International Journal of Innovative Technology and Interdisciplinary Sciences
www.IJITIS.org
ISSN:2613-7305
Volume 5, Issue 4, pp. 1033-1051, 2022
DOI: https://doi.org/10.15157/IJITIS.2022.5.4.1033-1051
Received October 13, 2022; Accepted November 24, 2022

On A* Graph Search Algorithm Heuristics Implementation Towards Efficient Path Planning in the Presence of Obstacles

Estela Pogaçe *[a1], Dimitrios A. Karras [b1,2]

[1]Canadian Institute of Technology, Tirana, Albania
[2] National and Kapodistrian University of Athens, Greece

*[a] estela.pogace@cit.edu.al; [b]dimitrios.karras@cit.edu.al

ABSTRACT

Nowadays it has become more and more important to reach a destination in a short time, in the shortest path between all the options and possibilities. This need is addressed by different search engines like google maps for example and it is common sense that the user is expecting the result in the least amount of time. The scope of this publication, is to help the reader understand the mechanism behind pathfinding algorithms integrated with heuristics and on how to choose between them in a given case study. Moreover, this paper aims at illustrating, after pathfinding algorithm selection, how to tweak and improve it in order to better fit the given setup scenario. With this respect, it will be shown how the A* algorithm computationally performs in a graph theoretic grid setup, initially in a small one and then, in a graph grid with a 10-fold increase of the initial setup dimensions. This experimental study compares two different heuristics in A* implementation, the Euclidean distance heuristic and the Chebyshev heuristic. Computational time results are compared with respect to the time taken to produce a final result in each case. Moreover, the total number of nodes involved in the path as well as the total cost estimated are considered per each case. These results are further compared with the ones derived when obstacles are introduced in the graph grid setup and how the algorithm will handle such scenarios is illustrated. This paper aims at providing information to researchers so that to understand what analysis needs to be done when selecting heuristics associated with pathfinding algorithms heuristics, and at providing relevant performance metrics regarding shortest path planning estimation between two predefined nodes in a graph grid setup. Moreover, aims at providing information on how the heuristics define the decision-making process in the A* algorithm and on how to weight the time/cost factors importance based on specific use cases.

Keywords: Search Algorithms; Pathfinding Algorithms; A* Algorithm; Heuristics; Euclidean Distance; Chebyshev Distance; Computational Time; Computational Performance; Graph Cost; Unity; C#.

1. INTRODUCTION

We live in the digitalized era and aim to make the world smaller and smaller every day by connecting through the network we know as the internet. It's this digitalization that has made it so easy for us to be everywhere at the same time, although virtually and if

needed, physically also. We may not know exactly how to reach an appointment location and almost by default we reach out for help to "Google Maps". Ever wondered about the logic behind all the path suggestions to arrive at the desired location, each of them takes a specific amount of time but the coloured route is the one that takes less time? Well, it's the search algorithms in the background making these results possible [1-4] These algorithms have been of interest for a long time and have continuously been improved, new ones have been released and each of them aims to target a specific flaw of the previous one [5-10]. The search needs to be swift and reliable, as it can be understandable, when you are searching for a location in "Google Maps" or from your "TomTom", you don't have the luxury to wait a long time until the results show up.

One extension of the search algorithms is "Pathfinding Algorithms" [1-10], which are built on top of graph search algorithms and explore the route between two destinations or nodes starting from point A and aiming to find the shortest and most reliable route to reach point B, comparing and deciding from all the different possibilities. The pathfinding algorithm we will be exploring is A* pathfinding [2] due to its nowadays popularity and combination of performance and accuracy

The A* method combines the actual cost from the starting point with a projected price to the endpoint to choose the next node to be evaluated. The estimated cost is determined by the heuristic function used by A*. Although being highly preferred in the realm of pathfinding algorithms, A* does not always produce the best results and generate the shortest path because it mainly relies on approximations and heuristics. A heuristic algorithm sacrifices precision and accuracy for speed in order to solve issues more quickly and effectively [4-6]. Every graph has many nodes or points that the algorithm must pass through in order to reach the target node. Each of these nodes' pathways has a numerical number, which is regarded as the path's weight. The cost of that route is calculated as the sum of all paths crossed.

Heuristics themselves might have their limitations in specific given scenarios and the individual should do a thorough research before deciding with which heuristic to feed the algorithm as in different scenarios, different heuristics perform better [4-13]

The goal of this paper is to present a comparative study in the performance of A* algorithm under heuristics and different distance metrics in different path finding scenarios, simulating realistic situations in the presence of obstacles. Although several studies exist in this field there has not been any relevant comparative analysis in the presence of obstacles, attempting realistic path finding conditions in the defined scenarios

2. HEURISTICS

Heuristic programming uses a practical approach that isn't always ideal, flawless, logical, or reasonable but is nevertheless sufficient for achieving a short-term objective. Heuristics are tactics developed from prior experiences with problems that are comparable. These methods focus on the use of easily available information to manage problem solving in machines and abstract problems [4-13].

2.1 Euclidean Distance Heuristic

The Euclidean distance would be the straight-line distance between two coordinates on a plane. Mathematically it would be calculated as:

If our coordinates are: A(x1, y1) and B(x2 , y2) then the Euclidean distance between the points A, N is

$$dist(A,B) = \sqrt{(x_A - x_B)^2+(y_A - y_B)^2} \quad (1)$$

2.2 Chebyshev Distance Heuristic

Another popular approach is the "Chebyshev Distance" which can be depicted as the movement of the King on a chessboard: it can go one step in any direction (up, down, left, right and verticals). The result of this is that the movement is as fast in a diagonal direction as it can be in a horizontal direction. Mathematically it would be given as:

$$d=\max(|xa\text{-}xb|,|ya\text{-}yb|) \quad (2)$$

2.3 Manhattan Distance Heuristic

The Manhattan distance calculates the distances between two vectors or points with real-valued coordinates. It is determined as the total of their Cartesian coordinates' absolute differences. Mathematically the Manhattan distance between two points p and q in an n-dimensional Euclidean space with coordinates (p1, p2,..., pn) and (q1, q2,..., qn) is given by:

$$|p1\text{-}q1|+|p2\text{-}q2|+\ldots\ldots+|pn\text{-}qn| \quad (3)$$

2.4 Comparison

Depending on the given grid, below are mentioned the criteria to choose a proper distance for the path finding heuristics applications. More specifically, it is proven that if we take in consideration a square grid:

- If only four directions of movement are allowed, Manhattan distance performs better
- If any direction is allowed, Euclidean distance is preferred.
- If eight directions are allowed, the Chebyshev distance is preferred

This paper attempts to deeply dive and compare the Euclidean Distance with the Chebyshev heuristics, as they are seemingly a little less limited than the Manhattan distance. There have been several comparisons in the literature [14-17] but not with this perspective and especially not in the context of graphs with non-walkable nodes, simulating obstacles in real world. This is the main novelty of this paper.

3. A* PATHFINDING ALGORITHM IMPLEMENTATION IN C#

In the comparison scenarios investigated in this paper, there will be no difference in the implementation of the source code regarding Euclidean and Chebyshev distances based heuristic cases, other than the suitable implementation and integration of the relevant heuristics in the context , however, of the grids and scenarios requirements. The implementation in this paper is based on C# following the same path as several other implementations in the literature [18-28].

In both cases we will need a PathNode class, which will hold all the required information for a node in the grid together with a reference to the grid itself. To explain each variable comment have been added in the code, following all rules of refactoring in programming, since authors plan is to make this code public, when finalized. As we have explained in the previous sections, as we are going to use the A* algorithm, we need to hold information about the coordinates of the nodes, which will be represented by our x and y variables, the gCost which is the cost of movement, the hCost which is the heuristic cost and the final f cost which will hold the information of gCost+hCost.

Since the A* algorithm, recreates the path from all the walked nodes until having reached the goal node, we need to keep a reference to the previous node for each current node, this information will be held in the variable: public PathNode previousNode;. Later on, in the implementation we will be discussing about the behaviour of the algorithm after introducing obstacles, in simple words, these obstacles will be nothing else but "unwalkable" nodes, which will be initialized at the start of the program.

The information whether a node is "walkable" or not, will be registered by the flag Boolean variable in the class, public bool isWalkable;

If the node is unwalkable it will be directly added to the closed list, removed from the open list of nodes to be explored and will not be considered anymore by the algorithm.

If the node is unwalkable it will be directly added to the closed list, removed from the open list of nodes to be explored and will not be considered anymore by the algorithm.

```
using System.Collections;
using System.Collections.Generic; using UnityEngine;

public class PathNode {
private Grid<PathNode> grid;
//x coordinate public int x;
//y coordinate public int y;
/** As we have previously discussed
* The function value "f", which is used by A*, is defined as
f(n)=g(n)+h(n)
As a result we will hold this information for each node*/
//g(n) value public int gCost;
//h(n) value public int hCost;
//f(n) value public int fCost;
/**W will also hold a reference to the previous node in order to take note of the path we have been traversing*/

public PathNode previousNode; public bool isWalkable;

public PathNode(Grid<PathNode> grid, int x, int y)
 { this.grid = grid;
this.x = x; this.y = y;
this.isWalkable = true;
}
public void SetIsWalkable(bool isWalkable) { this.isWalkable = isWalkable;
}
```

```
public void calculateFCost() { fCost = gCost + hCost;
}
public override string ToString() { return x + "," + y;
}
}
```

Let's continue with the implementation of our main class, where we will be implementing the A* Algorithm.

As a start we declare all the global variables that we will need throughout the implementation, therefore, we need an openList to hold all our nodes to explore and a global variable that holds all the nodes that have been explored in our case the closedList. If a node is present in the closed list, we don't take that into consideration anymore. Having said so, all our unwalkable nodes end up in the closed list.

We then need a global variable for the grid we will walk and a global variable that will hold the start time of the main functionality (FindPath) and a global variable that will hold the end time of the same functionality. This time will be in milliseconds and the reason we need to hold this information globally is because it will be logged in the console later on in our Testing class, this way we will be able to compare the timing results while using both heuristics.

Afterward we initialize the start node and the end node according to the information we have received as parameters in the FindPath functionality, then we initialize the open list with the start node while the closed list remains empty. As another step we need to initialize our grid nodes, setting the gCost to infinite and then calculating their fCost also, the previousNode is initialized to null since for the moment we don't have any data for the previous path.

Following the above steps, then, we initialize our start node, with the gCost set to 0 since we start right off here, with the h cost equals to the result of the functionality that uses a heuristic to produce a cost "calculateDistanceHeuristic" and the f cost =gCost+hCost.

We will open a cycle and continue to cycle up until the open list has nodes. We will set the current node to be the one with the lower fCost. To find out which is the node with the lowest fCost, we will need another functionality which we will name "getLowestFCostNode" which takes as a parameter the open list, and basically using the same logic to find out the minimum value of a vector, defines the node with the lowest fCost.

As a first thing we do, in the beginning of this loop, is check whether we have reached our goal node, and if so we return the calculated path that we took to reach that goal node, using the functionality the functionality "calculatedPath". This functionality returns a list of nodes and takes as a parameter the end node. But why the end node? As we have specified in the beginning, we hold reference to the previous node we walked, so now we are going to form back the path we took to reach the end node by tracing back the previous ones up until the previous node is not null meaning we have a previous node still. We are going to add the previous node to our final list of nodes, and set the current node with the value of the previous node, this is the logic behind our cycle.

After returning from our calculatedPath functionality, we remove the current node from the open list, indicating that it has already been searched and then add it to the closed list.

Then we need to cycle through the neighbours of the current node, for this we are going to build and use the functionality "getNeighborNodeList". We start off by taking as a parameter the current node and first of all check the node which is left of it if its valid meaning if we subtract to the current node x coordinate the value 1 , the result should be above or equals to 0, we add it to the list, then continue checking the node left down to it if its valid, meaning that if we subtract to the y value of the current node, the value 1, the result should be above or equals to 0. Then continue to search the left-up value, which is valid only if by adding 1 to the y value, the resulting value is smaller than the height of the grid. The same logic but reversed is followed to find and validate the neighbours on the right hand of the grid. Just like this we have our eight neighbour positions if they are valid.

After this operation we can start cycling through the neighbours of the current node. The first check we need to do is to see if the neighbour node is already on the closed list, and if it is, that means we have already examined it, so we simply continue.

Then we need to check if we have a smaller gCost from the current node to the neighbour node, than we had previously. This is why we need to have a variable holding a temporary gCost=currentNode.gCost + the calculated cost from the heuristic function to find out the distance from the current node to the neighbour node. If we do have a better cost, then, we save it by setting as the previous node of the neighbour our current node, then, we set our neighbour gCost to be our newly calculated temporary gCost, and set the hCost to be the distance between the neighbour node to the goal node and finally we set the fCost. Finally, we check, if the neighbour node is not already on our openList, and in this case we need to add it.

We have reached the end of our complete pathfinding algorithm implementation. As the last step, when being outside of the while loop, essentially this means that we have searched through the whole map and we could not find a path, in this case we simply return null.

3.1 Chebyshev distance heuristic mathematical formula interpreted in code and execution results

Lets start first off by translating into code the heuristic.

As we know, mathematically the Chebyshev Distance would be:

$$dist(A,B) = \max(|x_A - x_B|, |y_A - y_B|)$$ which in code would mean:

```
private int calculateDistanceHeuristic(PathNode a, PathNode b)
  {
     int xDistance = Mathf.Abs(a.x - b.x);
     int yDistance = Mathf.Abs(a.y - b.y);
     int remaining = Mathf.Abs(xDistance - yDistance);
     return  Mathf.Max(xDistance, yDistance);
  }
```

where PathNode a and PathNode b are two nodes that are being compared in a moment of the runtime of the program. The grid used in the aforementioned experiments is seen in Figure 1. The visual result after running the code would be shown in the Figure 2.

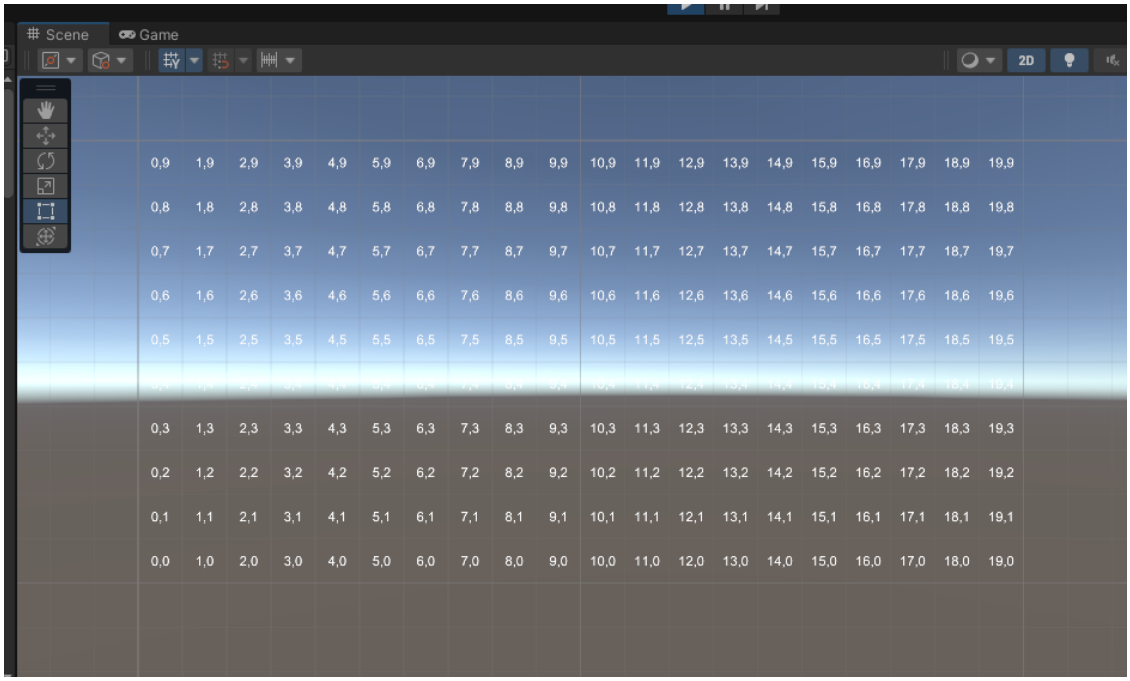

Figure 1. The grid involved in this paper.

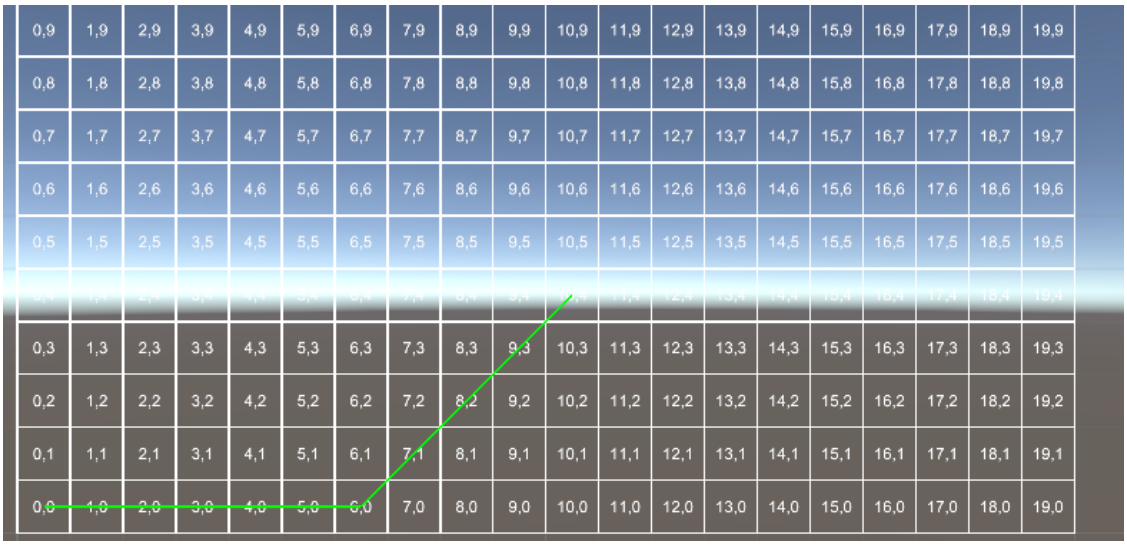

Figure 2. Chebyshev distance-based path following.

With a cost of 110 and number of walked nodes 11 as displayed by the system in the Figure 3.

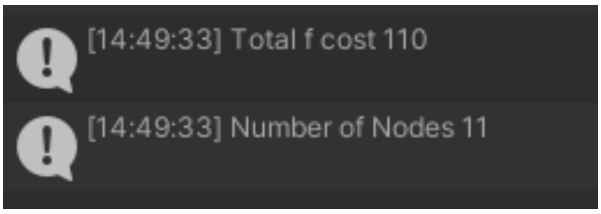

Figure 3: Chebyshev Distance Cost/Nodes result

3.2 Euclidean distance heuristic mathematical formula interpreted in code and execution results

As already mentioned, mathematically the Euclidean Heuristic would be represented as equation (1) where in C# code this equation would be translated as follows:

```
private int calculateDistanceHeuristic(PathNode a, PathNode b)
{
    int xDifference = a.x - b.x;
    int yDifference = a.y - b.y;

    int xSquare = (int)Math.Round(Mathf.Pow(xDifference, 2));
    int ySquare = (int)Math.Round(Mathf.Pow(yDifference, 2));

    int result = (int)Math.Round(Mathf.Sqrt(xSquare + ySquare));

    return result;

}
```

The reason why we have rounded the results is for facility in calculating the grid nodes. In this case, the visual result of the path would be as in Figure 4.

Figure 4. Euclidean distance-based path following.

With a cost of 111 and number of waked nodes 11 as in the following system message. Figure 5 depict Euclidean distance cost/nodes results

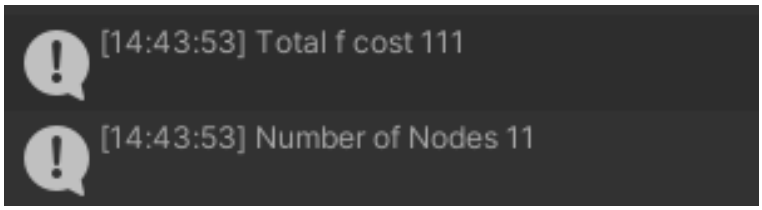

Figure 5: Euclidean Distance Cost/Nodes result

One conclusion we can draw from this test is that in the given conditions, the Euclidian heuristic is noticeably less costly but is equals in terms of number of path nodes.

Up until now we have explored the case when there are no obstacles in our grid, but what happens when there are some nodes which cannot be walked. How would our algorithm handle it and what would the impact be on the cost and number of nodes traveled? Would it necessarily mean that the cost will increase if we find obstacles in our way? These are some questions that we will try to answer on our upcoming section.

4. A* PATHFINDING ALGORITHM IMPLEMENTATION IN C# USING HEURISTICS WHEN OBSTACLES ARE INTRODUCED IN THE GRID

In order to continue the comparison between these two aforementioned heuristics, we will introduce obstacles. Introducing obstacles means nothing else rather than making some nodes in the grid “unwalkable”. One sensible and easy way to do so would be to introduce a variable in our PathNode class “isWalkable” which will be our flag that states if our algorithm will consider this node or not.

This node by default will be set to “true” and in some strategic cases, based on the previous result, it will be set to false. Two coordinates that have previously been considered per each heuristic will be set as “unwalkable”.

As we have previously done to create and initialize the grid in the testing class, the same method will be used to initialize the nodes that will not be walkable.

4.1: A* Pathfinding with Obstacles using Chebyshev Distance heuristic

From the results of the previous section, we noticed that two coordinates that were taken into consideration were (7,1) and (8,2).

These nodes in the sequel will be made “unwalkable”. In other words, we have put an obstacle, that is, we blacked out at (7,1) and (8,2).

The way how we could do this operation is by introducing in the Testing class two lines such as:

```
pathfinding.getNode(7,1).SetIsWalkable(false);
pathfinding.getNode(8,2).SetIsWalkable(false);
```

Figure 6 and 7 depicts respectively Chebyshev distance path with obstacles and Chebyshev distance cost/nodes result with obstacles

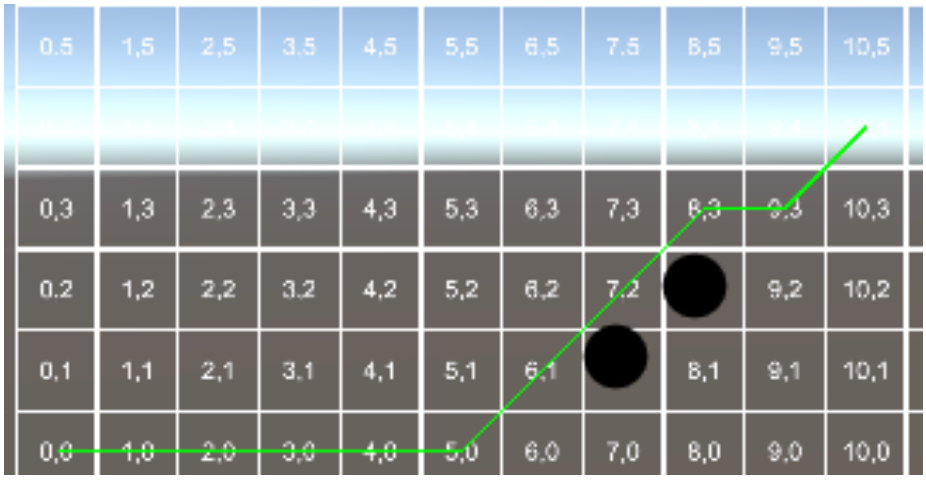

Figure 6. Chebyshev Distance Path with obstacles

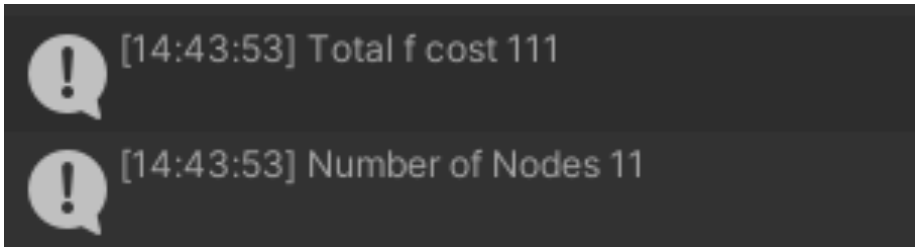

Figure 7. Chebyshev Distance Cost/Nodes result with obstacles

Here we notice something interesting in terms of cost and nodes. Compared with the previous result where the total cost was equals to 110, now the total f cost remained constant to the value of 110. It seems as if there is a linear dependability between the number of walked nodes and the cost, meaning, the cost increases if the number of walked nodes increases.

4.2: A* Pathfinding with Obstacles using Euclidean Distance heuristic

Again, following the same logic, based on the results in the previous section, we noticed that two coordinates that were taken into consideration were (2,2) and (3,2). These nodes are going to be made “unwalkable”, in other words, an obstacle will be put blacking out at (2,2) and (3,2). The way how this operation could be done is by introducing in the Testing class two lines such as

```
pathfinding.getNode(2,2).SetIsWalkable(false);
pathfinding.getNode(3,2).SetIsWalkable(false);
```

The new visual result would be as in the Figure 8 below. Figure 9 depicts the Euclidean distance cost/nodes result with obstacles

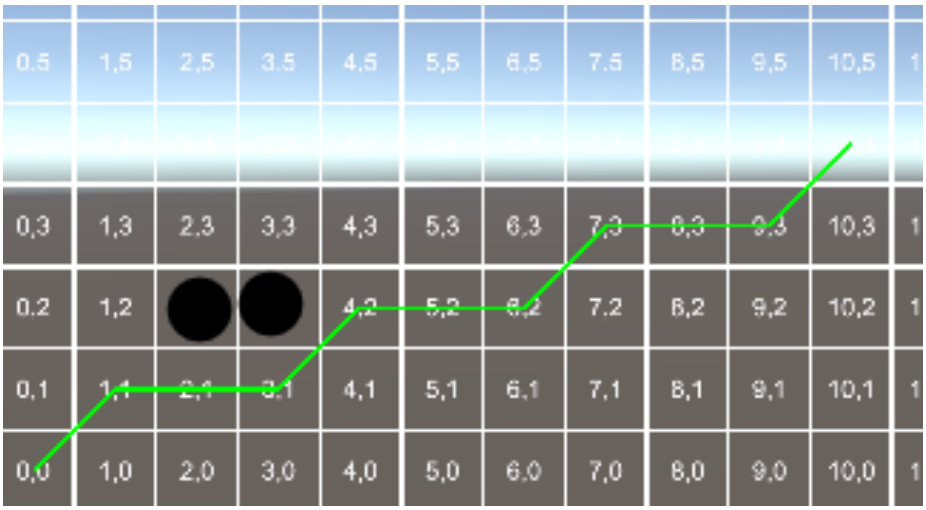

Figure 8. Euclidean Distance Path with obstacles

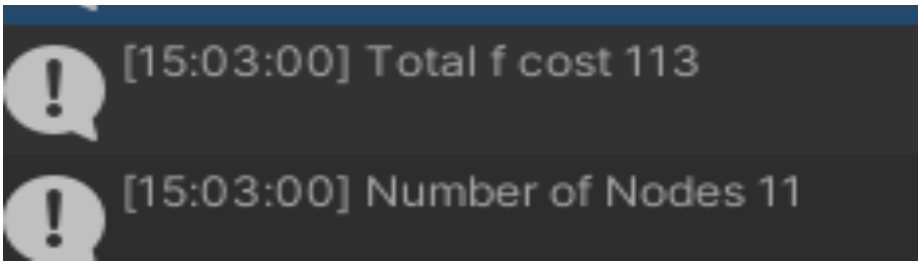

Figure 9. Euclidean Distance Cost/Nodes result with obstacles

When using the Euclidean distance heuristic, we notice a different pattern of behaviour in terms of cost and nodes. Compared with the previous result where the total cost was equals to 111 for 11 nodes, now the total f cost has increased to the value of 113 again following the same number of nodes. Which means, the cost is independent from the number of nodes. This kind of analysis needs to be taken into consideration when deciding between the heuristics that are to be chosen in a given scenario.

In our case, for the given conditions, the Chebyshev distance heuristic performed better in both cases, and we, also, saw that it produced better results even when introducing obstacles.

A very important metric to take into consideration is the timewise performance of the algorithm using different heuristics, and not only this, but how does the application behave when we take into consideration a larger graph. Would this affect the performance of the algorithm? Let's proceed in the next section with answering these questions.

5. A* PATHFINDING ALGORITHM: TIMEWISE PERFORMANCE IN A SMALL AND A LARGER GRAPH

The graph we have taken into consideration until now is a small one, with 20 nodes wide and 10 nodes high. So, for the small graph observation we will continue to use the example we considered until now. To measure the timing cost, we have introduced one global variable named:

public double totalMilliseconds = 0; , which will hold the value:

ts = endTime - startTime; which is calculated inside the FindPath functionality (see previous section).

That is, timestamp of end of the algorithm process – timestamp of start of the algorithm process. Its value will be logged in the console in the Testing class from the line :

Debug.Log("Total execution time in milliseconds " + pathfinding.totalMilliseconds);

5.1 Euclidean Heuristic vs Chebyshev Heuristic timewise performance without obstacles, in a small graph of 20 nodes width and 10 nodes length

When enabling the Euclidean heuristic and disabling the Chebyshev heuristic, after running the application we calculate a value of runtime equals to 2.352 milliseconds, see Figure 10:

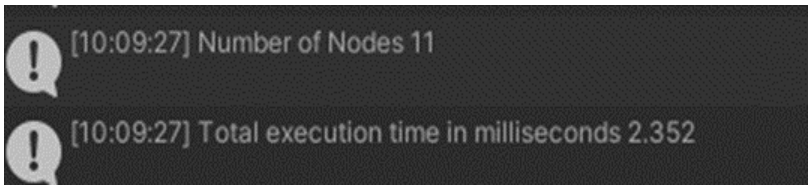

Figure 10. Euclidean Distance time result

When enabling the Chebyshev heuristic and disabling the Euclidean heuristic, after running the application we calculate a value of runtime equal to 2.9958 milliseconds, see Figure 11:

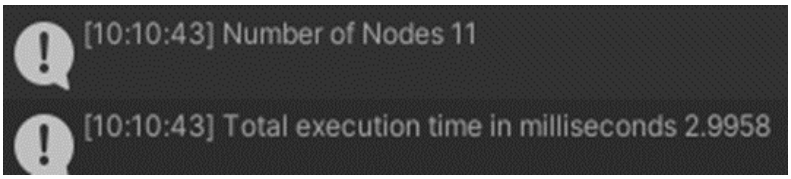

Figure 11. Chebyshev Distance time result

Interesting enough is that when we compare these timing results, with the results when we introduce obstacles, we see that in terms of total cost, we have a better cost in terms of fCost if the path does not contain obstacles but we have a worse timing with approximately 0.5 milliseconds.

5.2 Euclidean Heuristic vs Chebyshev Heuristic timewise performance without obstacles, in a larger graph of 200 nodes width and 100 nodes length

When enabling the Euclidean heuristic and disabling the Chebyshev heuristic, after running the application we calculate a value of runtime equals to 2.2915 milliseconds as in the system message below, see Figure 12.

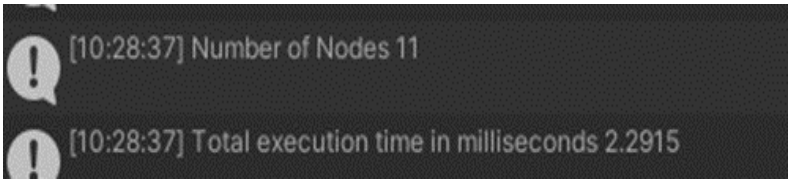

Figure 12. Euclidean Distance time result (larger graph)

When enabling the Chebyshev heuristic and disabling the Euclidean heuristic, after running the application we calculate a value of runtime equals to 2.9992 milliseconds as shown in the system message below, see Figure 13.

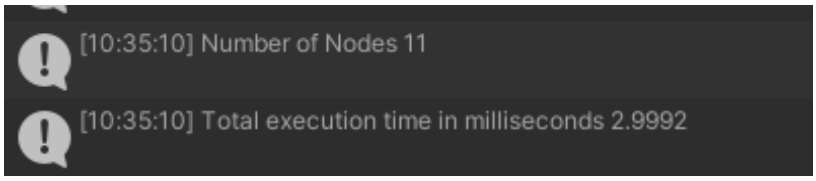

Figure 13. Chebyshev Distance time result (larger graph)

The result is that no matter how much we enlarge the graph, based on the theory of the A* algorithm, when we find a path from the start node to the end node, we return it without scanning the whole graph as we will analyze next too.

5.3 Euclidean Heuristic vs Chebyshev Heuristic timewise performance with obstacles, in a small graph of 20 nodes width and 10 nodes length

When enabling the Euclidean heuristic and disabling the Chebyshev heuristic, after running the application we calculate a value of runtime equals to 1.6632 milliseconds as can be seen in Figure 14.

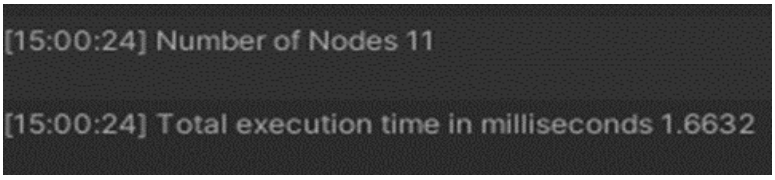

Figure 14. Euclidean Distance time result with obstacles

When enabling the Chebyshev heuristic and disabling the Euclidean heuristic, after running the application we calculate a value of runtime equals to 2.0521 milliseconds as can be seen in Figure 15.

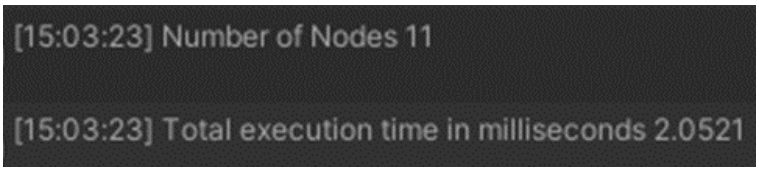

Figure 15. Chebyshev Distance time result with obstacles

We understand clearly from these results, that even though the Euclidean heuristic is more costly for the algorithm, it is notably more performant timewise, exactly 0.3889 milliseconds. So, depending on the case study, for example when the user might need to find the direction to somewhere, time would be an important factor into deciding which heuristic to choose from, and in our case the Euclidean heuristic is better performant.

5.4 Euclidean Heuristic vs Chebyshev Heuristic timewise performance with obstacles, in a larger graph of 200 nodes width and 100 nodes length

Let's say we increase the graph width and height ten times, in order to become 200 nodes width and 100 nodes height, how would it affect our performance using each heuristic? When enabling the Euclidean heuristic and disabling the Chebyshev heuristic,

after running the application we calculate a value of runtime equals to 1.9938 milliseconds, see Figure 16.

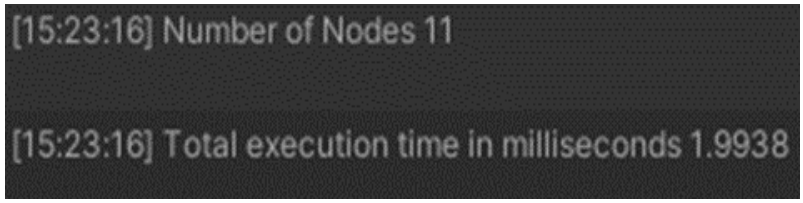

Figure 16. Euclidean Distance time result with obstacles (larger graph)

On the other hand, when enabling the Chebyshev heuristic and disabling the Euclidean heuristic, after running the application we calculate a value of runtime equals to 2.0386 milliseconds, see Figure 17.

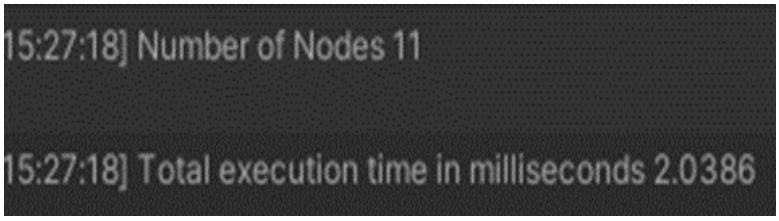

Figure 17. Chebyshev Distance time result with obstacles (larger graph)

What we understand from this result, is that the algorithm timing is not deteriorated the more we increase the graph dimensions, but why so, wouldn't it be more difficult to find the path in a larger graph, since we have more neighbors to examine? Finally, this answer is no, in fact, and this is where the A* pathfinding algorithm is more performant than other relevant ones, especially the ones more preferred than Dijkstra algorithm, from where it has its roots. Let's go back to previous sections where we have explicitly said that "Another drawback of the A* algorithm is that It is not optimal as it does not explore all the paths once it finds a solution.". Furthermore, our algorithm found the optimal path comparing the close neighbors until he goals node, it returned the final path. We can increase the graph dimensions as much as we want, but the time to find path until the goal node is going to be the same as we always scan the same neighbors and not the whole graph, as we would have done with the Dijkstra algorithm for example. Dijkstra algorithm is obviously an optimal search path algorithm while A* is only a suboptimal path finding algorithm based on heuristics.

5.5 Result Summary tables

In the Tables 1-4 below all results obtained through the experimental study of the paper are summarized as follows.

Table 1. Heuristic in Small Graph without Obstacles

Heuristic	Number of nodes	Time in ms	Number of Nodes2	Fcost
Euclidean	200	2.352	11	111
Chebyshev	200	2.9958	11	110

Table 2. Heuristic in Larger Graph without Obstacles

Heuristic	Number of nodes	Time in ms	Number of Nodes2	Fcost
Euclidean	2000	2.2915	11	111
Chebyshev	2000	2.9992	11	110

Table 3. Heuristic in Small Graph with Obstacles

Heuristic	Number of nodes	Time in ms	Number of Nodes2	Fcost
Euclidean	200	1.663	11	113
Chebyshev	200	2.0521	11	111

Table 4. Heuristic in Larger Graph with Obstacles

Heuristic	Number of nodes	Time in ms	Number of Nodes2	Fcost
Euclidean	2000	1.9938	11	113
Chebyshev	2000	2.0386	11	111

6. DISCUSSION AND CONCLUSION

Nowadays, in this very small world, it has become more and more important to be connected with each other through different networks. It has become more and more important to be able to reach each other virtually and physically. Scientists for years have been researching ways to shorten distances between two points but until now, no luck in any technology tested. There are even movies dedicated to this topic which would suggest that the only way to shorten a distance between two points, would be to bend the surface that connects them. Well, it is no wonder why up until now, humanity has had no luck in doing so.

Even though we cannot bend surfaces, we can mitigate the issue by exploring the shortest path that connects one starting point and one goal point, being mindful of the fact that we still need to keep a very important factor in mind which is time. The least time-consuming it is to reach a goal point, the better it is for us, and anyone means to pursue the fastest solution.

Computer scientists and mathematicians joining forces together have been researching and improving algorithms that help solve this issue. Some of these algorithms are more time consuming but produce a more reliable result and some others are notably faster but you are not so sure that the path result between point a and b is going to be the shortest, especially when there are obstacles introduced. Hence when developing an application, or designing a network, or in any other real-life scenario where we need to provide the shortest-path solution it is essential that we take the factor time and reliability into consideration. Sometimes it might just suffice to scan the closest neighbour nodes up until the goal node, and sometimes we might need to scan the whole graph taken into consideration in order to produce a result.

For example, let's say you are driving somewhere, from city "Alice" to city "Bob" you are using google maps and the destination is "Bob", imagine how much time it would take for the application to scan the whole map of the world and produce a result, even though these cities are located in the same state. It is just sensible in this case that

there is no need to scan the whole map (which to parallelize with our terminology would be our graph) in order to produce a result, but scanning the closest neighbours up until the destination would be enough.

This is one part of the analysis we need to do when choosing the algorithms to help us find a suitable path. Starting from this, in the last decade there have been made considerable improvements to algorithms of this nature. During the scope of this paper, we have explored the different search algorithms and their subcategories. We have explained pathfinding algorithms and scratched the surface of each type focusing on A* algorithm. There is a reason why we chose the A* one.

The A* algorithm, is a combination of efficiency and performance. It uses the logic of heuristics, the logic behind the algorithm brain, to help make a decision on which node to choose next, compared to the neighbours involved and the previous node. Before choosing which heuristic to use, we need to be mindful of which heuristic performs better in each case. The heuristic is nothing else than a mathematical expression which in our case will be translated in code, that produces a result value.

There are different mathematical expressions that calculate the distance between two points in a plan. Classical examples would be the Euclidean distance and the Manhattan distance. Another interesting one, but not so popular as the previous two, is the Chebyshev distance. It has been researched and studied, that the Manhattan distance heuristic produces a better result when the object is moving in four directions, the Chebyshev distance heuristic performs better when the object moves in eight directions and lastly the Euclidean distance allows the object to move in any direction. When choosing between them, it is understandable that if the object needs to move in more than four directions, as was our case study, then the Manhattan distance heuristic is the one to be dropped from the list.

When we implemented the A* pathfinding algorithm, our aim has been to produce a comparison result for graphs of different scales with the shortest number of nodes walked, with the lowest cost and with the best performance timewise. To this end, we explored the results when using Euclidean heuristic and then, when using the Chebyshev heuristic, by enabling and disabling each one at a time. To be confident in the results produced, we enlarged the graph ten times for both dimensions and analysed the new results. Taking these into consideration different scenarios we then compared both cases.

One interesting fact was common in both cases, that the timewise performance did not change considerably when we increased the dimensions ten times. This is because what we have said when we explained theoretically the A* pathfinding algorithm, that even though we might increase the graph surface, it will scan only the closest neighbours from the starting point to the goal point, finding the less costly result and then returning it. So, in both cases the surface scanned has been the same.

We notice that the number of walked nodes from point A to point B is the same for both heuristics, eleven to be exact, but the F cost is larger when using Euclidean heuristic. Although, we notice that the Euclidean heuristic is more performant timewise in all the cases taken into consideration. So, in a scenario when time is more important, the latest would have been chosen for the implementation. A rather interesting fact is that when we introduce obstacles, the number of walked nodes increases with two units in the Euclidean heuristic and one unit in the Chebyshev heuristic, but surprisingly the

execution time decreases, suggesting that the algorithm returns the first less costly result found even though timewise is not the fastest to be found.

To summarize our results, if we were to choose the heuristic based on the F cost, the Chebyshev heuristic would have won the competition, while, if we were to choose based on time performance when integrated in the algorithm, the Euclidean heuristic would have won. We have provided these results, for the user to choose wisely between them depending on the case study, after carefully weighting the importance of time versus the importance of the F cost.

ACKNOWLEDGMENT

A very special thank you goes to professor Dimitrios Karras who has supported me for this interesting research work. I hope the results are good enough to be on his level of professionalism and dedication.

CONFLICT OF INTERESTS

The authors would like to confirm that there is no conflict of interests associated with this publication and there is no financial fund for this work that can affect the research outcomes.

REFERENCES

[1] Felner A., Li J., Boyarski E., Ma H., Cohen L., Satish Kumar T.K. and Koenig S. Adding Heuristics to Conflict-Based Search for Multi- Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 2018; 28(1); 83-87.

[2] Foead D., Ghifari A., Kusuma M.B., Hanafiah N. and Eric Gunawan E. A Systematic Literature Review of A* Pathfinding. *Procedia Computer Science*, 2021; 179; 507-514.

[3] Harabor D. and Grastien A. (2011). Online Graph Pruning for Pathfinding on Grid Maps. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011; p. 1114-1119.

[4] Atzmon D., Li J., Felner A., Nachmani E., Shperberg S., Sturtevant N. and Koenig S. Multi-Directional Heuristic Search. *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-20)*, 2020; 4062-4068.

[5] Mathew G.E. Direction Based Heuristic for Pathfinding in Video Games. *Procedia Computer Science*, 2015; 47; 262-271.

[6] Elizondo-Leal J.C., Parra-González E.F., Ramírez-Torres J.G. The Exact Euclidean Distance Transform: A New Algorithm for Universal Path Planning. *International Journal of Advanced Robotic Systems*, 2013; 10(6); 1-10.

[7] Iqbal M.A., Panwar H., Singh S.P. Design and Implementation of Pathfinding Algorithms in Unity 3D. International Journal for Research in Applied Science and Engineering Technology, 2022; 10(4); 71-79.

[8] Shetty V.S.S., Shwetha R. Heuristic Solution for Optimized Path Finding in Smart Parking System. *International Journal of Engineering Research & Technology*, 2018; 6(15); 1-4.

[9] Suni Y.J., Ding X.Q., Jlang L.N. Heuristic Pathfinding Algorithm Based on Dijkstra. *3rd Annual International Conference on Electronics, Electrical Engineering and Information Science*, 2017; p. 421-425.

[10] Algfoor Z.A., Sunar M.S., Kolivand H. A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. International Journal of Computer Games Technology, 2015; 1-11.

[11] Randall M. (2020) Heuristic Search Part 1: Introduction and Basic Search Methods. Lancaster University. UK

[12] Torralba A. and Croitoru C. (2018) Comparing Heuristic Functions, Saarland University. Germany

[13] Cui X. and Shi H. A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security*, 2011; 11(1); 125-130.

[14] Bjornsson Y. and Halldorsson K. (2006) Improved Heuristics for Optimal Pathfinding on Game Maps. Reykjavik University. Island.

[15] Sidhu H.K. (2020) Performance Evaluation of Pathfinding Algorithms. (2020) Electronic Theses and Dissertations. University of Windsor. Canada.

[16] Östberg O. (2021) A comparison of algorithms for the purpose of path-finding in a 3D grid. Blekinge Institute of Technology. Sweden.

[17] Eiselt H.A., Sandblom C.L, Spielberg K., Richards E., Smith B.T., Laporte G. and Boffey B.T. (2000) Integer Programming and Network Models, 1st edition. Springer, Germany.

[18] Brainkart. A* Search: Concept, Algorithm, Implementation, Advantages, Disadvantages. Available at https://www.brainkart.com/. Accessed on 12 September 2022.

[19] Chowdera. Unity grid system (2) thermal diagram in Grid. Available at https://chowdera.com/. Accessed on 12 September 2022.

[20] Dot Net Core Tutorials. A* Search PathFinding Algorithm In C#. Available at https://dotnetcoretutorials.com/. Accessed on 10 September 2022.

[21] Free Code Camp. Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction. Available at https://www.freecodecamp.org/. Accessed on 18 September 2022.

[22] Growing with the web. A*pathfinding algorithm. Available at https://www.growingwiththeweb.com/. Accessed on 18 September 2022.

[23] Neo4j. Graph Search Algorithms. Available at https://neo4j.com/. Accessed on 20 September 2022.

[24] Towards Data Science. 3 distances that every data scientist should know. Available at https://towardsdatascience.com/. Accessed on 15 September 2022

[25] Unity Documentation. Creating a 2D game. Available at https://docs.unity3d.com/. Accessed on 15 September 2022

[26] Koçiaj I. An Overview Methodology for Writing Suitable Boolean Rules for Protein Signaling Pathways. *International Journal of Innovative Technology and Interdisciplinary Sciences*, 2021; 4(2); 691–705.

[27] Unity Documentation. Grid (User Manual on the basics of a grid in unity). Available at https://docs.unity3d.com/. Accessed on 15 September 2022.

[28] Unity Documentation. Navigation and Pathfinding. Available at https://docs.unity3d.com/. Accessed on 15 September 2022.