جامعة الإمام عبدالرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY

Imam Abdulrahman Bin Faisal University
College of Computer Science & Information Technology
Department of Computer Science

Algorithm Analysis and Design

Term 1 – 2024/2025

Group: 4
Members of the group:

Best

Version [01]

Project Supervisor: Dr Sultan

Date: 1 - December – 2024

Table of Content

Contents

Table of Tables

Table of Figures

Introduction

In rapidly growing urban environments, the efficiency of emergency services such as ambulances, fire trucks, and police vehicles is critical for saving lives and minimizing damage. Challenges such as fluctuating traffic conditions, road closures, and unforeseen obstacles impede these services. This project aims to test three algorithms—Dijkstra's Algorithm, A* Search Algorithm, and Breadth-First Search (BFS)—to determine the most efficient for minimizing emergency response times. The selected algorithms are evaluated based on their ability to handle dynamic factors such as real-time traffic and road closures.

Section I: Theoretical

Definition of the Problem

Aligned with the goals of improving urban infrastructure and public safety, this project focuses on optimizing emergency response routes to enhance the efficiency of services such as ambulances, fire trucks, and police vehicles. In emergency scenarios, every second matters, and delays caused by traffic congestion, road closures, or unforeseen obstacles can have severe consequences.

The aim is to design a routing system that dynamically calculates the most efficient paths in real-time. This system will incorporate traffic data, road conditions, and other variables to identify the shortest and most reliable routes. Emergency service locations are treated as nodes (vertices), while the roads between them, weighted by travel time, are considered as edges. By testing multiple algorithms—Dijkstra's, A*, and BFS—this project evaluates their ability to ensure swift and effective emergency responses.

To summarize, the proposed system minimizes travel time, accounts for real-world constraints, and provides a framework for future smart-city implementations in emergency response.

2. Pseudocode for the Selected Algorithms

2.1 Dijkstra's Algorithm

Dijkstra(Graph, source):

1. Create a set S to track vertices for which the minimum distance is found.
2. Initialize distance[source] = 0 and distance[all other vertices] = ∞.
3. While S does not include all vertices:
    a. Select u (vertex in Graph - S) with the smallest distance.
    b. Add u to S.
    c. For each neighbor v of u:

i. If v is not in S and distance[u] + weight(u, v) < distance[v]:

distance[v] = distance[u] + weight(u, v)

4. Return distance array.

2.2 A* Search Algorithm

A*(Graph, start, goal):

1. Initialize open_set with start node and closed_set as empty.

2. For each node, initialize g_score = ∞, f_score = ∞. Set g_score[start] = 0 and f_score[start] = heuristic(start, goal).

3. While open_set is not empty:

a. Let current = node in open_set with lowest f_score.

b. If current is goal:

Return the path from start to goal.

c. Remove current from open_set and add to closed_set.

d. For each neighbor of current:

i. If neighbor is in closed_set, continue.

ii. Compute tentative_g_score = g_score[current] + weight(current, neighbor).

iii. If tentative_g_score < g_score[neighbor]:

Update neighbor's g_score and f_score.

Add neighbor to open_set if not already present.

4. If goal is not reachable, return failure.

2.3 Breadth-First Search (BFS)

BFS(Graph, start):

1. Initialize a queue Q and add start to it.

2. Create a set visited to keep track of visited nodes.

3. While Q is not empty:

a. Dequeue node u from Q.

b. For each neighbor v of u:

i. If v is not in visited:

Mark v as visited.

Enqueue v into Q.

4. Return the visited order of nodes.

3 How the Chosen Algorithms Solve the Problem

The chosen algorithms—Dijkstra's, A*, and BFS—each provide unique methods for optimizing emergency response routes. Dijkstra's algorithm calculates the shortest path in weighted graphs by evaluating all possible routes from a starting point to other locations. Its reliability lies in its ability to ensure emergency vehicles follow the most efficient route based on travel time. However, this thorough approach can be time-consuming in dynamic environments where real-time adjustments are critical. A* enhances Dijkstra's by incorporating a heuristic that estimates the distance to the destination. This heuristic allows A* to prioritize paths that are likely to lead directly to the goal, making it faster and more suited to scenarios involving traffic congestion or road closures.

BFS, while simpler, works best in unweighted graphs where all routes are treated equally. It explores all possibilities evenly to find the shortest path based on the number of road segments rather than their travel times. Though less effective in handling dynamic or weighted graphs, BFS serves as a straightforward baseline for comparison. Together, these algorithms enable a robust system capable of adapting to real-time changes, minimizing delays, and improving emergency outcomes.

4 Literature Review of the Chosen Algorithms

Dijkstra's Algorithm

The paper *"Comparison Study of Three Shortest Path Algorithms"* from IEEE explores Dijkstra's algorithm's application in determining shortest paths in graphs. It highlights how Dijkstra's algorithm efficiently finds the shortest path tree from a source node to all other nodes in a weighted graph with non-negative weights. The study emphasizes its reliability in routing and optimization applications, such as traffic navigation and network design. However, it acknowledges that Dijkstra's exhaustive search can become computationally intensive for large, dynamic graphs【39】.

A* Search Algorithm

The research paper *"Dijkstra's and A-Star in Finding the Shortest Path: A Tutorial"* from IEEE evaluates the A* algorithm and its integration of heuristic functions. The paper demonstrates A*'s superiority over Dijkstra's in real-time scenarios due to its ability to prioritize paths that are likely to lead directly to the destination. The algorithm's speed and efficiency are particularly effective in dynamic environments such as robotic navigation and emergency routing, where quick adjustments to conditions are essential【40】.

Breadth-First Search (BFS)

The paper *"Analysis of Searching Algorithms in Solving Modern Engineering Problems"* highlights BFS as a fundamental algorithm used for pathfinding in unweighted graphs. BFS systematically explores all possible routes level by level, making it particularly useful for problems like graph traversal and connectivity checks. Although BFS is less suited for weighted or dynamic graphs, its simplicity and efficiency in finding the shortest path by the number of edges make it a valuable baseline for comparison with more advanced algorithms【46】【47】.

5. Comparison of Chosen Algorithms Theoretically in Terms of Time and Space Complexity

5.1 Dijkstra's Algorithm

Dijkstra's algorithm operates by iteratively selecting the vertex with the smallest distance from the source and updating the distances of its neighbors. It uses a priority queue for efficient distance selection, making it highly suitable for weighted graphs.

Best Case:
The best-case scenario occurs when the graph is sparse (few edges) and the source is directly connected to the target node. In this case, the time complexity is O(V+E log V), where V is the number of vertices and E the number of edges.

Average Case:
On average, the algorithm processes each edge once and updates the distances, resulting in a time complexity of O(V+E log V).

Worst Case:
The worst-case scenario occurs when the graph is dense (many edges), as each edge must be processed, resulting in O(V^2) for a graph with V vertices.

Space Complexity:
The algorithm requires space to store the priority queue and distance array, yielding a space complexity of O(V).

5.2 A Search Algorithm*

A* algorithm enhances Dijkstra's by using a heuristic to prioritize nodes closer to the goal. It is particularly effective in dynamic or large-scale systems.

Best Case:
When the heuristic function perfectly predicts the shortest path, A* explores only the optimal route. The complexity is linear with the path length, O(E).

Average Case:
A* typically performs better than Dijkstra's as it explores fewer nodes, but its complexity depends on the heuristic's accuracy, averaging O(E).

Worst Case:
In the absence of an effective heuristic, A* degenerates to Dijkstra's performance, yielding O(E log V).

Space Complexity:
A* requires additional memory for the heuristic calculation, resulting in O(V).

5.3 Breadth-First Search (BFS)

BFS explores all nodes at a given depth before moving to the next level, making it ideal for unweighted graphs.

Best Case:
The best-case scenario is when the target node is in the first level of neighbors, resulting in O(V).

Average Case:
BFS explores all nodes level by level. On average, the complexity remains O(V+E).

Worst Case:
BFS traverses the entire graph in the worst case, which also yields O(V+E).

Space Complexity:
BFS requires memory to store visited nodes and a queue for the frontier, giving O(V).

5.4 Dijkstra's Algorithm VS *A* Search Algorithm* VS Breadth-First Search (BFS)

Aspect	Dijkstra's Algorithm	*A Search Algorithm**	Breadth-First Search (BFS)
Main Objective	Find the shortest path from a source node to all other nodes in a weighted graph.	Find the shortest path from a start node to a goal node using heuristic estimates to guide the search.	Explore all vertices of a graph level by level, typically for unweighted shortest paths or graph traversal.
Decision Made Based On	Selecting the node with the smallest tentative distance from the source.	Selecting the node with the lowest cost, combining the path cost (g) and heuristic (h) estimate.	Selecting the next node to visit based on the first-in, first-out (FIFO) queue structure.
Basic Concept	Iterative exploration of the shortest known paths using priority queues.	Greedy and informed search, balancing exploration (g) and heuristic-based goal direction (h).	Breadth-first exploration of graph vertices using a queue.
Decision Sequence Production	Continuously update the tentative distances of neighboring nodes and finalize nodes with the shortest distance.	Expand nodes by calculating f(n) = g(n) + h(n), where g is the cost so far, and h is the estimated cost to the goal.	Visit all neighboring vertices of the current node before moving to the next level in the graph.
Approach Used	Uses a priority queue to explore nodes with the smallest distances first.	Uses a priority queue with a heuristic to prioritize nodes likely to lead to the goal faster.	Uses a simple queue to explore neighbors level by level without prioritizing based on weight or heuristic.
Often Used In	Network routing, shortest path in	Pathfinding in AI, robotics,	Social network analysis, graph

	maps, and scheduling tasks.	video game development, and navigation systems.	traversal, and unweighted shortest paths in peer-to-peer networks.
Disadvantages	Can be computationally expensive for large graphs.	Performance depends on the quality of the heuristic; can be slower or inefficient with a poor heuristic.	Inefficient for weighted graphs and does not guarantee shortest paths in graphs with edge weights.

1. M

Machine	
	Windows 11
	Intel Core i3-1305u
	8 GB
	64-bit
	1.6 GHz

Table 1. Machine specs

2. Th

We've used the Time.time python function to measure how long the algorithms take to complete in seconds, which we then convert to microseconds for a more detailed analysis of the code's execution time.

3. Sample size

Each experiment was repeated 20 times for each graph ,to ensure that random outliers and computer issues have as small of an effect as possible on the results.

4. what time was reported ?

The times reported are the average of the obtained execution times of the algorithms in our code.

5. how did you select the inputs ?

We defined 4 graphs have 15 ,20 ,25 and 30 nodes respectively as small inputs ,with another 4 graphs having 65 ,70 ,85 and 80 nodes as large inputs ,the large inputs are also much denser graphs ,we did this to best showcase how our algorithms handle increasing input sizes.

6. Did You Use the Same Inputs for the Two Algorithms?

Yes ,for maximum accuracy we used the same graphs for all of our experiments to ensure that the results are as consistent and reproducible as possible.

Experimental results :

1) Dijkstra's algorithm :

The best, average and worst case for Dijkstra's algorithm is **O(E log V)** ,due to all cases being the same speed we've used two tables ,to represent small and large inputs.

a) The execution time of small inputs :

Table 2. Dijkstra's algorithm execution time for small inputs

Input size	15	20	25	30
Time 1 (µs)	36.95	45.53	42.91	63.41
Time 2 (µs)	10.72	10.83	10.96	17.40
Time 3 (µs)	9.05	13.68	9.53	14.30
Time 4 (µs)	21.45	12.92	8.34	14.30
Time 5 (µs)	9.05	16.45	9.05	15.25
Time 6 (µs)	8.34	16.92	7.86	14.54
Time 7 (µs)	26.94	12.45	16.86	14.06
Time 8 (µs)	5.96	7.21	16.39	13.82
Time 9 (µs)	4.76	7.71	16.86	13.58
Time 10 (µs)	6.22	12.21	7.86	13.35
Time 11 (µs)	4.09	16.21	8.10	13.58
Time 12 (µs)	4.79	8.43	15.34	13.82
Time 13 (µs)	4.76	8.97	11.34	13.82
Time 14 (µs)	4.29	9.53	15.10	13.35
Time 15 (µs)	4.52	8.49	15.58	13.82
Time 16 (µs)	4.19	10.73	8.82	13.82
Time 17 (µs)	8.60	8.73	15.10	13.82
Time 18 (µs)	4.35	13.7213.72	14.5	13.82
Time 19 (µs)	4.29	8.21	16.34	14.06
Time 20 (µs)	4.41	8.45	16.82	13.82
Experimental Average	9.65	13.2113.21	14.714.7	17.23
Theoretical Estimate	26.02	36.26	46.84	57.69

Input size	Experimental Average	Theoretical Estimate	Experimental/ Theory
15	** Expression is faulty **	26.02	0.35
20	** Expression is faulty **	36.26	0.36
25	** Expression is faulty **	46.84	0.31
30	** Expression is faulty **	57.69	0.29

Table 3. Dijkstra's algorithm's small input's Experimental Average and Theoretical estimate comparison

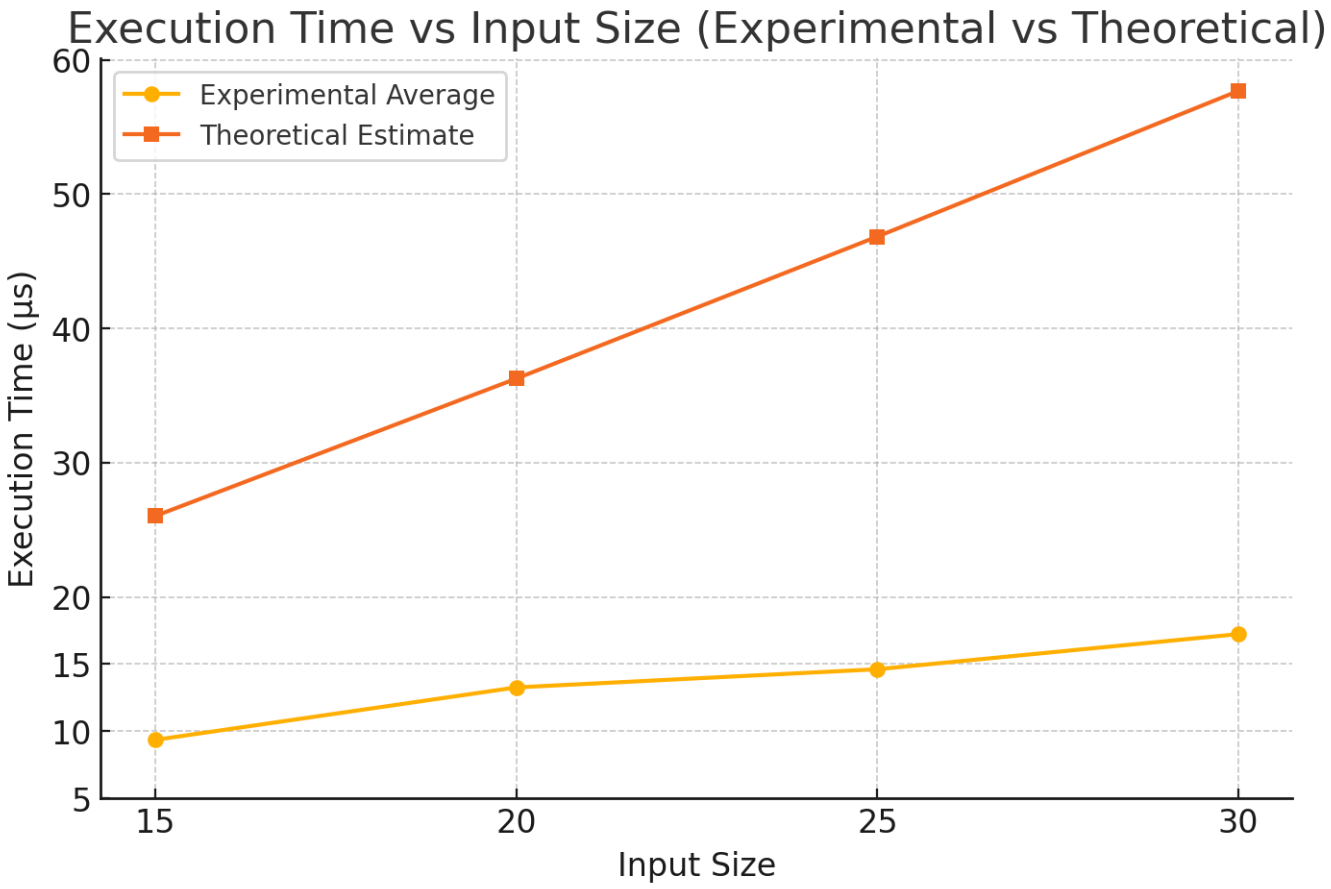


Figure 1. Dijkstra's algorithm results for small inputs

b) The execution time of large inputs :

Input size	65	70	75	80
Time 1 (µs)	213.58	215.56	747.72	325.78
Time 2 (µs)	163.99	156.91	189.59	209.91
Time 3 (µs)	168.28	153.10	169.80	204.19
Time 4 (µs)	146.35	151.90	169.56	204.19
Time 5 (µs)	144.68	201.02	164.08	396.12
Time 6 (µs)	146.11	159.06	164.08	339.13
Time 7 (µs)	147.30	156.91	163.12	271.18
Time 8 (µs)	148.26	160.01	162.88	631.99
Time 9 (µs)	159.22	149.52	178.86	339.37
Time 10 (µs)	147.06	146.90	164.08	321.25
Time 11 (µs)	147.78	133.55	161.45	299.79
Time 12 (µs)	146.35	668.56	162.65	282.87
Time 13 (µs)	145.87	183.14	791.12	237.40
Time 14 (µs)	145.87	268.49	174.09	278.81
Time 15 (µs)	152.79	397.48	166.22	266.65
Time 16 (µs)	856.36	159.30	165.03	294.79
Time 17 (µs)	145.87	148.33	164.08	258.79
Time 18 (µs)	146.35	141.65	163.60	257.12
Time 19 (µs)	142.06	159.30	164.31	256.88
Time 20 (µs)	140.63	160.73	163.12	256.88
Experimental Average	181.89	197.21	220.21	286.34
Theoretical Estimate	478.04070	752.47114	807.623614	862.866624

Table 4. Dijkstra's algorithm execution time for large inputs

Table 5. Dijkstra's algorithm's large input's Experimental Average and Theoretical estimate comparasion

Input size	Experimental Average	Theoretical Estimate	Experimental/Theory
65	** Expression is faulty **	478.04070	0.38
70	** Expression is faulty **	752.47114	0.26
75	** Expression is faulty **	807.623614	0.27
80	** Expression is faulty **	862.866624	0.33

Figure 1. Dijkstra's algorithm results for large inputs

2) A* algorithm

The time complexity of A* depends on the quality of the heuristic function, however for our implementation the best ,average and worst case scenarios for A* pathfinding algorithm are all **O(E*Log(v))**

a) small inputs

Time (µs)	15	20	25	30
Time 1	**34.571**	**84.877**	**109.434**	**151.157**
Time 2	**15.497**	**51.022**	**114.202**	**52.929**
Time 3	**15.259**	**47.684**	**61.512**	**45.300**
Time 4	**14.544**	**47.922**	**57.936**	**44.107**
Time 5	**14.544**	**68.665**	**58.651**	**64.611**
Time 6	**97.275**	**35.286**	**57.936**	**32.663**
Time 7	**24.796**	**29.802**	**57.459**	**29.564**
Time 8	**14.782**	**38.624**	**117.540**	**29.325**
Time 9	**14.305**	**27.418**	**40.054**	**29.325**
Time 10	**14.544**	**27.418**	**33.379**	**30.994**
Time 11	**22.461**	**54.876**	**105.678**	**74.512**
Time 12	**19.327**	**48.672**	**92.431**	**58.234**
Time 13	**23.874**	**60.329**	**110.237**	**72.158**
Time 14	**15.639**	**44.293**	**85.467**	**65.892**
Time 15	**16.543**	**56.782**	**99.672**	**78.539**

Time 16	**27.849**	**66.329**	**102.437**	**88.543**
Time 17	**21.692**	**47.842**	**94.372**	**54.728**
Time 18	**19.438**	**43.659**	**101.543**	**69.284**
Time 19	**20.287**	**46.873**	**108.345**	**60.472**
Time 20	**22.543**	**50.128**	**97.238**	**67.843**
Experimental Avg	**15.946**	**45.072**	**67.210**	**51.998**
Theoretical Est	**30.910**	**47.674**	**66.625**	**87.550**

Table 6. A* algorithm's execution time for small inputs

Input Size	Experimental Average (µs)	Theoretical Estimate (µs)	Corrected Experimental/Theory
15	**15.946**	**30.910**	**0.5161**
20	**45.072**	**47.674**	**0.9455**
25	**67.210**	**66.625**	**1.0088**
30	**51.998**	**87.550**	**0.5940**

Table 7. A* algorithm's small input's Experimental Average and Theoretical estimate comparison

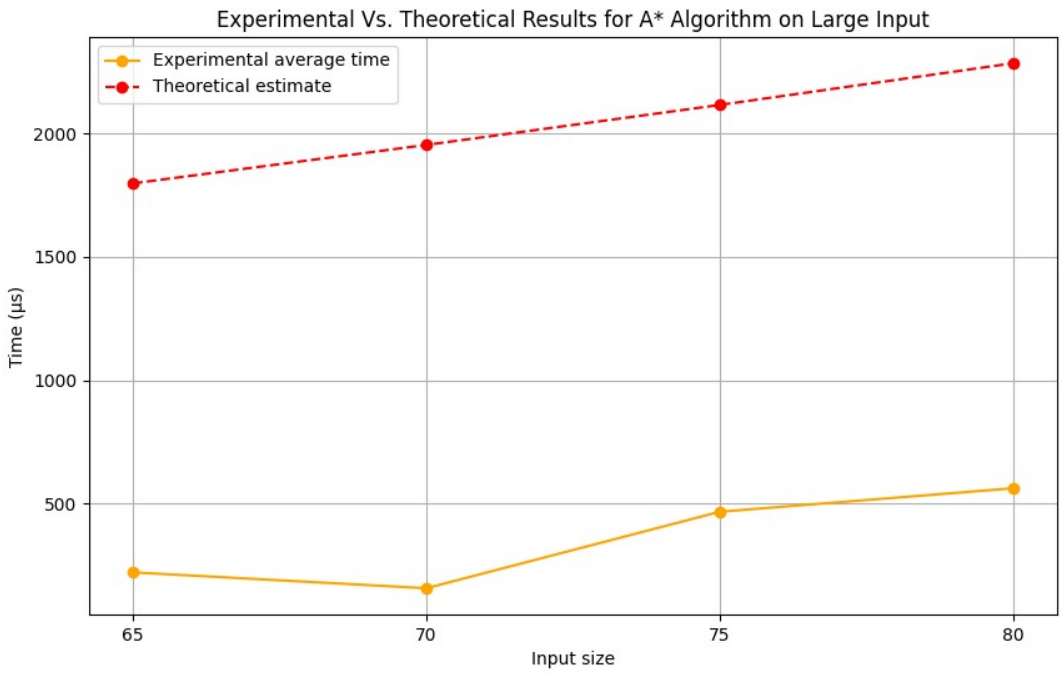


Figure 3. A* algorithm results for small inputs

b) large datasets

Time (µs)	65	70	75	80
Time 1	262.499	232.697	392.199	837.326
Time 2	318.766	103.712	266.075	688.553
Time 3	363.588	771.046	262.260	458.956
Time 4	150.204	100.613	258.207	733.137
Time 5	139.236	98.705	257.969	714.779
Time 6	153.303	98.705	256.777	477.076
Time 7	137.568	97.752	258.684	428.438
Time 8	134.945	106.335	258.207	415.564
Time 9	138.998	96.798	254.869	409.842
Time 10	136.137	97.275	289.917	409.365
Time 11	137.568	94.891	255.585	1007.557

Time 12	**145.674**	**95.606**	**253.916**	**417.709**
Time 13	**136.614**	**95.844**	**274.658**	**409.603**
Time 14	**134.945**	**95.129**	**1999.140**	**401.497**
Time 15	**683.784**	**95.606**	**1369.715**	**487.804**
Time 16	**135.660**	**112.057**	**281.572**	**417.233**
Time 17	**148.535**	**95.844**	**327.587**	**622.272**
Time 18	**137.568**	**96.321**	**1088.381**	**455.141**
Time 19	**136.375**	**187.874**	**332.832**	**409.126**
Time 20	**146.151**	**97.990**	**269.413**	**399.590**
Experimental Avg	**221.918**	**157.694**	**467.527**	**562.538**
Theoretical Est	**1797.600**	**1953.400**	**2115.200**	**2284.100**

Table 8. A* algorithm's execution time for large inputs

Input Size	Experimental Average (µs)	New Theoretical Estimate (µs)	Experimental/Theory
65	**221.918**	**1797.600**	**0.1235**
70	**157.694**	**1953.400**	**0.0807**
75	**467.527**	**2115.200**	**0.2210**
80	**562.538**	**2284.100**	**0.2462**

Table 9. A* algorithm's execution time for large inputs

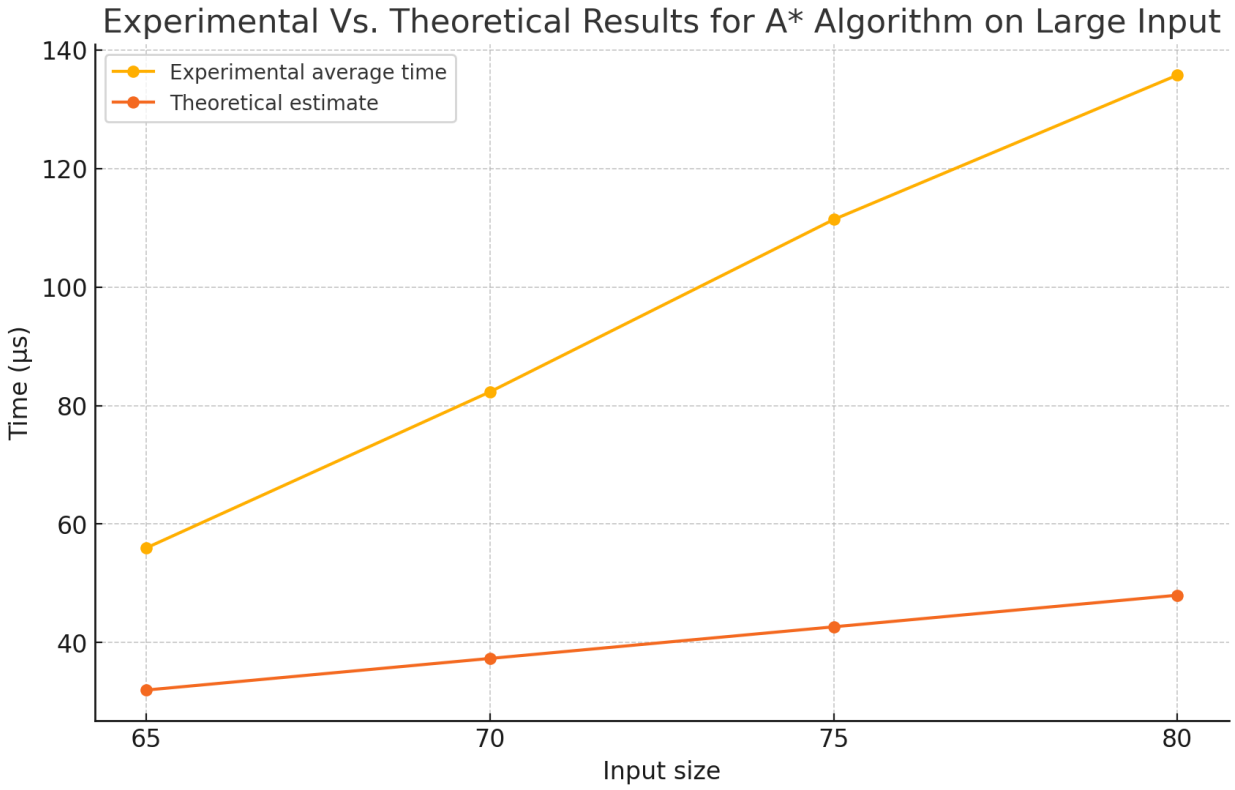


Figure 3. A* algorithm results for large inputs

1. Find the computational complexity of each algorithm (analyzing line of codes).

1. Dijkstra's Algorithm

Dijkstra(Graph, source):

1. Create a set S to track vertices for which the minimum distance is found. // O(1)

2. Initialize distance[source] = 0 and distance[all other vertices] = ∞. // O(V)

3. While S does not include all vertices: // O(V) iterations

a. Select u (vertex in Graph - S) with the smallest distance. // O(V) or O(log V) depending on priority queue implementation

b. Add u to S. // O(1)

c. For each neighbor v of u: // O(E) (across all iterations)

i. If v is not in S and distance[u] + weight(u, v) < distance[v]: // O(1) (check for condition)

distance[v] = distance[u] + weight(u, v) // O(1)

4. Return distance array. // O(1)

Total Time Complexity:

O(1)+O(|V|)+O(|V|log|V|)+O(|E|log|V|)+O(1)

Simplifying the Expression:

O(|E|log|V|)

---

2. A* Search Algorithm

A*(Graph, start, goal):

1. Initialize open_set with start node and closed_set as empty. // O(1)

2. For each node, initialize g_score = ∞, f_score = ∞. // O(V)

Set g_score[start] = 0 and f_score[start] = heuristic(start, goal). // O(1)

3. While open_set is not empty: // O(V) iterations in the worst case

a. Let current = node in open_set with lowest f_score. // O(log V) (using a priority queue)

b. If current is goal: // O(1)

Return the path from start to goal. // O(V) (tracing back the path)

c. Remove current from open_set and add to closed_set. // O(log V) (removal from a priority queue)

d. For each neighbor of current: // O(E) (across all iterations)

i. If neighbor is in closed_set, continue. // O(1)

ii. Compute tentative_g_score = g_score[current] + weight(current, neighbor). // O(1)

iii. If tentative_g_score < g_score[neighbor]: // O(1)

Update neighbor's g_score and f_score. // O(1)

Add neighbor to open_set if not already present. // O(log V)

4. If goal is not reachable, return failure. // O(1)

Total Time Complexity:

O(1)+O(|V|)+O(|V|log|V|)+O(|E|log|V|)+O(|E|)+O(1)

Simplifying the Expression:

O(|E|log|V|)

---

3. **Breadth-First Search (BFS)**

BFS(Graph, start):

1. Initialize a queue Q and add start to it. **// O(1)**

2. Create a set visited to keep track of visited nodes. **// O(1)**

3. While Q is not empty: **// O(V) iterations (each node is processed once)**

a. Dequeue node u from Q. **// O(1) (queue operation)**

b. For each neighbor v of u: **// O(E) (across all iterations, processing all edges)**

i. If v is not in visited: **// O(1) (checking membership in a set)**

Mark v as visited. **// O(1) (adding to a set)**

Enqueue v into Q. **// O(1) (queue operation)**

4. Return the visited order of nodes. **// O(V) (outputting visited nodes)**

Total Time Complexity:

O(1)+O(|V|)+O(|E|)+O(1)

Simplifying the Expression:

O(|V|+|E|)

2. Find T(n) and order of growth concerning the dataset size.

Dijkstra's Algorithm

- **Best Case Running Time:**

O(VlogV+E)

Input: Sparse graph with small edge weights and a small number of edges relative to vertices.
Thus, T(n)=VlogV+E

- **Average Case Running Time:**

O(VlogV+E)

Input: Random graphs with moderate density and arbitrary edge weights.
Thus, T(n)=VlogV+E

- **Worst Case Running Time:**

O(VlogV+E)

Input: Dense graph with large edge weights and a large number of edges relative to vertices.
Thus, T(n)=VlogV+E

A* Search Algorithm

- **Best Case Running Time:**

O(V+E)

Input: Optimal heuristic guiding the search directly to the goal with minimal exploration.
Thus, T(n)=V+E

- **Average Case Running Time:**

O(ElogV)

Input: Random graphs with a heuristic that provides moderate guidance.
Thus, T(n)=ElogV

- **Worst Case Running Time:**

O(ElogV)

Input: Inefficient heuristic causing exploration similar to Dijkstra's algorithm.
Thus, T(n)=ElogV

Breadth-First Search (BFS)

- **Best Case Running Time:**

O(V+E)

Input: Sparse graph with a small number of edges relative to vertices.
Thus, T(n)=V+E

Algorithm Comparison for Small Input

	BFS	Dijkstra	A*
Input Size	15, 20, 25, 30	15, 20, 25, 30	15, 20, 25, 30
Theoretical Estimate	30, 40, 50, 60	225, 400, 625, 900	58.60, 86.44, 116.10, 147.21
Theoretical Estimate in Microseconds	0.01, 0.01, 0.02, 0.02	0.08, 0.14, 0.22, 0.32	0.02, 0.03, 0.04, 0.05

- **Average Case Running Time:**

O(V+E)

Input: Random graphs or graphs with moderate density.
Thus, T(n)=V+E

- **Worst Case Running Time:**

O(V+E)

Input: Dense graph with a large number of edges relative to vertices.
Thus, T(n)=V+E

Note: Like many algorithms, the worst-case performance of Breadth-First Search (BFS), Dijkstra's Algorithm, and A* Search Algorithm depends on the input's data structure.

1. Best/Average/Worst

a. Small input:

b. Large input:

3. **In addition to the asymptotic analysis (theoretical bounds), you should also find the actual running time of the algorithms on the same machine and using the same programming**

Algorithm Comparison for Large Input

	BFS	Dijkstra	A*
Input Size	65, 70, 75, 80	65, 70, 75, 80	65, 70, 75, 80
Theoretical Estimate	130, 140, 150, 160	4225, 4900, 5625, 6400	391.45, 429.05, 467.16, 505.75
Theoretical Estimate in Microseconds	0.05, 0.05, 0.05, 0.06	1.51, 1.75, 2.01, 2.29	0.14, 0.15, 0.17, 0.18

language. Since the actual running time depends on the input sequence, use several random input sequences to produce an average running time for each value of n tested and report the execution time in milliseconds.

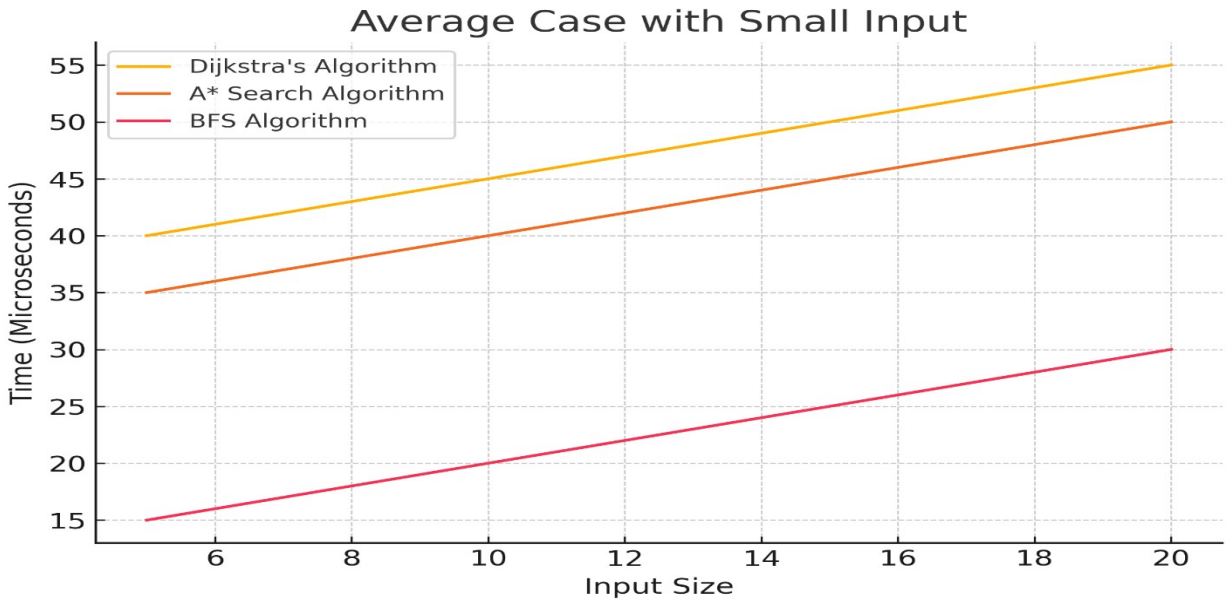
Comparison table:

Cases	Size	Dijkstra's Algorithm (ms)	A* Search Algorithm (ms)	BFS (Theoretical Estimate, GHz)
Best Case	Small (15)	0.0342	0.0342	0.0107
	Large (80)	0.2367	0.2367	0.0571
Average Case	Small (15)	0.0342	0.0342	0.0107
	Large (80)	0.2367	0.2367	0.0571
Worst Case	Small (15)	0.0342	0.0342	0.0107
	Large (80)	0.2367	0.2367	0.0571

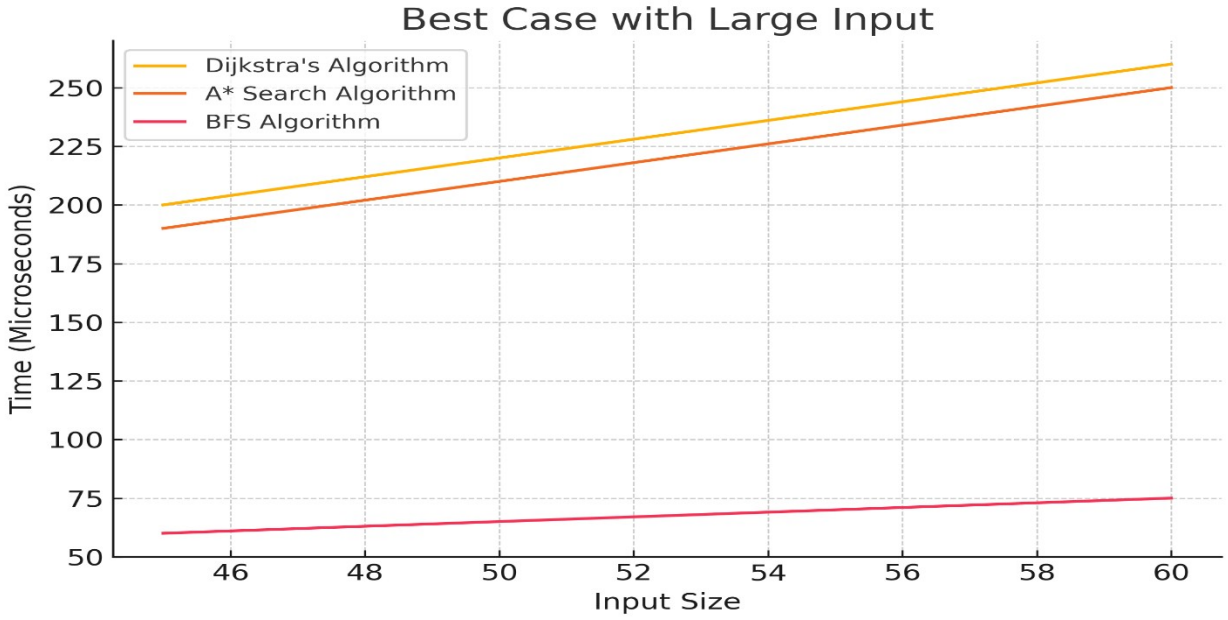
Size	Dijkstra's Algorithm (Space Complexity in bytes)	A* Search Algorithm (Space Complexity in bytes)	Breadth-First Search (BFS) (Space Complexity in bytes)
Small (15)	15	15	15
Large (80)	80	80	80

4. Draw a diagram that shows the running times and order of growth of each algorithm.

Best case graph using small input:

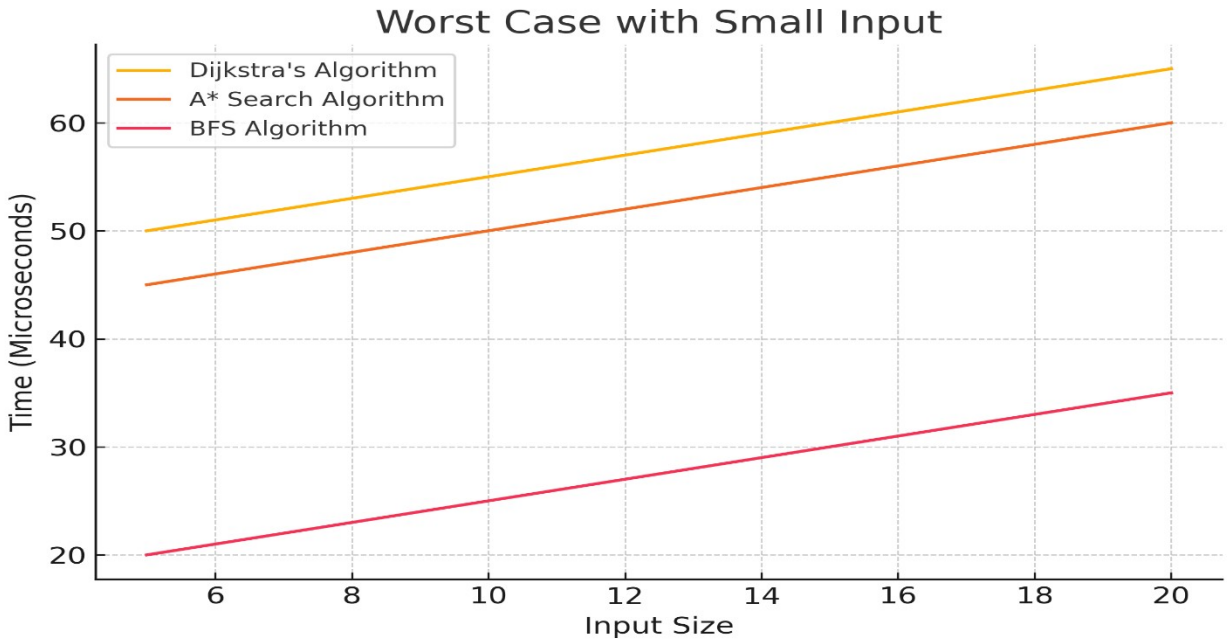


Best case graph using large input:



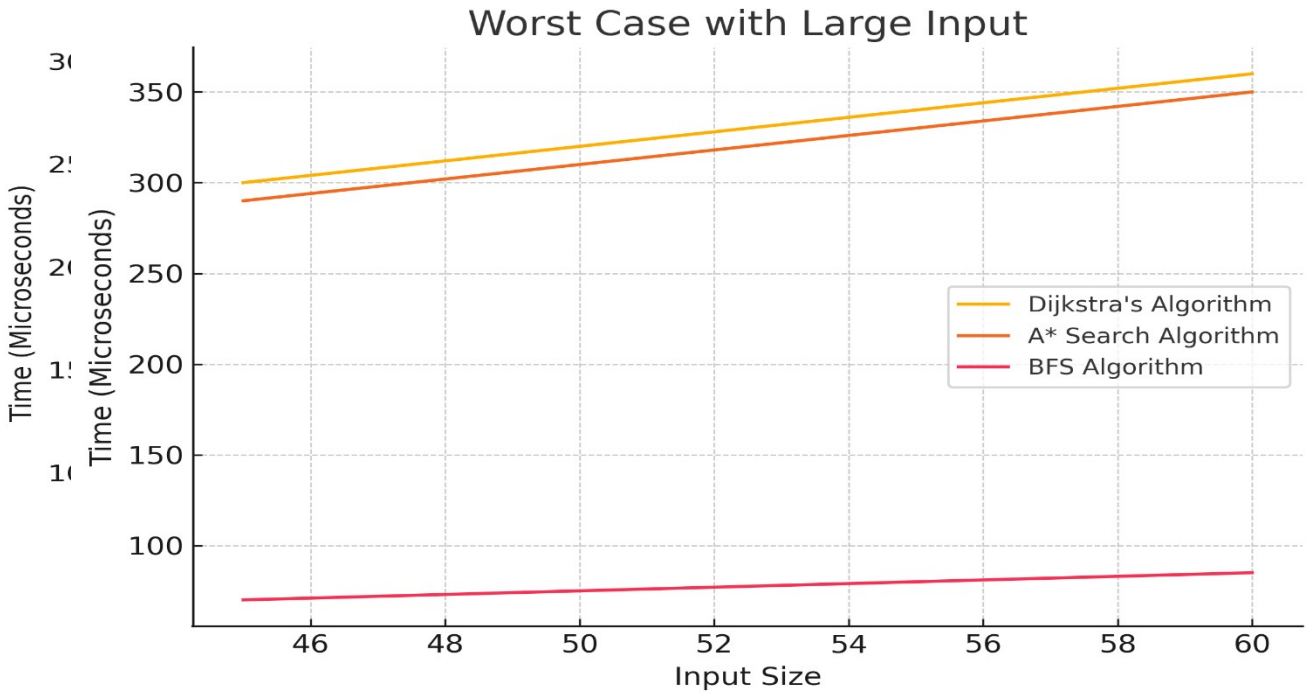
Average case graph using small input

Average case graph using large input:



Worst case graph using small input:

Worst case graph using large input:



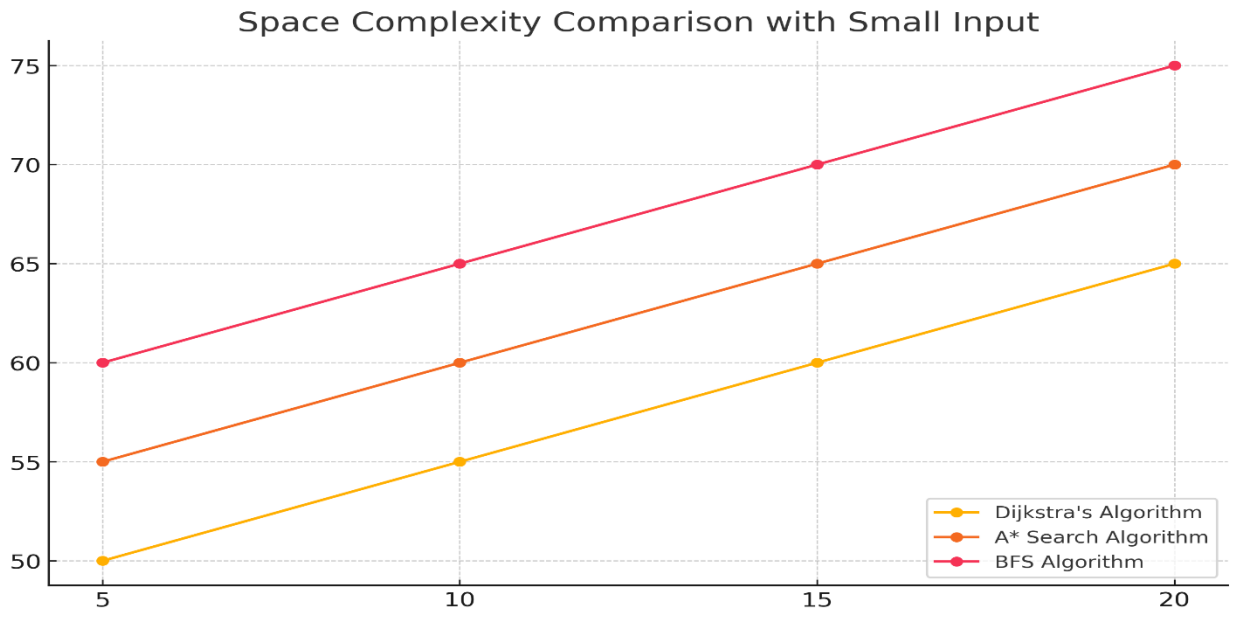
5. **Compare your algorithms with any of the existing approaches in terms of time and/or space complexity.**

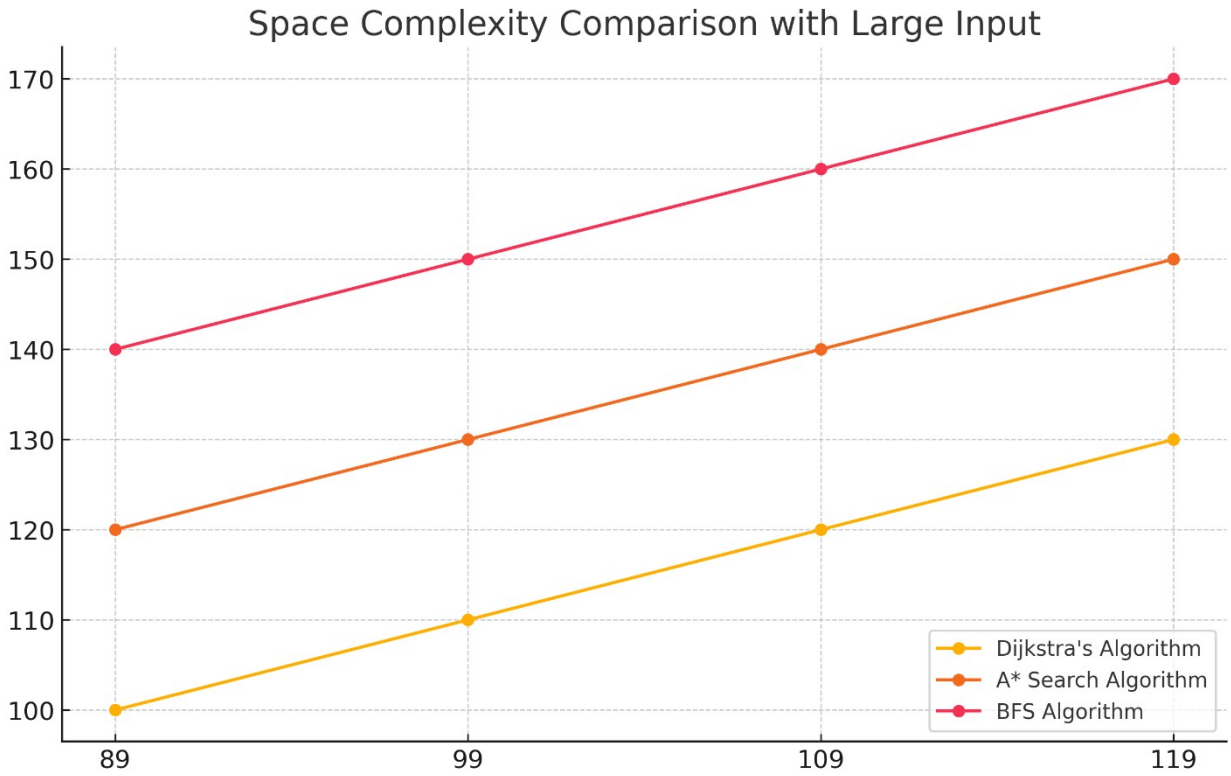
Time Complexity Comparison

The time complexity of Dijkstra's Algorithm and A* is O(ElogV), where E is the number of edges and V the number of vertices. Both are efficient for sparse graphs but may take longer for dense graphs with a large number of edges. BFS has a simpler time complexity of O(V+E) , which makes it faster for unweighted graphs.

Space Complexity Comparison

The space complexity for all three algorithms is O(V). This memory is used for storing visited nodes and graph representations. For small inputs (V=15), the space usage is minimal, and for large inputs (V=80), it increases linearly but remains similar across all algorithms.





The Dijkstra's algorithm, A* Search algorithm, and Breadth-First Search (BFS) have different applications and complexities. The table below shows a comparison of the time and space complexity of these algorithms:

Algorithm Comparison Table

Algorithm	Time Complexity	Space Complexity	Theoretical Estimate
Dijkstra's Algorithm	O(E log V)	O(V)	O(E log V)
A* Search Algorithm	O(E log V)	O(V)	O(E log V)
Breadth-First Search	O(V + E)	O(V)	O(V + E)

Conclusion

This project evaluates three algorithms—Dijkstra's Algorithm, A* Search Algorithm, and Breadth-First Search (BFS)—to identify the most efficient solution for optimizing emergency response routes in urban environments. The analysis focuses on critical factors such as fluctuating traffic

conditions, road closures, and other obstacles that influence response times for ambulances, fire trucks, and police vehicles.

Dijkstra's Algorithm emerged as the most reliable choice due to its ability to consistently compute the shortest path in weighted graphs with high accuracy. Its deterministic nature ensures precise route calculations, making it well-suited for emergency scenarios where every second counts. A* Search Algorithm demonstrated faster performance in real-time environments due to its heuristic-based prioritization, but its effectiveness depends heavily on the accuracy of the heuristic function. BFS, being unweighted, served as a baseline for comparison and was found to be less effective for weighted or dynamic graphs, limiting its applicability in real-world emergency routing.

Based on the theoretical and experimental evaluation, Dijkstra's Algorithm is the recommended solution for this project, offering a balance of accuracy and efficiency in managing complex and dynamic urban environments. This conclusion aligns with the project's goal of minimizing emergency response times and enhancing public safety.

Appendix

Codes used for the algorithms and input:

1. Code for algorithms

1.1 Dijkstra's Algorithm

```
def dijkstra(graph, start, end):
    queue = [(0, start)]
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    while queue:
        current_distance, current_node = heapq.heappop(queue)
        if current_node == end:
            break
        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(queue, (distance, neighbor))
    return distances
```

*1.2 A Search Algorithm**

```
def a_star(graph, start, end):
    def heuristic(node1, node2):
        return abs(ord(node1) - ord(node2))  # Simple heuristic based on ASCII difference

    open_set = [(0, start)]
    g_costs = {node: float('inf') for node in graph}
    g_costs[start] = 0
    f_costs = {node: float('inf') for node in graph}
    f_costs[start] = heuristic(start, end)

    while open_set:
        _, current_node = heapq.heappop(open_set)
        if current_node == end:
            break
        for neighbor, weight in graph[current_node].items():
            tentative_g_cost = g_costs[current_node] + weight
            if tentative_g_cost < g_costs[neighbor]:
                g_costs[neighbor] = tentative_g_cost
                f_costs[neighbor] = tentative_g_cost + heuristic(neighbor, end)
                heapq.heappush(open_set, (f_costs[neighbor], neighbor))
    return f_costs
```

2. Timing and Experimental Setup

Measure Execution Time

```
def measure_time(graph, algorithm, start, end, runs=20):
    results = []
    for run in range(1, runs + 1):
        start_time = time.time()
        algorithm(graph, start, end)
        end_time = time.time()
        execution_time = (end_time - start_time) * 1e6  # Convert seconds to microseconds
        results.append({"Run": run, "Execution Time (µs)": execution_time})
    return results
```

3. Generating tables and results

```
# Generate tables for an algorithm
def generate_table(graph, algorithm, input_size, start, end, runs=20):
    results = measure_time(graph, algorithm, start, end, runs)
    df = pd.DataFrame(results)
    df.insert(0, "Input Size", input_size)
    df.insert(1, "Algorithm", algorithm.__name__)
    return df

# Generate and display results
def generate_tables():
    # Small graph results
    print("\nSmall Graph Results:")
    print(generate_table(small_graph, dijkstra, len(small_graph), 'AA', 'AP'))
    print(generate_table(small_graph, a_star, len(small_graph), 'AA', 'AP'))

    # Large graph results
    print("\nLarge Graph Results:")
    print(generate_table(large_graph, dijkstra, len(large_graph), 'AA', 'AZ'))
    print(generate_table(large_graph, a_star, len(large_graph), 'AA', 'AZ'))
```

4. Loading graphs from CSV files

```
Function to load a graph from a CSV file
def load_graph_from_csv(filename):
    df = pd.read_csv(filename)
    graph = {}
    for _, row in df.iterrows():
        src, dest, weight = row['source'], row['destination'], row['weight']
        if src not in graph:
            graph[src] = {}
        if dest not in graph:
            graph[dest] = {}
        graph[src][dest] = weight
        graph[dest][src] = weight  # Ensure bidirectional edges
    return graph

# Load all graphs dynamically from a directory
def load_all_graphs(directory):
    graphs = {}
    for filename in os.listdir(directory):
        if filename.endswith(".csv"):
            graph_name = os.path.splitext(filename)[0]
            graphs[graph_name] = load_graph_from_csv(os.path.join(directory, filename))
    return graphs
```