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. It's 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 three pre-defined 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. The tool is meant for an educational context.

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 opposite side on the right, there is a red node. The green one represents the maze's starting position, while the red node marks the goal.

Clicking on a blank node in the grid turns it black, meaning it is a wall that can not be walked through. Clicking on a black node returns it blank (white colored) meaning it is walkable.

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

1. Algorithm - opens a drop down menu with all available maze solving algorithms. Clicking on one selects it to be used to solve mazes.

2. Generate maze - recursively generates a maze that fills the grid.

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

On the right of the header, there is a button that says Run Maze. Clicking it starts an animation of the selected algorithm attempting 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 colours 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 maze where each cell can be either a wall or empty. A path is an array of actions where each action can either be checking wether 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 wether 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 wether a position lies within the maze, a function to check wether 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* 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 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 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 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.

3.3 aMazing Language Interpreter

3.3.1 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. It 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.

3.3.2 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.

3.3.3 Evaluator

The evaluator will use the tokenizer and parser to get the inputs intermediate representation. It uses 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.

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 variables 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 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 encounters 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 search 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 its 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 and 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 statement when inside a code block and returns 0 when inside a function.

Return value `return x;`

Skips all following statement 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 statement 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 types and prints them to stdout separated by spaces.

`len`

A function that takes an array as 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 adds then second argument to 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 if 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 Discussion

5.1 Conclusion