COMS W4115 – FALL 2012

aML –" a-Mazing-Language"

Evan Drewry – **ewd2106**

Nikhil Helferty – **nh2407**

Sriramkumar Balasubramanian – **sb3457**

Timothy Giel – **tkg2104**

Introduction

A maze is a puzzle in the form of a series of branching passages through which a solver must a route. Actual mazes have existed since ancient times, serving as a means to confuse the traveler from finding his or her way out. Since then, the idea behind mazes has been extrapolated to construct a set of puzzles designed to challenge people to solve or find the route.

While the concept of maze solving might seem too restricted, maze exploration in general can be extrapolated to other fields like Graph theory and topology. Apart from this, there exist more than one way to solve mazes, which has led to the rise of the time and space analysis of these approaches. Also solving a maze can be likened to exploring a map which paves way for many practical uses of a language for solving mazes.

Having justified the existence of a language to solve mazes, we now introduce AML (A-mazing Language) which can be used to solve mazes by feeding instructions to a *bot* which is located at the entrance to the maze at time *0.* The maze in question can either be defined by the user in the form of text files or can be randomly generated by the standard library functions.

AML is designed to not only make the process of solving mazes easier to a programmer, but also to introduce programming to the common man through mazes.

Language Description

AML's design ensures the freedom of the user to implement many maze solving algorithms, while ensuring the ease of use of the language to traverse mazes. The language serves as an instruction set to the bot, hence the movement of the bot determines accessing of various data.

Here are some important features of the preliminary AML syntax:-

Primitive data types: "Integer", "Boolean", "Cell"

Literals: Integer literals, T/F, null

Non-primitive data-types: “List<Type>”

Operators: {+, -, *, /, %, ^}, {:=, ==}, {and, or, not}

Implicit Functions: “move_up(), move_down(), move_left(), move_right(),revert(), moveTo(<Cell>)”

Implicit Variables: {Visited: List<Cell>}, {Current_Position: Cell}

Keywords: “function (args) : <Type>”, “if then”, “exit”, “return”

*(**Note:** The usage of implicit functions and variables to allow no confusion to the programmer regarding declaring variables to control the bot. A lot of functions will be created in the standard library to make the language more accessible to non-programmers.)*

Sample Programs

These are generally how programs in AML would be structured.

Depth First Search

```
#load “ScenarioA.map”

function MainLoop():void{
	DFS();
}


Function DFS():void{
	Cell node := Current_Position;
	if (node.target()) then{
		Exit;
	}
	if(Visit_Unvisited(Current_Position)) then{
		DFS();
	}
	else{
		if (node.source()) then{
			Exit;
```

```
        }


                revert();

                DFS();

        }


}


function Visit_Unvisited(Cell node):boolean{

        if (node.hasLeft() and not Visited(node.left)) then{

                Move_left();

        }

        else if (node.hasTop() and not Visited(node.top) then{

                Move_top();

        }

        else if (node.hasRight() and not Visited(node.right) then{

                Move_right();

        }

        else if (node.hasBottom() and not Visited(node.bottom) then{

                Move_bottom();

        }

        else{

                Return F;

        }

        Return T;

}
```

Breadth First Search

```
#load-randomized

function MainLoop():void{
```

```
        List<Cell> toGo := {Current_Position};
        BFS (toGo);
}

function BFS (List<Cell> toGo):void{
        If (not (empty (toGo))) then {
                Cell node := toGo.remove();
                If (node.target()) then{
                        Exit;
                }
                moveTo(node);
                addToGo(node, toGo);
                BFS(toGo);
        }
}

function addToGo(Cell node, List<Cell> toGo):void{
        If (node.hasLeft() and not Visited(node.left)) then{
                toGo.add(node.left);
        }
        if (node.hasTop() and not Visited(node.top) then{
                toGo.add(node.top);
        }
        if (node.hasRight() and not Visited(node.right) then{
                toGo.add(node.right);
        }
        if (node.hasBottom() and not Visited(node.bottom) then{
                toGo.add(node.left);
        }
}
```