walls, visited nodes, or paths.
- `generateMaze`: Resets the board and generates a maze using recursive division and updates the `grid` state.
- `run`: Starts the visualization. If the code editor is displayed, it runs the code in the editor.

3.4.6 Styling

All components are styled using Tailwind CSS, which is a CSS framework that defines a list of utility CSS classes that can be used to build designs directly in the markup.

Example

```
<div className="flex items-center space-x-2">
    <div className="node start" />
    <span>Start Node</span>
</div>
```

Tailwind scans the components for class names, generating the corresponding styles and then writing them to a static CSS file [9]. For this reason, a CSS file must be included in which Tailwind can be imported.

3.5 aMazing Language Interpreter

3.5.1 Tokenizer

The first step of interpreting a string is to convert it to an array of tokens. A token consists of its type, the text it represents, the line number it started on and the column number it started on.

The input string is read one character at a time and that character is added to a string containing the current token. When whitespace is encountered or the current token string would become invalid, the current token string is matched against keyword, symbols, integer literal rules and variable naming rules. If it matches any of them, a new token of that type is pushed to the resulting array and the current token string is reset to begin accumulating the next token.

3.5.2 Parser

Second is to convert the token array of tokens into intermediate representation. The intermediate representation is a tree-like structure where each node contains the type of expression or statement it represents as well as what other values and nodes it consists of.

The input array is treated as a stack from which tokens are consumed as subexpressions are parsed. The main parse function will look at the next token and depending on what it is, it will call different helper functions capable of parsing that specific type of statement. Those functions will in turn call different helpers to parse the expressions they consist of which will call helpers to parse their subexpressions. Once all subexpressions are parsed they can be combined into the full expression node and returned to be combined into a full statement node and so on.

3.5.3 Evaluator

The evaluator will use the tokenizer and parser to get the inputs intermediate representation. It uses an environment frame to store a table for looking up the values associated with variable names as well as what frame to fall back on if a name could not be found.

The predefined values are defined in a new environment frame before starting the evaluation of the intermediate representation in a new frame with that as fallback. Similarly to the parser, when evaluating statements, helper functions are called to parse the expressions that make them up which in turn call helper functions that evaluate their subexpressions and so on. Eventually, a function is returned that will evaluate the main function of the input using this same method.

3.5.4 Maze Solving Evaluator

To evaluate in maze mode, the solver wrapper is used. The regular evaluator is used but the functions passed by the solver wrapper are exposed to the program as additional predefined values. Additionally, the main function is called with the end position's coordinates as arguments. A tuple of a maze solver and an array of strings is returned. The array of strings represents the program's standard output and will be filled with the strings printed during execution after the maze solver is called.

4 aMazing Language Specification

4.1 Tokens

4.1.1 Comments

The first # character in a line and all characters after it until the next new line or end of input is ignored.

Example

```
var x = 2; # this is a declaration
#abc # var var var ?
```

4.1.2 Naming Rules

A variable name is case sensitive and must start with a letter a-z or A-Z or an underscore and every following character must be a letter a-z or A-Z, an underscore or a digit 0-9. A variable cannot share a name with any of the reserved keywords.

Example

```
x
_x
x1
_123
```

4.1.3 Integers

An integer literal is case insensitive and must start with a digit 0-9 and every following character must be a letter a-z or A-Z, an underscore or a digit 0-9. Underscores are ignored for purposes of determining its value.

If it begins with `0b` it is a binary integer and can only contain binary digits. At least one non-underscore character must follow.

If it begins with `0x` it is a hexadecimal integer and can only contain hexadecimal digits. At least one non-underscore character must follow.

Otherwise it is a decimal integer and can only contain decimal digits.

Example

```
1234      → 1234
1_234     → 1234
0x1f      → 31
0x_1__f_  → 31
0b01010   → 10
0b1_010   → 10
```

4.1.4 Reserved Keywords

- `var`
- `fn`
- `if`
- `else`
- `while`
- `for`
- `return`
- `continue`
- `break`

4.1.5 Operators

Unary Prefix Operators All unary prefix operators have the same precedence which is higher than all binary operators. All unary prefix operators are right-to-left associative.

`!`	Logical NOT
`+`	Unary plus
`-`	Unary minus

Binary Operators Binary operators are listed top to bottom in descending precedence. Operators on the same line have the same precedence. All binary operators are left-to-right associative.

- `* / %` Multiplication, division and modulo
- `+ -` Addition and subtraction
- `< <= > >=` The common relational operators
- `== !=` Equality and inequality
- `&&` Logical AND
- `||` Logical OR

Example

```
x[y]()        → (x[y])()
!-x           → !(-x)

x + y * z     → x + (y * z)
x + y - z     → (x + y) - z
x && y || z   → (x && y) || z

-x[y]         → -(x[y])
-x + -y       → (-x) + (-y)
x() + y[z]    → (x()) + (y[z])
```

4.1.6 Other Symbols

All symbols (including operators) are parsed greedily, not ending a symbol before it would become invalid or encounter whitespace.

- `( )` Overriding precedence
- `{ }` Grouping statements
- `[ ]` Creating arrays
- `=` Declaration and assignment
- `,` Separating elements
- `;` End statement

Example

```
===   → == =
=!=   → = !=
||||  → || ||
```

4.2 Syntax

4.2.1 Expressions

Expressions must be of one of the following forms.

name	Variable name
integer	Integer literal
`(` *expression* `)`	Overriding precedence
`[` *expression* ...`]`	Array literal
`fn (` *expression* ...`) {` *statement* ...`}`	Function literal
expression `(` *expression* ...`)`	Function call
expression `(` *expression* `)`	Array subscripting
unary-prefix-operator expression	Unary prefix operator
expression binary-operator expression	Binary operator

Example

```
[1, 2, 3] + [x, y, z]
fn () {}
(fn (x, y) { return x + y; })()
-x[y]
1 * (1 + 1)
```

4.2.2 Statements

Statements must be of one of the following forms.

`;`	No operation
expression `;`	Expression, ignoring result
`var` *name* `=` *expression* `;`	Variable declaration
expression `=` *expression* `;`	Variable assignment
`if (` *expression* `)` *statement*	If conditional
`if (` *expression* `)` *statement* `else` *statement*	If-else conditional
`while (` *expression* `)` *statement*	While loop
`for (` *expression/assignment/declaration* `;` *expression* `;` *expression/assignment* `)` *statement*	For loop
`return ;`	Return
`return` *expression* `;`	Return value
`continue ;`	Continue
`break ;`	Break
`{` *statement* ...`}`	Code block

Example

```
var x = 1;
x = 2;
if (x) return x; else if (y) continue; else break;
while (f(x));
{ var x = 1; x = 2; }
```

4.3 Types

There are three types: integer, array and function. Variables, elements, parameters and return values do not have types associated with them and can store, accept and return different types than they did originally. No two values of different types are equal.

Integer An integer stores a whole number. Two integers are equal if they represent the same number.

Array An array stores a reference to an ordered list of elements that can be of different types. Elements can be modified, added or removed from the list and all arrays storing the same reference will observe the effect. Two arrays are equal if they store the same reference.

Function A function stores a reference to an object that can be called with a list of ordered values that can be of different types as its arguments to return a value of any type. Two functions are equal if they store the same reference.

4.4 Environment

The first frame created which points to nothing contains all predefined values. This is what the program frame, where the user written code runs, points to. After executing, the program frame must contain a variable named `main` that stores a function. The number and types of the arguments this function is called with will depend on the use case.

Creation When a statement inside a conditional, loop or code block is executed, it will do so inside a newly created frame pointing to the frame the condition, loop or code block was executed inside. Additionally a function object will store what frame it was created in and when called, the body will be executed in a new frame pointing to the one stored in the function object.

The function body is executed in the same frame that the parameters were declared in.

Usage When reading or assigning to a variable, the name will be searched for within the current frame and if it has not been declared, it will recursively search the frame it points to. If it has not been declared and it does not point to anything, an error is thrown. When declaring a variable, if the name has not been declared in the current frame, it will be, otherwise an error is thrown. Note that if a variable is read or assigned to but would be declared later in the same scope, the reading or assigning will still search the frame pointed to.

Example

```
var x = 1;
{
    print(x); # 1
    x = 3;
    var x = 2;
    print(x); # 2
}
print(x) # 3
```

4.5 Semantics

4.5.1 Truth Value

An integer is truthy if it is not 0. An array is truthy if its length is not 0. A function is always truthy.

4.5.2 Operators

Function call `f(x)`
Takes a function as its primary operand and optionally a list of values between the parentheses as arguments to the function. Returns the value returned by its body or 0 if no return value was returned.

Array subscripting `a[i]`
Takes an array as its primary operand and an integer between the brackets to use as index. The index must be between 0 (inclusive) and the length of the array (exclusive) where 0 corresponds to the first element in the array. Returns the value stored at that index which can be assigned to.

Logical NOT `!x`

Takes a value as operand and returns 0 if it's truthy and 1 otherwise.

Unary plus `+i`

Takes an integer and returns the same integer.

Unary minus `-i`

Takes an integer and returns its negation.

Multiplication `i*i`

Takes two integers and returns their product.

Division `i/i`

Takes two integers and returns the floor of their quotient. If the right operand is 0 an exception is thrown.

Modulo `i%i`

Takes two integers and returns the integer $a - b\left\lfloor\frac{a}{b}\right\rfloor$ where a is the left operand and b the right operand. If the right operand is 0 an exception is thrown.

Addition / Concatenation `x+x`

Takes two integers or two arrays. If they are integers it returns their sum. If they are arrays it returns a new array containing all the elements in the left operand followed by all the elements in the right operand.

Subtraction `i-i`

Takes two integers and returns their difference.

Less than `i<i`

Takes two integers and returns 1 if the left operand is less than the right operand or 0 otherwise.

Less than or equal to `i<=i`

Takes two integers and returns 1 if the left operand is less than or equal to the right operand or 0 otherwise.

Greater than `i>i`

Takes two integers and returns 1 if the left operand is greater than the right operand or 0 otherwise.

Greater than or equal to `i>=i`

Takes two integers and returns 1 if the left operand is greater than or equal to the right operand or 0 otherwise.

Equality `x==x`

Takes two values and returns 1 if they are equal or 0 otherwise.

Inequality `x!=x`

Takes two values and returns 0 if they are equal or 1 otherwise.

Logical AND `x&&x`

Evaluates the left operand and returns it if it is not truthy. Otherwise it evaluates the right operand and returns it.

Logical NOT `x||x`

Evaluates the left operand and returns it if it is truthy. Otherwise it evaluates the right operand and returns it.

4.5.3 Statements

No operation `;`

Does nothing.

Expression `x;`

Evaluates the expression, discarding its value.

Variable declaration `var n = x;`

Declares a variable with the name following the var keyword in the current frame and assigns it the value on the right hand side of the equals sign.

Variable assignment `x = x;`

If the left hand side of the equals sign is a variable name, that variable is assigned the value on the right hand side of the equals sign.

If the left hand sign is an array subscripting expression, the element at that index of the array is assigned the value on the right hand side of the equals sign.

Otherwise an error is thrown.

If conditional `if (x) s`

If the value inside the parentheses is truthy the statement following them is executed in a new frame. Any returns, continues and breaks are passed through.

If-else conditional `if (x) s else (x) s`

If the value inside the parentheses is truthy the statement following them is executed in a new frame. Otherwise the statement following the else keyword is executed in a new frame. Any returns, continues and breaks are passed through.

While loop `while (x) s`

Evaluates the expression inside the parentheses in a new frame. If it is truthy the statement following them is evaluated. If it is a return, the return is carried through. If it is a break, the loop stops. Otherwise, this is repeated.

For loop `for (x; x; x) s`

Evaluates the first expression inside the parentheses in a new frame. Evaluates the second expression inside the parentheses. If that second expression is truthy, the statement following the parentheses is evaluated. If it is a return, the return is carried through. If it is a break, the loop stops. Otherwise, the third expression inside the parentheses is evaluated and everything except the evaluation of the first expression is repeated.

Return `return;`

Skips all following statements when inside a code block and returns 0 when inside a function.

Return value `return x;`

Skips all following statements when inside a code block and returns the value following the return keyword when inside a function.

Continue `continue;`

Skips all following statements when inside a code block.

Break `break;`

Skips all following statements when inside a code block and stops the loop when inside a while loop.

Code block `{s...}`

Evaluates all statements inside the curly brackets in a new frame. If a statement is a return, continue or break, the statements following it are not evaluated and the return, continue or break is passed through.

4.5.4 Predefined Values

`true`

The integer 1.

`false`

The integer 0.

`panic`

A function that takes no arguments and throws an error.

`print`

A function that takes any number of arguments of any type and prints them to stdout separated by spaces.

`len`

A function that takes an array as its only argument and returns its length as an integer.

`push`

A function that takes an array as its first argument and a value of any type as its second argument and then adds the second argument to the end of the array, increasing its length by one. Returns 0.

`pop`

A function that takes an array as its only argument. If the array is empty, an error is thrown. Otherwise the last element of the array is removed, reducing its length by one. Returns the removed element.

`is_int`

A function that takes a value as its only argument and returns 1 if it is an integer or 0 otherwise.

`is_arr`

A function that takes a value as its only argument and returns 1 if it is an array or 0 otherwise.

`is_fun`

A function that takes a value as its only argument and returns 1 if it is a function or 0 otherwise.

Example

```
print(true, false, len([3, 2]));      # "1 0 2"
var x = [1, 2]; push(x, 3); print(x); # "[1, 2, 3]"
var x = [1, 2]; print(pop(x), x);     # "3, [1, 2]"
print(is_int(true));                  # "1"
print(is_arr([3, 2][0]));             # "0"
print(is_fun(is_fun));                # "1"
```

4.6 Maze Mode

Maze mode is meant to aid in creating maze solving algorithms using the language and is what is used by aMazing Solver.

4.6.1 Mazes

A maze is a grid of cells with a current position and a goal position. Each cell in the grid can be either a wall or not a wall and a valid maze solving algorithm should never occupy the same space as a wall. The top left cell has x coordinate 0 and y coordinate 0 and the x and y coordinates of cells increase as you move right and down through the maze respectively.

4.6.2 The main Function

The main function will be called with two integers, The first being the x coordinate of the goal and the second being the y coordinate of the goal. If the main function returns when the current position is the same as the goal position and the current position has never overlapped with a wall or exceeded the boundaries of the maze, it has successfully solved the maze. The return value is ignored.

4.6.3 Additional Predefined Values

`right`
The integer 0.

`up`
The integer 1.

`left`
The integer 2.

`down`
The integer 3.

`get_x`
A function that takes no arguments and returns the current x coordinate within the maze as an integer.

`get_y`
A function that takes no arguments and returns the current y coordinate within the maze as an integer.

`in_bound`
A function that takes two integer arguments and returns 1 if the position with x value equal to the first argument and y value equal to the second argument is within the bounds of the maze. Otherwise it returns 0.

`is_wall`
A function that takes two integer arguments and returns 1 if the position with x value equal to the first argument and y value equal to the second argument is within the bounds of the maze or if that position is occupied by a wall. Otherwise it returns 0.

`move`
A function that takes one integer value. If the value is 0, the current x coordinate is incremented by one. If the value is 1, the current y coordinate is decremented by one. If the value is 2, the current x coordinate is decremented by one. If the value is 3, the current y coordinate is incremented by one. Otherwise an error is thrown.

Example

```
# Moves down if there is no wall below.
if (!is_wall(get_x(), get_y() + 1)) {
    move(down);
}
```

5 Testing

The Jest [10] testing framework is used for all tests.

5.1 Path Verification

The maze.ts file that defines the verify_path function has above 88% statement coverage. None of the remaining uncovered lines are part of the verify_path function.

5.2 Maze Solving Algorithms

Both implementations of each maze solving algorithm have above 80% statement coverage. The reason it is not higher is because they are easier to test manually on the final website where larger mazes can be automatically generated and the paths they take can be visualized.

5.3 aMazing Language Interpreter

Each part of the interpreter has above 90% statement coverage. All remaining statements are for error handling, some of which are unreachable and the rest unreachable given input generated from the previous layer of the interpreter. These are kept in case a bug was missed and they end up being reachable or in case the interpreter is refactored and the same errors are no longer handled in the earlier stages.

5.4 Website

The website does not have automated tests because that is outside the functionality of Jest.

6 Discussion

6.1 Technical Decisions

Before visualizing an algorithm, it must complete and return. This means that an algorithm stuck in an infinite loop will crash the page, but also that it makes implementing them much simpler. It also makes it easier to create a website and algorithm separately before joining them together, making it possible to work on different parts in parallel.

When updating the maze on the website, the immediate visual feedback is decoupled from the underlying state which is only updated after the user completes an action. This provides great performance boosts for larger mazes.

In the aMazing Language, whenever the next subexpression could be of different forms, it is always possible to disambiguate just from the very next token. For example, a function literal starts with `fn` instead of using the JavaScript arrow function syntax. This makes parsing easier without a GLR parser.

6.2 Strengths

The code is modular, with a well defined interface for different components to interact with each other. This makes it easy to add or change some parts of the code without modifying the rest.

The website provides a clean and intuitive interface. It makes it easier to focus on understanding the algorithms instead of how to use and navigate the website.

The fact that you can view the code of the predefined algorithms makes it possible to get a more in-depth understanding of the algorithms. Additionally, it makes it easier to learn the syntax of the language without needing to look up the documentation.

6.3 Weaknesses

Unfortunately, the website cannot use the same maze type we agreed on at the beginning of the project and instead implements a separate grid type. This makes the code somewhat less readable and coherent but was necessary in order to store the visual states like visited and path.

Not all algorithms halt when presented with an impossible maze. This could have been fixed in multiple ways but was ignored in order to prioritize other functionality.

Despite initial ambitions, there is only one maze generation algorithm available on the website.

The website does not work well for mobile devices. The header does not fit the page and the grid does not have touch controls.

6.4 Conclusion

We have created a website where users can visualize different maze solving algorithms. Additionally, we defined a new programming language and created an interpreter for it built into the website. This language can be used to create custom maze solving algorithms to be visualized on the website.

We are very pleased with the end result and feel we have accomplished our goals. However, the website has not been tested in a real setting and so we cannot determine how effectively it aids in learning.

References

[1] Wikipedia contributors, "A* search algorithm — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1273031415, 2025. [Online; accessed 5-March-2025].

[2] E. Darulova and T. Wrigstad, "Programkonstruktion och datastrukturer 1DL201 DL201 HT2024 — Uppsala Universitet," 2024–2025. [Course Lecture Notes; accessed 5-March-2025].

[3] Spanning Tree, "How dijkstra's algorithm works — YouTube." https://www.youtube.com/watch?v=EFg3u_E6eHU, 2020. [Online; accessed 5-March-2025].

[4] Wikipedia contributors, "Maze-solving algorithm — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Maze-solving_algorithm&oldid=1275380789, 2025. [Online; accessed 5-March-2025].

[5] Wikipedia contributors, "Maze generation algorithms — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Maze_generation_algorithm#Recursive_division_method, 2025. [Online; accessed 7-March-2025].

[6] React, "Your first component." https://react.dev/learn/your-first-component, 2025. [Online; accessed: 6-March-2025].

[7] React, "Passing data deeply with context." https://react.dev/learn/passing-data-deeply-with-context, 2025. [Online; accessed: 6-March-2025].

[8] React, "Built-in react hooks." https://react.dev/reference/react/hooks, 2025. [Online; accessed: 6-March-2025].

[9] Tailwind, "Styling with utility classes." https://tailwindcss.com/docs/styling-with-utility-classes, 2025. [Online; accessed: 6-March-2025].

[10] Jest Contributors, "Jest: Delightful javascript testing." https://jestjs.io/. [Online; accessed 5-March-2025].