Ticket to Tech

Group Members: Carrie Houston, Makenzie Kadjeski, & Jeny Thomas

Summary:

This project compares the amount of time it takes for several different path planning algorithms- Breadth-First Search, Depth-First Search, A Search, and Bidirectional Search- to find the shortest path between two given vertices. This program receives user input as the start and end point, prints the path taken, and the time it took to compute that path. Our data is based off the buildings at Tennessee Tech and their relative locations to each other.*

1. Overview of Problem:

On college campuses, one of the biggest challenges is navigating between classes in a timely manner. Knowing the optimal path is essential, especially when students have consecutive classes and time constraints.

These challenges inspired our project, Ticket to Tech. With the assistance of Generative AI, we implemented four different searching algorithms to ensure students take the best path, including BFS (Breadth First Search), DFS (Depth First Search), A* Search, and Bidirectional Search. Our objective was to find the most efficient algorithm for determining the optimal path between two places.

2. Objective/Research:

Before experimentation, we brainstormed initial questions we wanted to answer. What are the time efficiencies of each algorithm? How would the algorithms' results vary with different input sizes? Are certain algorithms more efficient with certain input sizes?

Based on our research, we expect bidirectional search to be the most efficient algorithm. Bidirectional search implements BFS in two different directions. Bidirectional search is used in puzzle solving, pathfinding in AI navigation systems, and network analysis (GeeksForGeeks). The BFS algorithm starts from a vertex and explores all the adjacent vertices before going deeper. BFS real life applications include shortest pathfinding, bipartite check, and level-order traversals (Radian). The DFS algorithm begins at a vertex, follows a path as deep as possible, and then backtracks to explore alternatives. DFS algorithms are applied in topological sort, cycle detection, and searching for connected components. (Radian). A* uses a heuristic function to determine which path is most optimal. A heuristic function provides an estimated cost of reaching a given node, which allows the algorithm to prioritize promising paths. A* is used in video games, navigation systems, robotics, and network systems (Kumar).

Since bidirectional search uses two searching algorithms at once, we believe that its implementation will be more efficient than any single searching algorithm. The objective of our project was to determine which algorithm was the most efficient for finding the optimal path between two places.

3. Description of Experiment(s):

Three real-world datasets of different sizes were constructed to map Tennessee Tech's campus. Each dataset was represented as an adjacency matrix and used as a graph input for each algorithm. To compare test sizes, we had a small graph with four vertices, a medium graph with eight vertices, and a large graph with sixteen vertices. The four vertices on the small graph were the Ashraf Islam Engineering Building (AIEB), Bruner Hall, the Library, and Roaden University Center (RUC). The eight vertices on the medium graph were the AIEB, Bruner Hall, Lab Science Commons (LSC), Library, RUC, Red Parking Lot, Prescott Hall, and Stonecipher Lecture Hall. The sixteen vertices on the large graph were the AIEB, Bruner Hall, the Library, RUC, the Red Parking Lot, Prescott Hall, the LSC, Stonecipher Lecture Hall, Brown Hall, Clement Hall, the Bryan Fine Arts Building (BFA), Memorial Gym, Bell Hall, Derryberry, the Fitness Center, and Ray Morris Hall. Initially, we were planning to implement weighted graphs; however, we realized that BFS counted the number of edges between vertices rather than the cost of each edge. From this observation, we decided to implement an unweighted connected graph in the C++ programming language. Later, we implemented our program in Python on Kaggle.

Once we finished generating the input, we began to implement BFS, DFS, A* Search, and Bidirectional Search and using base code provided by Generative AI. The original base code was designed to interpret the graph input as a coordinate plane, which conflicted with our adjacency matrix graph representation. To address this problem, we applied prompt engineering to alter the code, allowing the algorithms to correctly

navigate the graph using the matrix structure. We then updated the code to print out the building names rather than the numbers for our vertices. As a result, our code was more user-friendly and interactive.

Once implementing these steps, the code was executed on Makenzie's MacBook Pro featuring an Apple M3 pro chip, 18 GB memory, and the macOS Sequoia 15.6.1 operating system. We tested the code five times on each graph size, using the distance between the two farthest points on the small graph (AIEB to RUC) as a baseline. This same point was tested across all three graphs to determine whether the presence of extra vertices affected execution time. In addition, we also tested the two furthest vertices for each graph size and measured the execution time. Specifically, we used AIEB to RUC on the small graph, RUC to Stonecipher on the medium graph, and library parking lot to Bell Hall on the large graph.

Finally, we compared brute force execution time to the other algorithm's execution times for the path from Bruner to RUC. This comparison was limited to the small and medium graphs, as brute force could not efficiently handle the large dataset. The performance metrics were the same for three of the algorithms. For BFS, DFS, and A*, the time complexity is $O(|V|^2)$. (Cite:1,3,4). The time complexity for Bidirectional Search is $O(bd/2)$ where "b" is the branch factor and "d" is the distance (Cite:1).

4. Graphs/Tables:

AIEB to RUC vs. Execution Time (µs)			
Algorithm	**Small**	**Medium**	**Large**
BFS	93.2	88.4	96.4
DFS	23.4	32.6	35.4
A*	37.8	46	50.2
Bidirectional Search	16.4	23.2	18.4

Table 1: Average execution times for the path from AIEB to RUC.

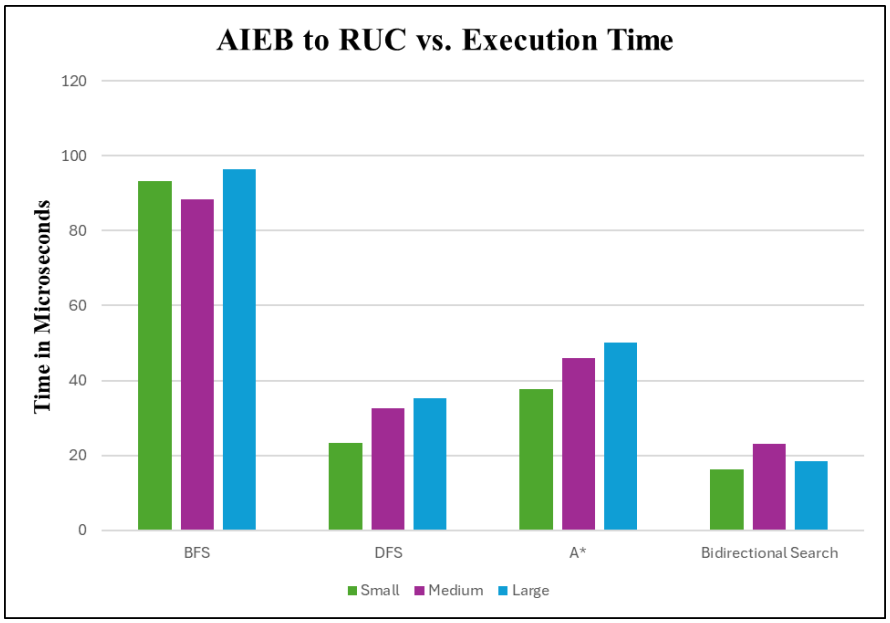

Figure 1: This figure illustrates Table 1 shown above.

Graph Size vs. Farthest Point Execution Time (µs)			
Algorithm	**Small**	**Medium**	**Large**
BFS	93.2	56.6	99.4
DFS	23.4	17.6	28.2
A*	37.8	38.6	73.6
Bidirectional Search	16.4	18.8	30.2

Table 2: This table shows the average execution time of five trials from each graph. The small graph's furthest path was from AIEB to RUC. The medium graph path's furthest path was from the RUC to Stonecipher. The large graph's furthest path was from the Red Parking Lot to Bell Hall.

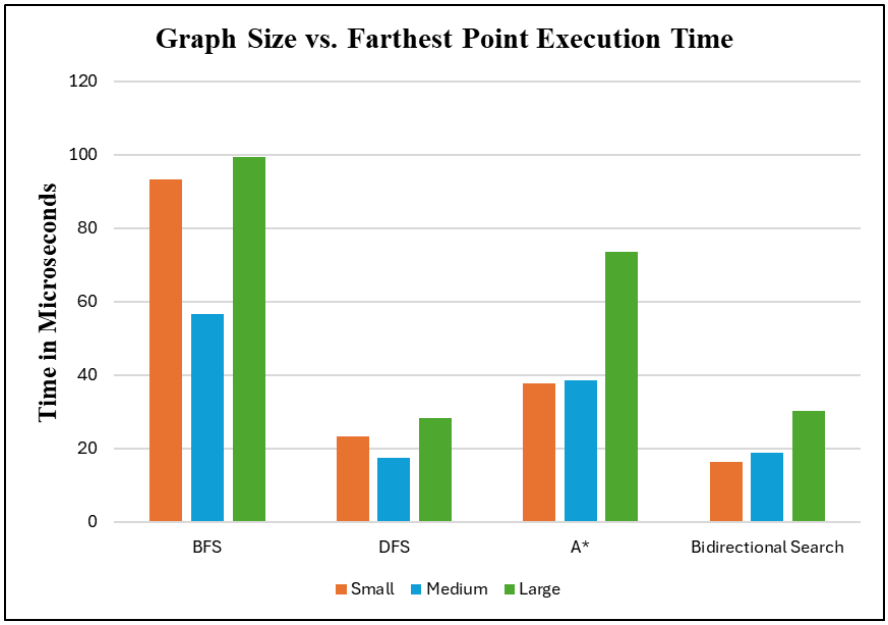

Figure 2: This figure illustrates table 2.

Average of Brute Force Method vs. Path Planning Algorithms		
Algorithm	**Small**	**Medium**
Brute Force	16 µs	211.8 µs
BFS	44.4 µs	52.2 µs
DFS	11.4 µs	19.4 µs
A*	24.2 µs	30.8 µs
Bidirectional Search	10.2 µs	11.8 µs

Table 3: This table compares the brute force to the four other path planning algorithms implemented. Five trials were averaged to generate the data in microseconds.

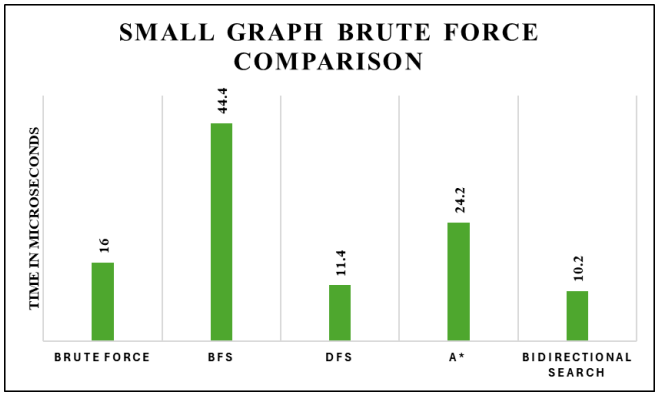

Figure 3: This figure shows the execution time difference between brute force and the other algorithms for the small graph. The data is shown in Table 3.

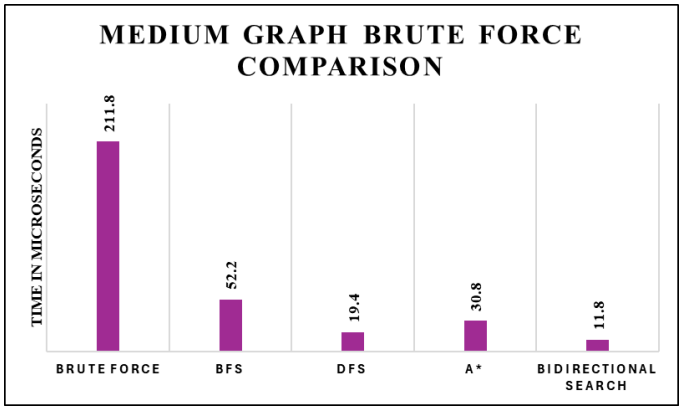

Figure 4: This figure shows the execution time difference between brute force and the other algorithms for the medium graph. The data is shown in Table 3.

5. Results:

In these results, BFS had the longest execution time of all the algorithms, which indicated that searching through the graph level by level takes more work than the other methods. A* had the next longest execution time, followed by DFS. Normally, A* would be much faster than DFS. However, the graphs were unweighted, so the heuristic for A* did not provide a strong advantage. Thus, A* explored nodes similarly to DFS, which results in a similar execution time. Lastly, bidirectional search had the smallest execution time, making it the most efficient algorithm. Based on Table 1, BFS and Bidirectional Search were the most consistent algorithms. Overall, the results behaved similarly as the graph size increased. They clearly highlight how bidirectional search can outperform the other approaches in unweighted graphs.

6. Interpretation:

We tested each algorithm five times with the same input and averaged the results to ensure accurate measurements. We measured performance based on execution time, which is system dependent. We started with a small 4x4 graph to establish a baseline. To evaluate each algorithm's performance scaled, we doubled the input size to create a medium 8x8 graph and a large 16x16 graph. Based on Figure 1 and Table 1, increasing the number of vertices in the graph led to less efficient times in general when searching for the same path. BFS ran faster on the medium graph than the small graph, marking the only exception. We then compared the execution times for the longest source-to-destination paths across all the graphs.

Based on the data from Table 2 and Figure 2, Bidirectional Search was the most efficient algorithm when finding the furthest path and BFS was the least efficient. From Figures 3 and 4, we interpreted that brute force was more efficient with a scaled-down input size. For the smaller graph, the average brute force execution time was less than the average BFS and A* algorithm execution time. However, the brute force execution time increased significantly in the medium graph compared to the other algorithms.

Some restrictions in this experiment were our input size and time. Since we used real life data, our data size was limited to the number of buildings at Tennessee Tech which restricted the scale of the graphs. Our largest data set was a 16x16 matrix, which is not a large input size, so our project might not fully reflect each algorithm's performance on large data sets. Next time, we would like to implement our program that includes every building on campus. Additionally, we would like to use weighted graphs, with the weight reflecting the distance between each building. In the future, we could find the most efficient route rather than the most efficient algorithm.

Further questions include the following:

- Would BFS have been more efficient if the graph was sparse?
- Would the algorithms' execution times improve using an Adjacency List?
- Graph compared to the Adjacency Matrix Graph?
- Should we ask if A* would've worked better if we used a weighted graph?

7. References:

GeeksforGeeks. (2008). GeeksforGeeks. https://www.geeksforgeeks.org/ -used in objective/research

Levitin, A. (2012). *Introduction to the design & analysis of algorithms*. Pearson. -used in objective/research

Kumar, Rajesh. (2024). DataCamp. The A* Algorithm: A Complete Guide | DataCamp

OpenAI. (2025). *ChatGPT* [Large language model]. OpenAI. https://chat.openai.com -experimentation

Radian, C. (2025). *Section 3.5 DFS_BFS Presentation*. -used in objective/research

8. Responsibilities Table:

Group Member	Duties	Contribution %
Carrie Houston	Wrote report, created graphs, calculated performance metrics, created PowerPoint	**33.33%**
Makenzie Kadjeski	Created Jupyter notebook, modified code, tested data, created PowerPoint	**33.33%**
Jeny Thomas	Wrote report, created graphs, imported data, took average of data sets, created PowerPoint	**33.33%**