INTERNATIONAL JOURNAL OF Telecommunications AIR DEFENSE COLLEGE

Enhancing the Hexagon Path Planning Algorithm for Dense Obstacle Environments (Iterative Hexagon Algorithm)

Citation: Arnaoot , H.; Abdin , H.; *Inter. Jour. of Telecommunications, **IJT'2024**, Vol. 04, Issue 02, pp. 1-10, 2024.*

Editor-in-Chief: Youssef Fayed.

Received: 26/05/2024.

Accepted: date 15/07/2024.
Published: date 30/07/2024.

Hany Arnaoot, Hesham A.Abdin

* Correspondence: hanyarnaoot@yahoo.com; Tel.: (+2 01068485694)
Contributing authors: hesham.ahmed@aiet.edu.eg

Abstract: This study intends to improve the Hexagon Path Planning algorithm's application in dense obstacle situations by reducing computation time, allowing for real-time path planning with a dynamic search area that starts small and grows until a path is identified (iterative Hexagon visibility Graph). To evaluate the performance of the suggested algorithm, a novel assessment approach is provided that takes into account a large number of mazes rather than just a few. The key to the proposed algorithm's efficiency comes in the use of iterative Hexagon visibility Graph, applying rules to identify and hence reject paths likely to be long, thus saving time and compute resources, prioritizing nodes with the likelihood of being part of the solution. This study also introduced the concept of checking Node-to-node visibility as needed, rather than a creating a complete visibility map, and then solve. This smart search reduces the computational load caused by node-to-node visibility checks complexity which increase by increasing the nodes number greatly. The proposed algorithm can find the shortest path or indicate that there isn't one. It successfully completed mazes with 250 rectangular obstacles (equal to 2000 different nodes). Notably, the proposed methodology saved computing time greatly when compared to the usual method of building a visibility graph and then solving it using the Dijkstra algorithm: by roughly 45.56% without filter application and an astonishing 95.85% when a hexagon filter was used.

Keywords: path plan, graph theory, geometrical algorithm, visibility graph, Autonomous Navigation, roadmap, swarm path planning, Dijkstra, Hexagon algorithm.

1. INTRODUCTION

Autonomous robots/vehicles number and applications increased rapidly Over the past decade e.g. [1], [2], [3], [4], [5]. A basic component of autonomous robots/vehicle control system is the determination of a collision-free path in a given environment, which is known as path planning.
path planning algorithm (whether it's local or global path planning). In autonomous navigation, path planning is a major problem, the problem is more complicated when dealing with unknown or partially known environments.
The following are examples of common approaches used to create a geometric path[6],[7],[8]:
Heuristic-based methods are known as:

- Ant Colony Optimization (ACO);
- Particle Swarm Optimization (PSO);
- Genetic Algorithm (GA);
- Neural Network (NN);

Classical methods are known as:

- Subgoal Network (SN);

- Road Map;
- Potential Field (PF);
- Cell Decomposition (CD);

2. RESEARCH GAP

This study seeks to improve the Hexagon Path Planning algorithm (a visibility graph path plan algorithm that can determine the shortest path or indicate whether a valid solution does not exist) utilized in dense obstacle environments. This is accomplished by lowering calculation time, allowing for real-time path planning in a dense obstacle area. The main contribution of this work is to obtain a significant reduction in computation time when compared to the Hexagon algorithm. This is accomplished by imitating the human style of maze solving by examining the nodes that are more likely to be part of the solution before other nodes, as opposed to the typical method of constructing a visibility map and then solving the Dijkstra algorithm, which needs all nodes' distances and visibility statuses checked, which typically includes evaluating unnecessary nodes that are not part of the solution.

3. THE PROPOSED ALGORITHM

the hexagon visibility graph algorithm [9] introduced a new concept to the visibility graph algorithms which is to check the node-to-node visibility as the search needs, so some nodes may not be checked and still give the same path as in the case of applying visibility graph algorithms common approach. usually, a visibility graph table is created then the data is processed by the Dijkstra algorithm to find the shortest path. Thus, saving a lot of computational resources.

However, search time in the visibility graph algorithm can be calculated using the general formula Calculation time T is a function of (n^2 log n), where n is the total number of obstacle vertices [10]. That is why a method to limit the searched space (reduce the obstacle number considered in path investigation) is needed. Many researchers used a geometrical shape to filter obstacles, e.g. rectangle shape in Dynamic Visibility Graph DVG Visibility Graph-Based path planning [11], Equilateral shape in Equilateral-Space Oriented Visibility Graph (ESOVG) [6]. Ellipse shape in Elliptical Concave Visibility Graph ECoVG [12].while searching most/all of the space will result in increased possibilities and hence long calculation time and huge computational load. Searching a small part of the space will limit possibilities and hence path does not exist result will be reached although there might be one.

The problem is when path is not found, the algorithm cannot determine if it is because it does not exist or because the obstacles node considered for path search are not enough. When searching for a new path fails, and retrying the search with more obstacles a new visibility graph must be created to adapt for the new obstacles and the Dijkstra algorithm needs to be applied again on the new matrix until a path is found or all obstacles are considered for search then the path does not exist is obtained. This means the extra effort for the previous trials is in vain. Hence, adjusting of the shape parameters subject must be treated with extreme care as it will affect performance greatly.

The main advantage iterative hexagon path planning possesses is that each processing iteration data is kept, and new data are added. i.e., the system does not start from ground zero in each iteration. Thus, saving time and reducing computational load [6]. Also, the suggested algorithm has a set of improvements to improve performance even compared to the hexagon path planning algorithm.

The general flow chart of the proposed algorithm can be found in Figure 1 The original Hexagon Path Planning Algorithm steps in green background shapes, while the suggested algorithm with red background shapes.

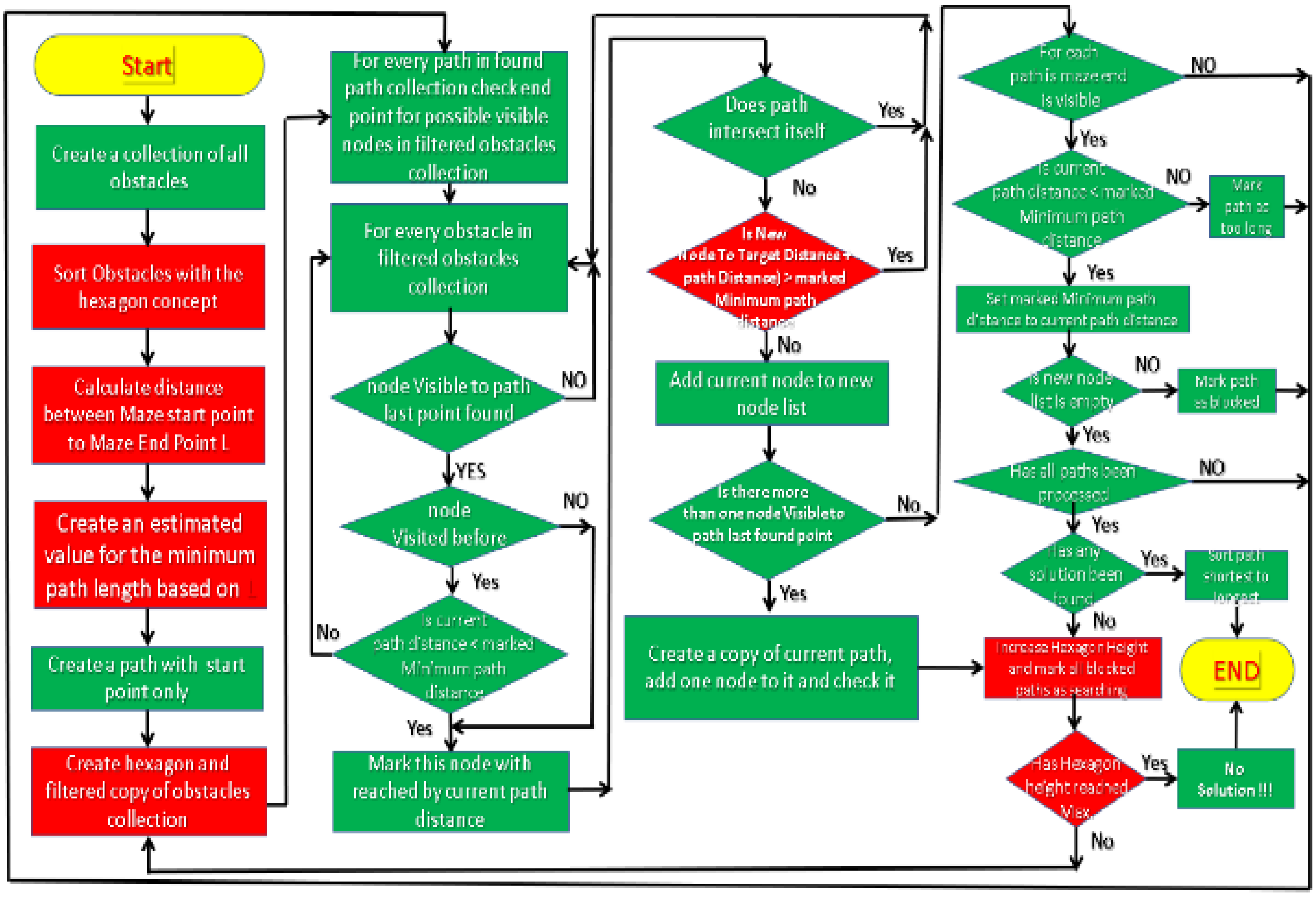

Start
Create a collection of all obstacles
Sort Obstacles with the hexagon concept
Calculate distance between Maze start point to Maze End Point L
Create an estimated value for the minimum path length based on L
Create a path with start point only
Create hexagon and filtered copy of obstacles collection
For every path in found path collection check end point for possible visible nodes in filtered obstacles collection
For every obstacle in filtered obstacles collection
node Visible to path last point found
NO
YES
node Visited before
NO
Yes
Is current path distance < marked Minimum path distance
No
Yes
Mark this node with reached by current path distance
Does path intersect itself
Yes
No
Is New Node To Target Distance + path Distance) > marked Minimum path distance
Yes
No
Add current node to new node list
Is there more than one node Visible to path last found point
No
Yes
Create a copy of current path, add one node to it and check it
For each path is maze end is visible
NO
Yes
Is current path distance < marked Minimum path distance
NO
Mark path as too long
Yes
Set marked Minimum path distance to current path distance
Is new node list is empty
NO
Mark path as blocked
Yes
Has all paths been processed
NO
Yes
Has any solution been found
Yes
Sort path shortest to longest
No
Increase Hexagon Height and mark all blocked paths as searching
END
Has Hexagon height reached Max
Yes
No Solution!!!
No

Figure 1. The proposed algorithm flow chart, the green color refers to the hexagon Path Planning algorithm, the red color refers to the new suggested algorithm modification

3.1. Iterative hexagon concept

The suggested algorithm is a modified version of the hexagon visibility graph [9]. the main change is to make it iterative and improve the search process to reduce odds. the suggested hexagon changeable extents guarantee the path will be found see Figure 2.

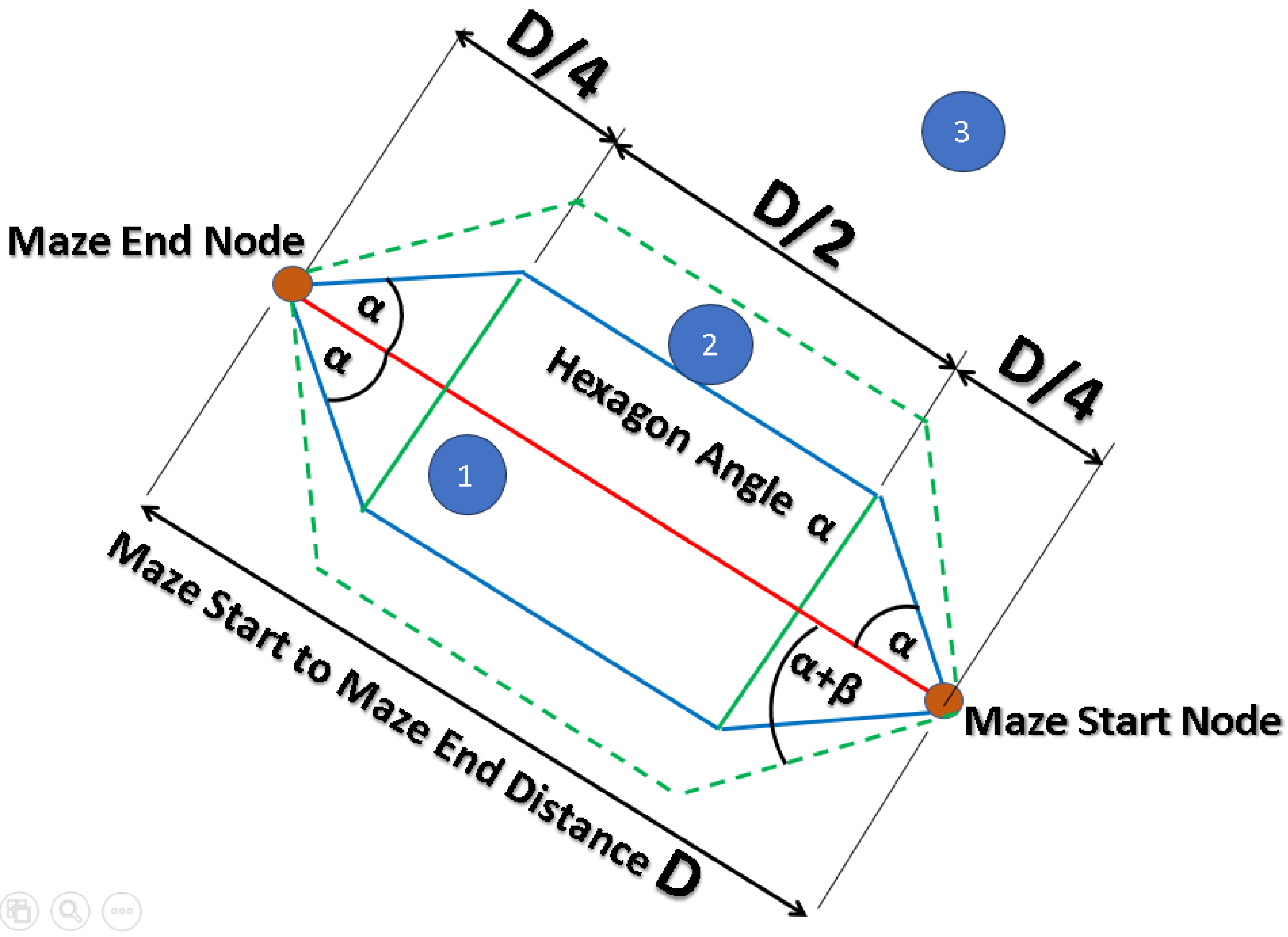


Figure 2 The start hexagon with angle α in continuous blue lines, the second Hexagon in dashed blue line obtained by increasing the angle α with β

The hexagon nodes are calculated by the method introduced in [9] with initial angle β, then mark the obstacle that are in the search area (the hexagon with the blue solid lines e.g. obstacle number 1). This obstacle collection will be considered for path search to solve the maze. The obstacle outside will not be considered like obstacle 2 and 3. If path was not found, increase the hexagon angle α with β to be (α + β) then recalculate hexagon nodes, (the hexagon with the green dashed lines e.g. obstacle number 1 and 2). Obstacle 3 will not be considered for path search in the first or the second iteration.

The following is a pseudo-code for path search:

```
let hexagon angle α= initial value e.g. 25°
do-while
  (path not found = true and obstacle count > searched obstacles number)
{
calculate the vertices of the hexagon using angle α.
For each obstacle in the obstacle space
            {if (check inside polygon (obstacle co-ordinates, hexagon vertices) = = true)
{set obstacle search include=true
  searched obstacles number +=}
}
```

```
}
If (path was found = = false)
(Show message path between node (start node coordinates and node (end node coordinates) does not exist)
end
```

3.2. Node-To-Node Visibility Check Obstacle Sorting

Path search depends on node-to-node visibility check. To speed this operation all obstacles are ordered by their center distance to the start node. this will increase the possibility to start with the shortest path which will reduce calculation time and the number of operations to get the shortest path. Figure 3 shows an example, the order of the obstacle is random. If obstacle number 2 was processed before obstacle number 1 then, an unnecessarily long path will be obtained. Also, that path may be blocked. Which is a waste of computational resources.

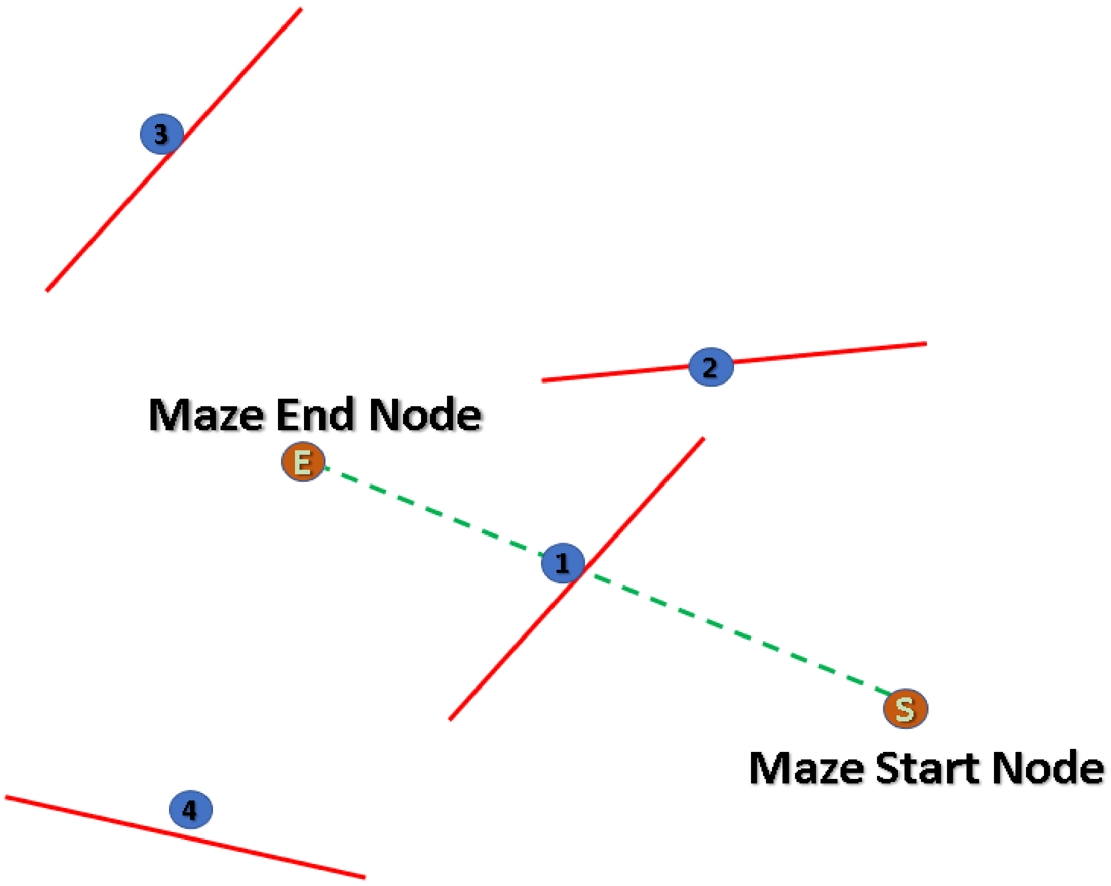


Figure 3 illustration of the obstacle sort concept, red lines are obstacles, green dashed line is the maze start to maze end node line

The obstacle sorting algorithm is as follows:

1. Find the middle node for each obstacle.
2. Construct the maze start node to the maze end node line.
3. Find the node on the maze start-end line (constructed in the previous step) that has the shortest distance, i.e., the obstacle center to that node will be the perpendicular maze start-end line (perpendicular node).

4. If the node is located between the maze start-end nodes, then the obstacle distance is the distance between the perpendicular node from the previous step and the obstacle middle node.
5. Sort the obstacles in ascending order by obstacle distance.

   These steps did boost the performance of the algorithm a little. However, the authors personally believe that the performance can be boosted further by a more sophisticated algorithm for Node-To-Node Visibility Check Obstacle Sorting or a way to check with the near obstacles first before going to the far ones to tell that it is blocked stop and return the false value. This will not waste time.

   The following is the pseudo code:

```
{  for each obstacle in obstacle list
        {middle node x = (obstacle node1 x + obstacle node2 x)/2
        middle node y = (obstacle node1 y + obstacle node2 y)/2
   loop}
        maze line = new line (maze start node, maze end node)
}
```

It worth mentioning that to guarantee the safety of the robot or vehicle, all obstacles are considered for visibility check not only those inside the hexagon.

3.3. Rough Shortest Path

During the search for a solution, the Rough Shortest Path can be used to eliminate unnecessary long paths. This is done in a way similar to the shortest solution path distance concept. When a solution is found, a flag is raised. In the shortest solution path distance concept, the path distance up to the path to the last node is compared to the shortest solution path distance. If the searched path is longer shortest found solution is rejected.

The Rough Shortest Path concept will add path distance to the distance between the current node and maze end as a straight line. distance between the current node and maze end is the shortest theoretical value that could make the current path a solution (there will be turns that will increase the distance significantly). If the sum of the path length and theoretical length to the maze endnode is greater than the shortest path length found, then this path length will be too long. The path will be marked as too long and will not be further processed. Figure 4 is an example of the Rough Shortest Path concept. The blue line is the shortest path found so far. But there are still other nodes that have not been investigated.

The path that goes from the maze start node to node number three, the distance to the maze endnode is calculated (the straight line even if it goes through an obstacle). As seen in the figure the distance between the maze starts nodes and node three (green continuous line) plus the distance from node three to the maze endnode (green dashed line) is longer than the distance from maze start node to node four and then to the maze endnode (shortest found so far maze solution path blue continuous line). Based on the currently illustrated concept the path will not be further processed. Although it will provide a valid solution (e.g., maze start to node three to node two then to maze endnode) the solution will be longer. This will be a waste of time and computational resources keeping in mind that the shortest path is required, and all other paths are not of importance. The order by which obstacles are being investigated influences the performance of the Shortest Path Concept. If the obstacle with nodes three and one was investigated before the obstacle with nodes two and four then the concept will not be applied since the path (maze start to node three to node two to maze end) will be already investigated. That is why the obstacle sorting step mentioned earlier in section discussed earlier is very important.

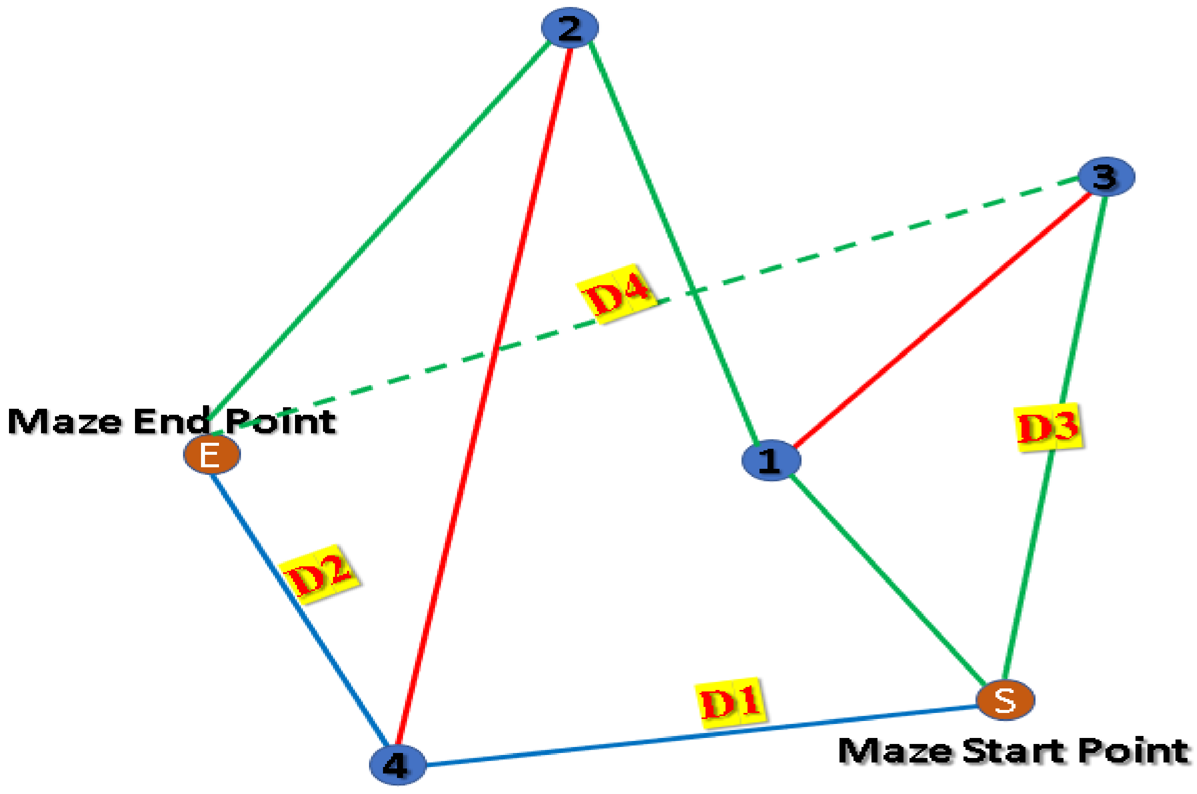


Figure 4. Rough Shortest Path Concept illustration, the found shortest path (Start node to Node two to Maze End Node) in blue, obstacles as two red lines, the path being investigated in (start node to node three) green continuous line, (dashed and con)

In Figure 4 to go from the maze start node denoted with ***S*** to maze end node denoted with ***E*** there are several paths, the shortest path found during search is from *S >> 4 >> E* with ***distance* = *D1+D2***. To investigate other possible paths, at node*3*, the distance from maze start node is D3 which is less than least found solution ***distance=D1+D2.*** there a possibility that a shorter path passing through node 3. before going through with possibility investigation, the distance from ***node 3 (D4)*** is evaluated.

Comparing the calculated distance, it is found that *D1+D2< D3+ D4.* this means the path going through ***node 3*** should be neglected and stop further processing. This will save checking ***node*** 3 to maze start node with ***S*** which will return false and will reject the paths *S >> 3 >>2 >>E* and *S>> 3 >> 4 >> E* which are longer i.e. useless, also time and resources consuming. For large mazes this will reduce possibilities significant because eliminating a branch in a tree will in turn eliminate all child node and their child and their child child.

3.4. Estimated Shortest Path Length

In some cases, especially when the number of obstacles is relatively small, filtering obstacles may prevent finding a maze solution. Also, filtering may result in planning a path that goes through neglected obstacles. In that case, the Estimated Shortest Path Length concept is a good solution. The theory is that the shortest path may not be found in the search beginning. The shortest path might be found later. So, the shortest path concept will not be applied until then. To overcome this an initial shortest path value is estimated and used in the check. The value is a ratio of the distance between the maze start node to the maze end node distance. the ratio must be greater than one, e.g., 1.2 the distance. If no path was found, then the ratio is increased by a step e.g., to 1.4, and all paths that were marked as too long are marked as being searched and processed again until a path is found.

3.5. Obstacle Hexagon Sorting

The concept of a hexagon sorting is used to prioritize maze obstacles during search, i.e. it is not used to filter or reject obstacle nodes from being part of searched path nodes.

Before going further, the fact that the obstacles in the space exist in two different sets as follows:

1. Obstacles set that are used to check for node-to-node visibility. Algorithm in section 3.2 is applied to this set.
2. Obstacles set that are used to create nodes that a possible path can go through. The algorithm in section3.5 is applied to this set.

So, from the above classification, an obstacle in the space can be in both sets or the first set only (all obstacles are considered in the node-to-node visibility check for safety as mentioned earlier). the algorithm in section **Error! Reference source not found.** and algorithm in section **Error! Reference source not found.** can be used gether because each algorithm sorts different set of obstacles

It speeds up pathfinding by putting the obstacles that are more likely to This is done by creating multiple hexagon shapes with the same diagonal (the line connecting maze start node to maze end node) and increasing hexagon height. The obstacle is marked with a hexagon (green, brown and blue) that its middle node lies in. The linear distance between the obstacle's middle node and the start node is calculated. Then obstacles are sorted by the hexagon number (obstacles in smaller hexagons first then obstacles in bigger hexagons) then by a distance smaller to bigger. This will cause the nodes that are more likely to be part of the solution to be on top of the obstacle list. An example of the previously mentioned technique can be seen in Figure 5, obstacle 1 is in the smallest hexagon as well as obstacle 2, but obstacle 1 is closer to the maze start node so obstacle 1 will be on top of the list than obstacle 2. Although obstacle 3 is closer to the maze start node than 2 obstacle 3 is in a bigger hexagon (the brown hexagon). at the end of the list comes obstacle 4 as it lies on the biggest (blue hexagon).

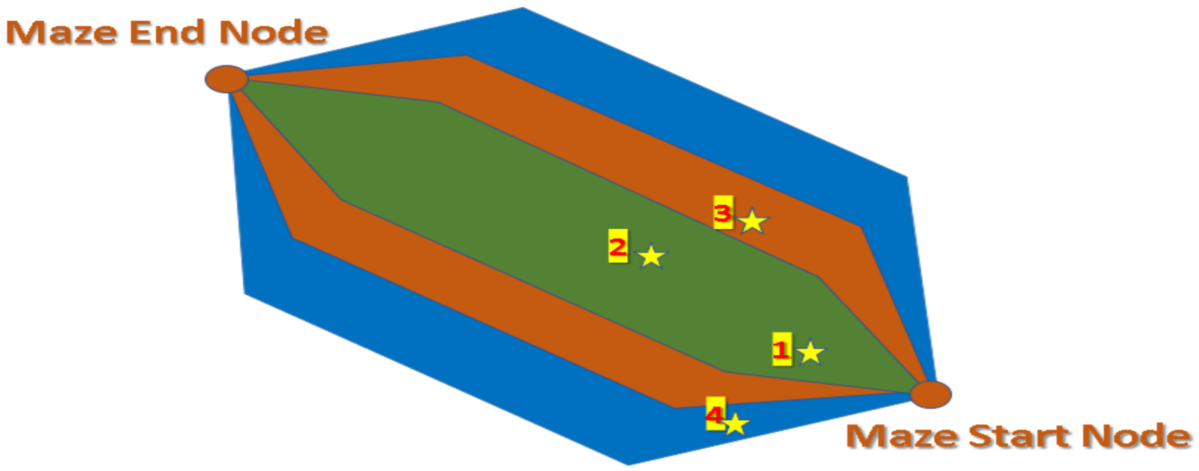


Figure 5. The Obstacle Hexagon Sorting Concept

Here is a pseudo code for this

```
1. Initialize Temp Hexagon Nodes (0) and Temp Hexagon Nodes (3) with Maze Start Node and Maze End
   Node, respectively.
2. Calculate Hexagon Middle Node as the mid node between Maze Start Node and Maze End Node.
3. Calculate Hexagon Middle Near Start Node as the mid node between Maze Start Node and Hexagon Middle
   Node.
4. Calculate Hexagon Near End Middle Node as the mid node between Hexagon Middle Node and Maze End
   Node.
5. Iterate over Temp Hexagon Height Ratio from 0.15D to 2D with a step of 0.2D:
Where D is the linear distance between Maze Start to End nodes.
    a. Calculate Hexagon Middle Length as Maze Start to End Distance multiplied by Hexagon Height Ratio.
    b. Extend Hexagon Middle Near Start Node along the Maze Start to End Angle by Hexagon Middle Length
       / 2 to get Temp Hexagon Nodes (1) and Temp Hexagon Nodes (5).
    c. Extend Hexagon Near End Middle Node along the Maze Start to End Angle by Hexagon Middle Length /
       2 to get Temp Hexagon Nodes (2) and Temp Hexagon Nodes (4).
    d. Check each obstacle (indexed by S):
        If Obstacles(S). Hexagon Number is 0:
            - Check if Obstacles(S). Obstacle Middle Node is inside the polygon defined by Temp Hexagon
              Nodes.
            - If true, assign Hexagon Number to Obstacles(S). Hexagon Number.
    e. Decrement Hexagon Number by 1.
6. Sort the obstacles (Obstacles) based on Hexagon Number in descending order, and then by Center Distance to
Start Node in descending order to get Sorted.
```

3.6. Random maze creator

To test any path plan algorithm, there must be different mazes to test with. So, a random maze generator was created. Two types of random mazes can be created in this research that can be generated by the created software, which are:

1. The lines maze represents the random size (up to a predefined size) and random orientation lines.
2. The rectangle maze, represents buildings in a city, stacked goods in a warehouse, etc...

To generate a random maze, the following data must be provided:

1. Maze type (lines, rectangles).
2. Maze size (width, height)
3. The number of obstacles in the maze.
4. Maximum obstacle length.

The following are the steps of creating a random maze:

1. Choose the maze start node and the maze end node to be at maze diagonal to ensure that the solution path should go across as many obstacles as possible
2. Choose a random node that is within the Maze size (width, height)
3. In the case of line choose a random orientation and random length that is between 0.2 to maximum obstacle length (to not get a very small obstacle that may be impractical), then the calculate obstacle second node coordinates.
4. In the case of a rectangle choose a random width and random height that is between 0.2 and to maximum obstacle length, then calculate four obstacles' coordinates.
5. Add the newly generated obstacle(s) to the obstacles list.
6. Repeat steps 2 to 4 until the required number of obstacles is reached.

since geometric algorithms are supposed to find the shortest path, this will yield a similar path (nearly the same path length and the same number of turns) with different methods. Thus, making Computation time the main assessment criterion. Usually, the comparison between two or more methods is done by comparing the time needed to solve a certain maze as suggested by [1, 7-9, 13, 14]. Other researchers suggested a standard maze to assess the path-planning algorithm [15].

The main problem is that one or two mazes may not yield a fair judgment of the performance of a certain algorithm in general particularly from the node of view of time.

This is believed to be due to the following:

1. Obstacle distribution.
2. Obstacle order in the obstacle list given to the algorithm.
3. Used algorithm priority and search method.
4. Used computer capabilities.

1. To reduce the effect of these factors, it is suggested to create a large number of random mazes with varying obstacle number that covers the expected working zone of the path plan algorithm. The creation of a random maze for algorithm judgment requires making some roles to ensure that the maze will fit. for example, if the maze is large compared to obstacle number and size the start and end nodes might be visible, eliminating the need for a path planning algorithm. In other cases, a solution might not exist. So, the following steps are considered in random maze creation:
2. Set minimum maze width (say 50 units).
3. Set maze increase with obstacle number to value relatively smaller than in step 1 (say 4 units).
4. The maze width is calculated by multiplying the required obstacle number by the maze increase with obstacle number in Step 2
5. The maze length is estimated as the double maze width
6. A random maze is created using the technique mentioned earlier with the maze height, width, and the number of obstacles.
7. The maze is solved by an algorithm that is guaranteed to find the path or tell that there is not one.
8. If no solution is found or the maze start and end nodes are visible to each other repeat steps 5 and 6.
9. The solution time, path length, and path nodes are recorded for later comparison
10. Steps 3 through 8 are repeated until all mazes with the number of obstacles required are created e.g., 1 to 200 obstacles mazes.

The steps for the suggested path plan assessment method are as follows:

1. Create random mazes varying in obstacle number using the mazes generation roles mentioned earlier.
2. The created random mazes are solved by the method(s) that need to be assessed and a curve is a plot for each criterion.
3. A curve fitting is applied to remove noise (relatively high or lower criteria than the expected trend)
4. Draw a comparison plot for methods being evaluated (solution time, path length, and the number of turns).
5. If a rough number is needed sum up the result for each algorithm and each criterion over the different mazes.
6. Compare methods based on the percentage, not the absolute number or the curve behavior.

3.7. Calculated Data Storage

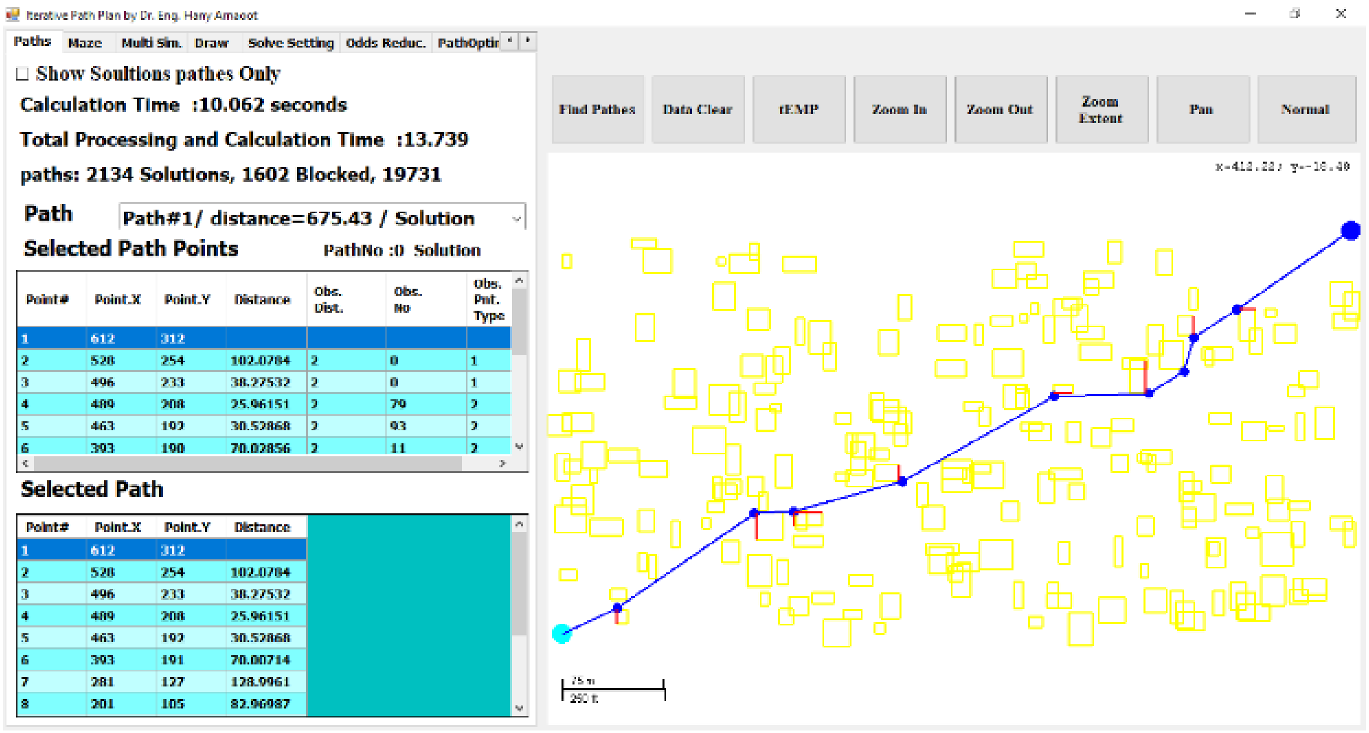
The maze solving needs continuous calculation of many parameters e.g., node to node visibility or node to node distance. In many cases, the same calculation may be needed more than one time. This is the case when a path is under search and visits a node then another path visits the same node again. Instead of recalculating and wasting valuable time, a matrix that stores the data is checked first, if there is a stored value then the stored value will be returned, otherwise, the value is calculated and stored in case this value is needed again.

4. THE DEVELOPED SOFTWARE PACKAGE

Software is needed to apply the currently presented algorithm. The software was created using Visual Basic.Net 2019 software Integrated Development Environment IDE. The software is divided into two parts:

1. Graphical user interface GUI (graphically represents the data, handles file loading and saving, gets user settings from displayed controls, creates graphs)
2. Calculation class (pathfinding)

The calculation class is completely independent of the Graphical user interface GUI. The purpose of creating a stand-alone class is to possess the ability to deploy the algorithm at different platforms (P.C., microcontroller, server...) with no/minor recoding. This will save the recording time as well as testing and debugging time. also, it will simplify creating updates. i.e., this will transform a desktop application for research to a microcontroller or server application real-world application as simple as possible. The Developed software GUI is in Figure 6.



☐ Show Soultions pathes Only
Calculation Time :10.062 seconds
Total Processing and Calculation Time :13.739
paths: 2134 Solutions, 1602 Blocked, 19731
Path: Path#1/ distance=675.43 / Solution
Selected Path Points   PathNo :0 Solution

Point#	Point.X	Point.Y	Distance	Obs. Dist.	Obs. No	Obs. Pnt. Type
1	612	312				
2	520	254	102.0704	2	0	1
3	496	233	38.27532	2	0	1
4	489	208	25.96151	2	79	2
5	463	192	30.52868	2	93	2
6	393	190	70.02856	2	11	2

Selected Path

Point#	Point.X	Point.Y	Distance
1	612	312	
2	520	254	102.0704
3	496	233	38.27532
4	489	208	25.96151
5	463	192	30.52868
6	393	191	70.00714
7	281	127	128.9961
8	201	105	82.96987

Figure 6. Developed software GUI.

To plot the maze and the results, an open-source project called Map Window Geographic Information System GIS ActiveX [16] was chosen. It is a free tool that is used by many researchers around the world e.g. [17],[18].

5. RESULTS

for a fair comparison, the suggested algorithm and the traditional way of creating a visibility map then solve the Dijkstra algorithm using the same functions (e.g., the algorithm mentioned earlier in section The node-to-node windowed visibility check). This is because the Dijkstra algorithm takes on average half of the time needed to solve the maze and the other half to create node-to-node visibility check. It is worth mentioning that the case of obstacle lines intersecting does not affect the proposed algorithm, i.e., obstacles in the form of intersecting polygons are permitted.

The implementation of the proposed algorithm on a sample of random mazes created by the algorithm mentioned earlier in **Error! Reference source not found.** . 250 random mazes were generated starting rom one rectangular-shaped obstacle maze (4 different lines with 8 different nodes) to 250 rectangular obstacle maze (1000 different lines with 2000 different nodes). The results were obtained by running an executable of the code file, not the code through the Visual Studio (to exclude the Visual Studio developing environment load effect on the time) on a clean machine while turning off all other applications. The used machine was a core i5 with 12 Gigabyte RAM

5.1. Sample of Maze solving Results

A sample of the 250 mazes generated with 50 rectangular-shaped obstacles as in 250 rectangular-shaped obstacles as in Figure 7 are as follows:

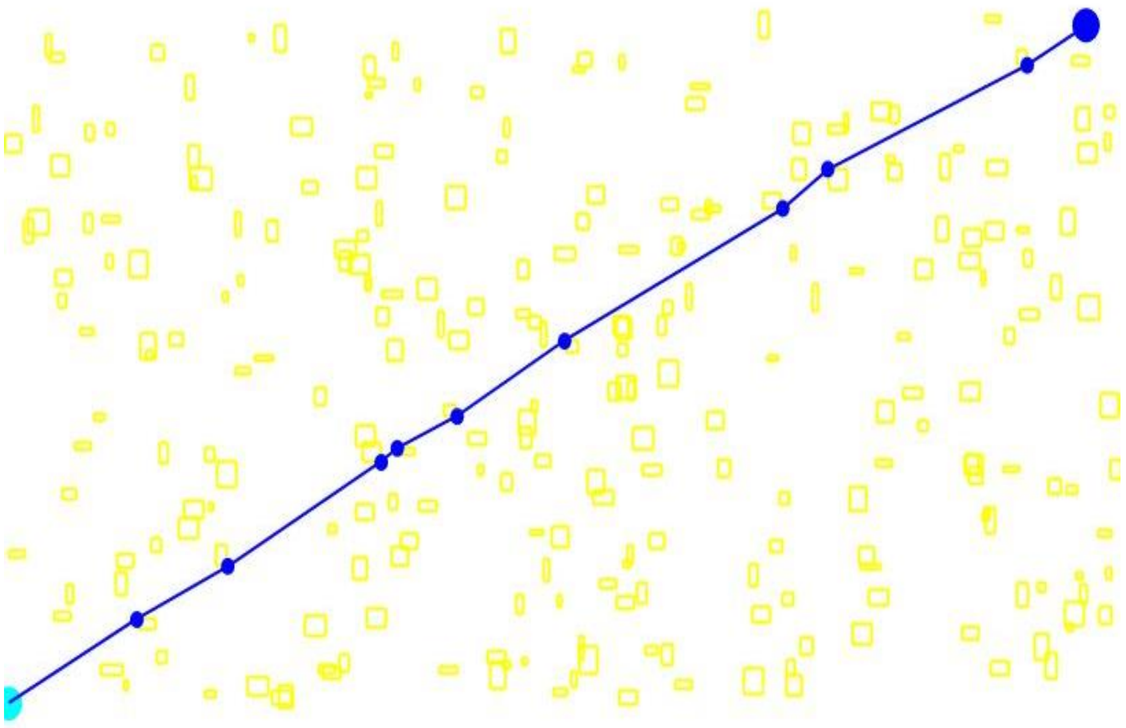


Figure 7. a maze with 250 rectangular-shaped obstacles in yellow solved by the suggested algorithm (solution in blue line).

5.2. Maze solving Results Comparison

Applying the proposed algorithm in section **Error! Reference source not found.** to the process of solving he 250 mazes created earlier yielded the following graph Figure 8, where the horizontal axis is the number of obstacles in the maze and the vertical axis is the time needed to solve in milliseconds.

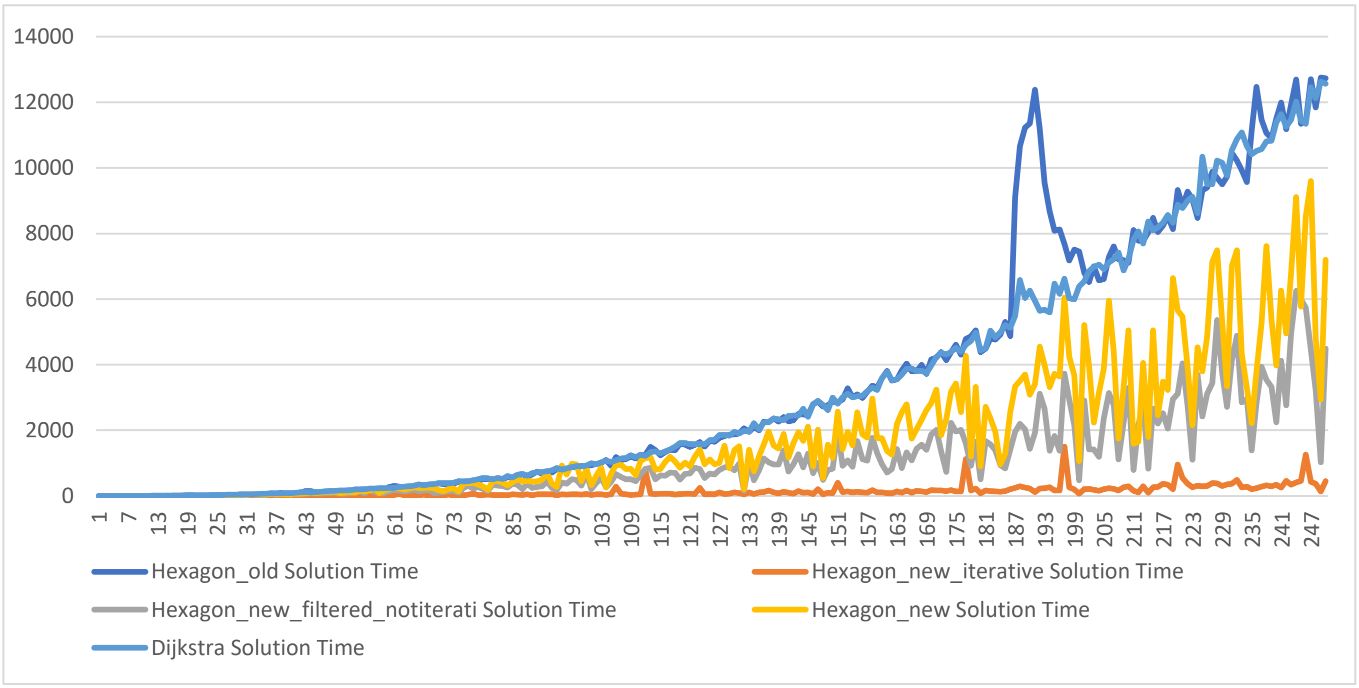


Figure 8 Application of Multi-Maze Solve Curve to 250 different mazes results from time comparison. Orange line is Dijkstra Results, the grey line is Suggested Result, and the blue line is Suggested with hexagon iterative filter results.

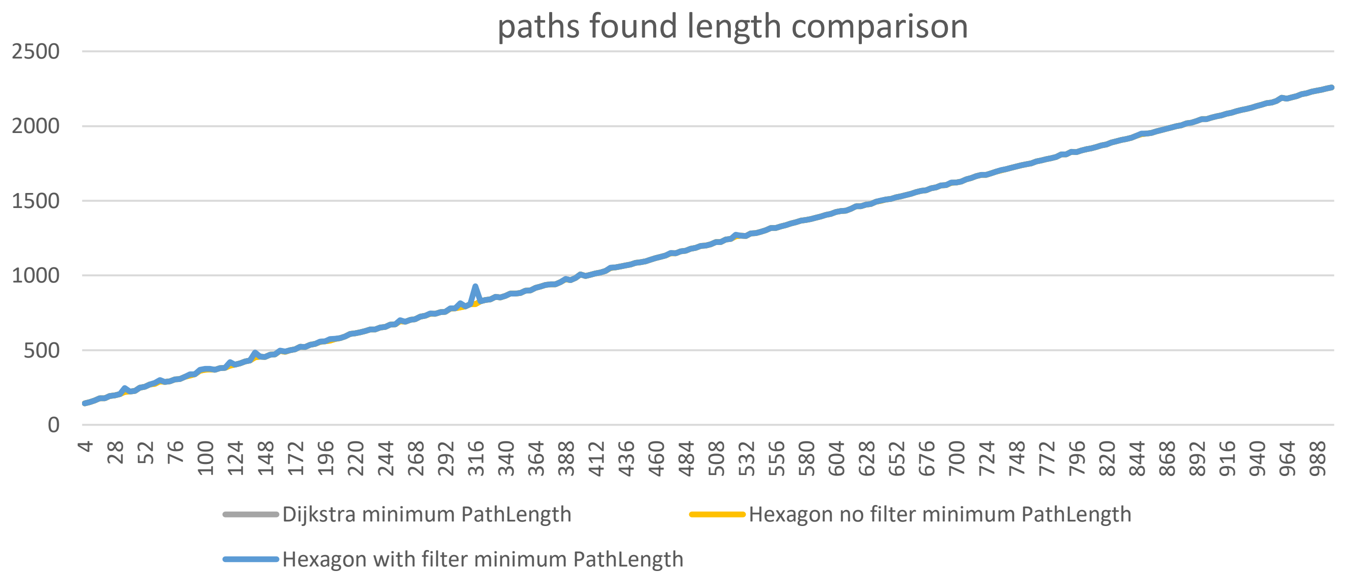


Figure 9. Application of Multi-Maze Solve Curve to 250 different mazes result length comparison. The orange line is Dijkstra Results, Grey line Suggested Result, blue line is Suggested with hexagon iterative filter results.

It is seen that the proposed method is always faster than the Dijkstra algorithm. But with a varying reduction ratio. This is due to the varying nature of the mazes created which assure the randomness of mazes structure.

The following table, Table 1 summarizes the (Creating a visibility map then solve by Dijkstra, old Hexagon, Hexagon new iterative, Hexagon new filtered without iteration, Hexagon new) maze solve time

summation Comparison summary data in the previous graph. The first column ***(Time in seconds)*** is the summation of all 250 mazes in Figure 8 Application of Multi-Maze Solve Curve to 250 different mazes results from time comparison. Orange line is Dijkstra Results, the grey line is Suggested Result, and the blue line is Suggested with hexagon iterative filter results.Figure 8 solving time. the second column is the ratio of the method time in previous column to creating a visibility map then solve by Dijkstra.
Summation was chosen as an index of the performance to reflect the whole spectrum not selecting some of them, however, taking the average will give the same percentage because they are all result of solving the same number of mazes which in this case 250 mazes.

Table 1 solve time summation Comparison summary

Method name		**Total of 250 maze solve Time in seconds**	**Ratio% to creating a visibility map then solve by Dijkstra**
Create Visibility Map Then Dijkstra		820.039	100%
Old Hexagon Visibility Graph		864.309	105.39%
The current suggested algorithm	Hexagon New only (no iteration, no filter)	446.412	54.44%
	Hexagon New Filtered Without Iteration	278.790	33.99%
	Hexagon New Iterative	34.041	4.15%

5.3. COMMENTS ON THE RESULTS

the shortest path found by the three methods considered returned the same nodes and hence length. however, in one or two cases there was a slight difference for the filer application. By investigating the results. It was found that two obstacle nodes are very close, one of them was removed by filtering. This means the reason is the filtering action, not the algorithm itself. the path length increase ratio was less than 1% of the total length which can be neglected due to two reasons:

1. huge calculation time reduction compared to minimal path change if exists
2. the small increase ratio

the hexagon and hexagon iterative methods' time is much varying compared to the traditional way of creating a visibility map and then solving the Dijkstra algorithm. this can be explained by the nature of the traditional method where a visibility graph is prepared for all nodes, which in the case of 2000 obstacles will require 4,000,000 million checks. node-to-node visibility check must check the path node against the 2,000-obstacle meaning 8,000,000,000 intersection check. That is why it takes a long time even with a modern computer. While the suggested algorithm reduces this enormous calculation by checking node-to-node visibility as needed, so in most cases it reaches the desired path from early trials with much less time than the traditional method, while in the other minor cases, it has to check almost all paths which will consume time close but never exceeds the traditional way. Overall, there is a huge time savings.
The need for showing the spectrum was motivated by the above fact, because showing 5 10 or even 50 selected mazes may not reflect the real performance

6. CONCLUSION

- The proposed algorithm reduced Path planning computational time significantly compared to the traditional way of creating a visibility map then solve the Dijkstra algorithm Dijkstra and the hexagon algorithm without applying a filter and the reduction is increased when filtering obstacles with hexagon. The reduction ratio is variable this is believed to be due to the random distribution of the maze obstacles and size.

- Hexagon is suitable for both small and large obstacles maze.
- obstacles sorting and prioritizing significantly reduced the calculation time.
- Creating a matrix to store node-to-node distances or node-to-node visibility reduces the computation time significantly.
- One or two mazes are not enough to assess the performance of the path-planning algorithm, increasing the number of mazes will yield a better understanding.
- Repeating the test could help reduce the odds of criteria results in randomly created mazes for better algorithm judgment.
- Calculated Data Storage (saving the result of an operation so that recalculation is not needed) reduces calculation time significantly in the algorithms that perform calculations as needed.
- The suggested algorithm creates a set of valid different solutions, not one solution as in the case of the Dijkstra algorithm. The created set of solutions can be valuable when other factors rather than path length (e.g., path safety) are of concern because no further processing is needed to obtain them.
- The suggested algorithm achieved a significant time and computational load reduction at all investigated mazes. However, it was intended for use in solving dense obstacle mazes.
- Obstacle filtration can significantly reduce time and computational load. Also, it enables solving a maze more complicated than the abilities of the processing unit. However, the filtration parameter (e.g., in the method presented in this work hexagon height, hexagon increase ratio should be chosen with care to achieve the best performance.
- When using obstacle filtration, it is advised to check the validity of the path with all obstacles not only the filtered ones, as this may lead to a path that will collide with the walls of the maze being searched. However, checking the validity of the path with all obstacles will significantly increase the solution time. To overcome this dilemma, it is suggested to create two filters, one for path searching and a bigger one for path validity check.

References

1. Kim, J., *3D path planner of an autonomous underwater vehicle to track an emitter using frequency and azimuth–elevation angle measurements.* IET Radar, Sonar & Navigation, 2020. **14**(8): p. 8.
2. Michele Gabrio Antonelli, P.B.Z., Andrea De Marcellis, Elia Palange, *Autonomous robot for cleaning photovoltaic panels in desert zones.* mechatronics, 2020. **68**.
3. Pagilla, Y.W.P.R., *A novel 3D path following control framework for robots performing surface finishing tasks.* Mechatronics, 2021. **76**.
4. Costanzo, p.M., *Control of robotic object pivoting based on tactile sensing.* Mechatronics, 2021. **76**.
5. Masami Hirayamaa, b., , Jose Guivant a, Jayantha Katupitiyaa, Mark Whittya, *Path planning for autonomous bulldozers.* Mechatronics, 2019. **58**: p. 20-38.
6. Nor Badariyah Abdul Latip, R.O., Sanjoy Kumar Debnath, *Optimal Path Planning using Equilateral Spaces Oriented Visibility Graph Method.* International Journal of Electrical and Computer Engineering (IJECE), 2018. **7**(6): p. 3046~3051.
7. Mohd Nadhir Ab Wahab , S.N.-M., Adham Atyabi *A comparative review on mobile robot path planning Classical or meta-heuristic methods.* Annual Reviews in Control, 2020. **50**: p. 232-253.

8. LbAnishPandeycD.R.K.ParhidA.Jagadeeshe, B.K.P., *A review On path planning strategies for navigation of mobile robot.* Defence Technology,elsevier, 2019. **15**(4): p. 25.

9. Elsayed Ibrahim Mohamed Arnaoot, H.M., *Hexagon Visibility Graph Path Planning Algorithm* IET Radar, Sonar & Navigation, 2022. **16**(12): p. 15.

10. Prof. Dr. Mark de Berg, D.O.C., Dr. Marc van Kreveld, Prof. Dr. Mark Overmars, *Computational geometry_ algorithms and applications*, ed. V.B. Heidelberg. 2008.

11. Han Pang Huang, S.Y.C., *Dynamic Visibility Graph for Path Planning*, in *2004 IEEElRSl International Conference on Intelligent Robots and Systems*. 2004, IEEE: Sendai, Japan

12. Sanjoy Kumar Debnath, R.O., Susama Bagchi, Marwan Nafea, Ranesh Kumar Naha, Elia Nadira Sabudin, *Energy Efficient Elliptical Concave Visibility Graph Algorithm for Unmanned Aerial Vehicle in an Obstacle-rich Environment* in *2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS 2020)*. Shah Alam, Malaysia.

13. Debnath, S.K., Rosli Omar, and Nor Badariyah Abdul Latip, *A review on energy-efficient path planning algorithms for unmanned air vehicles.* In Computational Science and Technology,Springer, 2019. **523-532**.

14. Hanlin Niua, Y.L., Al Savvarisa, Antonios Tsourdosa, *An energy-efficient path planning algorithm for unmanned surface vehicles.* Ocean Engineering, 2018.

15. Woo, J.K.a.S.H.A., *Reference Test Maps for Path Planning Algorithm Test.* International Journal of Control, Automation and Systems, 2018. **16**(1-5).

16. Daniel P. Ames, *MAPWINGIS REFERENCE MANUAL,A function guide for the free MapWindow GIS ActiveX map component.* 2018.

17. Lei, X.W., Yuhui; Liao, Weihong; Jiang, Yunzhong; Tian, Yu; Wang, Hao, *Development of efficient and cost-effective distributed hydrological modeling tool MWEasyDHM based on open-source MapWindow GIS.* Computers & Geosciences., 2011. **37**(9): p. 14.

18. Arnaoot, H.M., *Design of a Naval War Training Simulator* in *The 2021 International Telecommunications Conference, ITC*. 2021, IEEE: Alexandria , Egypt.