1. Use of GenAI

Here are all the ways in which we used generative AI tools:

- Brainstorming graph ideas (naming the graphs etc.)
- Troubleshooting the graph visualization algorithm as we were not that familiar with the libraries used
- Coming up with ways to store graphs in a .txt file

2. Describes the Problem

The goal of this project is to find and highlight the shortcomings of the A* graph traversal algorithm by deliberately designing graphs that induce worst-case performance. More specifically, we need to construct a number of graphs that force the program to expand vast parts of the search space to make the solution stray from the optimal path. After, we can make subtle variations to our graphs to highlight what factors are to blame for the weak performance. Lastly we will test these graphs along with their variants on an actual implementation of the A* algorithm to highlight how our constructs reduce performance.

The resulting data will allow us to make an argument, backed by concrete data, that certain graph shapes can cause A* to struggle immensely and stray from the actual optimal solution.

3. The Solutions

By brainstorming graph ideas based on the working principles of the A* algorithm, we came up with graphs that would be very inefficient to solve with A*. Later, we made slight variations of these graphs to show why A* struggles so much with them.

4. Ideas Used/Developed

We decided to have three main parts to our program. Firstly, we knew we need a robust implementation of A* in C++ to keep things efficient, then we needed a way to represent these graphs in a .txt file for ease of use and lastly, we wanted a way to easily visualize these graphs using the same format as the storage solution so that it would be easy for us to show them in our report while making sure we don't have to convert them between two different standards.

In addition, we classify a graph as hard when the percentage of vertices that A* actually expands is high relative to the total vertex count, and we regard our design as successful when a relatively inconsequential modification to the graph causes that expansion percentage to change dramatically.

5. Technical Highlights of the Implementation

5.1 A* Algorithm

The program implements and evaluates the A* algorithm on various graph structures. It reads graph data from text files in a designated folder where each file contains the information to a graph including the nodes with 2D coordinates, edges with weights, and designated start and goal nodes. For each graph, the program runs the A* algorithm multiple times to get accurate performance metrics, and the program outputs statistics including average nodes expanded, average steps taken, execution time, and the final path cost.

Our implementation of A* follows the standard algorithm: we use an open list stored in a binary-heap priority queue with keys determined by the evaluation function f(n)=g(n)+h(n) where g(n) is the minimum cost from the initial node to n and is the heuristic function of the cost from n to the target. Each step selects the node with the lowest f and expanding it ensures that the first time the goal is extracted from the queue we have discovered an optimal path. In all of our experiments the only heuristic is the Euclidean distance from each node's (x,y) coordinates to the goal coordinates' (x,y). Because straight-line distance never overestimates the shortest-path cost in graphs with non-negative edge weights, it is admissible and remains cheap to calculate. We monitor node-expansions, maximum queue size, and wall time to illustrate how this one geographically motivated heuristic works with a range of challenging graph topologies and in what circumstances even a perfect straight-line estimate will not be able to contain the combinatorial explosion.

5.2 Graph Storage Solution

To make the dataset separate from the program, we implemented a way to store graphs in a designated folder. The graphs are stored in .txt files using this format:

- First line : Graph name
- Second line : Number of nodes
- Next N lines : Coordinates for each node (x, y pairs)
    - For example: 24   13 represents a node at position (24, 13)
- After coordinates : Start and goal nodes
    - For example: 0   19 means the start node is node 0 and the goal node is node 19
- Next line : Number of edges
- Remaining lines : Edge definitions (source, destination, weight)
    - For example: 0 1 10 represents an edge from node 0 to node 1 with weight 10

This solution makes it easier for other graphs to be used as it separates the graphs from the actual program.

5.3 Graph Visualization

To make the explanations easier, we implemented a program in python to visualize a given graph. This program visualizes graph data stored in text files from a designated folder. It creates a visual representation of each graph using NetworkX and Matplotlib, displaying nodes as circles (with start nodes in green and goal nodes in red) and edges with their weights. Each graph is shown in a separate window with proper coordinate scaling, grid lines, and axis labels. The program reads graph files that contain information about node coordinates, start/goal nodes, and weighted edges between nodes. It then renders these graphs with clear node labels, edge weight labels, and a legend explaining the color coding.

The visualization includes features like automatic tick mark generation for readable axis values and proper padding around the graph to ensure all elements are visible. This tool is particularly useful for visualizing pathfinding problems, allowing users to see the structure of different graph scenarios that might be used with algorithms like A*.

6. Datasets Used in the Experiments

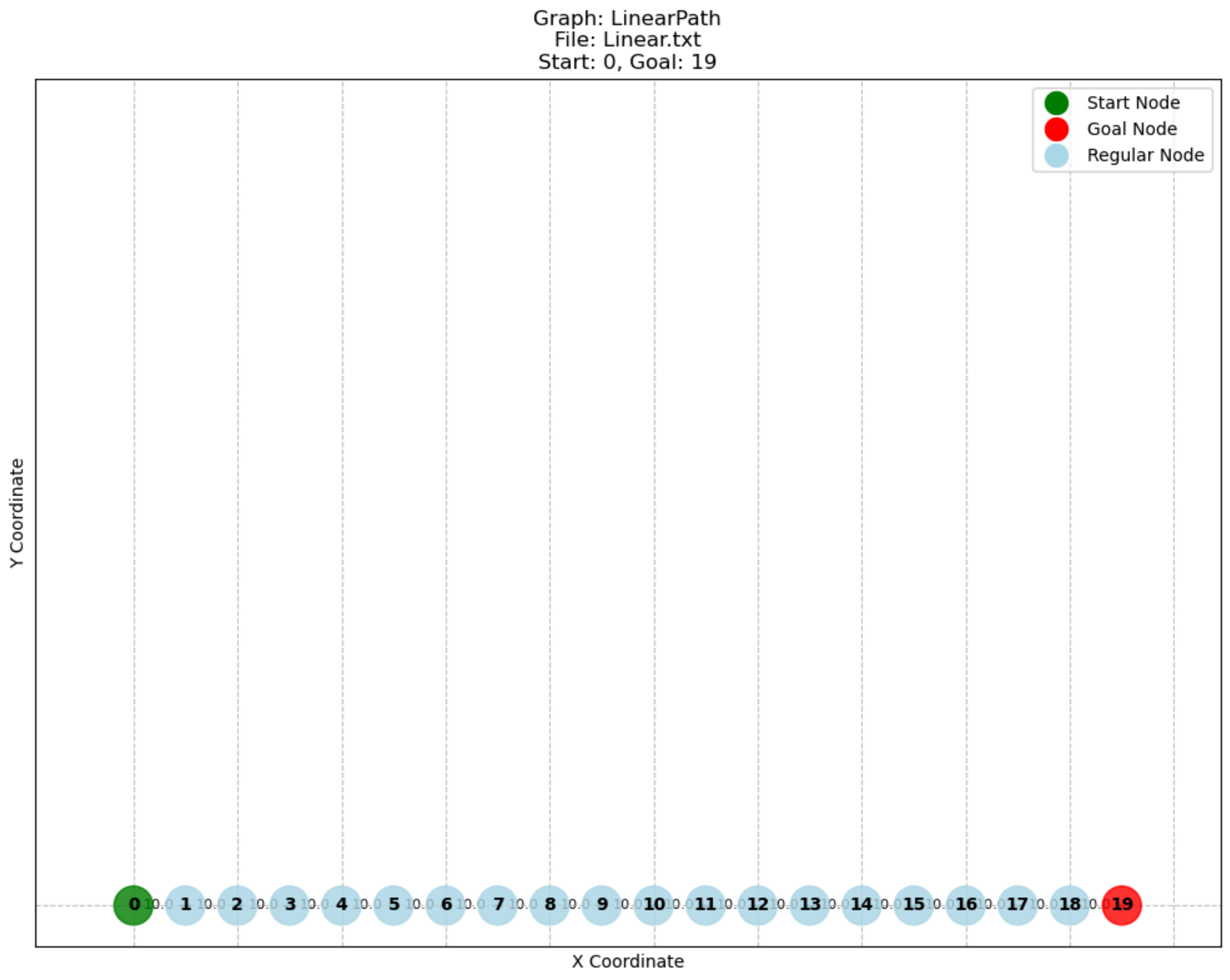
6.1 Original Designed Graphs

Here are the graphs we designed for this project:

1. **Basic Linear Path:** This graph was more for testing than anything else. We wanted to test the A* algorithm testing parameters and the Graph visualization program to make sure everything was working as expected.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Basic Linear Path	20	0 0 10 0 20 0 30 0 40 0 50 0 60 0 70 0	0 & 19	19	0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8	10 10 10 10 10 10 10 10

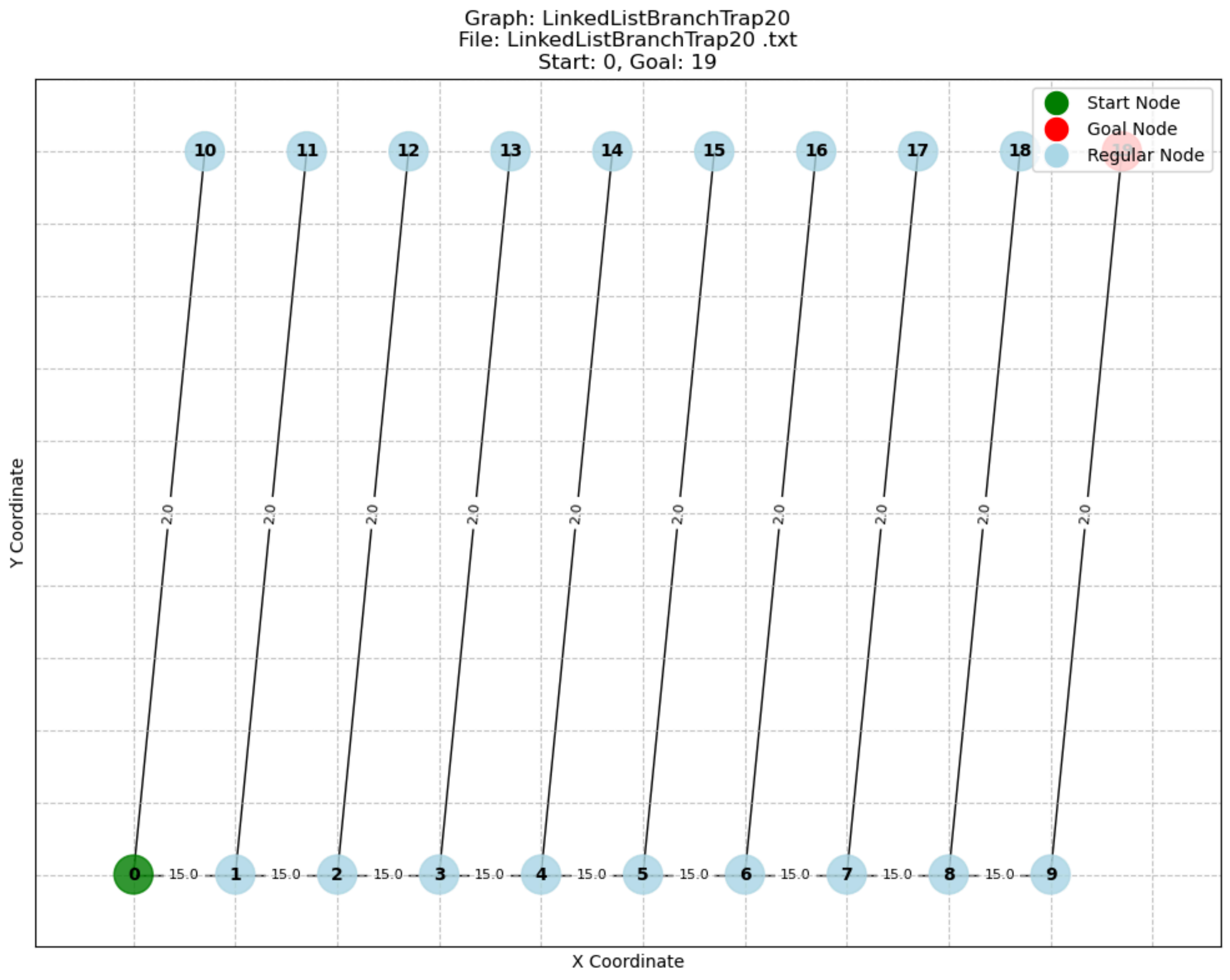
		80 0 90 0 100 0 110 0 120 0 130 0 140 0 150 0 160 0 170 0 180 0 190 0			8 9 9 10 10 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19	10 10 10 10 10 10 10 10 10 10 10



2. **Branching Linked List:** In this graph, since the function looks for a low weighted edge and a closer distance to the goal node, it keeps thinking that exploring the

branches can lead to the optimal solution as they both have a low edge weight and are closer to the goal node.

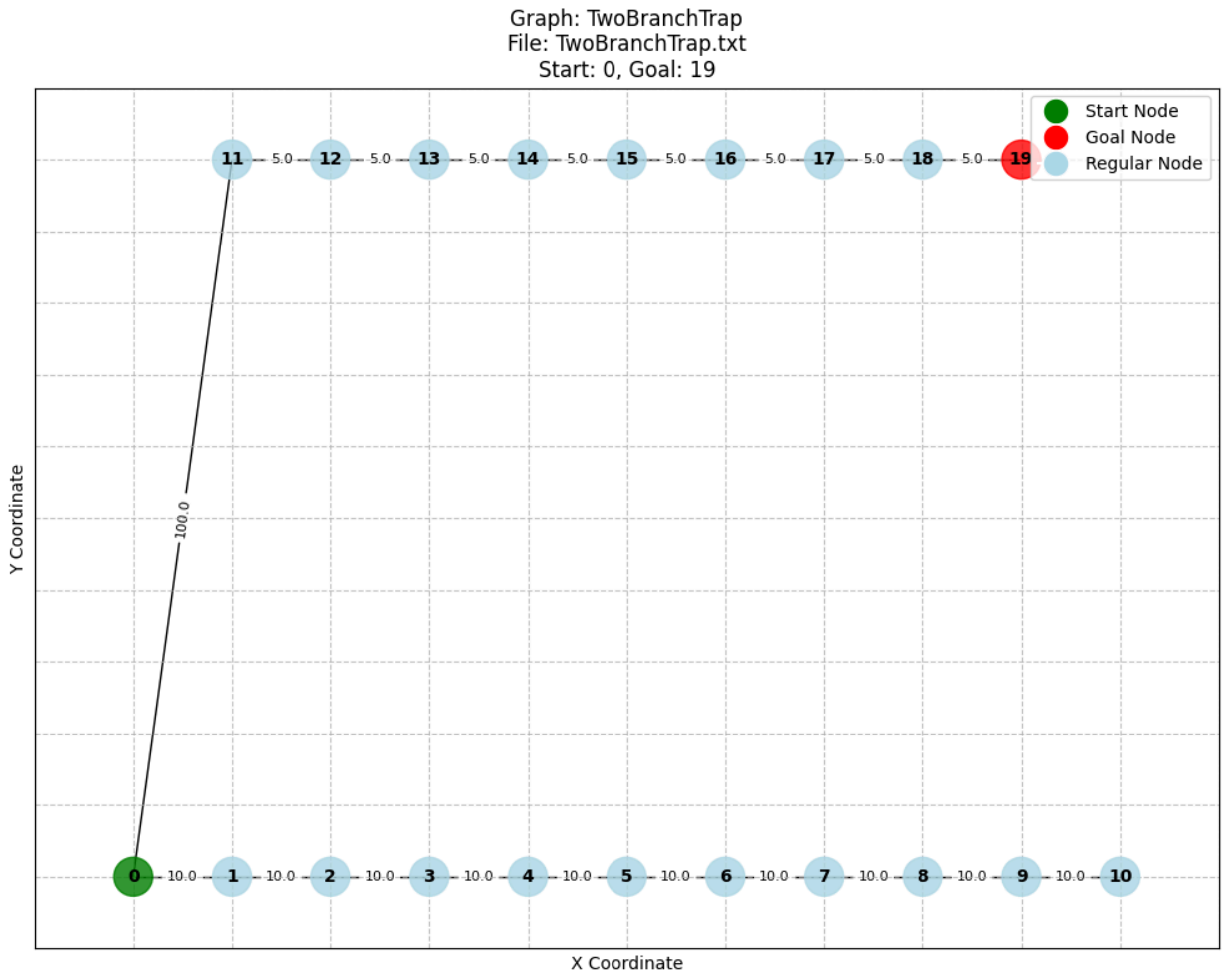
Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Branching Linked List	20	0 0	0 & 19	19	0 1	15
		10 0			1 2	15
		20 0			2 3	15
		30 0			3 4	15
		40 0			4 5	15
		50 0			5 6	15
		60 0			6 7	15
		70 0			7 8	15
		80 0			8 9	15
		90 0			0 10	2
		7 10			1 11	2
		17 10			2 12	2
		27 10			3 13	2
		37 10			4 14	2
		47 10			5 15	2
		57 10			6 16	2
		67 10			7 17	2
		77 10			8 18	2
		87 10			9 19	2
		97 10				



3. **Two Branch Trap:** In this graph since getting closer to the goal node is so expensive, the algorithm gets on the wrong path at the very beginning and reaches a dead end. As a result it has to backtrack all the way and pass the starting node to go to the correct path.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Two Branch Trap	20	0 0 5 0 10 0 15 0 20 0 25 0	0 & 19	19	0 1 1 2 2 3 3 4 4 5 5 6	10 10 10 10 10 10

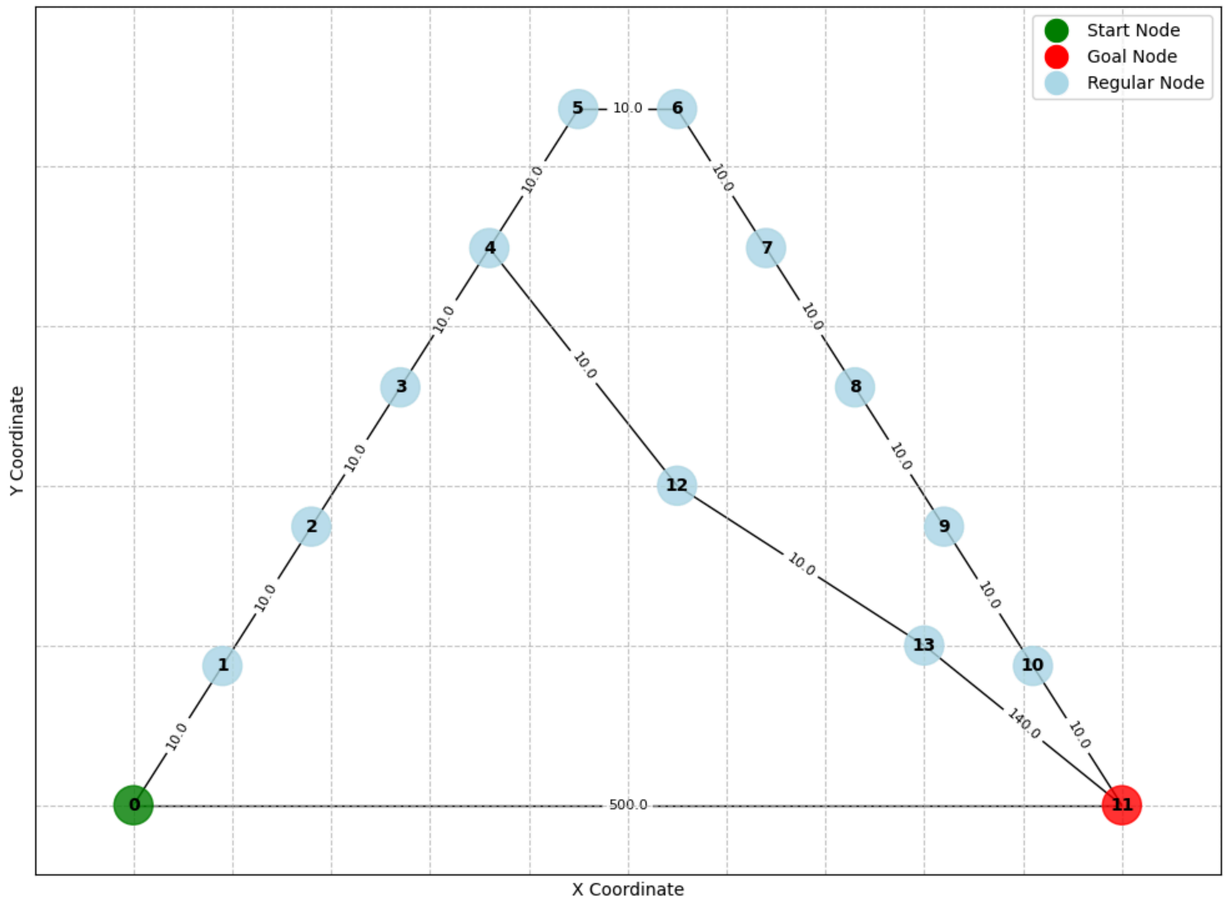
		30 0 35 0 40 0 45 0 50 0 5 10 10 10 15 10 20 10 25 10 30 10 35 10 40 10 45 10			6 7 7 8 8 9 9 10 0 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19	10 10 10 10 100 5 5 5 5 5 5 5 5



4. **Short Expensive Path:** In this graph, the algorithm quickly gets to a point where it has to decide between a shorter path that gets closer to the end goal faster or a path that gets further away. However once it gets to the neighbor of the end node, it realizes that the edge weight is too high for it to make sense. As a result it has to backtrack to the longer route.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Short Expensive Path	14	0 0 9 4.36 18 8.72 27 138 36 17.44 45 21.79 55 21.79 64 17.44 73 138 82 8.72 91 4.36 100 0 55 10 80 5	0 & 11	15	0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 4 12 12 13 13 11 0 11	10 10 10 10 10 10 10 10 10 10 10 10 10 140 500

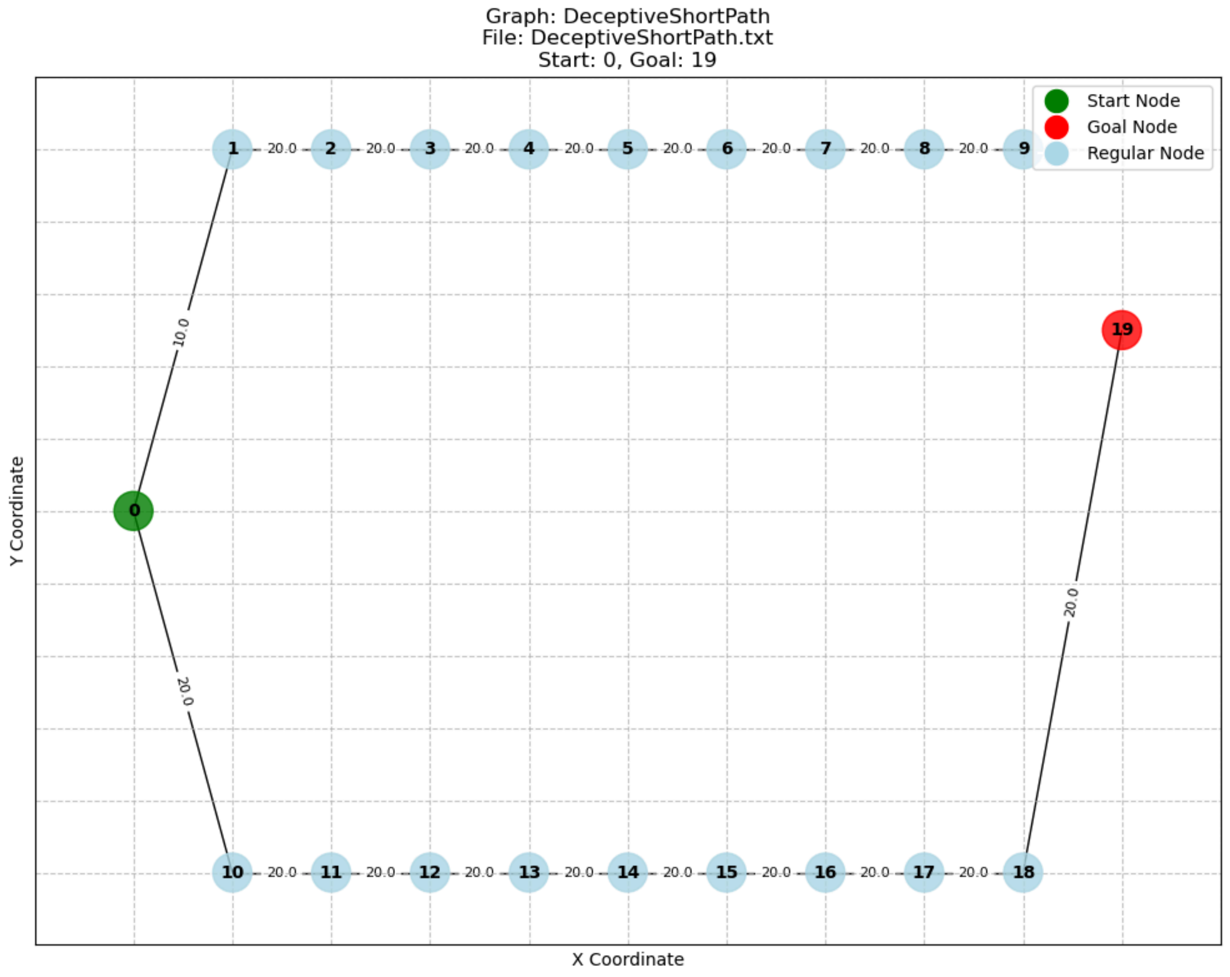
Graph: ShortExpensivePath
File: ShortExpensivePath.txt
Start: 0, Goal: 11



5. **Deceptive Short Path:** In this graph, the algorithm has to decide at the first step which of the two paths it will go down. As a result of the combination of both the lower edge weight and the fact that the top node is closer to the end goal, it tries the path at the top. Only to realise that it reaches a dead end, at which point it has to backtrack all the way to the starting node to change the course from the beginning.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Deceptive Short Path	20	0 10 10 20 20. 20 30 20	0 & 19	20	0 10 0 1 1 2 2 3	20 10 20 20

		40 20 50 20 60 20 70 20 80 20 90 20 10 0 20 0 30 0 40 0 50 0 60 0 70 0 80 0 90 0 100 15			3 4 4 5 5 6 6 7 7 8 8 9 10 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19 0 10	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20



6.2 Variations of the Graphs

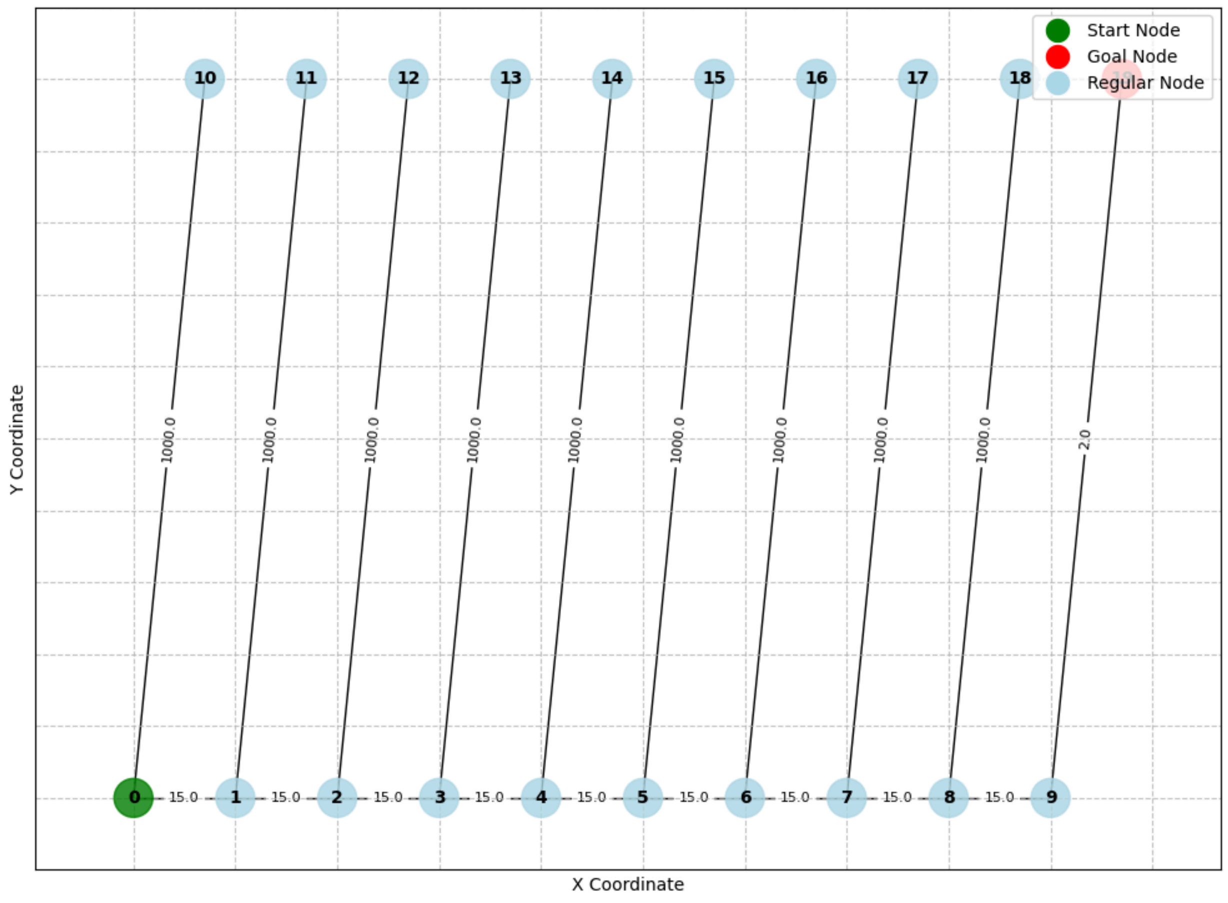
Here are the variations we came up with where we tried to make small changes in the graphs that have a sizable effect on the number of vertices traversed until the solution is found.

1. **Basic Linear Path:** Since this graph was mostly meant for testing, we did not modify it to have a variant of it as any changes we would make would either be inconsequential or too drastic to the point where it did not make sense.

2. **Branching Linked List:** For this graph, we can think of the branch edge weights getting more expensive which stops the algorithm from checking the branches, thus not losing time.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Branching Linked List Variant	20	0 0	0 & 19	19	0 1	15
		10 0			1 2	15
		20 0			2 3	15
		30 0			3 4	15
		40 0			4 5	15
		50 0			5 6	15
		60 0			6 7	15
		70 0			7 8	15
		80 0			8 9	15
		90 0			0 10	1000
		7 10			1 11	1000
		17 10			2 12	1000
		27 10			3 13	1000
		37 10			4 14	1000
		47 10			5 15	1000
		57 10			6 16	1000
		67 10			7 17	1000
		77 10			8 18	1000
		87 10			9 19	2
		97 10				

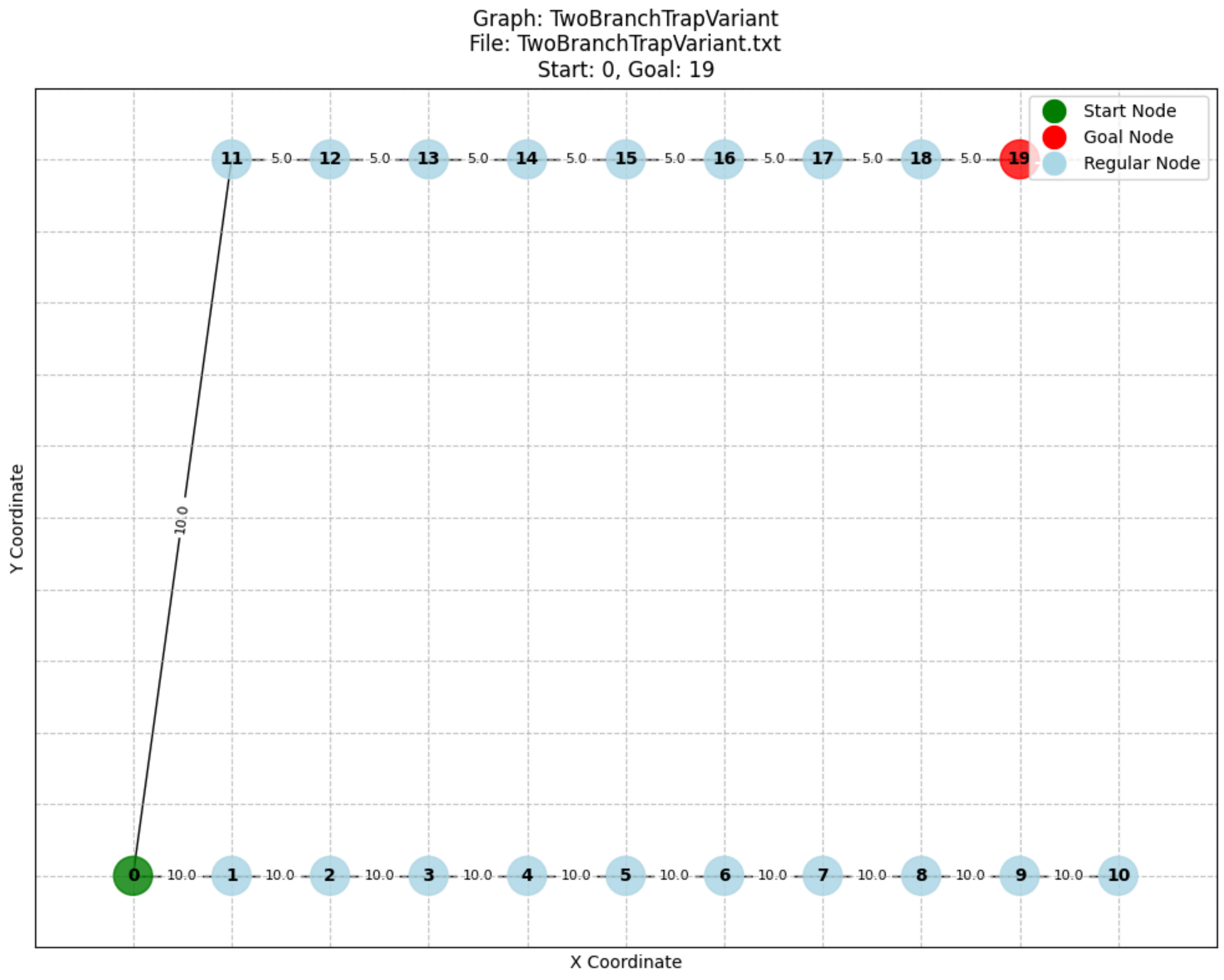
Graph: LinkedListBranchTrapVariant
File: LinkedListBranchTrapVariant.txt
Start: 0, Goal: 19



3. **Two Branch Trap:** For this graph, the edge weight being adjusted for one edge going from node 0 to node 11 results in the algorithm making the right choice in the very beginning. For a real life application we can think of this as a road being expanded so that it takes less time to travel and thus has a lower edge weight.

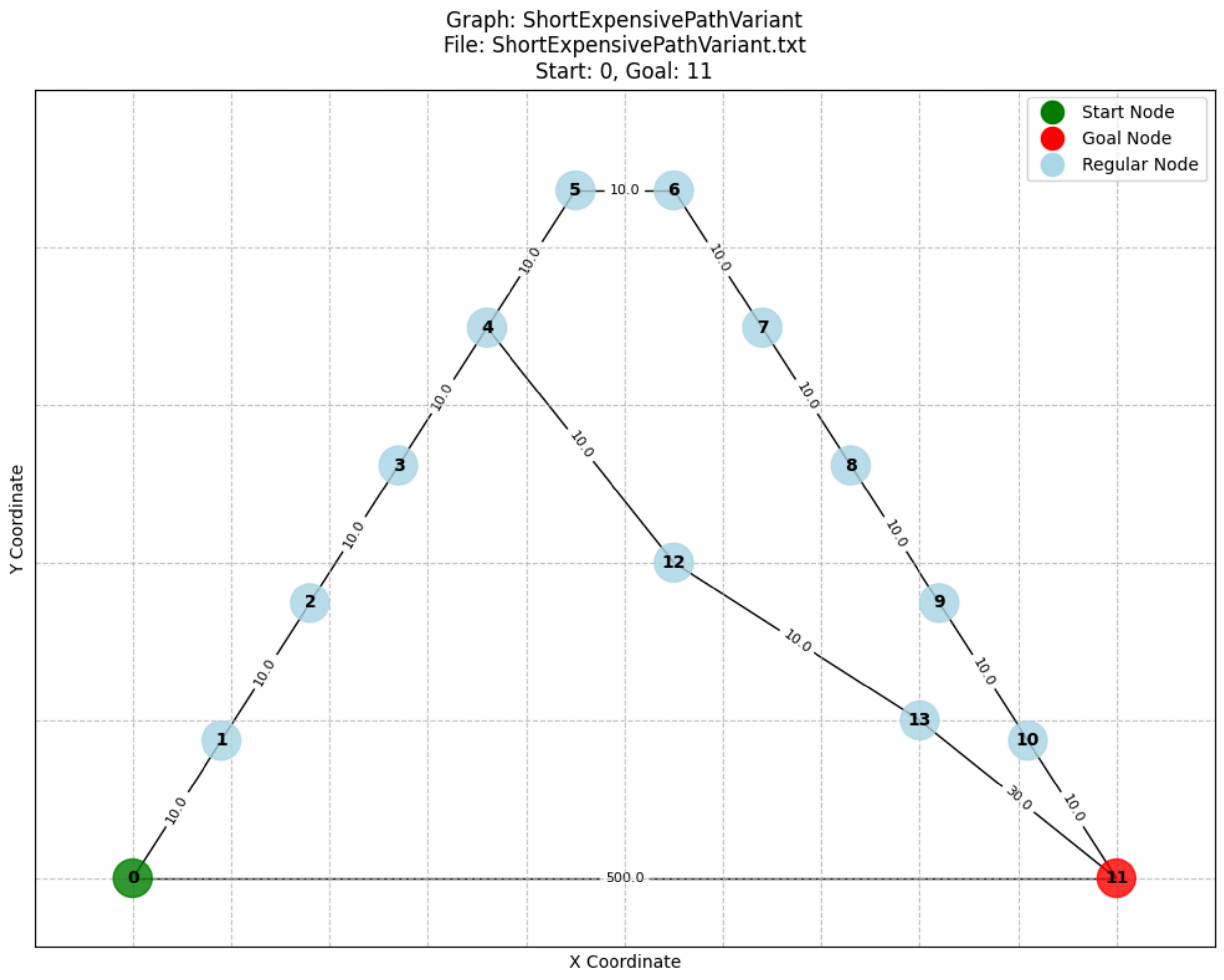
Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Two Branch Trap Variant	20	0 0 5 0 10 0 15 0 20 0 25 0	0 & 19	19	0 1 1 2 2 3 3 4 4 5 5 6	10 10 10 10 10 10

		30 0 35 0 40 0 45 0 50 0 5 10 10 10 15 10 20 10 25 10 30 10 35 10 40 10 45 10			6 7 7 8 8 9 9 10 0 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19	10 10 10 10 10 5 5 5 5 5 5 5 5



4. **Short Expensive Path:** For this graph, similar to the previous one, one edge weight being adjusted is enough to make A* solve it efficiently. The edge in question is from node 13 to node 11. This is where the algorithm normally starts to backtrack as the edge weight is too high. We can think of this as an old crumbling road being restored so that it isn't such a big deal to cross anymore.

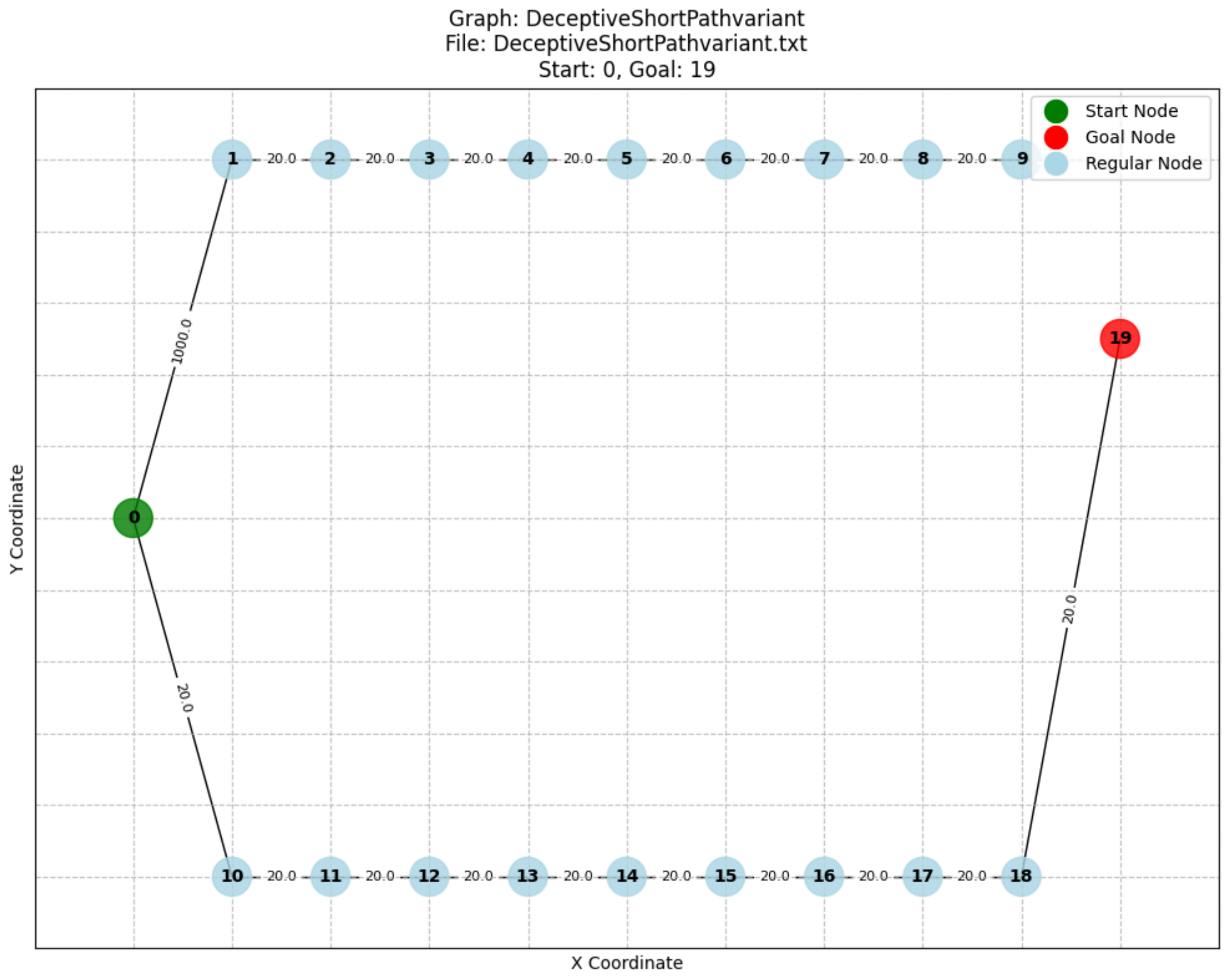
Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Short Expensive Path Variant	14	0 0 9 4.36 18 8.72 27 13.08 36 17.44 45 21.79 55 21.79 64 17.44 73 13.08 82 8.72 91 4.36 100 0 55 10 80 5	0 & 11	15	0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 4 12 12 13 13 11 0 11	10 10 10 10 10 10 10 10 10 10 10 10 10 30 500



5. **Deceptive Short Path:** For this graph, simply adding an edge between the nodes 9 and 19 that doesn't have an abnormally high edge weight solves the problem. We can think of this as a short and cheap road that was built between the goal and the already existing road that ended at node 9.

Name	Vertex Count	Vertices	Start & End Nodes	Edge Count	Edges	Edge Weight
Deceptive Short Path Variant	20	0 10 10 20 20 20 30 20 40 20 50 20	0 & 19	20	0 10 0 1 1 2 2 3 3 4 4 5	20 1000 20 20 20 20

		60 20 70 20 80 20 90 20 10 0 20 0 30 0 40 0 50 0 60 0 70 0 80 0 90 0 100 15			5 6 6 7 7 8 8 9 10 11 11 12 12 13 13 14 14 15 15 16 16 17 17 18 18 19 0 10	20 20 20 20 20 20 20 20 20 20 20 20 20 20



7. The performance with Comparison with Proper Competitors

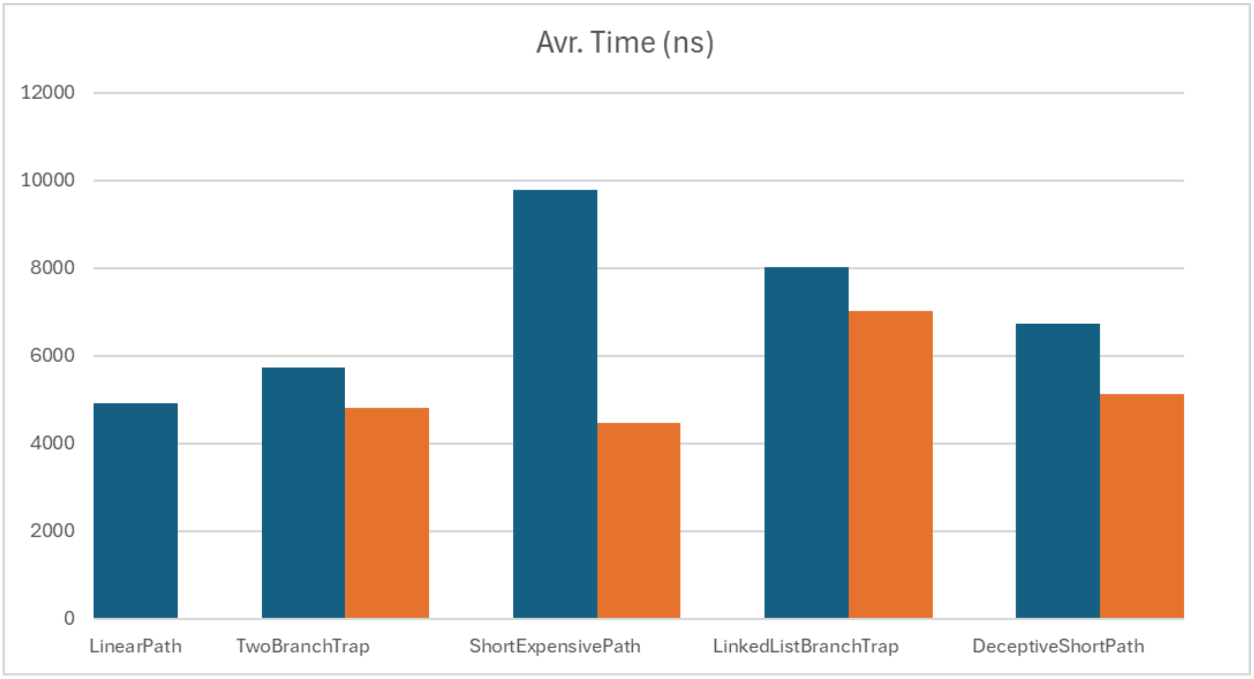
The experiments were run on a laptop with an Intel i7-12700H that has a base clock speed of 2.3GHz and 16 GB of RAM running Windows 11.
After testing the new variant graphs against the original ones we are left with this table:

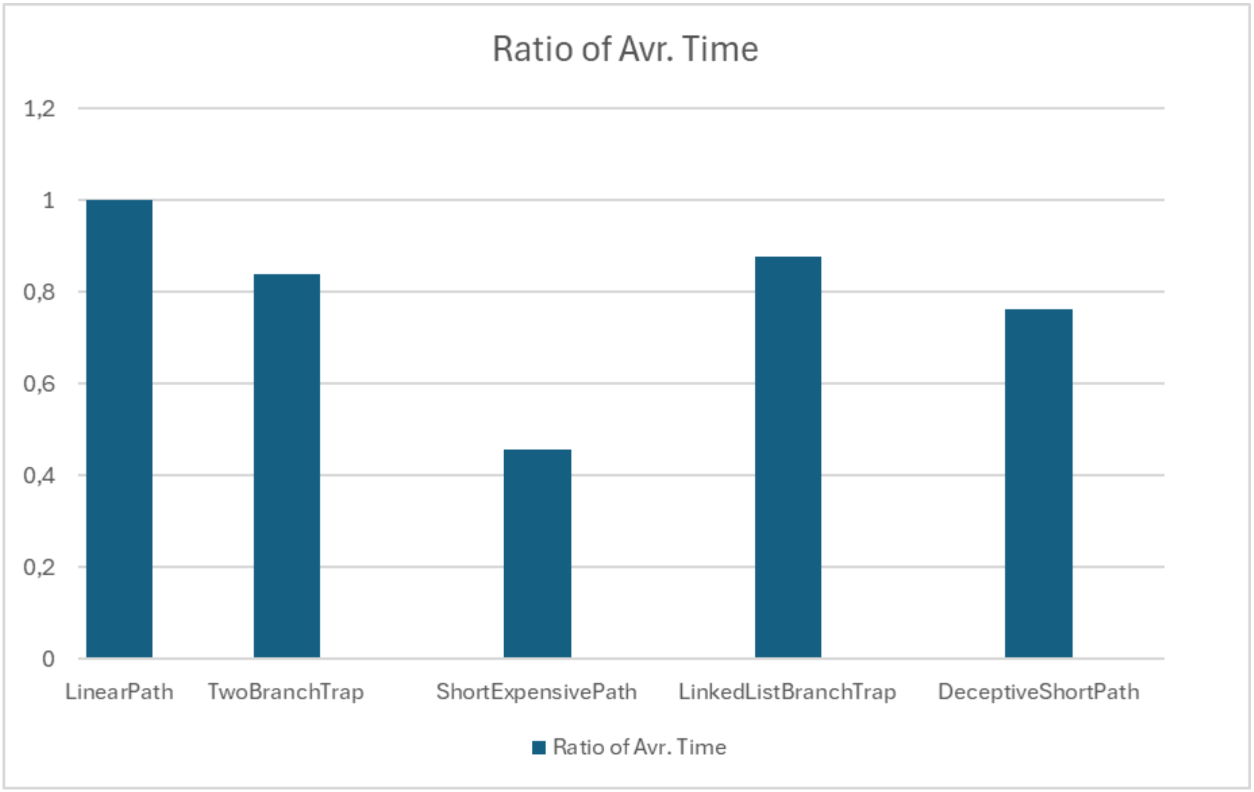
Graph Name	Version	Avr. Time (ns)	Nodes	Avr. Nodes Expanded	Ratio of Avr. Time	Ratio of Avr. Nodes Expanded
Basic Linear Path	Regular	4920	20	20	1	1
Two Branch Trap	Regular	5740	20	20	0,83902439	0,50
	Variant	4816	20	10		
Short Expensive Path	Regular	9779	14	14	0,455772574	0,571428571
	Variant	4457	14	8		
Branching Linked List	Regular	8009	20	20	0,875390186	0,55
	Variant	7011	20	11		
Deceptive Short Path	Regular	6740	20	20	0,762166172	0,55
	Variant	5137	18	11		

7.1 Visualizations

7.1.1 Time Comparison

Looking at the table form before we can analyze the graphs based on two main factors. First is the average time taken and how the regular graphs compare to their variations.

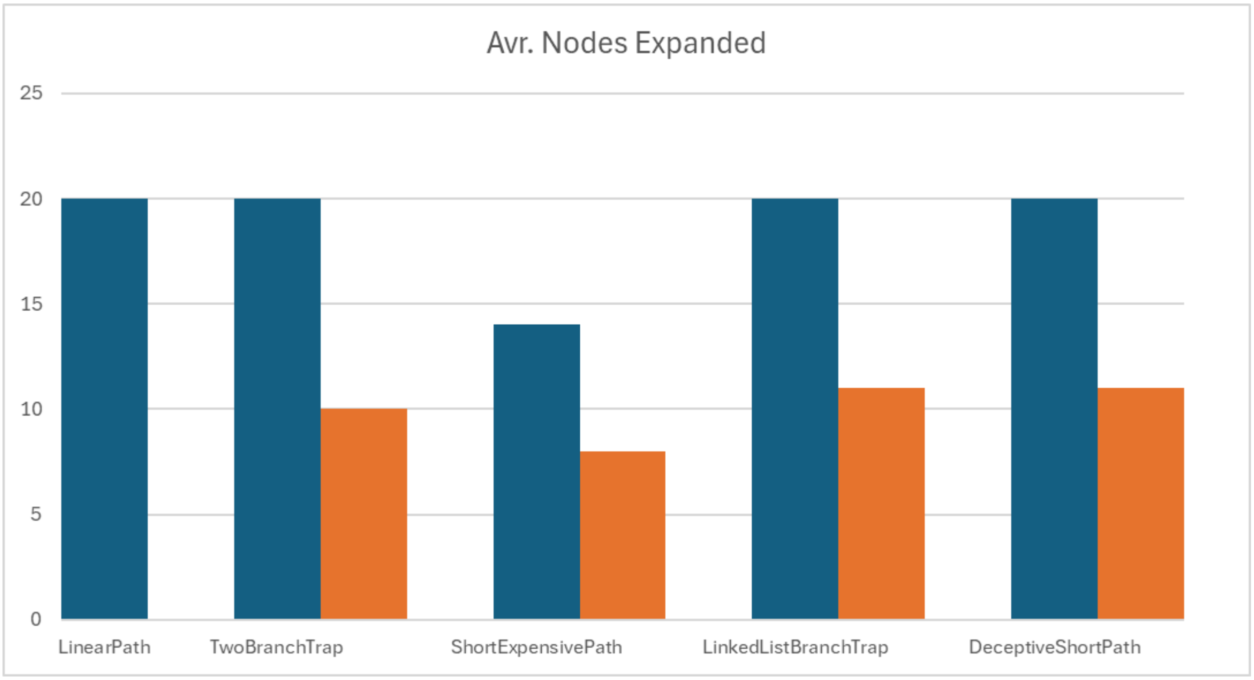


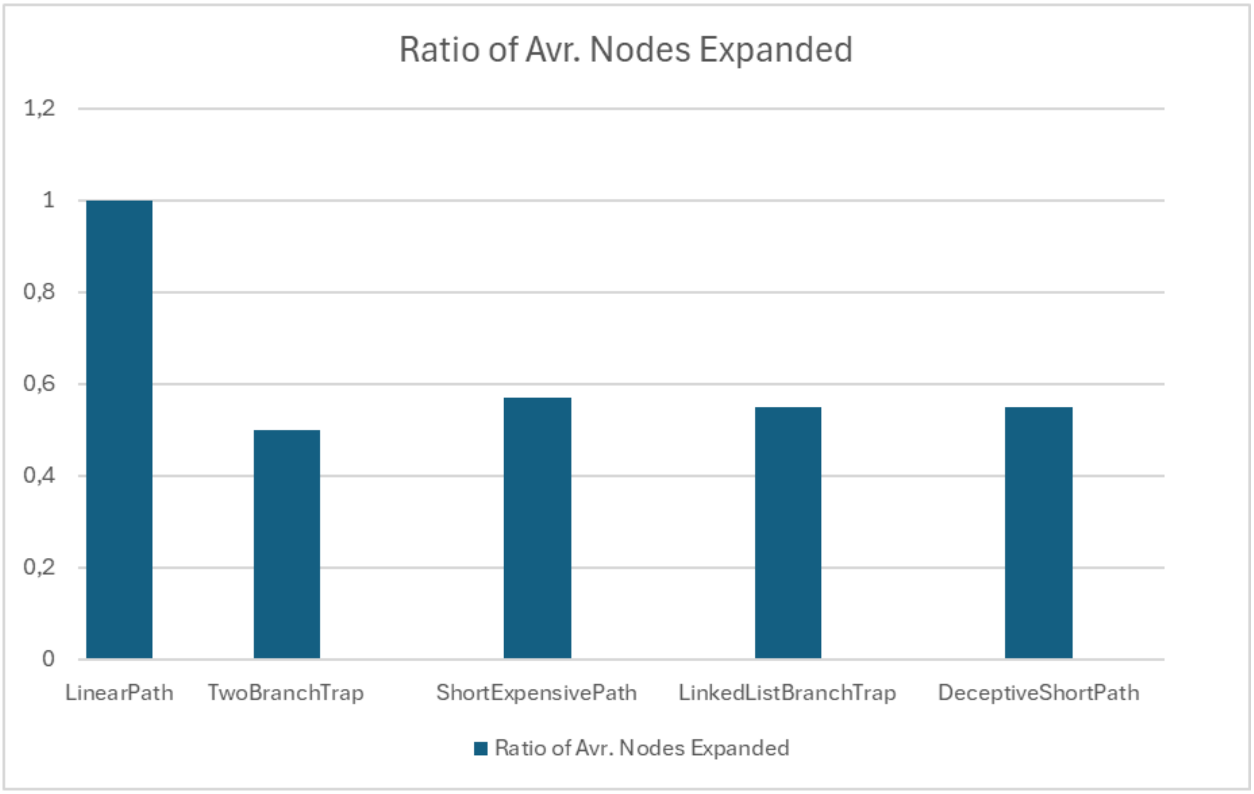


Looking at these two graphs we see a side by side comparison of the time it took for the algorithm to finish execution and on the second one we see the ratio of time for each type of graph between the original and variant kinds. From the second graph we see that the time it took to run the algorithm reduced by anywhere from %15 to %45.

7.1.2 Nodes Expanded Comparison

The second factor we can use in our analysis is the number of nodes expanded while the algorithm runs.





This is where we see an even more drastic change compared to the time analysis. By looking at the second graph we can see that for the graphs that we made variants for, the number of nodes expanded reduced by around %50.

Work Distribution Table

Name and ID	Tasks
Janok N. Dincer 24141212X	Added the measurements to the A* implementation Coming up with the original graphs Graph visualization code Visualizing the original graphs Writing the report Graph Storage Solution
Cagan Cakir 24140489X	A* Implementation in C++ Coming up with the original graphs Coming up with the variant graphs Visualizing the variant graphs

P.S. We did the entire project together, meeting up multiple times. Anything that is credited to one of us is simply because that person had more impact in it, in reality we did every part together.