Exploring Efficiency: A Comprehensive Analysis and Comparison of Shortest Path Finding Algorithms

Manisha Kumari[1]

***Abstract*— The following paper focuses on analysing the graph algorithm used for finding the shortest path between a source and a destination. The algorithms chosen are the A* search algorithm, Dijkstra's algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, BFS, and Johnson's algorithm. The metric of analysis is mainly the time and space complexity required by the algorithms. Ultimately, we tend to provide the respective strengths and weaknesses of the algorithms based on the analysis done.**

I. Introduction

Path-finding algorithms are used daily, be it in maps or computer networks. With evaluation in programming, these algorithms have also improved. With this project, we are trying to see this evaluation through experimentation. Starting from Dijkstra's algorithm to Johnson's algorithm, we will put these algorithms to the test. The experimentation will involve passing the algorithms through different kinds of graphs and data sizes. We have divided our experiment into 4 parts, which will be discussed in the data section.

II. code link

https://github.com/iit2018062/graph/tree/main.

III. Data

For the following comparison, I have used NetworkX in Python, and the erdos renyi graph in NetworkX helps to return a random graph, also known as an Erdős-Rényi graph or a binomial graph. The G_{np} model takes the following parameters:

- N: (int) number of vertices/nodes in the graph.
- p: (float) which is the probability of edge creation.
- $Seed$: (int) which can be None by default. This indicates a random number generation state.
- $directed$: (bool) which indicates whether the graph needs to be directed or not.

This technique of graph creation has a complexity of $O(n^2)$.

We have four kinds of experimental data:

Experiment 1: Algorithms: Dijkstra's algorithm, Bellman-Ford, Floyd-Warshall, Johnson's algorithm

For the comparative study, we experimented with the following types of graphs:

- Weighted and directed graph
- Weighted and undirected graph
- Acyclic graph
- Cyclic graph
- Dense graph (for this we have used `nx.complete_graph`, this helps to generate a graph where all pair of distinct vertices have edges connecting them)
- Sparse graph (we used fast `gnp_random_graph` for this, which has a lesser complexity in the case of a sparse graph, $O(n + m)$, where n is the number of nodes and m is the expected number of edges).

Experiment 2:

Study one graph with negative weights, for this, we have used Bellman-Ford, Floyd-Warshall, and Johnson's algorithms.

Experiment 3:

We analysed the BFS algorithm along with algorithms like Bellman-Ford, Floyd-Warshall, and Johnson's algorithm on unweighted graphs, as BFS is applied only to unweighted graphs.

Experiment 4:

We utilised the NetworkX library to leverage its suite of algorithms. We then structured the experiment to compare these algorithms across two distinct types of graphs. To facilitate the A*

algorithm's functionality, we incorporated a heuristic function, defined as the absolute difference between vertex 1 and vertex 2.

IV. Algorithms

We will discuss the various algorithms below.

A. Dijkstra's Algorithm

This algorithm was created and published by Dr. Edsger W. Dijkstra. For a weighted graph, the Dijkstra algorithm helps us find the shortest path from the source vertex to all the vertices in a given graph. This algorithm cannot be used for a graph with negative weights. We have used Dijkstra to find the distance between the source vertex and the destination vertex. The way this algorithm works is:

Complexity:

The time complexity of Dijkstra's algorithm is $O(E \log V)$, where E is the number of edges and V is the number of vertices. The auxiliary space complexity is $O(V)$.

Applications:

Dijkstra's algorithm finds applications in various domains, including route planning in maps (such as Google Maps), computer networking, transportation systems, and traffic control.

Algorithm 1 Dijkstra's Algorithm

Data: graph, source, destination
Result: Shortest distance from source to dest.

Initialize distance array with size n (total number of vertices) Initialize values in distance as infinity
foreach *vertex* v *in graph* **do**
  $Distance[v] := \infty$
end
$Distance$[source] $:= 0$ `// Distance from // source`
`// Create a priority queue` q `(min) // heap`
q.add($(Distance$[source], source$)$)
while q *is not empty* **do**
  $U := q$.top() ; `// Extract the // topmost element (vertex with // minimum distance)`
  q.pop()
  if $U == destination$ **then**
    return $Distance[u]$
  end
  foreach *neighbor* V *of* U **do**
    $dist := Distance[U]$ + graph.edge(U, V)
    if $dist < Distance[V]$ **then**
      $Distance[V] := dist$
      q.push$(Distance[V], V)$
    end
  end
end
return INF ; `// indicates no path // from source to destination.`

B. Bellman–Ford algorithm

Just like Dijkstra's algorithm, the Bellman-Ford algorithm tries to find the shortest distance between the source and all other vertices in a weighted graph. The important difference it has from Dijkstra is its capability to handle edge weights, which are negative. This algorithm is slower than Dijkstra. The Bellman-Ford algorithm fails to find the shortest path if there are any negative cycles in the graph. The algorithm is based on the principle of relaxation, which states that in a graph with n vertices, all edges need to be relaxed n-1 times. We have used this to find the shortest path between the source and destination.

Complexity:

- **Best Case:** $O(E)$, where E is the number of edges. This occurs when we stop the iteration if the change in the distance array stops after 1 or 2 iterations.
- **Average and Worst Case:** $O(E \cdot V)$, where V is the number of vertices. The time complexity when the graph is disconnected comes around $O(E \cdot V^2)$.

Auxiliary Space:

$O(V)$, where V is the number of vertices.

Algorithm 2 Bellman-Ford Algorithm

Data: graph, edges, vertices, src, dest
Result: Shortest distances from source to all vertices or "Negative cycle"

```
for each vertex v ∈ vertices do
    Dist[v] ← ∞
end
Dist[src] ← 0 ; // Distance from source
for i from 1 to n − 1 do
    for each edge (u, v) ∈ edges do
        if Dist[v] > Dist[u] + weight(u, v) then
            Dist[v] ← Dist[u] + weight(u, v);
              // Update distance
        end
    end
end
for each edge (u, v) ∈ edges do
    if Dist[v] > Dist[u] + weight(u, v) then
        return "Negative cycle"
    end
end
return Dist[dest]
```

Algorithm 3 Floyd-Warshall Algorithm

Data: graph, source, V, destination
Result: Shortest distances between source and destination

```
// Initialize a distance matrix with the same
//dimension as the graph
for i from 0 to V do
    for j from 0 to V do
        Dist[i][j] ← graph[i][j]
    end
end
for k from 0 to V do
    for i from 0 to V do
        for j from 0 to V do
            if Dist[i][j] > (Dist[i][k] + Dist[k][j])
              and Dist[k][j] ≠ ∞ and Dist[i][k] ≠
              ∞ then
                Dist[i][j] ← Dist[i][k] + Dist[k][j];
                  // Update distance
            end
        end
    end
end
return Dist[source][destination]
```

Applications:

This algorithm with some variation is used in distance-vector routing protocols for example the Routing Information Protocol (RIP).

C. Floyd–Warshall

The algorithm finds the shortest paths using a specific source. The algorithm works for both directed and undirected weighted graphs. The weights for this algorithm can be negative or positive; however, it fails to deal with a negative cycle like the Bellman-Ford algorithm. The core concept of this algorithm is based on dynamic programming.

Complexity:

The time complexity for the algorithm is $O(V^3)$, where V is the number of vertices. The space complexity comes around $O(V^2)$, the space used for storing distance.

Applications:

The algorithm is used in computer networking, flight connectivity, Geographic Information Systems, transitive closure of directed graph problem, Kleene's algorithm etc.

D. Johnson's Algorithm

Johnson's algorithm tries to find the shortest path between the source and all other vertices. Just like Floyd Warshall, which can handle negative weight, it can also deal with negative weight. The algorithm uses a combination of Dijkstra's algorithm and the Bellman-Ford algorithm. We know from our previous discussion that Dijkstra's algorithm does not work with negative weights. Johnson's algorithm overcomes this problem by re-weighting all the edges to make the weights positive.

The weights are re-weighted using the Bellman-Ford algorithm. The weight between two vertices u and v is re-weighed using $weight(u, v) +$ bellmanWeight(u) − bellmanWeight(v). This ensures each edge is increased by the same amount, and all negative weights become positive.

We utilise the previous Dijkstra's algorithm to

find the distance between the source and destination in this algorithm.

Algorithm 4 Bellman-Ford Algorithm

Data: graph, edges, vertices, src
Result: Shortest distances from source to all vertices or "Negative cycle"

```
for each vertex v ∈ vertices+1 do
    Dist[v] ← ∞
end
Dist[vertices] ← 0 ;        // Distance from source
for each vertex v ∈ vertices+1 do
    edges.add([vertices, v, 0])
end
for i from 1 to n − 1 do
    for each edge (u, v) ∈ edges do
        if Dist[v] > Dist[u] + weight(u, v) then
            Dist[v] ← Dist[u] + weight(u, v);
              // Update distance
        end
    end
end
for each edge (u, v) ∈ edges do
    if Dist[v] > Dist[u] + weight(u, v) then
        return "Negative cycle"
    end
end
for each vertex v ∈ vertices do
    distance[v] ← Dist[v]
end
return distance
```

Complexity:

The time complexity of the algorithm is approximately $O(V \cdot V \log(V) + VE)$. Here, Dijkstra's algorithm is called V times, and Bellman-Ford has a time complexity of $O(VE)$. Note: V is the number of vertices, and E is the number of edges.

The auxiliary space for the algorithm is around $O(V \cdot V)$.

Applications:

Just like other path-finding algorithms, it is used in networks, computer science, and operational research.

Algorithm 5 Johnson(graph, V)

```
// initializing an array named edges to store edges of the graph   ▷ V is the number of vertices
edges ← []
    for i from 0 to V do
        for j from 0 to V do
            if graph[i][j] ≠ 0 then
                edges.add([i, j, graph[i][j]])
            end
        end
    end
// calling the Bellman algorithm
Weight_from_bellman_ford ← BellmanFord(edges, graph, V)
// making the modified graph with new weights
modifiedGraph ← CreateMatrix(len(graph), len(graph))
    for i in range V do
        for j in range graph[v].length do
            if graph[i][j] ≠ 0 then
                modifiedGraph[i][j] ← graph[i][j] +
                Weight_from_bellman_ford[i] −
                Weight_from_bellman_ford[j]
            end
        end
    end
    return DijkstraAlgorithm(modifiedGraph, src, dest)
    =0
```

E. A* search

This algorithm helps find the shortest path between the source and destination of a weighted graph. A* takes in the weight of the edges plus the estimated cost between the nodes, also known as heuristics (h). It is an informed search algorithm. The way the algorithm works is explained below:

Complexity:

The time complexity of the A* algorithm depends on the heuristic function used. Generally, it can be approximated to be around $O(E^V)$, where E is the branching factor of a node, and V is the number of nodes on the path. The space complexity of A* also comes around to be $O(E^V)$. The time complexity heavily depends on the type of heuristic

Algorithm 6 A* Algorithm

```
Data: graph, start, end
Result: Shortest path from start to end
for vertex in graph do
    vertex.score ← INF
    vertex.visited ← false
    vertex.heuristic_value ← INF
end
startVertex.score ← 0
startVertex.heuristic_value ← 0
while true do
    currentVertex ← minimum(graph)
    currentVertex.visited ← true
    for neighbour in currentVertex.neighbours do
        if neighbour.visited == false then
            score ← calculateScore (currentVertex, neighbour)
            if score < neighbour.score then
                neighbour.score ← score
                neighbour.heuristic_value ← score + calculateHeuristicScore (neighbour, end)
                fromNode[neighbour] ← currentVertex
            end
        end
    end
    if currentVertex == end then
        return buildPath (end)
    end
    if minimum(graph).score == INF then
        return NoPath
    end
end
```

Algorithm 7 Find Minimum Node in Graph

```
Data: graph
Result: Minimum node in the graph
Result ← null
Tmp ← INF
for node in graph do
    if node.visited == and node.heuristic_value < Tmp then
        Result ← node
        Tmp ← node.heuristic_value
    end
end
```

function chosen. A better heuristic function results in fewer nodes to visit. The best A* heuristic function aims to achieve a branching factor of 1.

Applications:

A* algorithms are widely used in various fields, including game development. One of the most popular applications is in Tower Defense Games. Additionally, they are utilized in path-finding applications, robotics, network routing, and artificial intelligence systems.

F. BFS (Breadth-First Search)

This algorithm is applied in unweighted graphs only. The algorithm takes in V(number of nodes), E(number of edges), source node, and destination node. The aim is to find the shortest path from source to destination.

Algorithm 8 Breadth-First Search (BFS)

```
Data: graph, source, parent, destination
Result: Shortest distances from source to all other nodes
queue ← Queue (); for i from 0 to V do
    Dist[i] ← INF;
end
Dist[source] ← 0; queue.add (source);
while not queue.isEmpty () do
    node ← queue.dequeue (); for neighbour in graph[node] do
        if Dist[neighbour] == INF then
            parent[neighbour] ← node
            Dist[neighbour] ← Dist[node] + 1 queue.add (neighbour)
        end
    end
end
```

Complexity:

The time complexity of this algorithm is $O(V+E)$, where V is the number of vertices and E is the number of edges. The space complexity is $O(V)$.

Applications:

This algorithm is typically used in unweighted graphs. However, due to this restriction, it is not

Algorithm 9 Print Path

Data: graph, source, destination, V
Result: Prints the shortest path from source to destination
par ← array of size V; dist ← array of size V; `bfs` (graph, source, par, dist);
if *dist[destination]* == INF **then**
  return;
end
path ← []; current$_n$*ode*← destination; path.append(destination);
while *par[current$_n$ode]*≠ *-1* **do**
  path.append(par[current$_n$*ode*]); *current$_n$ode*← par[current$_n$*ode*]; **end**
  Print path;

widely used in real-life applications. Its modified version, Dijkstra's algorithm, is more commonly employed in various fields such as network routing, path-finding, and artificial intelligence systems.

V. Algorithms Analysis

For the weighted graph, random weights between 0.1 to 1 were chosen to prevent integer overflow.

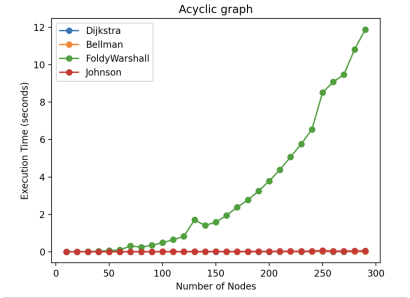
VI. Experimental Setup

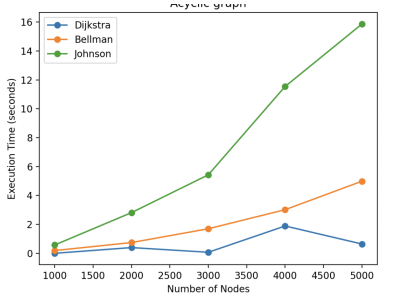
The study was conducted using Python code in the macOS environment. Python libraries such as `networkx`, `time`, `matplotlib.pyplot`, and `random` were utilized. The experiments were performed on a single computer with the following specifications:

1) Python version: 3.12.0, executed in IntelliJ IDEA.
2) **Hardware Specifications:**
   - Model Name: MacBook Air
   - Model Identifier: MacBookAir10,1
   - Chip: Apple M1
   - Total Number of Cores: 8 (4 performance and 4 efficiency)
   - Memory: 8 GB
   - System Firmware Version: 8419.60.44
   - OS Loader Version: 8419.60.44
3) Configuration Settings: Edge weights were set between 0.1 to 1, with edge probability ranging from 0.5 to 0.7. Each algorithm was run multiple times to account for result randomness.To maintain a uniform result, the codes were run in an environment where no other processes were running on the laptop. The system's CPU utilization and memory allocation were monitored to account for sudden spikes. These algorithms use a lot of memory allocations which could lead to intensive work by garbage collectors. To mitigate this, object creations were minimized and also while running the code other processes were closed to have low activity in the system.

A. Experiment 1:

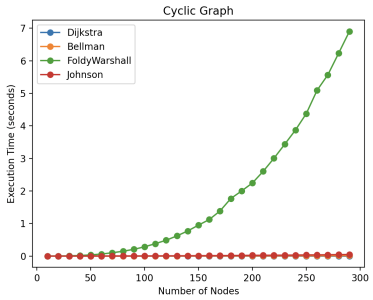
Acyclic Graph:

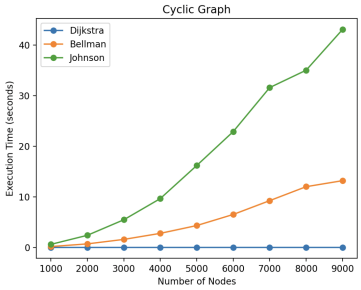




We generated an acyclic graph using the `NetworkX` library in Python, ensuring that the graph has no cycles by leveraging the `is_directed_acyclic_graph` function. We ran our code for 300 vertices and then scaled up to several vertex numbers, including 5000. As expected, the Floyd-Warshall algorithm takes the maximum time due to its cubic complexity. Following that, Johnson's algorithm is next in terms of execution time, followed by Bellman-Ford, and finally Dijkstra's algorithm. Therefore, for this experiment, the algorithms exhibit the expected behaviour.

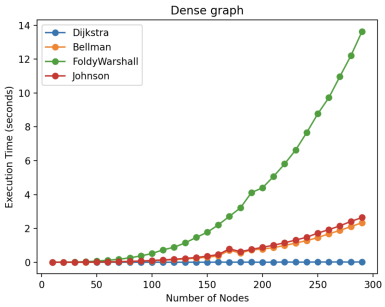
Cyclic Graph:

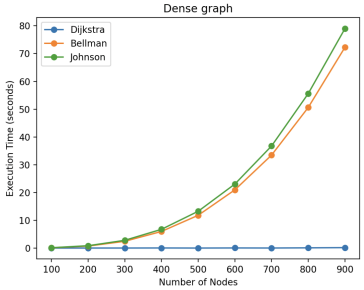


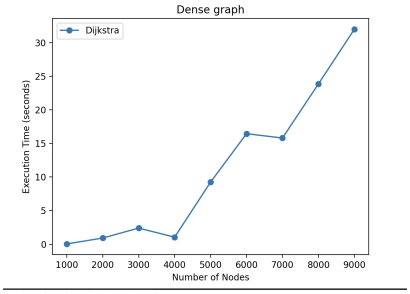


Just like acyclic graphs, cyclic graphs also showed desired outputs. We ran our algorithms for the first 300 nodes and then for 9000 nodes. The fastest algorithm turns out to be Dijkstra, while the slowest is Floyd-Warshall.

Dense Graph:

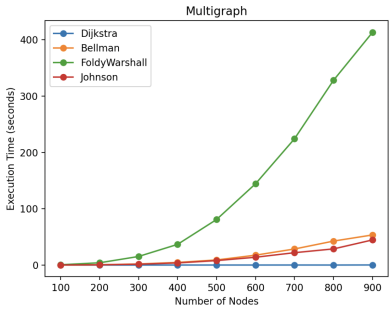






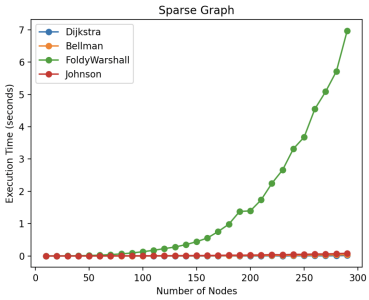
For the dense graph, we used a complete graph from Networkx. This ensures a fully connected graph where any unique pair of vertices has an edge. In this scenario, all algorithms seem to run quite slowly, except for Dijkstra's algorithm. The graph shows Floyd-Warshall as the slowest, while Dijkstra again wins the race.

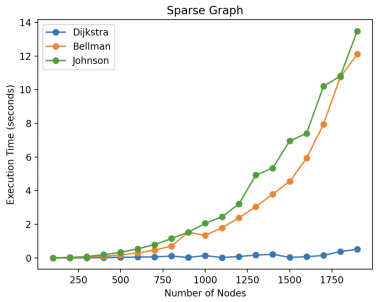
Multi-Graph:



For multi-graphs, we used Multi Graph from Networkx. We have represented weights for edge (u,v) as the sum of all the weights between (u,v). The algorithms seem to work slower than they did in dense graphs. I got the expected result, with Floyd-Warshall as the slowest and Dijkstra as the fastest. However, Johnson's algorithm seems to take less time than Bellman's in some cases. This could be the result of sparsity in graph.

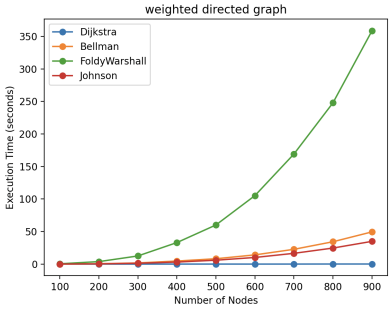
Sparse-Graph:





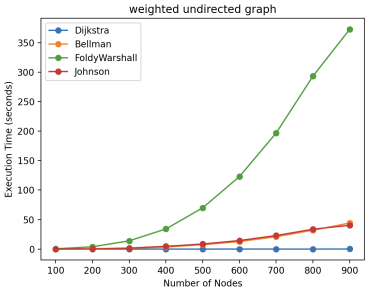
For this, we used `fast_gnp_random_graph` with a probability of each edge of 0.01. This ensured a sparse graph. The order of algorithms based on time complexity came out to be Dijkstra, Bellman, Johnson, and the slowest, Floyd-Warshall.

Weighted directed-Graph:



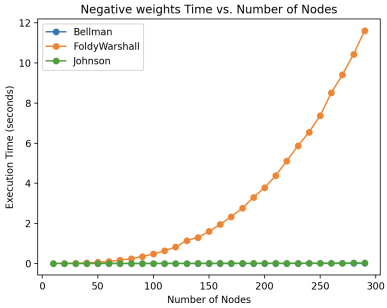
For this experiment, we ran our code on 900 nodes. Surprisingly, in this case, Johnson's algorithm outperformed Bellman's algorithm. The order of algorithms in terms of complexity followed Floyd-Warshall, Bellman, and Johnson, and the fastest was Dijkstra. One possible reason could be the density of the graph, which in this case is 0.5. This leads to a little sparse graph, as Johnson's algorithm pre-processes the graph before and hence can deal with this sparsity, leading to better performance than Bellman.

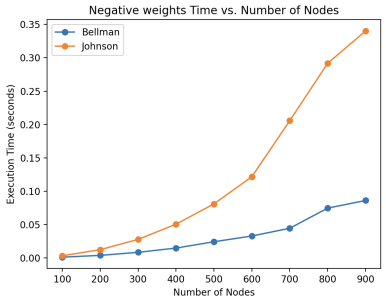
weighted undirected-Graph:



For this, we ran algorithms for 900 nodes with a probability of each edge being 0.5. The graph shows the order of complexity according to the theoretical order of complexity for these algorithms.

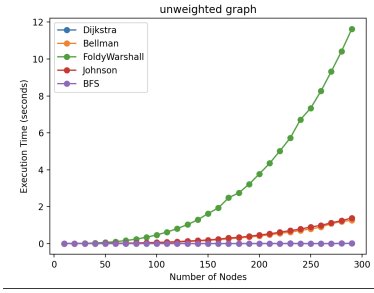
B. Experiment 2:

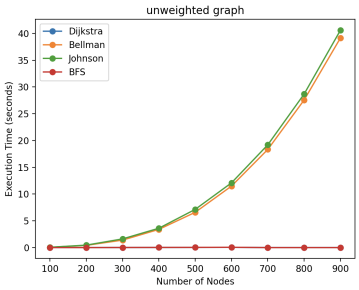




For this experiment, the main challenge was generating a random graph with negative weight edges such that these edges don't make a negative cycle. So to overcome this, we first generated an acyclic graph and then assigned the edges random positive and negative values. Doing so ensured a non-cyclic graph, giving better results. We got the experimental order of graphs based on complexity, which same with the theoretical order. This experiment required a lot of memory utilization, therefore I ran it multiple time to compensate the effects of time taken by garbage collector.

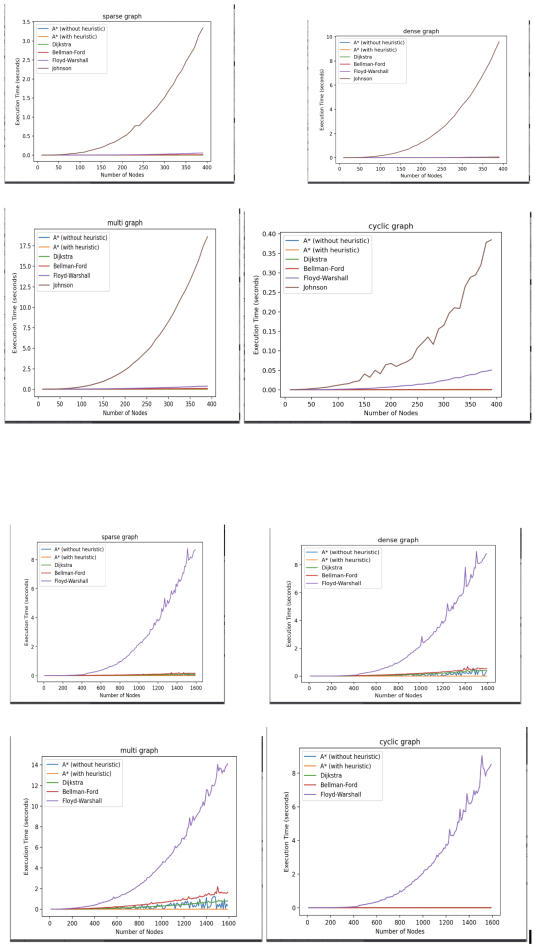
C. Experiment 3:

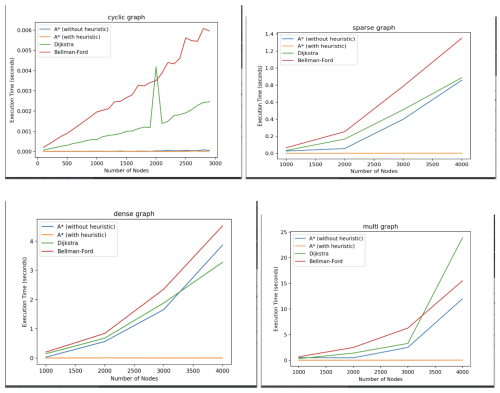


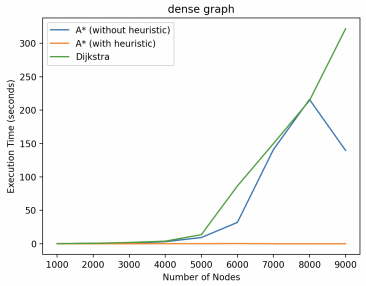


The experiment uses an unweighted and undirected graph. We experimented with 300 and 900 nodes. Dijkstra's algorithm in the unweighted graph is the same as BFS. The time complexity graph for this experiment coincides with the theoretical order of complexities for all algorithms.

D. Experiment 4:







This experiment used the Networkx library and utilised it to implement different algorithms directly. The algorithms use four kinds of graphs: sparse graphs, dense graphs, multigraphs, and cyclic graphs. The graph is represented as an adjacency list with weights varying between 0.1 and 1.

1) The algorithm worked fastest for cyclic graphs, as expected for A* searches. It can effectively use its heuristic to find the shortest edge among multiple edges. The cyclic graph helps in faster convergence in the case of Dijkstra.
2) One surprising thing to notice was that Floyd-Warshall's algorithm outperformed Johnson's algorithm. One possible reason is the caching efficiency of Floyd-Warshall over Johnson. On top of this, Johnson's algorithm has the overhead of modifying the graph best on output from Bellman's algorithm. In addition, randomness in the graph, or the environment supporting graph structure to have an advantage in Floyd-Warshall can be a reason.
3) A* The search algorithm came to be the fastest, and it is interesting to see how quick it is compared to Dijkstra.
4) NetworkX has several optimisations for A*

search, so even without heuristics, it seems to be slightly outperforming Dijkstra, which should not have been the case. Overall, the outcome showed the expected ranking of algorithms, except for a few exceptions, as mentioned.

VII. Conclusion:

TABLE I
SPACE AND TIME COMPLEXITY OF VARIOUS ALGORITHMS

Algorithm	Space Complexity	Time Complexity
Dijkstra	$O(\|V\| + \|E\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Johnson	$O(V^2)$	$O(V^2 \log V + VE)$
Floyd-Warshall	$O(V^2)$	$O(V^3)$
A* Search	$O(V)$	$O(b^d)$
BFS	$O(V)$	$O(\|V\| + \|E\|)$
Bellman-Ford	$O(V)$	$O(VE)$

In Experiment 1, we observed that if the graph is too dense, Bellman and Johnson's algorithms tend to have a very narrow difference in complexity. This is because the time complexity of Bellman's algorithm, $O(VE)$, increases when the number of edges E is increases. Johnson's algorithm has a complexity of $O(V \cdot V \cdot \log(V) + VE)$. In this case, when VE becomes larger than $V^2 \log(V)$, it leads to a less favorable difference in performance between Bellman and Johnson.

Overall, from experiment 1 we found that for all kinds of graphs, the best algorithm to choose is Dijkstra's algorithm. However, Dijkstra's algorithm cannot handle negative weights.

For space complexity as well, we observed Floyd-Warshall and Johnson to be at the higher end.

The experiments helped us to analyze different algorithms with different kinds of data, taking space complexity, time complexity, and implementation details into account. From the experiments, we can conclude that for graphs with non-negative weights, choosing the A* search algorithm is the most effective way to find the shortest path. However, the effectiveness of A* depends heavily on the heuristic function used. With a good heuristic function, we can optimize the branching factor to a large extent, making A* the fastest algorithm.

In the case of negative weights, Bellman's algorithm seems to provide the fastest result; however, the algorithm has the drawback of not being able to handle negative weights.

We have looked over different algorithms and how they perform with different types of graphs. It's important to note that results from these algorithms are sensitive to the environment we use. With all experiment results, we can conclude that A* search and Dijksta's algorithm can be used to greatly optimize any path search application. However, for edges with negative weights, one needs to use Bellman, Johnson or Flody-warshall.

References

[1] Wikipedia contributors, *Bellman-Ford algorithm*, *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm

[2] GeeksforGeeks, *Dijkstra's Shortest Path Algorithm*, *GeeksforGeeks*, 2023, September 12. [Online]. Available: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[3] Wikipedia contributors, *Dijkstra's algorithm*, *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[4] Wikipedia contributors, *Floyd-Warshall algorithm*, *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

[5] Wikipedia contributors, *Johnson's algorithm*, *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Johnson%27s_algorithm

[6] GeeksforGeeks, *Johnson's Algorithm for All Pairs Shortest Paths — Implementation*, *GeeksforGeeks*, 2024. [Online]. Available: https://www.geeksforgeeks.org/johnsons-algorithm-for-all-pairs-shortest-paths-impl

[7] Wikipedia contributors, *A* search algorithm*, *Wikipedia*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm

[8] NetworkX documentation, *networkx.generators.classic.complete_graph*, *NetworkX*, 2024. [Online]. Available: https://networkx.org/documentation/stable/reference/generated/networkx.generators.classic.complete_graph.html