CALIFORNIA STATE UNIVERSITY
FRESNO

PROJECT REPORT
(CSCI 264)
On

MAZE SOLVING USING AI AIGORITHMS

TO
DAVID RUBY

BY
Sri Ramya Nimmagadda -109492883
Jeevana Kala Bommannagari - 109504843

Introduction

Imagine that an artificially intelligent robot approaches a valley. Looking down the cliff into the valley, the robots sees a large, man-made, labyrinth structure. The structure has an entrance, an exit, and is surrounded by water. The robot can, of course, simply enter the structure and wander around, until it finally reaches an exit point. However, since the robot is programmed with AI, it can do a lot better. The robot can scan the maze into its memory and perform image processing against it, converting the pixels in the image into a data representation of the maze. With the maze analyzed, an algorithm can be ran against it to determine a solution path through the maze.

There are a number of different maze solving algorithms, that is, automated methods for the solving of mazes. The random mouse, wall follower, Pledge, and Trémaux's algorithms are designed to be used inside the maze by a traveler with no prior knowledge of the maze, whereas the dead-end filling and shortest path algorithms are designed to be used by a person or computer program that can see the whole maze at once.

Many maze solving algorithms are closely related to graph theory. If one pulled and stretched out the paths in the maze in the proper way, the result could be made to resemble a tree.

Tremaux's Algorithm

Tremaux's algorithm, invented by Charles Pierre Tremaux, is an efficient method to find the way out of a maze that requires drawing lines on the floor to mark a path, and is guaranteed to work for all mazes that have well-defined passages. A path is either unvisited, marked once or marked twice. Every time a direction is chosen it is marked by drawing a line on the floor (from junction to junction). In the beginning a random direction is chosen (if there is more than one). On arriving at a junction that has not been visited before (no other marks), pick a random direction that is not marked (and mark the path). When arriving at a marked junction and if your current path is marked only once then turn around and walk back (and mark the path a second time). If this is not the case, pick the direction with the fewest marks (and mark it, as always). When you finally reach the solution, paths marked exactly once will indicate a direct way back to the start. If there is no exit, this method will take you back to the start where all paths are marked twice. In this case each path is walked down exactly twice, once in each direction. The resulting walk is called a bidirectional double-tracing.

Tremaux works better on complex classic style mazes (fixed-size pixel tiles) with many passages in many directions. Using either algorithm, the AI will determine a solution and may successfully traverse the maze.

This method is similar to actually walking through the maze. Only the immediate tiles visible to you can be followed. The robot begins walking in a random direction and continue until it hits a wall. It then turns right, until a path is available to walk. Each time the algorithm takes a step, it marks the tile as visited. The algorithm always tries to first visit an unvisited tile. However, if no new tiles are found, it will backtrack over a visited tile, incrementing the mark on the tile to 2, until it finds an unvisited tile. It will never visit the same tile more than twice (with the only exception being if the algorithm is stuck in a dead-end, in which case it will re-visit a tile it's already backtracked over, in order to exit the trap. The maze solution is all tiles that have been visited just once. Tremaux requires that each tile in the path is 1-unit in size with no obstructions in the path, other than walls. That is, the maze must contain well-defined, narrow passages. Tremaux does not require knowing the exit point when beginning the search through the maze. Since it essentially performs a depth-first search, it simply needs to know when to stop.

A* Search Algorithm

One possible algorithm finds the shortest path by implementing a breadth-first search, while another, the A* algorithm, uses a heuristic technique. The breadth-first search algorithm uses a queue to visit cells in increasing distance order from the start until the finish is reached. Each visited cell needs to keep track of its distance from the start or which adjacent cell nearer to the start caused it to be added to the queue. When the finish location is found, follow the path of cells backwards to the start, which is the shortest path. The breadth-first search in its simplest form has its limitations, like finding the shortest path in weighted graphs.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

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

where *n* is the last node on the path, *g*(*n*) is the cost of the path from the start node to *n*, and *h*(*n*) is a heuristic that estimates the cost of the cheapest path from *n* to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

In order for the AI to use A* search to find a solution path, it requires knowing the exit point. It uses this data in its heuristic calculation.

Manhattan Distance

```
this.manhattan = function(pos0, pos1) {
var d1 = Math.abs(pos1.x - pos0.x);
var d2 = Math.abs(pos1.y - pos0.y);
return d1 + d2;
```

```
        }
```

A* search tends to be more flexible and is usually able to search around obstacles and variable-sized paths. However, A* has other requirements and limitations. It must know the exit point before beginning its calculations. It also needs an explicit heuristic defined. Movement with an A* search algorithm is typically 4-directional or 8-directional, including diagonals. A* search takes longer, exploring almost all of the map to locate a solution. This is most likely due to the directions of the maze tunnels leading away from the exit point and the high-degree of backtracking required.

Code

index .html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Mazes</title>
        <link rel='stylesheet' href='css/bootstrap.min.css' />
        <link rel='stylesheet' href='css/bootstrap-responsive.min.css' />
        <link rel='stylesheet' href='css/style.css' />
        <script type="text/javascript">
</script>
    </head>
    <body style="background-color:#FAFD95">
        <div class='container' style='margin-left: 30px;'>
        <h1 style="color:orange;"> Maze Solving Using Different AI
Algorithms</h1>
        <div class="pull-left" style='margin: 0 0 25px 0;'>

    <div id='customDiv' class='pull-left' style='cursor: pointer; margin: 0 0 0 10px;'>
```

```
<div class='pull-left'>
<i id='customDivIcon' class='icon-plus-sign'></i>
                    </div>
                    <div class='pull-left' style='margin: 0 0 0 5px;'>
                        Build  Maze
                    </div>
                </div>
                <div style='clear: both;'></div>
<div id='custom' class='well' style='display: none; margin: 8px 0 0 0;'>
                    <p>
                        <div class='pull-left'>
                            <a href='?maze={"start": {"x": 3, "y": 0},
"end": {"x": 6, "y": 0}, "width": 20, "height": 10, "map": "
    ***.**.*************
    ***....*************
    **..****.........***
    *..****..*******.***
    *.****..********...*
    *.****.***********.*
    *.****........***..*
    *.**************..**
    *................***
    ********************"}'>'G' Maze</a>
                        </div>
        <div class='pull-left' style='margin: 0 5px 0 5px;'>|</div>
                        <div class='pull-left'>
                            <a href='?maze={"start": {"x": 2, "y": 0},
"end": {"x": 19, "y": 3}, "width": 20, "height": 10, "map": "
    **.*****************
    **.****............*
    **......*****.******
    **.*********........
```

```
        *..*********.*******
        **....******.***.***
        *****......*.***.***
        **....****.*.....***
        **.***.....*****...*
        ********************"}'>simple Maze</a>
                                        </div>
<div id='canvasStatus'></div>
<div id='canvasDiv' style='float: left;'>
<canvas id='imageView'>
</canvas>
</div>
<div style='clear: both;'></div>
<div style='margin: 25px 0 0 0;'>
<div class='pull-left'>
<button id='btnGo' class='btn btn-primary btn-large'
style="background-color:red">Go!</button>
</div>
<div class='pull-right' style='margin:7px 0 0 25px;'>
<div class='pull-right' style='margin:7px 0 0 25px;'>
<div class="btn-group" data-toggle-name="algorithmType" data-toggle="buttons-radio">
<button type="button" value="scripts/tremauxAlgorithm.js" class="btn active"
data-toggle="button" style="background-color:pink">Tremaux Algorithm</button>
<button type="button" value="scripts/aStarAlgorithm.js" class="btn" data-toggle="button"
style="background-color:lightgreen">A* Search</button></div>
<input type="hidden" id="algorithmType" name="algorithmType"
value="scripts/tremauxAlgorithm.js" /></div></div><div style='clear: both;'></div>
          <script type='text/javascript' src='scripts/tremauxAlgorithm.js'></script>
          <script type='text/javascript' src='scripts/walker.js'></script>
          <script type='text/javascript' src='scripts/maze.js'></script>
</script>
</body>
</html>
```

Walker.js

```
function walkerManager(context, maze) {
	this.context = context,
	this.maze = maze,
	this.x = maze.start.x,
	this.y = maze.start.y,
	this.lastX = -1,
	this.lastY = -1,
	this.visited = createArray(this.maze.width, this.maze.height),

	this.init = function() {
		// Clear array to all zeros.
		for (var x = 0; x < this.maze.width; x++) {
			for (var y = 0; y < this.maze.height; y++) {
				this.visited[x][y] = 0;
			}
		}

		// Set starting point.
		this.visited[this.x][this.y] = 1;

		// Draw starting point.
		this.draw();
	},

	this.draw = function() {
		this.context.fillStyle = 'rgb(255, 100, 100)';
		this.context.fillRect(this.x * 10, this.y * 10, 10, 10);
	},

	this.move = function(direction, backtrack) {
		var changed = false;
```

```
oldX = this.x;
oldY = this.y;

if (backtrack || !this.hasVisited(direction)) {
	// Get the new x,y after moving.
	var point = this.getXYForDirection(direction);

	// Check if this is a valid move.
	if (this.canMove(point.x, point.y)) {
		this.x = point.x;
		this.y = point.y;
		changed = true;
	}
}

if (changed) {
	this.context.fillStyle = 'rgb(' + (backtrack ? 100 : 255) + ', 0, 0)';
	this.context.fillRect(oldX * 10, oldY * 10, 10, 10);

	this.lastX = oldX;
	this.lastY = oldY;

	// Mark this tile as visited (possibly twice).
	this.visited[this.x][this.y]++;

	if (backtrack) {
		// We've turned around, so don't visit this last tile again.
		this.visited[this.lastX][this.lastY] = 2;
	}

	if (this.visited[oldX][oldY] == 2 && this.visited[this.x][this.y] == 1) {
```

```
                                        // Found an unwalked tile while backtracking. Mark our last
tile back to 1 so we can visit it again to exit this path.
                                        this.visited[oldX][oldY] = 1;
                                        this.context.fillStyle = 'rgb(255, 0, 0)';
                                        this.context.fillRect(oldX * 10, oldY * 10, 10, 10);
                                }
                        }

                        return changed;
                },

                this.canMove = function(x, y) {
                        return (maze.isOpen(x, y) && this.visited[x][y] < 2);
                },

                this.hasVisited = function(direction) {
                        // Get the new x,y after moving.
                        var point = this.getXYForDirection(direction);

                        // Check if this point has already been visited.
                        return (this.visited[point.x][point.y] > 0);
                },

                this.getXYForDirection = function(direction) {
                        var point = {};

                        switch (direction) {
                                case 0: point.x = this.x; point.y = this.y - 1; break;
                                case 1: point.x = this.x + 1; point.y = this.y; break;
                                case 2: point.x = this.x; point.y = this.y + 1; break;
                                case 3: point.x = this.x - 1; point.y = this.y; break;
                        };
```

```
                    return point;
          }
};

function createArray(length) {
    var arr = new Array(length || 0),
        i = length;

    if (arguments.length > 1) {
        var args = Array.prototype.slice.call(arguments, 1);
        while(i--) arr[length-1 - i] = createArray.apply(this, args);
    }

    return arr;
}
```

Tremauxalgorithm.js

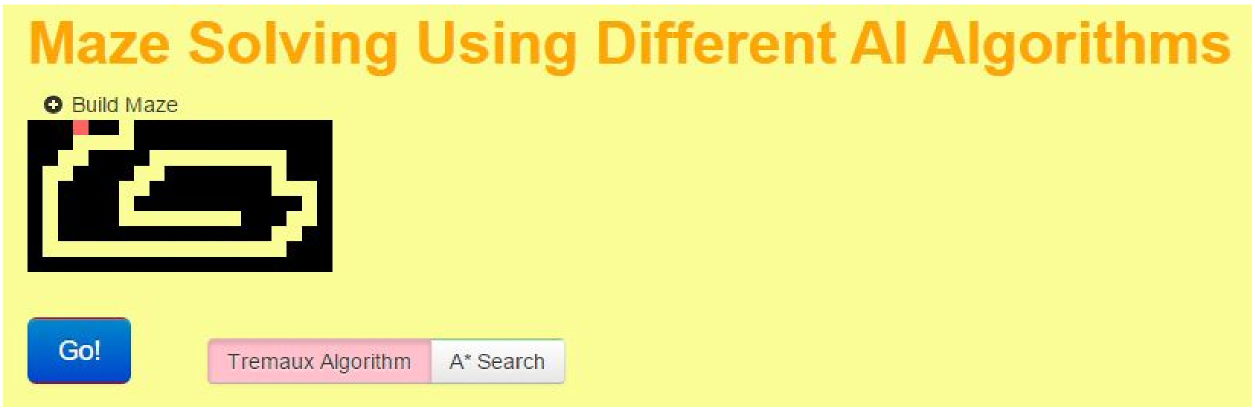
```
function searchAlgorithm(walker) {
          this.walker = walker,
          this.direction = 0,
          this.end = walker.maze.end,

          this.step = function() {
                    var startingDirection = this.direction;

                    while (!this.walker.move(this.direction)) {
                              // Hit a wall. Turn to the right.
                              this.direction++;

                              if (this.direction > 3) {
                                        this.direction = 0;
```

```
            }

            if (this.direction == startingDirection) {
                // We've turned in a complete circle with no new path available. Time to backtrack.
                while (!this.walker.move(this.direction, true)) {
                    // Hit a wall. Turn to the right.
                    this.direction++;

                    if (this.direction > 3) {
                        this.direction = 0;
                    }
                }

                break;
            }
        }

        this.walker.draw();
    },

    this.isDone = function() {
        return (walker.x == walker.maze.end.x && walker.y == walker.maze.end.y);
    },

    this.solve = function() {
        // Draw solution path.
        for (var x = 0; x < this.walker.maze.width; x++) {
            for (var y = 0; y < this.walker.maze.height; y++) {
                if (this.walker.visited[x][y] == 1) {
                    this.walker.context.fillStyle = 'red';
```

```
                                        this.walker.context.fillRect(x * 10, y * 10, 10, 10);
                                }
                        }
                }
        }
};
```

Output

Initial page



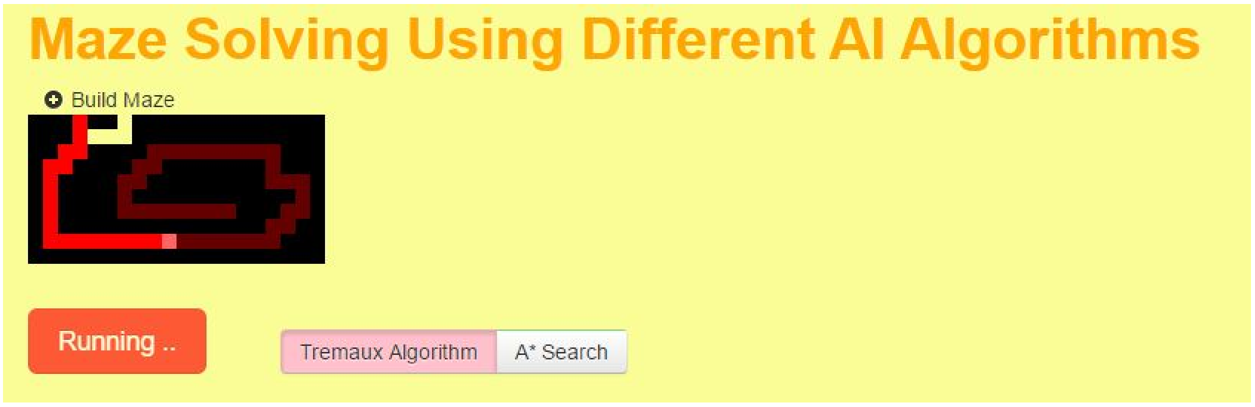
Path finding using Tremaux Algorithm

The Tremaux algorithm is similar to actually walking through the maze. Only the immediate tiles visible to you can be followed. It begins in a random direction and continues until it hits a wall. It then turns right, until a path is available to walk. Each time the algorithm takes a step, it marks the tile as visited. The algorithm always tries to first visit an un-visited tile. However, if no new tiles are found, it will backtrack over a visited tile.

It will never visit the same tile more than twice (with the only exception being if the algorithm is stuck in a dead-end, in which case it will re-visit a tile it's already backtracked over, in order to exit the trap).

The maze solution is all tiles that have been visited once.

Step 1:

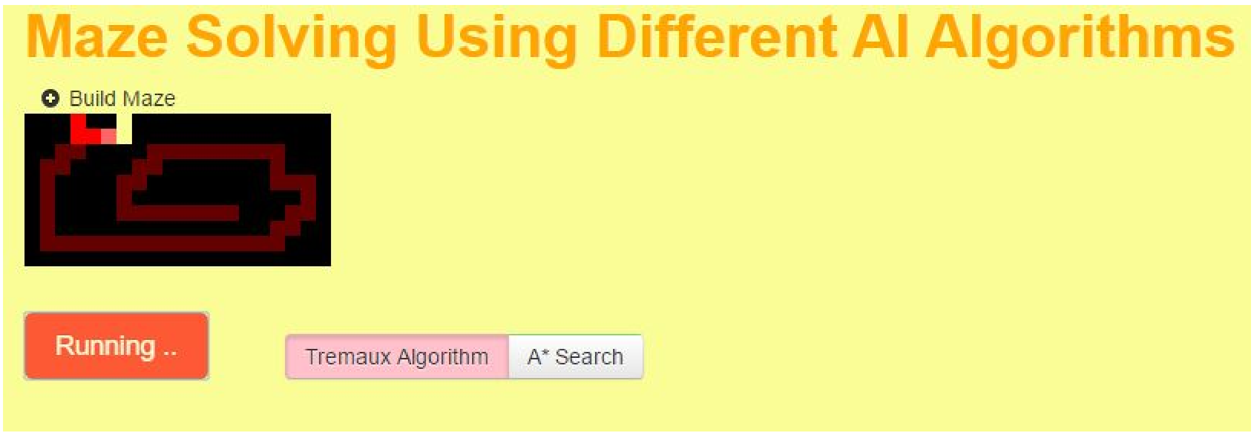

Maze Solving Using Different AI Algorithms

Build Maze

Running ..

Tremaux Algorithm | A* Search

Step 2:

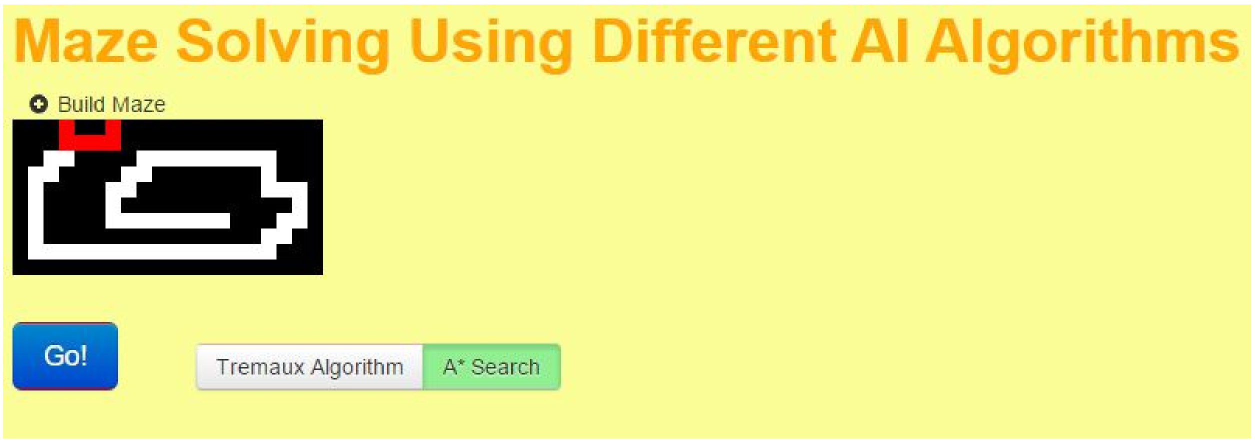

Maze Solving Using Different AI Algorithms

Build Maze

Running ..

Tremaux Algorithm | A* Search

Path finding using A* algorithm


Maze Solving Using Different AI Algorithms

Build Maze

Go!

Tremaux Algorithm | A* Search

A* search is similar to having an aerial view of the maze before walking through it. It uses the maze end point to calculate a score from each neighboring tile. It intelligently walks down multiple paths, seemingly at the same time, until it reaches the end.

CONCLUSION

We Implemented maze using different AI Algorithm. The maze is created using java code and the path generation is done using javascript. We provided web page for our project using basic HTML and CSS. The A* and Tremaux pathfinding algorithms are implemented and the pro's and con's of each Algorithm are studied.