Human-Like Path Finding Algorithm Based on Classic A*

McGill University - Comp 400 Project

Author: Yuzhou Guo

Professor: Clark Verbrugge

April 20th, 2019

Contents

1. **Introduction**
2. **Problem Area and Related Work**
   - 2.1. The A* Algorithm
   - 2.2. Bézier Curve
3. **Design and Methodology**
   - 3.1. Visualize using web browser in languages of JavaScript, HTML and CSS
   - 3.2. Classic A* with Imported Game Map
   - 3.3. Improve A* with Angle Smoothing
   - 3.4. A* improved with Angle Smoothing and Wall Avoiding - one step
   - 3.5. A* improved with Angle Smoothing and Wall Avoiding - two step
   - 3.6. Visually improved using Bézier Curve
4. **Result**
5. **Conclusion**
6. **Future Work**

List of Figures

1. Construction of a quartic Bézier curve
2. Random nodes classic A* algorithm
3. An example showing the imported map “arena.txt” interpreted as the grid
4. An example showing the imported map “den312d.txt” interpreted as the grid
5. Solution path of control case after Angle Smoothing
6. Solution path of control case after Angle Smoothing and Wall Avoiding - one step
7. Solution path of control case after Angle Smoothing and Wall Avoiding - two steps
8. Solution path of control case after Smoothing, Wall Avoiding and Bézier Curve
9. Summary diagram of the trend - Four attributes vs. Various stages of the algorithm

List of Code Pieces

1. Angle Smoothing by adding a certain penalty
2. Wall Avoiding by adding a certain penalty - one step
3. Wall Avoiding by adding a certain penalty - two steps

List of Table

1. Summary diagram of the trend - Four attributes vs. Various stages of the algorithm

1 Introduction

The human-like path aimed to be generated based on the classic A* algorithm falls into the category of shortest path finding algorithms. The algorithm itself is a variant of the A* with the application of Bézier curve on top of it to improve the visual demonstration at the end. Generation of the algorithm can be separated into four stages within a linear development: The Normal A*, Improved version with Angle Smoothing, Improved version with Angle Smoothing and one-step Wall Avoiding and Improved version with Angle Smoothing and two-steps Wall Avoiding.

The result can be seen comparing the attributes of time consumed, exploration percentage of the map/landscape and percentage of sharp turns and nodes-directly-along-the-wall arise in the solution path by running 100 test cases. Among the four versions of algorithms from the research, the time consumed and exploration percentage of the map become large as more and more constraints applied to the algorithm. The use of the Bézier curve may be explored more in the future to improve the smoothing feature.

2 Problem Area and Related Work

Trying to improve the shortest path to a more human-like version is not a new trend and there has been some research done before related to this field, namely the classic shortest path finding algorithm A* and the Bézier Curve algorithm for smoothing and drawing a curve among a certain number of control points.

2.1 A* - The shortest path algorithm

The classic A* algorithm was developed in 1968[1] to find the minimum cost path within a certain graph or graph-like landscape. "A* was originally designed for finding least-cost paths when the cost of a path is the sum of its edge costs, but it has been shown that A* can be used to find optimal paths for any problem satisfying the conditions of a cost algebra"[2].

In other words, the A* algorithm starts from a specific starting node of a graph and aims to find a path to the given goal node having the smallest cost. Specifically, A* selects the path that minimizes the f value from the function

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

Where g(n) is the cost so far and h(n) is the heuristic value of the current node. A* is an informed search, which means it not only calculates the cost along the solution path so far but

also tries to pick the next move as close to the destination as possible. While finding the shortest path between nodes, the heuristic function is usually the straight-line distance from the current node to the destination node.

Implementation wise, a priority queue is usually used while application. Then "At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbours are updated accordingly, and these neighbours are added to the queue"[3]. The main focus of this research will be adding penalty values to the nodes so that their status within the priority queue is controllable and in some way predictable. Thus by controlling which nodes to visit first from the queue, we can sort of adjust the solution path to be more human-like.

2.2 Bézier Curve

Bézier Curve is a parametric curve commonly used in computer vision and other related fields[4].

The main idea is that, say a series of points connected is given, set the start point and the end point to be anchor points and all points in between as control points. Let BP0, P1, …, Pn denote the Bézier curve determined by any selection of points P0, P1, …, Pn.

Write the mathematical representation of the Bézier Curve as recursive form[5]:

$$\mathbf{B}_{\mathbf{P}_0}(t) = \mathbf{P}_0, \text{ and}$$
$$\mathbf{B}(t) = \mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_n}(t) = (1-t)\mathbf{B}_{\mathbf{P}_0\mathbf{P}_1\ldots\mathbf{P}_{n-1}}(t) + t\mathbf{B}_{\mathbf{P}_1\mathbf{P}_2\ldots\mathbf{P}_n}(t)$$

As an diagram demonstration of the formula:

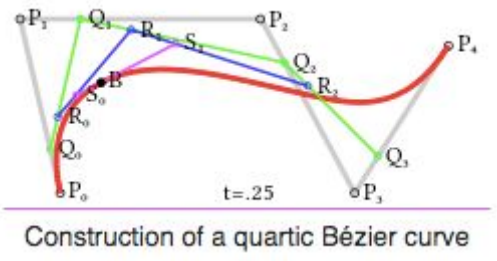

Figure 1: Construction of a quartic Bézier curve

The application of the Bézier curve will be used on top of the variation of the A* algorithm as a better visualization from the web browser.

3 Design and Methodology

The main idea of this research is that based on the foundation of the A* algorithm, a variant shall be made so that it "takes some steps back" and produces a more human-like path by sacrificing some of the path property such as the time-consuming.

Within the scope of this particular research, a two-dimensional grid of nodes drawn on a web canvas will be used to represent the landscape of a map with obstacles. Visually, black nodes represent obstacles and white nodes represent the free area. For each node, eight directions in total are capable to move which are upper-left, straight up, upper-right, left, right, bottom-left, straight down and bottom right. A start node and an end node will be chosen, and as the initial state of the research, the classic A* algorithm will find out the shortest path between these two nodes.

Various version of algorithms improved based on A* will be developed and their functionality and properties will be tested on the grid.

3.1 Visualize using web browser in languages of JavaScript, HTML and CSS

Before diving into further research, the visualization issue shall be solved to make debugging and showing the path itself visually easier.

Without the using of any visualization tool encapsulating the source code (HTML 5, OpenCV, JavaScript, etc), the best a programmer can do to implement the A* algorithm has to be printing out the path on the console. Usually, the grid will be drawn in the format of String and the message on console showing the path is usually in the format of

$$(1, 0) \rightarrow (2, 4) \rightarrow (3, 6) \rightarrow \ldots \rightarrow (56, 20)$$

Each pair of numbers represents the x value and y value of the node, the programmer needs to follow the entire string to understand the solution path, which is not readable or developable when huge scale map is being used.

A JavaScript framework called "p5.js" is chosen to be used in this research in the Chrome web browser. Thus, all source code of this research project will be in the languages of JavaScript and associated HTML and CSS.

Briefly speaking, the function draw() in the “p5” framework is a continuously recursive function repeating code within. The framework itself has a rich library of visual demonstration functions such as drawing canvas or majority types of shapes.

Drawing a piece of canvas and nodes in the shape of ellipses, then choose the very top-left node as the start node and the bottom-right node as the end node. In the draw() function, we trace back the solution path of the current state and show it as a connective line so that the path can be shown non-stop due to the recursive feature of the function.

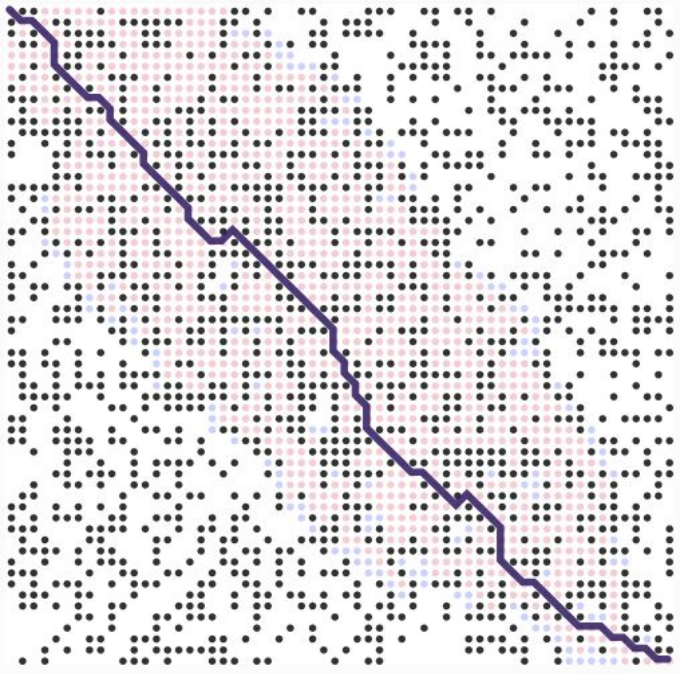

Figure 2: Random nodes classic A* algorithm

From Figure 2, the purple path represents the solution path, the light blue nodes represent the nodes in the open set (nodes plan to visit) and the pink nodes represent the nodes in the closed set (nodes has already been visited).

Nodes in the open/closed set are shown to demonstrate the exploration percentage of the current algorithm using. More attributes other than exploration percentage will be added and introduced in the following chapters.

3.2 Classic A* with Imported Game Map

With the purpose of setting up different difficulty levels to test the resulting algorithm and make the whole process controllable, external game maps in the form of text are imported to the “p5” framework workplace[6]. An additional function was added to help identify whether a specific node is an obstacle or not according to the map String provided.

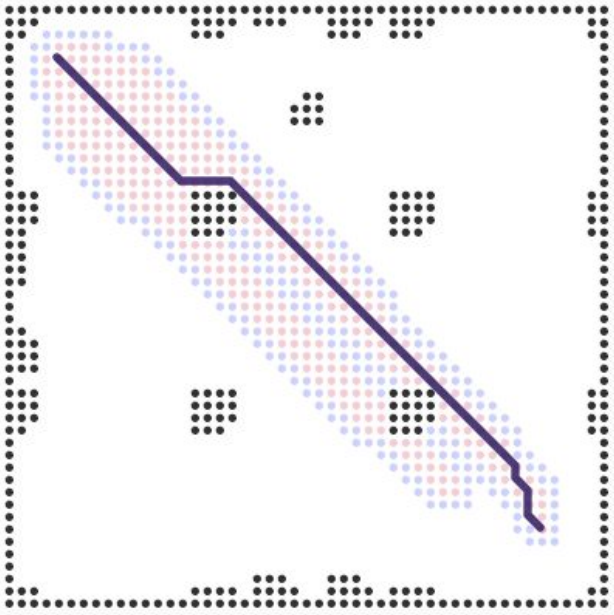

Figure 3: An example showing the imported map “arena.txt” interpreted as the grid

The new challenge is that since continuously drawing nodes and the solution path consumes time, and after testing with a huge map, it has been shown that hours can be costed to run through the classic A* algorithm in the framework.

As a result of finding a balance between complicated map and relatively small scale map, the map “den312d.txt” with start node at position (7, 7) and end node at position (7, 70) has been chosen to be the control case throughout the research.

Figure 4 shows the solution path of implementation of the classic A* algorithm. This path will remain as the “initial path” from the classic algorithm and be used to compare to various stages of the improved version occasionally.

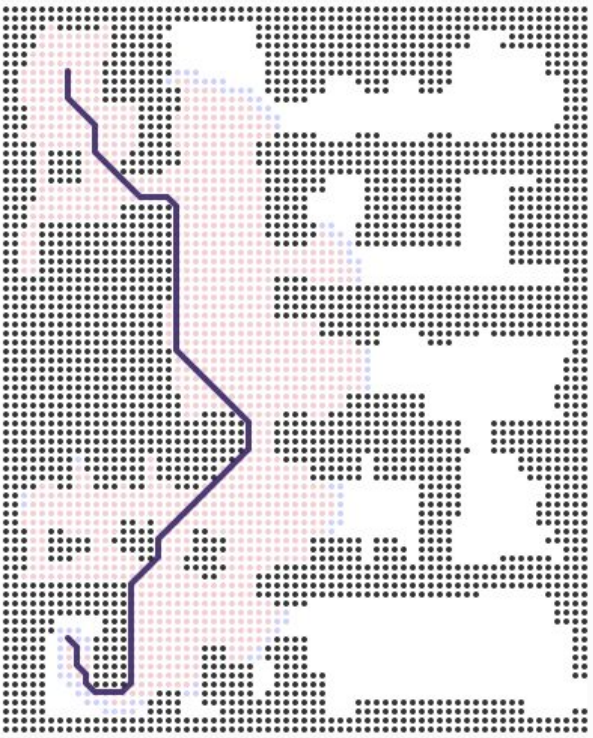

Figure 4: An example showing the imported map “den312d.txt” interpreted as the grid

3.3 Improve A* with Angle Smoothing

The first adjustment can be made on the algorithm is the decision of nodes to explore based on the angle degree.

Since the graph is treated as a discrete grid in this research and eight directions in total are possible for a node to explore. Degrees possible are 0 degrees (going back), 45 degrees, 90 degrees, 135 degrees and 180 degrees. Consider that this is a variant A* algorithm and the heuristic value in the A* value function must be admissible, 0 degrees which means the path is repeating itself and going back will not be part of the research consideration.

In practice, the angle of three control points will be calculated: one of the neighbours of the current node now exploring, the current node and the previous node of the current node. If the angle is 135 degrees or 180 degrees, then it is considered to be a decision that a human being will make to step out. The main purpose of this angle smoothing attempt is to avoid sharp turn like 90 degrees.

```
var result = find_angle(neighbor, current, parent);
result = (result * 180) / Math.PI;
```

```
if(result <= 90){
  tempG = tempG + 3;
}
```

Code 1: Angle Smoothing by adding a certain penalty

From Code 1, a simple decision has been made that: If the neighbour visiting can cause an angle smaller than 90 degrees to be formed, then a penalty value of 3 will be added to the g value (cost so far) for this particular node.

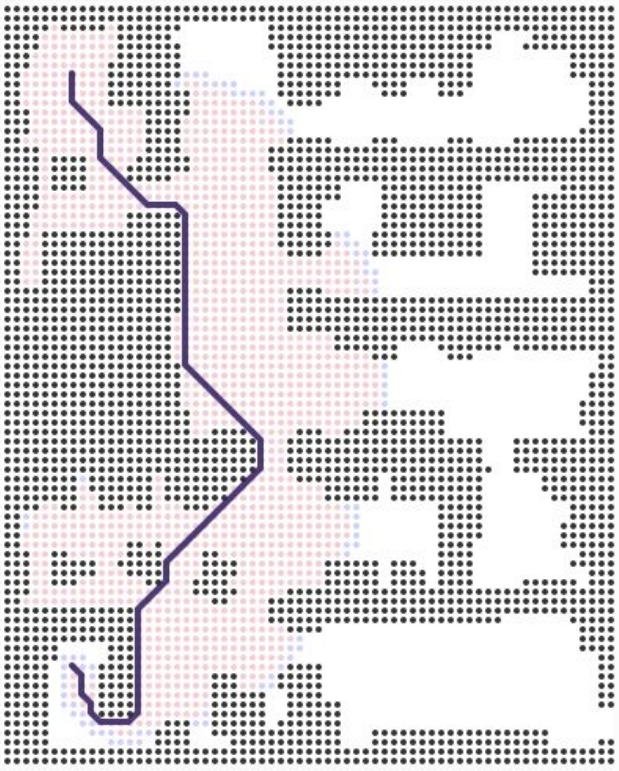

Figure 5: Solution path of control case after Angle Smoothing

Figure 5 shows the new solution path after implementing the Angle Smoothing idea. Visually, the solution path has no difference compared to the original path given by the classic A* algorithm. The reason is that since A* itself is an algorithm finding for the shortest path between nodes, all the 90 degrees angles or sharp turns have already been replaced by a series of diagonal steps if possible.

Nevertheless, in the result chapter, further discussion will be introduced and it turns out that even though this Angle Smoothing process doesn't help the solution path itself looks more

human-like, it improves the speed/time consumed to generate the path. This is because bad nodes will be pushed even further within the priority queue due to the penalty value added on, which eventually helps improve the overall performance.

3.4 A* improved with Angle Smoothing and Wall Avoiding - one step

Other than the angle degree, another important feature of human walking is that, if possible, a human being will try not to walk completely along the wall or any obstacle. Moreover, if one needs to walk through a parallel wall or obstacles, he or she may choose to walk in the center of the path instead of preferring either side.

Based on this, similar to the penalty given in the Angle Smoothing process, a certain penalty will be given when the current neighbour of the node exploring is a wall node. As a result, this node will be pushed back to the priority queue and considered later in the algorithm.

```
var wallPenal = 0;
var goThrough = neighbor.neighbors;
for (var p = 0; p < goThrough.length; p++) {
  if(goThrough[p].wall === true){
    wallPenal = wallPenal + 0.4;
  }
}
```

Code 2: Wall Avoiding by adding a certain penalty - one step

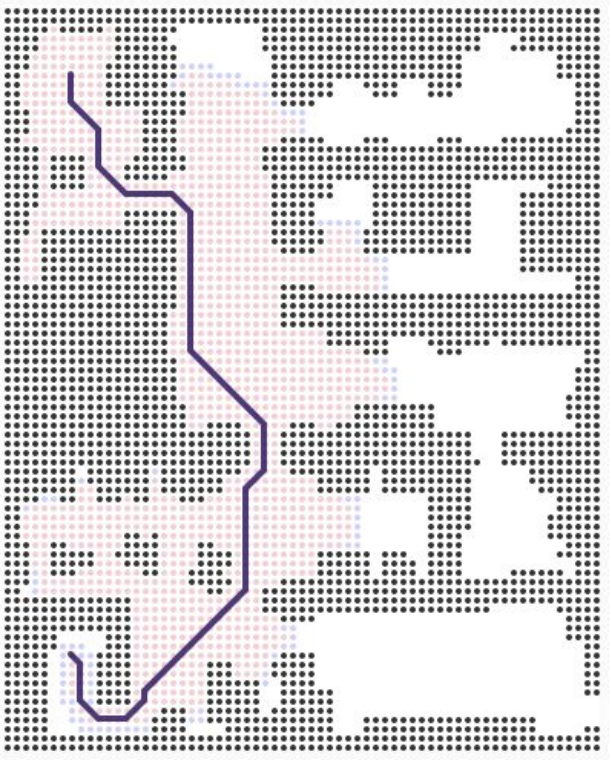

Figure 6: Solution path of control case after Angle Smoothing and Wall Avoiding - one step

From Figure 6, it is obvious that the solution path is now one node/step away from the obstacles detected. From now on, a node will not be chosen to be part of the path unless it is the only solution to make.

3.5 A* improved with Angle Smoothing and Wall Avoiding - two step

What introduced in section 3.4 is the Wall Avoiding with only one step, meaning only searching for the wall nodes within the neighbours of the current node. With the two-step version of the same algorithm, not only the neighbours of the current node but also the neighbours of the neighbours will be detected for any wall nodes arising.

Implementation-wise, this is a searching for-loop embedded within a for loop.

```
var wallPenal = 0;
var goThrough = neighbor.neighbors;
for (var p = 0; p < goThrough.length; p++) {
  if(goThrough[p].wall === true){
    wallPenal = wallPenal + 0.4;
```

```
    }
    var neis = goThrough[p].neighbors;
    for (var q = 0; q < neis.length; q++) {
      if(neis[q].wall === true){
        wallPenal = wallPenal + 0.1;
      }
    }
}
```

Code 3: Wall Avoiding by adding a certain penalty - two steps

Notice that the penalty value for the second-step wall detected is smaller than the first round detected since the balance between exploration/efficiency and human-like path must be found, which means the “bad nodes” shouldn’t be pushed back to the queue too hard.

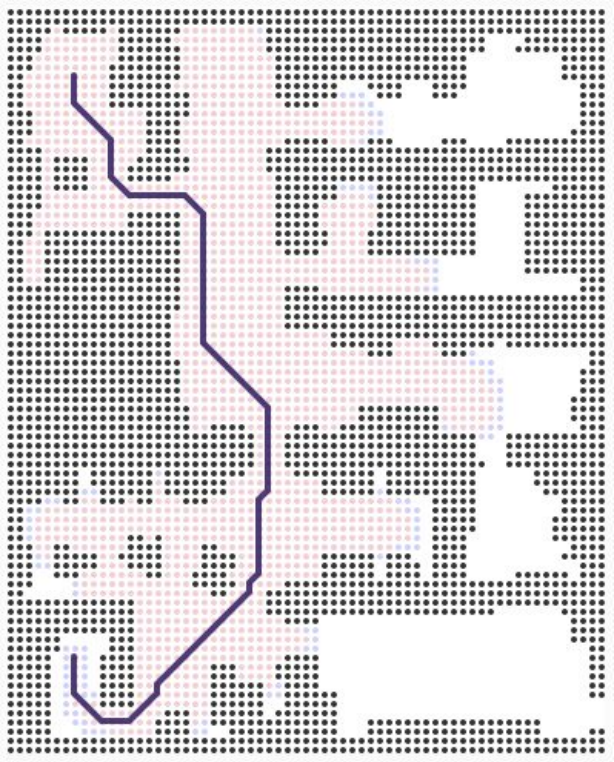

Figure 7: Solution path of control case after Angle Smoothing and Wall Avoiding - two steps

From Figure 7, after implementing the Wall Avoiding algorithm for two steps, an obvious observation is that the exploration area has rapidly become larger compared to previous versions due to the second-round penalty adding. The relationship between the human-like

feature and the exploration percentage of the map will be discussed further in the result chapter.

3.6 Visually improved using Bézier Curve

After the solution path has been found at the end of the program, from another perspective, these are all anchor points or control points which can be eventually developed as Bézier curves to make the path even more smooth.

Since the number of nodes in the solution path is large, to make the implementation side easier, the cubic Bézier curve calculation will be done connectively every three nodes so that the whole path will become a Bézier curve in a discrete way.

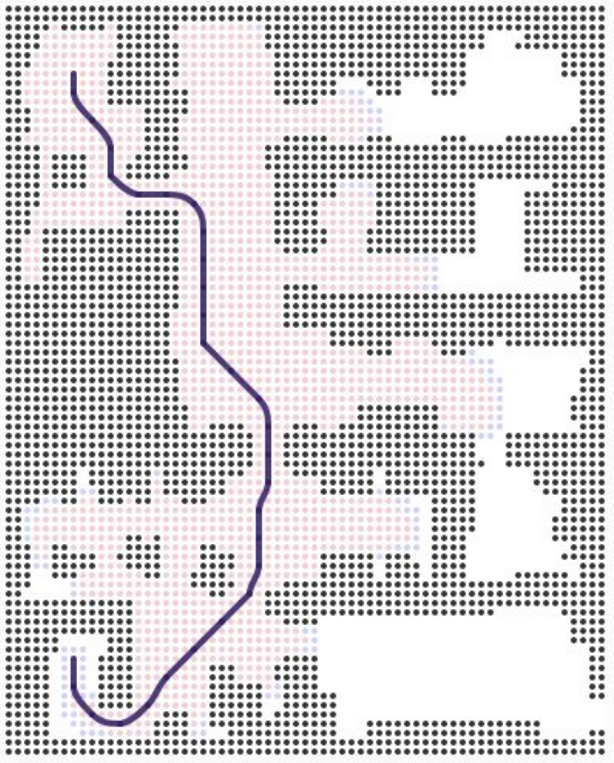

Figure 8: Solution path of control case after Smoothing, Wall Avoiding and Bézier Curve

The solution path now looks elegant and smooth but there are still issues due to the implementation of discretely connecting curves together, which will be further discussed in the Future Work chapter.

Since the Bézier Curve is drawn by calling the built-in function from the “p5” framework, the curve itself is directly drawn on the canvas created, which means the number of nodes being sharp or along the wall won’t change. Again, this Bézier Curving process is just an additional step added on the work before, the main algorithm and nodes forming the solution path remain exactly the same as before.

4 Result

To have a overview of the properties of various versions so far, four attributes are considered while analyzing: the time consumed to have the solution path, the exploration percentage, the percentage of sharp turns within the solution path and the percentage of nodes along the wall within the solution path.

Four attributes are defined as follows:

- **Time consumed to have the solution path (seconds)**
    - Period of time from the program running to the solution path has been found, or no solution for the case
- **Exploration percentage (%)**
    - Number of nodes in the open set array and closed set array, divided by all the white nodes (free area) of the map
- **Percentage of sharp turns (%)**
    - Turns smaller than or equal to 90 degrees in the solution path, divided by the total number of turns in the solution path
- **Percentage of nodes along the wall (%)**
    - Number of nodes have walls as one of the neighbours, divided by the total number of nodes in the solution path

100 test cases have been run on the control case map “den312d.txt” with four versions of algorithms improved from A* and normal A* itself. To see a trend of how attributes change according to which version used, the average number of the four attributes from 100 cases was chosen to form the analyzation diagram.

To decide whether a path is human-like or not, the latter two attributes must be noticed carefully, namely the percentage of sharps turn in the solution path and percentage of nodes along the wall in the solution path. Higher the number of these two factors, it means the solution path is sort of away from the human-like characteristic aimed from the beginning of the research.

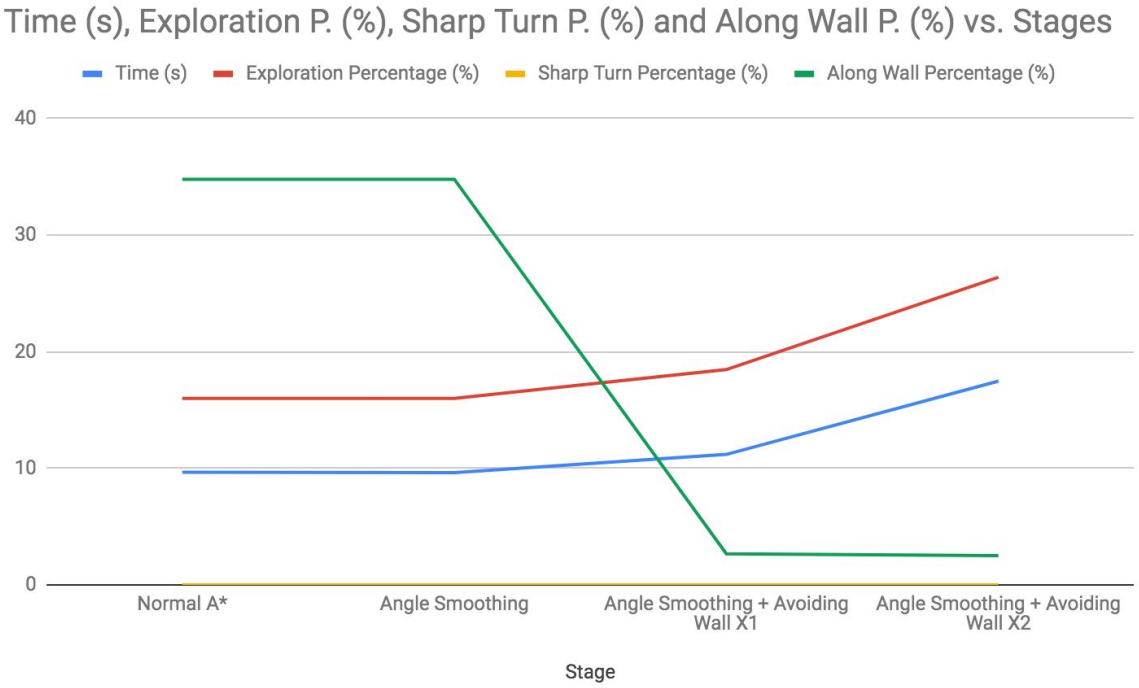

Figure 9: Summary diagram of the trend - Four attributes vs. Various stages of the algorithm

Stage	Time (s)	Exploration Percentage (%)	Sharp Turn Percentage (%)	Along Wall Percentage (%)
Normal A*	9.6708	16.0035	0	34.7819
Angle Smoothing	9.6295	16.0035	0	34.7895
Angle Smoothing + Avoiding Wall X1	11.194	18.4652	0	2.6697
Angle Smoothing + Avoiding Wall X2	17.479	26.3853	0	2.5202

Table 1: Summary diagram of the trend - Four attributes vs. Various stages of the algorithm

From the diagram, there is a clear trend that as more and more constraints are added to the algorithm, the percentage of nodes along the wall has become lower and lower.

There is a violent down of the “Along Wall Percentage” curve when one-step Wall Avoiding has been implemented: the percentage of nodes along the wall has dropped approximately 30%; After the two-step Wall Avoiding implemented, there is another small dropped down which is about 0.1%.

At the same time, the time consumed to run the algorithm and the exploration percentage of the map has been generally gone up as algorithms with more constraints and properties are being used. The exploration percentage of the map has increased by approximately 10% after two-step Wall Avoiding has been implemented, which means the cost of using the two-step version of Wall Avoiding algorithm is a bit huge considering time and exploration rate.

In fact, the version of the improved algorithm which has a nice balance between time consumed, exploration percentage and human-like property should be one with Angle Smoothing and one-step Wall Avoiding. Without raising the cost of time and exploration a lot (only 2%), this version successfully avoiding nodes being along with the obstacles for most of the time.

Another thing to point out is that, as discussed before, even though the Angle Smoothing process doesn't help with finding a more human-like path, it still improved the time consumed to find the path by 0.05 seconds in average from 100 cases. With other attributes about staying the same, this is still progress worthing doing and discussing.

5 Conclusion

As a conclusion, among the four versions of algorithms from the research, the time consumed and exploration percentage of the map become large as more constraints applied to the algorithm in a linear way (Angle Smoothing, then Angle Smoothing + one-step Wall Avoiding, then two-step Wall Avoiding). Although after the implementation of one-step Wall Avoiding which brings the percentage of nodes along the wall by more than 30%, the application of two-step Wall Avoiding doesn't work as efficiently as before and only brings the percentage down by 0.1%.

Considering sometimes with a limited budget of time and space, the version of Angle Smoothing and one-step Wall Avoiding can be the best choice since it lowers the percentage of nodes along the wall a lot, but only explores a little bit more and uses a bit more time.

6 Future Work

The main focus for future work on this research project will be further exploring the use of Bézier Curve for better visualization. The application, for now, is drawing cubic Bézier curve every three nodes and connecting curve fragments together to form a complete path. However, sometimes the relatively sharp point in the solution path is coincidently just the

“connective point” of the discrete application. Then, in this case, this point still remains to be relatively sharp and just not involved in the process of Bézier Curving.

To improve, four order or even more order Bézier Curve can be visited to have a more generalized solution path.

References

[1] Hart, Peter E.; Nilsson, Nils J.; Raphael, Bertram (1972-12-01). "Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'" (PDF). ACM SIGART Bulletin (37): 28–29. doi:10.1145/1056777.1056779. ISSN 0163-5719.

[2] Edelkamp, Stefan; Jabbar, Shahid; Lluch-Lafuente, Alberto (2005). *Cost-Algebraic Heuristic Search*. Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI): 1362–1367.

[3] Wikipedia. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-6. [Online; accessed 21-April-2019]

[4] Michael E. Mortenson (1999). *Mathematics for Computer Graphics Applications*. Industrial Press Inc. p. 264. ISBN 9780831131111.

[5] Wikipedia. Bézier curve. https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-6. [Online; accessed 21-April-2019]

[6] Nathan Sturtevant. 2D Pathfinding Benchmarks. https://movingai.com/benchmarks/grid.html [Online; accessed 21-April-2019]