Project Report

On

Analysis of Pathfinding Algorithms

Submitted by

Jhalak Gupta (170001024)

Ashwini Jha (170001012)

Computer Science and Engineering

2nd year

Under the Guidance of

Dr. Kapil Ahuja

Department of Computer Science and Engineering

Indian Institute of Technology Indore

Spring 2019

1. Introduction

Pathfinding is the plotting, by a computer application, of the shortest route between two points. It is a more practical variant on solving mazes. Pathfinding is closely related to the shortest path problem, within graph theory, which examines how to identify the path that best meets some criteria (shortest, cheapest, fastest, etc) between two points in a large network.

Two primary problems of a pathfinding are (1) to find a path between two nodes in a graph; and (2) the shortest path problem - to find the optimal shortest path. The first problem is solved by basic algorithms like Breadth- first search and depth-first search by exhausting all possibilities. The more complicated problem is finding the optimal path. Bellman-Ford, A, Dijkstra's algorithm and some other algorithms can be used to solve the second problem.*

- *To study the basic pathfinding algorithms, particularly A*, Dijkstra, BFS, Bellman-Ford.*
- *To find out examples where some of these algorithms might fail to find the shortest path to the target.*
- *To analyse the complexity of the A* algorithm and compare it with other commonly used algorithms.*
- *To study various optimizations done till now and, if possible, try to propose our own optimized A* algorithm.*

2. Algorithm Analysis

A algorithm: A* is a variant of Dijkstra's algorithm commonly used in games. A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end.*

F-cost is calculated as follows:

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

f(n) = F-cost of the current node 'n'

g(n) = exact cost of the path from initial node to the current node 'n'

h(n) = heuristic function that estimates the cost of the cheapest path from current node 'n' to the goal

Time Complexity : Depends on heuristic chosen

Dijkstra's algorithm: This algorithm begins with a start node and an "open set" of candidate nodes. At each step, the node in the open set with the lowest distance from the start is examined. The node is marked "closed", and all nodes adjacent to it are added to the open set if they have not already been examined. This process repeats until a path to the destination has been found. Since the lowest distance nodes are examined first, the first time the destination is found, the path to it will be the shortest path.

Time Complexity (using heap) : E log(V)

Greedy Best First Search algorithm: This algorithm is based on just heuristics and just unlike A, doesn't adds the weight of that node.*

A* vs Dijkstra vs Greedy Best-first Search

Parameter	A* Search Algorithm	Dijkstra's Algorithm	Greedy Best-first Search
PATHFINDING ON A 25 X 25 GRID			
Execution Time (sec)	0.676	7.825	0.577
Distance Covered	758.82	758.82	758.82
Computed Nodes	35	441	29
PATHFINDING ON A 50 X 50 GRID			
Execution Time (sec)	3.136	67.749	2.807
Distance Covered	762.25	750.53	788.11
Computed Nodes	72	1752	64
PATHFINDING ON A 100 X 100 GRID			
Execution Time (sec)	17.162	937.54	16.223
Distance Covered	749.32	738.11	783.96
Computed Nodes	132	7064	125

Comparing Heuristics

Parameter	Euclidean	Manhatten	Overestimate
PATHFINDING ON A 25 X 25 GRID			
Execution Time (sec)	0.585	0.599	0.562
Distance Covered	742.25	742.25	742.25
Computed Nodes	31	31	31
PATHFINDING ON A 50 X 50 GRID			
Execution Time (sec)	3.422	2.578	2.199
Distance Covered	790.54	770.54	790.54
Computed Nodes	97	74	64
PATHFINDING ON A 100 X 100 GRID			
Execution Time (sec)	17.468	17.851	15.568
Distance Covered	779.32	800.53	836.39
Computed Nodes	154	154	139

3. Algorithm design

Optimization of A Search Algorithm*

Following are the changes that could be done to optimise the A Search Algorithm:*

- *A simple onList flag can be set for each node thereby removing the need to search the Closed Set repeatedly, thus making the algorithm more time efficient.*
- *Open Set can be made a priority queue instead of an array, mainly because the size of the Open Set continues to grow over time and so the cost of finding and removing the smallest entry keeps increasing. Priority queue will help do the operations in O(logn) which is better than finding smallest element in array in O(n).*
- *Adding neighbors with lesser f-cost to a different set from Open Set.*

Selection Of Heuristics

Selection of heuristic function is an important part of ensuring the best A performance. Ideally H is equal to the cost necessary to reach the goal node. In this case A* would always follow perfect path, and would not waste time traversing unnecessary nodes. If overestimated value of H is chosen, the goal node is found faster, but at a cost of optimality. In some cases that may lead to situations where the algorithm fails to find path at all, despite the fact, that path exists. If underestimated value of H is chosen, A* will always find the best possible path. The smaller H is chosen, the longer it will take for algorithm to find path. In the worst-case scenario, H = 0, A* provides the same performance as Dijkstra's algorithm.*

Estimated heuristic cost is considered admissible, if it does not overestimate the cost to reach the goal.

The different type of heuristics used are:

- *Manhatten Distance:*
  h = abs(current_cell.x – goal.x) + abs(current_cell.y – goal.y)

- *Euclidean Distance:*
  h = sqrt((current_cell.x – goal.x)^2 + (current_cell.y – goal.y)^2)

- *Diagonal Distance*
  h = max(abs(current_cell.x – goal.x), abs(current_cell.y – goal.y))

4. Implementation and Results

```
function removeFromArray(arr,elt){
	for(var i =arr.length-1;i>=0;i--){
		if(arr[i] == elt) {
			arr.splice(i,1);
		}
	}
}

function heuristic(a,b){
	var d = dist(a.i,a.j,b.i,b.j);
	return d;
}

function PathDistance(shortestPath)
{
	var d=0;
	for(var i=1;i<shortestPath.length;i++)
	{
		d += dist(shortestPath[i].i*w
+w/2,shortestPath[i].j*h+h/2,shortestPath[i-1].i*w+w/2,shortestPath[i-1].j*h+h/2);
	}
	return d;
}

function SwitchGraph(x_id){
	var x = document.getElementById(x_id).value;
	if(x_id=="A"){
```

```
		if(x=="Show A*"){
			document.getElementById(x_id).value="Hide A*";
			showA = true;
			showFinalPath();
		}
		else if(x=="Hide A*"){
			document.getElementById(x_id).value="Show A*";
			showA = false;
			showFinalPath();
		}
	}
	if(x_id=="D"){
		if(x=="Show Dijkstra"){
			document.getElementById(x_id).value="Hide Dijkstra";
			showDij = true;
			showFinalPath();
		}
		else if(x=="Hide Dijkstra"){
			document.getElementById(x_id).value="Show Dijkstra";
			showDij = false;
			showFinalPath();
		}
	}
	if(x_id=="AOpp"){
		if(x=="Show A* Backward"){
			document.getElementById(x_id).value="Hide A* Backward";
			showAOpp = true;
			showFinalPath();
		}
		else if(x=="Hide A* Backward"){
			document.getElementById(x_id).value="Show A* Backward";
			showAOpp = false;
			showFinalPath();
		}
	}

}

function showFinalPath(){
	if(endLoop){
		if(showA){
			noFill();
			stroke(210,50,160);
			strokeWeight(w/2);
			beginShape();
			for(var i=0;i<finalPath.length;i++){
				vertex(finalPath[i].i*w+w/2,finalPath[i].j*h+h/2);
```

```
                }
                endShape();
            }
            if(showDij){
                noFill();
                stroke(0,50,160);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<finalPathDij.length;i++){
                    vertex(finalPathDij[i].i*w+w/2,finalPathDij[i].j*h+h/2);
                }
                endShape();
            }
            if(showAOpp){
                noFill();
                stroke(50,250,50);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<finalPathOpp.length;i++){
                    vertex(finalPathOpp[i].i*w+w/2,finalPathOpp[i].j*h+h/2);
                }
                endShape();
            }
        }

}

var cols = 20;
var rows = 20;
var w,h;
var endLoop = false;
//A*
var grid = new Array(cols);
var openSet = [];
var closedSet = [];
var start;
var end;
var path = [];
var doneA = false;
var noSolA = false;
var stepsA = 1;
var distanceA = 0;
var showA = true;
var finalPath = [];
//Dijkstra
var gridDij = new Array(cols);
var openSetDij = [];
```

```
var closedSetDij = [];
var startDij;
var endDij;
var pathDij = [];
var doneDij = false;
var noSolDij = false;
var stepsDij = 1;
var distanceDij = 0;
var showDij = true;
var finalPathDij = [];
//Aopp
var gridOpp = new Array(cols);
var openSetOpp = [];
var closedSetOpp = [];
var startOpp;
var endOpp;
var pathOpp = [];
var doneAOpp = false;
var noSolAOpp = false;
var stepsAOpp = 1;
var distanceAOpp = 0;
var showAOpp = false;
var finalPathOpp = [];


// DEFINING PROPERTIES OF EACH CELL/SPOT
function Spot(i, j) {
	this.i = i;
	this.j = j;
	this.f = 0;
	this.g = 0;
	this.h = 0;
	this.neighbors = [];
	this.previous = undefined;
	this.wall = false;

	this.show = function(col) {
		fill(col);
		if(this.wall){
			fill(0);
		}
		noStroke();
		ellipse(this.i*w+w/2,this.j*h+h/2,w-1,h-1);
	}

	this.addNeighbors = function(grid){
		var i = this.i;
```

```
            var j = this.j;
            if(i<cols-1){
                  this.neighbors.push(grid[i+1][j]);
            }
            if(i>0){
                  this.neighbors.push(grid[i-1][j]);
            }
            if(j<rows-1){
                  this.neighbors.push(grid[i][j+1]);
            }
            if(j>0){
                  this.neighbors.push(grid[i][j-1]);
            }
            if(i>0 && j>0){
                  this.neighbors.push(grid[i-1][j-1]);
            }
            if(i<cols-1 && j>0){
                  this.neighbors.push(grid[i+1][j-1]);
            }
            if(i>0 && j<rows-1){
                  this.neighbors.push(grid[i-1][j+1]);
            }
            if(i<cols-1 && j<rows-1){
                  this.neighbors.push(grid[i+1][j+1]);
            }
      }
}


// SETUP FUNCTION

function setup() {
      createCanvas(500,500);
      console.log('A*');

      w=width/cols;
      h=height/rows;
      for(var i=0;i<cols;i++){
            grid[i] = new Array(rows);
      }

      for(var i=0;i<cols;i++) {
            for(var j=0;j<rows;j++) {
                  grid[i][j] = new Spot(i,j);
            }
      }
```

```
	//DIJKSTRA
	for(var i=0;i<cols;i++){
		gridDij[i] = new Array(rows);
	}

	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			gridDij[i][j] = new Spot(i,j);
		}
	}

	//A* OPPOSITE
	for(var i=0;i<cols;i++){
		gridOpp[i] = new Array(rows);
	}

	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			gridOpp[i][j] = new Spot(i,j);
		}
	}

	// RANDOMIZE OBSTACLES
	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			if(random(1) < 0.3){
				grid[i][j].wall = true;
				gridDij[i][j].wall = true;
				gridOpp[i][j].wall = true;
			}
		}
	}


	// ADDING NEIGHBOURS OF EACH CELL
	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			grid[i][j].addNeighbors(grid);
		}
	}

	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			gridDij[i][j].addNeighbors(gridDij);
		}
	}
```

```
	for(var i=0;i<cols;i++) {
		for(var j=0;j<rows;j++) {
			gridOpp[i][j].addNeighbors(gridOpp);
		}
	}


	start = grid[0][0];
	end = grid[cols-1][rows-1];
	start.wall = false;
	end.wall = false;

	startDij = gridDij[0][0];
	endDij = gridDij[cols-1][rows-1];
	startDij.wall = false;
	endDij.wall = false;

	startOpp = gridOpp[cols-1][rows-1];
	endOpp = gridOpp[0][0];
	startOpp.wall = false;
	endOpp.wall = false;

	openSet.push(start);
	openSetDij.push(startDij);
	openSetOpp.push(startOpp);


}


// DRAW FUNCTION
function draw() {

	if(openSet.length > 0) {

		var winner = 0;
		for(var i=0;i<openSet.length;i++){
			if(openSet[i].f<openSet[winner].f){
				winner = i;
			}
		}
		var current = openSet[winner];

		if(doneA){
			distanceA = PathDistance(path);
		}
```

```
        if(current===end && !doneA){
                //noLoop();
                doneA = true;
                console.log("A* DONE in "+stepsA+" steps!");
        }

        if(!doneA){
                removeFromArray(openSet,current);
                closedSet.push(current);
                stepsA++;
        }

        var neighbors = current.neighbors;

        for(var i=0;i<neighbors.length;i++){
                var neighbor = neighbors[i];

                if(!closedSet.includes(neighbor) && !neighbor.wall){
                        var tempG = current.g + 1;

                        var newPath = false;
                        if(openSet.includes(neighbor)){
                                if(tempG < neighbor.g){
                                        neighbor.g = tempG;
                                        newPath = true;
                                }
                        } else {
                                neighbor.g = tempG;
                                openSet.push(neighbor);
                                newPath = true;
                        }

                        if(newPath){
                                neighbor.h = heuristic(neighbor,end);
                                neighbor.f = neighbor.g + neighbor.h;
                                neighbor.previous = current;
                        }
                }
        }

} else {
        console.log('No Solution');
        noSolA = true;
        //noLoop();
        //return;
}
```

```
//DIJKSTRA
if(openSetDij.length > 0) {

	var winnerDij = 0;
	for(var i=0;i<openSetDij.length;i++){
		if(openSetDij[i].f<openSetDij[winnerDij].f){
			winnerDij = i;
		}
	}
	var currentDij = openSetDij[winnerDij];

	if(doneDij){
		distanceDij = PathDistance(pathDij);
	}

	if(currentDij===endDij  && !doneDij){
		//noLoop();
		doneDij = true;
		console.log("DIJKSTRA DONE in "+stepsDij+" steps!");
	}

	if(!doneDij){
		removeFromArray(openSetDij,currentDij);
		closedSetDij.push(currentDij);
		stepsDij++;
	}

	var neighborsDij = currentDij.neighbors;

	for(var i=0;i<neighborsDij.length;i++){
		var neighborDij = neighborsDij[i];

		if(!closedSetDij.includes(neighborDij) && !neighborDij.wall){
			var tempGDij = currentDij.g + 1;

			var newPathDij = false;
			if(openSetDij.includes(neighborDij)){
				if(tempGDij < neighborDij.g){
					neighborDij.g = tempGDij;
					newPathDij = true;
				}
			} else {
				neighborDij.g = tempGDij;
				openSetDij.push(neighborDij);
				newPathDij = true;
			}
```

```
                        if(newPathDij){
                                neighborDij.f = neighborDij.g;
                                neighborDij.previous = currentDij;
                        }
                }
        }

} else {
        console.log('No Solution');
        noSolDij = true;
}
//END DIJKSTRA


// A* OPPOSITE
if(openSetOpp.length > 0) {
        var winnerOpp = 0;
        for(var i=0;i<openSetOpp.length;i++){
                if(openSetOpp[i].f<openSetOpp[winnerOpp].f){
                        winnerOpp = i;
                }
        }
        var currentOpp = openSetOpp[winnerOpp];

        if(doneAOpp){
                distanceAOpp = PathDistance(pathOpp);
        }

        if(currentOpp===endOpp && !doneAOpp){
                //noLoop();
                doneAOpp = true;
                console.log("A* OPPOSITE DONE in "+stepsAOpp+" steps!");
        }

        if(!doneAOpp){
                removeFromArray(openSetOpp,currentOpp);
                closedSetOpp.push(currentOpp);
                stepsAOpp++;
        }

        var neighborsOpp = currentOpp.neighbors;

        for(var i=0;i<neighborsOpp.length;i++){
                var neighborOpp = neighborsOpp[i];

                if(!closedSetOpp.includes(neighborOpp) && !neighborOpp.wall){
                        var tempGOpp = currentOpp.g + 1;
```

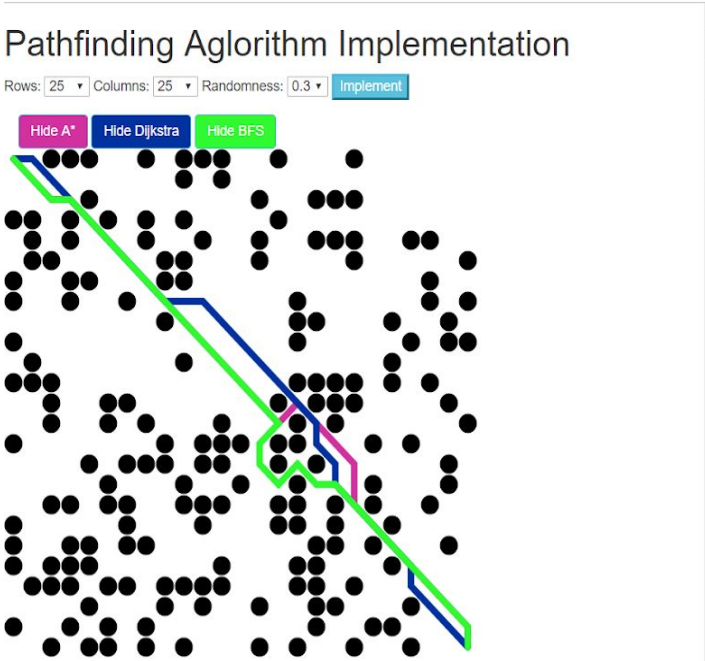
```
                        var newPathOpp = false;
                        if(openSetOpp.includes(neighborOpp)){
                                if(tempGOpp < neighborOpp.g){
                                        neighborOpp.g = tempGOpp;
                                        newPathOpp = true;
                                }
                        } else {
                                neighborOpp.g = tempGOpp;
                                openSetOpp.push(neighborOpp);
                                newPathOpp = true;
                        }

                        if(newPathOpp){
                                neighborOpp.h = heuristic(neighborOpp,endOpp);
                                neighborOpp.f = neighborOpp.g + neighborOpp.h;
                                neighborOpp.previous = currentOpp;
                        }
                }
        }

} else {
        console.log('No Solution');
        noSolAOpp = true;
        //noLoop();
        //return;
}

// END A* OPPOSITE


background(255);

// SHOWING THE GRID
for (var i=0;i<cols;i++){
        for(var j=0;j<rows;j++){
                grid[i][j].show(color(255));
        }
}

if(!noSolA){
        path = [];
        var temp = current;
        path.push(temp);
        while(temp.previous){
                path.push(temp.previous);
                temp = temp.previous;
```

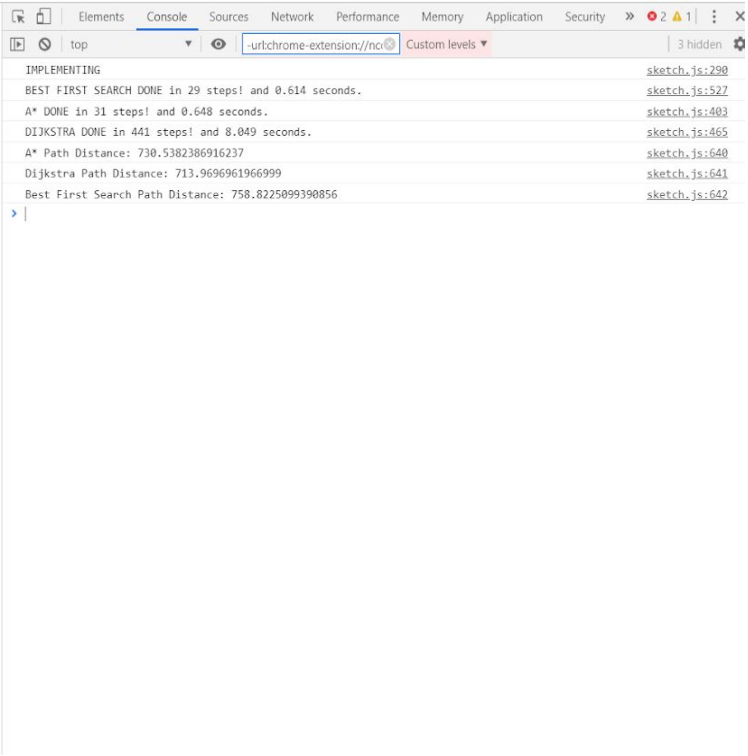
```
		}

		if(showA){
			noFill();
			stroke(210,50,160);
			strokeWeight(w/2);
			beginShape();
			for(var i=0;i<path.length;i++){
				vertex(path[i].i*w+w/2,path[i].j*h+h/2);
			}
			endShape();
		}
	}


	if(!noSolDij){
		pathDij = [];
		var tempDij = currentDij;
		pathDij.push(tempDij);
		while(tempDij.previous){
			pathDij.push(tempDij.previous);
			tempDij = tempDij.previous;
		}

		if(showDij){
			noFill();
			stroke(0,50,160);
			strokeWeight(w/2);
			beginShape();
			for(var i=0;i<pathDij.length;i++){
				vertex(pathDij[i].i*w+w/2,pathDij[i].j*h+h/2);
			}
			endShape();
		}

	}

	if(!noSolAOpp){
		pathOpp = [];
		var tempOpp = currentOpp;
		pathOpp.push(tempOpp);
		while(tempOpp.previous){
			pathOpp.push(tempOpp.previous);
			tempOpp = tempOpp.previous;
		}

		if(showAOpp){
```

```
                noFill();
                stroke(50,250,50);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<pathOpp.length;i++){
                        vertex(pathOpp[i].i*w+w/2,pathOpp[i].j*h+h/2);
                }
                endShape();
        }

        if(endLoop)
        {
                console.log("A* Path Distance: "+distanceA);
                console.log("Dijkstra Path Distance: "+distanceDij);
                console.log("A* OPP Path Distance: "+distanceAOpp);
                noLoop();
                finalPath = path;
                //path = [];
                finalPathDij = pathDij;
                //pathDij = [];
                finalPathOpp = pathOpp;
                //pathOpp = [];
                noFill();
                stroke(255,255,255);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<path.length;i++){
                        vertex(path[i].i*w+w/2,path[i].j*h+h/2);
                }
                endShape();

                noFill();
                stroke(255,255,255);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<pathDij.length;i++){
                        vertex(pathDij[i].i*w+w/2,pathDij[i].j*h+h/2);
                }
                endShape();

                noFill();
                stroke(255,255,255);
                strokeWeight(w/2);
                beginShape();
                for(var i=0;i<pathOpp.length;i++){
                        vertex(pathOpp[i].i*w+w/2,pathOpp[i].j*h+h/2);
                }
```

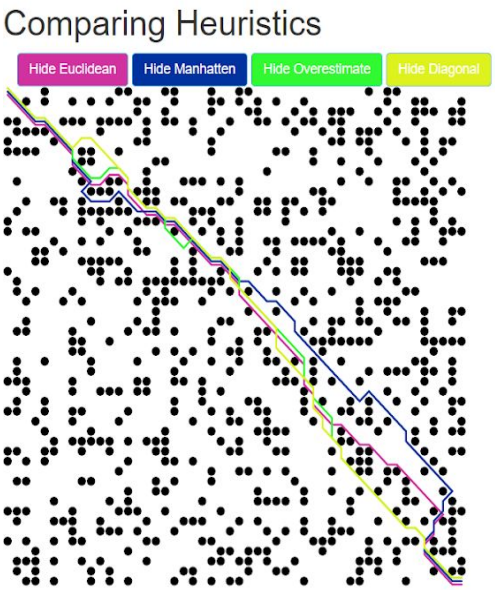
```
                        endShape();

                        showFinalPath();
                        return;
                }

                if(doneA && doneDij && doneAOpp){
                        endLoop = true;
                }
        }
}
```

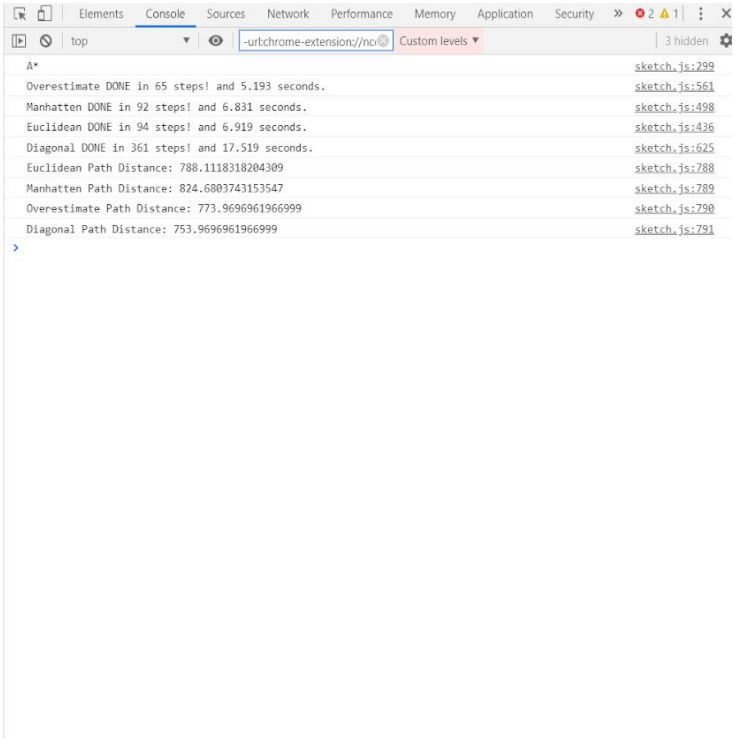
Result of comparing different pathfinding algorithms:





Result of comparing different heuristics:





5. Conclusions and Future Work

Based on the results of this research and evaluation conducted, it can be concluded that:

- *A*, Dijkstra's and Greedy Best First Search can be used to find the shortest path in maze.*
- *Dijkstra's algorithm gives the most optimal solution as the path found by it is the shortest of all though it takes a lot of time to find the solution.*
- *Greedy Best First Search is the fastest algorithm but its solution is not always the shortest path.*
- *A* search algorithm gives the solution close to that of the shortest path (Dijkstra's) but time taken by A* is very less than that of Dijkstra.*

- *Comparing A*, Dijkstra and Best First Search algorithms execution times in different size two dimensional grids, the slowest was Dijkstra and the fastest was Best First Search.*
- *A* is the best algorithm in pathfinding especially in Maze game / grids. This is supported by the minimal computing process needed and a relatively short searching time.*

Some suggestions that can be taken from this research are:

- *To continue this research with a modified A* algorithm to get better computation results.*
- *A*'s variants like Theta* , IDA* , HPA*, LPA* can be optimized further as per the requirements of the game.*
- *A* is still under research to find faster and more optimal algorithms.*
- *There are several space search methods such as navigation mesh which can be added to further optimize the algorithm.*

6. Code Repository

https://github.com/jhalak27/DAA-Project-

Active URL of the visualization for our code of different algorithms

Comparing various pathfinding algorithm: https://jhalak27.github.io/DAA-Project-/JS code/BestFS, A and Dijkstra/

Comparing various Heuristics: https://jhalak27.github.io/DAA-Project-/JS code/BestFS, A and Dijkstra/

7. References

- *https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games*
- *https://www.researchgate.net/publication/303369993_A_Review_on_Algorithms_for_Pathfinding_in_Computer_Games*
- *https://www.semanticscholar.org/paper/Investigation-of-the-*-(Star)-Search-Algorithms%3A-TI-Nosrati-Karimi/831ff239ba77b2a8eaed473ffbfa22d61b7f5d19*
- *https://ieeexplore.ieee.org/abstract/document/6095032/*
- *http://agris.fao.org/agris-search/search.do?recordID=LV2014000195*
- *https://p5js.org/ (library used to visualize the JavaScript code)*