`

INVESTIGATION

Comparing Dijkstra's Algorithm and A Algorithm*

How does the pathfinding efficiency of Dijkstra's algorithm compare to that of A* algorithm in terms of time complexity on different types of graphs?

AUTHOR: DAKSH SINGHAL

Table of Contents

1. Introduction

Pathfinding algorithms, highly essential tools in computer sciences, are used to determine the *shortest path for a given problem*. Determining the shortest path to a given problem is highly important in numerous fields such as network routing (where network transfer routes are optimized to enhance the performance of a network), and in autonomous navigation (for robotics as they determine the optimal route autonomous systems to navigate through complex environments). *(Cormen, Thomas H., et al.)*

As a result of increasing growth of database sizes, the demand for algorithms that can solve the *Shortest Path Problem* has severely increased. The Shortest Path Problem requires finding the shortest distance between two nodes in a graph, and these graphs typically have a minimum cost associated to them. Globally, a myriad of algorithms are used to solve this problem. Of these algorithms, the Bellman-Ford Algorithm, Dijkstra's Algorithm and A* Algorithm are the most popular in terms of their usage. *(Sedgewick, Robert, and Kevin Wayne)*

This essay will focus on comparing the time complexity and performance of pathfinding algorithms whilst traversing graphs. The two algorithms that would be analysed are the: *Dijkstra's Algorithm* (which explores all the possible paths) and the *A* Algorithm* (which utilizes heuristics to find the best path). *Time Complexity* is a term used for referring to the duration of time it takes for an algorithm to run when given a set of a certain size of input values. Comparing these algorithms will assist in determining and comparing the performance of these algorithms in different kinds of situations. Hence, the question*: How does the pathfinding efficiency of Dijkstra's algorithm compare to that of the A* algorithm in terms of time complexity on different types of graphs?* This essay relates to Topic 5 of the IB Higher Level Computer Science course.

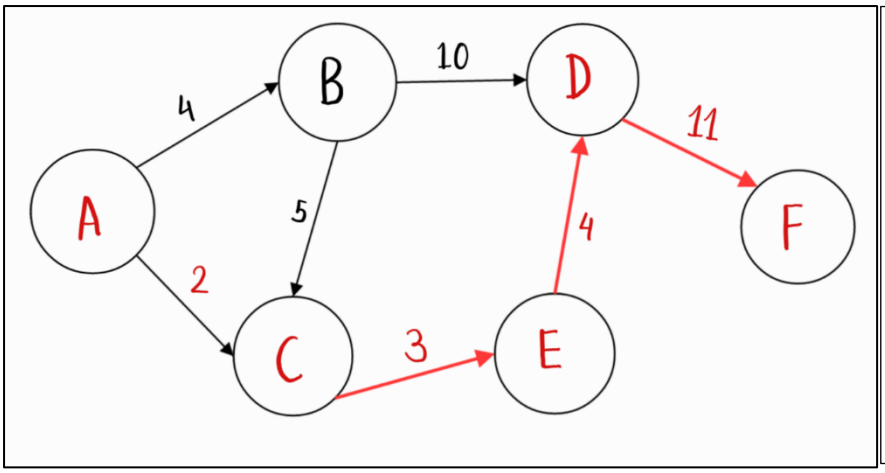
2. Background Information

2.1: Pathfinding on Weighted Graphs

At their core, pathfinding algorithms function by searching a graph from its vertex and then proceeding to explore all the adjacent nodes until the destination node is found. The two main objectives of pathfinding are to: *find a path between two nodes,* or to *find the most optimal path.*

These graphs consist of two elements: *nodes (vertices)* and *edges (paths).* In essence, a node represents the location inside the graph, this can be a grid for a game or even a waypoint inside a navigation system. Meanwhile, an edge represents the connection between two nodes. There are two types of graphs: *Weighted Graphs* and *Unweighted Graphs.* In a weighted graph, these edges represent a quantifiable cost or distance, whilst in an unweighted graph all of these edges are considered equal. *(Cormen, Thomas H., et al. Pg: 589)*

An example of a weighted graph is shown in Figure 2.1.1 below:



Edges and their weights

- AB (4)
- AC (2)
- BC (5)
- CE (3)
- BD (10)
- ED (4)
- DF (11)

Figure 2.1.1: Self-Made Illustration of a Weighted Graph

As shown in the figure above, the weighted graph has the following nodes: *(A,B,C,D,E,F)*, the weight of their edges are denoted on the graph's adjacent right. If an algorithm were to begin

at A and its destination node was F, then it would explore all possible nodes by traversing each edge to find the shortest distance. The path with shortest cumulative sum of these edges that is able to traverse from A to F is labelled as the *'Optimal Path',* this has been showcased as the red path on the graph.

2.2: The Shortest Path Problem (SPP)

The Shortest Path Problem is a mathematical problem in Graph Theory which aims to find the shortest possible distance between two selected nodes of a weighted graph. The cumulative sum of the edges between a path are its distance, and the shortest of these distances is the solution to the Shortest Path Problem.

This can be represented mathematically as:

$$\textbf{Minimize} \sum_{e \in E} w_e x_e$$

Here, w_e is the weight of the edge e, and x_e is the binary variable that indicates whether edge e is included in the path (or not); if it isn't then $x_e = 0$, and if it is $x_e = 1$. It's important to ensure that the path is valid and therefore *Flow Conservation Constraints* are applied: this balances the flow into and out of the function for the source *vertex,* r. Which can be represented mathematically as:

$$\sum_{i \in V} x_{ir} - \sum_{i \in V} x_{ri} = -(n-1)$$

Here, $(n-1)$ edges will be traversed, and the Flow Conversation Constraint ensures that the difference between the flow out of, and into the function is equal to $-(n-1)$.

A *Flow Conservation Constraint* is applied for *vertices i and j* , where j is any vertex excluding the source. It is highly important to ensure that no more than one edge can enter or exit the destination vertex. This can be represented mathematically as:

$$\sum_{i \in V} x_{ij} - \sum_{i \in V} x_{ji} = \begin{cases} 1 \\ 0 \end{cases}$$

Here, if j is the destination vertex, then $\sum_{i \in V} x_{ij} - \sum_{i \in V} x_{ji} = 1$ *else* $\sum_{i \in V} x_{ij} - \sum_{i \in V} x_{ji} = 0$.

It's important to ensure that the values assigned to the *edges* are *non-negative integers*. These edges are translated to binary based on their inclusion, for those that are included (0) or excluded (1). Hence, a *Non-Negativity Constraint* is applied, which is represented mathematically as: $x_e \in Z^+$ for $e \in E$. *(Larsen, Jesper, and Jens Clausen)*

2.3: Shortest Path Algorithms

2.3.1: Dijkstra's Algorithm

Dijkstra's Algorithm is a pathfinding algorithm that was created by Edsger W. Dijkstra in 1956. Beginning at the source node (chosen node), the algorithm analyses the graph to find the shortest path between the source node and other nodes. The algorithm only functions on edges with non-negative weights, which sometimes makes it restrictive in certain conditions. To function properly, the algorithm keeps track of the currently known *shortest distance*. Once an even shorter path is the values are updated. Once the shortest path has been found, the algorithm marks it as *"visited"* and adds it to the path. The process doesn't stop until all the nodes in the graph have been marked as "visited". In essence, this connects the source node to all the other nodes and thus finds the shortest path possible needed to traverse each node.

A Dijkstra's Algorithm is an example of a "Greedy Algorithm". Due to its employment of optimizations, it can be contrasted with a "naïve algorithm". Implementing naïve algorithms is very computationally expensive as it would search through every single one of the vertices in each iteration to find the shortest distance. *(Agubata, Immaculate Chidinma, et al.)*

```
FUNCTION dijkstra_algorithm(graph, start_node)
    unvisited_nodes = list of all nodes in graph

    shortest_path = empty dictionary
    previous_nodes = empty dictionary

    max_value = a very large number

    FOR each node IN unvisited_nodes DO
        shortest_path[node] = max_value
    END FOR

    shortest_path[start_node] = 0

    WHILE unvisited_nodes is not empty DO
        current_min_node = null
        FOR each node IN unvisited_nodes DO
            IF current_min_node IS null THEN
                current_min_node = node
            ELSE IF shortest_path[node] < shortest_path[current_min_node] THEN
                current_min_node = node
            END IF
        END FOR

        neighbors = get outgoing edges of current_min_node from graph


        FOR each neighbor IN neighbors DO
            tentative_value = shortest_path[current_min_node] + value of edge(current_min_node, neighbor)

            IF tentative_value < shortest_path[neighbor] THEN
                shortest_path[neighbor] = tentative_value
                previous_nodes[neighbor] = current_min_node
            END IF
        END FOR

        REMOVE current_min_node FROM unvisited_nodes
    END WHILE

    RETURN previous_nodes, shortest_path
END FUNCTION

```

Figure 2.3.1: Pseudocode for Dijkstra's Algorithm

The Pseudocode for the Dijkstra's Algorithm above has been broken down below:

Declaration: The shortest_path dictionary is used to store the distances from the source nodes to every other node. The previous_node dictionary on the other hand is used to store the most optimal path to reach every node.

The algorithm sets the distance from the source node to every node as max_value as the distances are initially unknown. The distance for the starting node is set to 0 because there is no cost for traversing back to itself.

The algorithm then executes a while loop until there are no unvisited nodes:

- In each iteration, the unvisited node (current_min_node) with the shortest distance is identified into the shortest_path dictionary.
- The algorithm then checks if the neighbours of current_min_node have been visited. If it isn't then the potential distance from the source node to the neighbour is calculated.
- The algorithm then checks if the newly calculated distance to a neighbour is shorter than the previously known distance. If it is, the shortest distance is updated and so is the path leading to the neighbour. This process is also known as "*Relaxing the edge"*.

Completion: The process continues until all the nodes have been "visited". When the process is finished, the shortest_path dictionary contains the shortest distance from the source node to every other node.

The algorithm's use has been demonstrated on the weighted graph in 2.1.1 below.

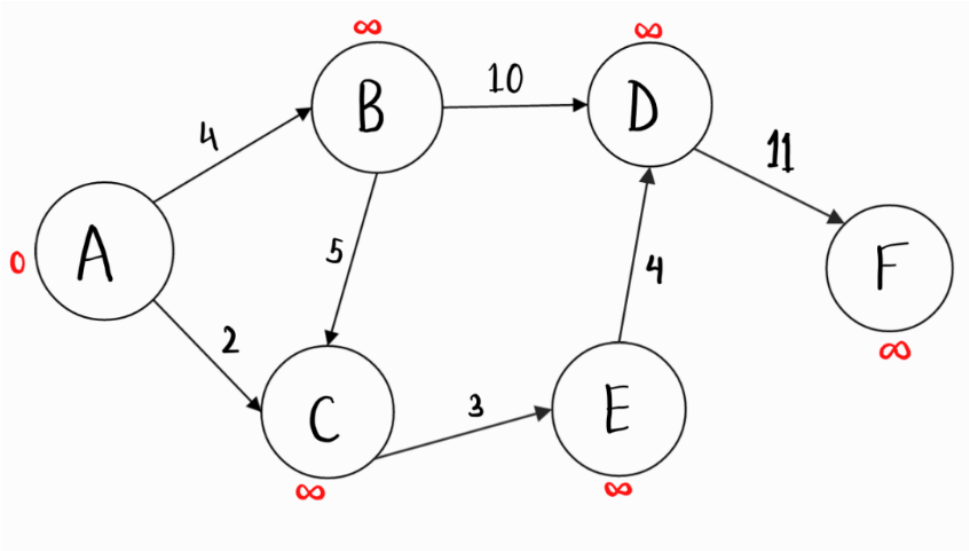


Figure 2.2.1: Implementation of Dijkstra's Algorithm on a Weighted Graph

As seen in the diagram, initially the distance from A (Source Node) to every other node is ∞, as their paths are "unvisited". Next, the algorithm traverses to the neighbouring vertices of B and C. The edge from A to B is relaxed: $0 + 4 = 4$, the edge from A to C is relaxed: $0 + 2 = 2$.

The algorithm will then select C and traverse to its neighbours. C to B is relaxed: $2 + 5 = 7$ and C to E is relaxed: $2 + 3 = 5$. Therefore, E becomes the next node. At the end, this entire process continues until all the vertices are mapped into the path. The following results would be obtained:

- A → A: 0 (starting point)
- A → B: 4 (via direct edge from A to B)
- A → C: 2 (via direct edge from A to C)
- A → D: 9 (A → C → E → D)
- A → E: 5 (A → C → E)
- **A → F: 20 (A → C → E → D → F)** (Final Path)

The *time complexity* of a Dijkstra's algorithm varies based on what type of data structures are implemented, this has been showed in the table below:

- V = Number of Vertices
- E = Number of Edges

Worst Case Analysis	Best Case Analysis
For Adjacency Matrices (Without Priority Queues): Here, all unvisited nodes need to be scanned which takes $O(V)$ time. This process is then repeated for all nodes which makes the time complexity of $\mathbf{O(V^2)}$	The best-case analysis occurs when the graph is sparse and there are very few nodes to explore. The best-case analysis occurs with adjacency list and priority queue: $\mathbf{O((V+E)\log V)}$. However, the actual runtime during best-case analysis will be even lower as some edges and nodes will need to be processed.
For Binary Heaps: A binary heap is used to store and extract nodes with the smallest distances and thus the time complexity becomes $O(\log V)$**.** Updating the distances takes a time of $O(E)$. This makes the final complexity of $\mathbf{O((V+E)\log V)}$**.**	
For Fibonacci Heaps: A Fibonacci heap is used to optimize the priority queue operations, and the node takes $O(\log V)$, and updating the shortest distance takes $O(1)$. Thus, the final time complexity takes $\mathbf{O(E + V\log V)}$**.**	

2.3.2: A* Algorithm

The A* Algorithm is a pathfinding algorithm that was created by Peter Hart, Nils Nilsson and Betram Raphael at Stanford Research Institute in 1968. A* Algorithm is a best-first search algorithm, it starts from the source node and aims to find the smallest path (in terms of cost) to the target node. It is an extension of the Dijkstra's Algorithm but one that also includes a heuristic function. At each iteration of its main loop, it identifies the path with the lowest cost and extends it. The chosen path minimizes:

$$\boldsymbol{f(n) = g(n) + h(n),}$$

Here, n is the next node on the path, g(n) is the cost of path from source node to n, h(n) is a heuristic function used to estimate the cheapest path from n to target node.

If the heuristic is correctly chosen it is said to be either *admissible* or *consistent:*

- *Admissible:* $\boldsymbol{h(n) \leq h^*(n)}$, where $\boldsymbol{h^*(n)}$ is the true cost of reaching the goal n. This ensures that the algorithm never overestimates its goal.
- *Consistent (Monotonicity Condition):* $\boldsymbol{h(n) \leq c(n, n') + h(n'),}$ where $\boldsymbol{c(n, n')}$ is the cost of moving from node n to its nearest neighbour n'. This ensures that the estimated cost does not exceed the sum of the cost of moving to the nearest neighbour and estimated cost from n' to target.

If the heuristic is incorrectly chosen, then h(n) becomes 0 which simplifies the total function to $\boldsymbol{f(n) = g(n) + 0 = g(n)}$ which in turn degenerates the A* algorithm to a Dijkstra's Algorithm.

(Hart, Peter, Nils Nilsson, and Bertram Raphael)

```
function reconstruct_path(cameFrom, current)
    total_path := {current}
    while current is in cameFrom.Keys:
        current := cameFrom[current]
        total_path.prepend(current)
    return total_path

function A_Star(start, goal, h)
    openSet := {start}

    cameFrom := an empty map

    gScore := map with default value of Infinity
    gScore[start] := 0

    fScore := map with default value of Infinity
    fScore[start] := h(start)

    while openSet is not empty:
        current := the node in openSet having the lowest fScore[] value

        if current = goal:
            return reconstruct_path(cameFrom, current)

        openSet.Remove(current)

        for each neighbor of current:
            tentative_gScore := gScore[current] + d(current, neighbor)

            if tentative_gScore < gScore[neighbor]:
                cameFrom[neighbor] := current
                gScore[neighbor] := tentative_gScore
                fScore[neighbor] := tentative_gScore + h(neighbor)

                if neighbor not in openSet:
                    openSet.add(neighbor)

    return failure
```

Figure 2.3.1: Pseudocode for A Algorithm*

The Pseudocode for the A* Algorithm above has been broken down below:

Declaration: The cameFrom dictionary stores the best previous node for every node, the gScore dictionary stores the lowest known cost to reach each node from the start, and the fScore dictionary stores the estimated cost of reaching the goal through each node.

The openSet initializes alongside the start node and represents a set of discovered nodes in need of evaluation. The gScore for every node is set to ∞ (except the start node), and fScore is initialized using the heuristic function h(start).

Every iteration, the node current with the lowest fScore[] value is chosen and is selected from openSet. If the current node and target node are equal, the algorithm reconstructs the path.

If the neighbor is not in cameFrom then it signals the evaluation hasn't occurred which causes the algorithm to calculate the tentative distance: tentative_gScore is calculated as the sum of gScore[current] and the cost is expressed as d(current, neighbor).

If tentative_gScore < gScore[neighbor], it means a better path has been found and the algorithm updates cameFrom[neighbor] to current, gScore[neighbor] to tentative_gScore, and fScore[neighbor] to the sum of gScore[neighbor] and the heuristic h(neighbor).

If neighbor is not in openSet then it is added for evaluation for the next iteration. If openSet ever becomes empty, then the algorithm returns failure.

The algorithm's use has been demonstrated on the weighted graph in 2.3.2 below.

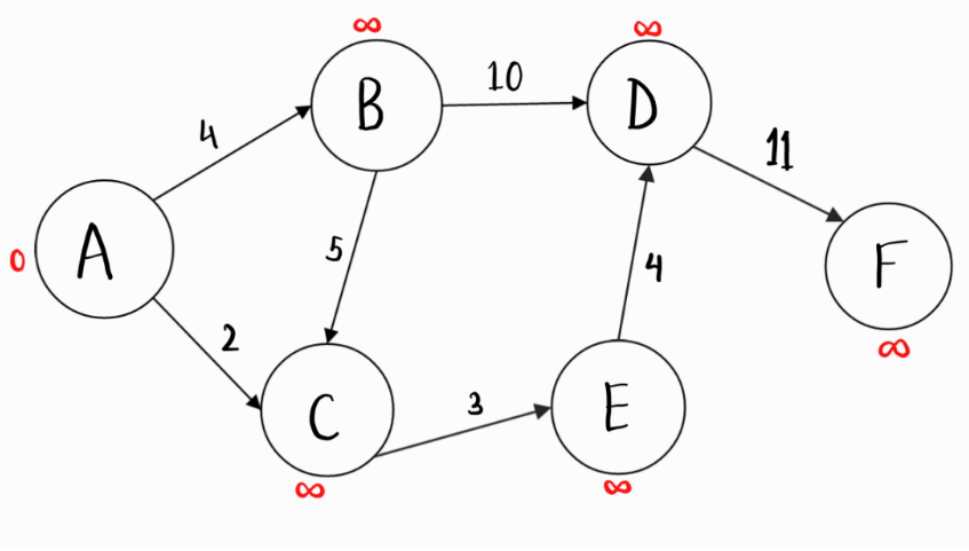


Figure 2.2.1: Implementation of A Algorithm on a Weighted Graph*

Initially the distance from A (Source Node) to every other node is set to ∞, except A (which is set to 0). The heuristic function h(n) is used to estimate the cost to reach F.

The algorithm starts by evaluating vertices B and C. Path from A to B is relaxed to 0+4=4, A to C is relaxed to 0+2=2. Given that C has a lower cost (including heuristic estimate), A* selects C as the next. The tentative distance to B is 2+5=7, and to E is 2+3=5. Hence, C is chosen next. This process continues until F is reached, which gives the shortest path as:

A → C → E → D → F

The *time complexity* of a A* algorithm varies based on what type of data structures are implemented, this has been showed below:

- V = Number of Vertices
- E = Number of Edges

- **Worst Case Time Complexity:** When every edge is examined and heuristic doesn't effectively guide searching, then it has a time complexity of $\boldsymbol{O(E)}$**.** If a priority queue such as a binary heap is used, complexity is typically $\mathbf{O((V+E)\log V)}$**.** However, in the case of Fibonacci heaps, the complexity improves to $\mathbf{O((E+V\log V)}$**.**
- **Best Case Time Complexity**: The heuristic function effectively reduces search space due to being well informed which causes the time complexity of $\boldsymbol{O(E)}$**.**

3. Hypothesis

3.1. Applied Theory

The experiment will investigate the relationship between execution time **(T)** in nanoseconds and number of vertices in the graph **(V).** V will be systematically increased to establish correlations between the variables. The performance of these algorithms under varying graph conditions will be compared. Both, sparse and dense graphs will be constructed and represented as E edges. Where, for sparse graphs, $\boldsymbol{E = O(V)}$ and for dense graphs, $\boldsymbol{E = O(V^2)}$.

Dijkstra's Algorithm when implemented using an adjacency matrix will have a time complexity of: $\mathbf{T_{Dijkstra} = O(V^2)}$**.** If a priority queue is implemented for extracting the vertex efficiently instead, the minimum distance will be performed in $\mathbf{O(\log V)}$ time. Relaxation occurs for the adjacent and extracted vertices making the time complexity: $\mathbf{T_{Dijkstra} = O((V+E)\log V)}$**.**

For the A* algorithm, the performance is heavily dependent on the effectiveness of heuristics. Both, Manhattan and Euclidean Distances can significantly reduce search spaces which will reduce node expansions. The execution time can be approximated to: $\mathbf{T_{A*} = O((V + E)\log V).}$

Relaxation will then be utilized which will change execution times. Number of relaxations is dependent on number of edges. Edge relaxation involves checking and updating distances to neighbouring vertices in $\mathbf{O(1)}$ time. For Dijkstra's Algorithm, time complexity in presence of priority queues will become: $\mathbf{T_{Dijkstra} = O(E \log V)}$ and A* becomes $\mathbf{T_{A^*} = O(E \log V)}$

3.2. Hypothesis

As the number of vertices will increase, the execution time of the algorithms is expected to show differing behaviour based on the type of graph. For Sparse Graphs, where number of edges $\boldsymbol{E \approx O(V)}$, the Dijkstra's algorithm can be expected to perform efficiently as it has a time complexity of $\boldsymbol{O(V^2)}$ when using an adjacency matrix, however A* can be expected to have an even lower execution time due to the guidance of heuristics as their implementation would reduce the frequency of unnecessary node expansions.

For Dense Graphs, $\boldsymbol{E \approx O(V^2)}$, A* should outperform Dijkstra's as its heuristic function should cut off non-optimal paths and reduce total search space which in turn would reduce execution time. As the heuristic function receives more information, the performance gain of A* over Dijkstra's will also increase as the difference in their total search spaces would become even greater.

Therefore, a logarithmic/power relationship between V and T can be hypothesized for both algorithms if dense graphs are used. As for sparse graphs, a polynomial relationship can be

hypothesized between V and T. A* algorithm has also been hypothesized to have a superior execution time in structured graph scenarios.

4. Methodology

4.1. Graph Representation and Generation

A myriad of weighted graphs ($\mathbf{G(V, E)}$) will be constructed from synthetic data. V would represent the nodes/vertices within a network, and E (edges) would represent direct connections between them. The weight of edges will show travelling costs.

Number of vertices ($|\mathbf{V}|$) will vary from 100 to 1000.

Number of edges ($|\mathbf{E}|$) will be adjusted to create sparse and dense graphs.

- Sparse Graphs: $|\mathbf{E}| \approx \mathbf{O}(|\mathbf{V}|)$
- Dense Graphs: $|\mathbf{E}| \approx \mathbf{O}(|\mathbf{V}|^{\mathbf{2}})$

Directed graphs will be generated using Python alongside NetworkX Library. The weight of edges will be assigned using a uniform distribution. The weights (travel costs) will range from 1 to 10. For all directed edges between vertices u and v, the weight will be generated as:

$$\boldsymbol{w \sim U(1, 10)}$$

Iteratively, vertices and edges will be added to the graph based on the parameters.

4.2. Independent Variables

The independent variables in this investigation are the graph's parameters (Number of Vertices (|V|) and Number of Edges (|E|)). By systematically varying these parameters, the sizes of graphs will change thereby creating Sparse and Dense Graphs.

Number of vertices will be increased in increments of 100. $|\boldsymbol{V}| \in [\mathbf{100}, \mathbf{200}, \mathbf{300}, \mathbf{500}, \mathbf{1000}]$

The edge density will vary by adjusting the number of edges:

- Sparse Graphs: $|\mathbf{E}| = \mathbf{O}(|\mathbf{V}|)$**,** every vertex will have 1-3 edges.
- Dense Graphs: $|\mathbf{E}| = \mathbf{O}(|\mathbf{V^2}|)$**,** all vertices can be connected to each other.

Number of edges will be calculated by $\mathbf{Sparse}\ |\mathbf{E}| \approx \mathbf{2} \times |\mathbf{V}|$ and $\mathbf{Dense}\ |\mathbf{E}| \approx \mathbf{0.5} \times |\mathbf{V^2}|$

4.3. Dependant Variables

The dependant variable in this investigation is execution time (measured in nanoseconds) of the algorithm. This will be measured using the time.perf_counter_ns() function. The formula for this is: $\boldsymbol{T_{execution} = t_{end} - t_{start}}$. Another performance metric $\boldsymbol{N_{expanded}}$ will be used to measure algorithmic efficiency.

4.4. Controlled Variables

The following variables will be kept constant throughout the experiment:

Variable	Description	Specification

Programming Language & IDE	IDE, Coding Language and Environment	Python 3.x Visual Studio Code 1.79, Python 3.11
Heuristic (A*)	The heuristic will be the Euclidean distance between the nodes, formula for which is constant.	$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Graph Representation	Weighted Graphs will be constantly referenced as nx.DiGraph()	nx.DiGraph()
Random Seed	To keep graph structures identical, a fixed random seed will be used throughout.	random.seed(42)
Hardware	Experiment will be run on the same device to minimize variations due to processing speeds of alternating devices	Acer Aspire Lite 16 (Windows 11 Home, 8 GB RAM, intel core i5)
Input Data	Randomly generated graph data will be same for both algorithms	Randomly generated graph data
Initial Conditions	The source node will remain the same for each run of every algorithm.	v_0

4.5. Procedure

No.	Procedure
1	Install necessary python libraries. (NetworkX, Matplotlib, Numpy) `PS C:\Users\[redacted]\Downloads\Project> pip install networkx matplotlib numpy`
2	Create a .py file
3	Define a function generate_graph(num_vertices, density) to create a directed graph with a chosen number of vertices. Make it so that the weights of the edges are randomly generated from 1-10.
4	Define two functions. dijkstra(graph, start) & a_star(graph, start, goal)
5	Initialize empty sets to store execution times and the number of edges for both types of graphs. For each vertex count, num_vertices_list (100, 200, 300, ..., 1000)
6	Generate a sparse graph and record the execution time for Sparse and Dijkstra's Algorithm. Store the collected data and number of edges in lists.
7	Use Matplotlib to plot the execution time with the number of vertices in a graph

Figure 4.5.1: Procedure Table

(Refer to Appendix A for the Code)

5. Experiment Results

5.1. Tabular Data Results

The table below shows the mean execution time for the Dijkstra's and A* algorithm on Sparse Graphs. *(Refer to Appendix B for Raw Data)*

Vertices	Edges	Execution Time (Nanoseconds)	
		Dijkstra's (Sparse)	A* (Sparse)
100	990	677825.03	264862.95
200	3980	1728050.03	613225.20
300	8970	3393524.98	3203475.93
400	15960	6484614.98	6825113.86
500	24950	10383849.97	5033688.78
600	35940	18799300.02	16312000.47
700	48930	26963299.98	18990850.09
800	63920	55681000.01	52549500.04
900	80910	71420399.99	25539350.02
1000	99900	74375699.98	58843900.02

Figure 5.1.1: Effect of Vertices on Average Execution Time of Dijkstra's and A Algorithms on Sparse Graphs*

The table below shows the mean execution time for the Dijkstra's and A* algorithm on Dense Graphs. *(Refer to Appendix B for Raw Data)*

Vertices	Edges	Execution Time (Nanoseconds)	
		Dijkstra's (Dense)	A* (Dense)
100	4950	2275999	1239000
200	19900	9345000	9753499
300	44850	24726100	18715800
400	79800	52430600	24362599

500	124750	78538100	66637600
600	179700	121960000	55843299
700	244650	159604300	50566400
800	319600	220764799	65785299
900	404500	523508899	705460399
1000	499500	310178400	21775200

Figure 5.1.2: Effect of Vertices on Average Execution Time of Dijkstra's and A Algorithms on Dense Graphs*

5.2. Graphical Representation of Data

The execution time is graphed against the number of vertices to represent the complexity of the algorithms. The Figures below show the relationship between the algorithm's number of vertices and their execution times. The data has been represented in **non-linear regression models.**

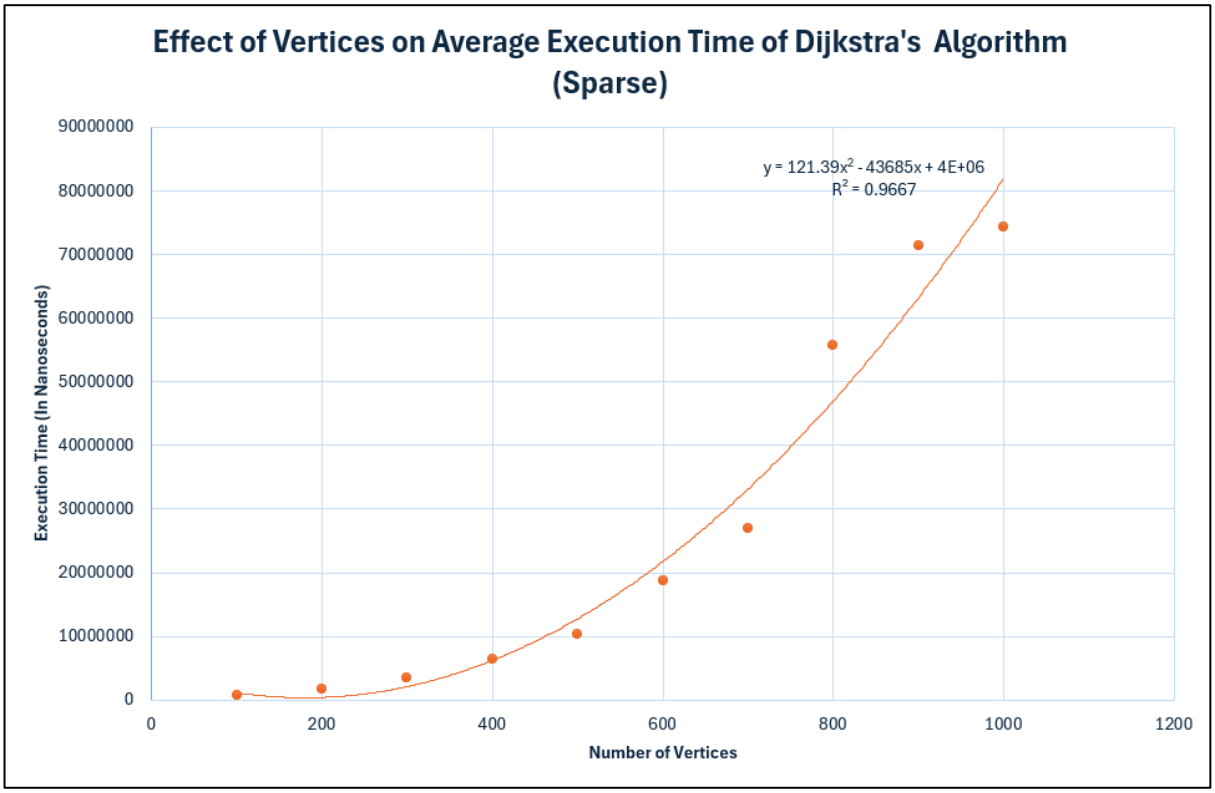


Figure 5.2.1: Execution Time of Dijkstra's Algorithm (Sparse) against Number of Vertices (Quadratic)

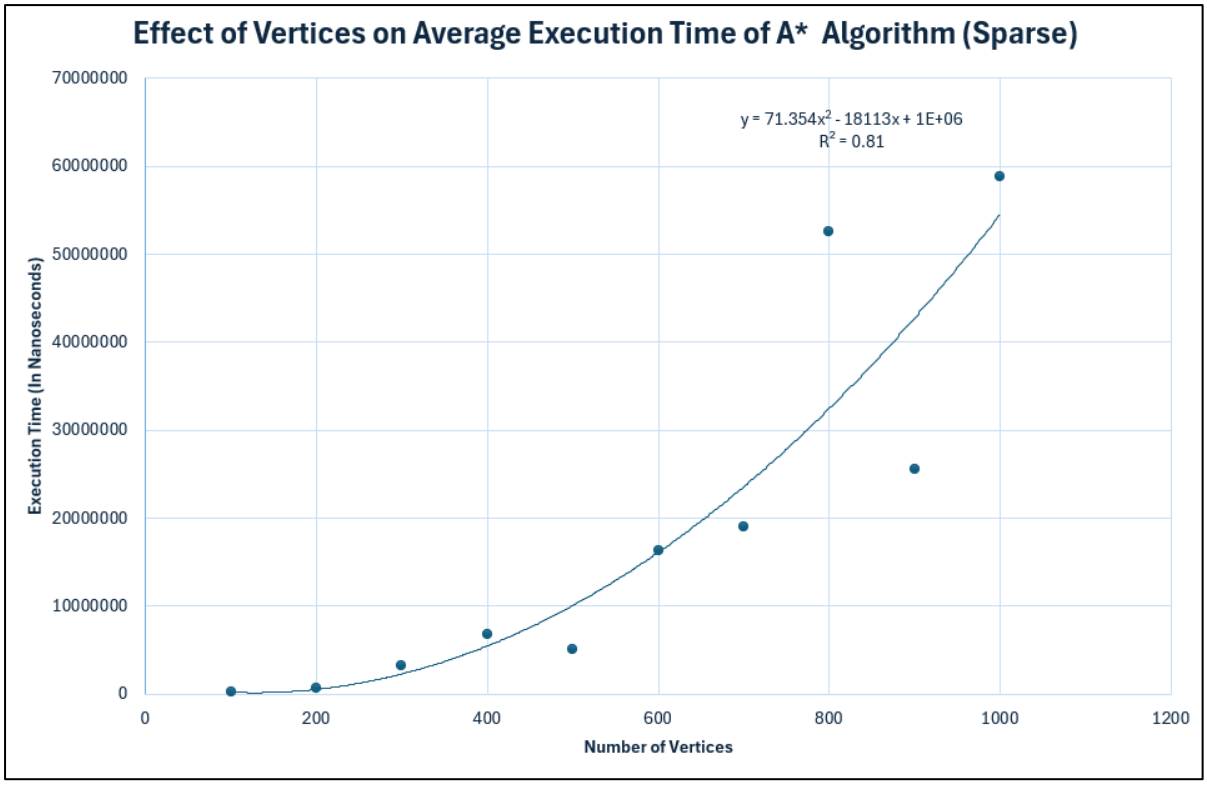


Figure 5.2.2: Execution Time of A Algorithm (Sparse) against Number of Vertices (Quadratic)*

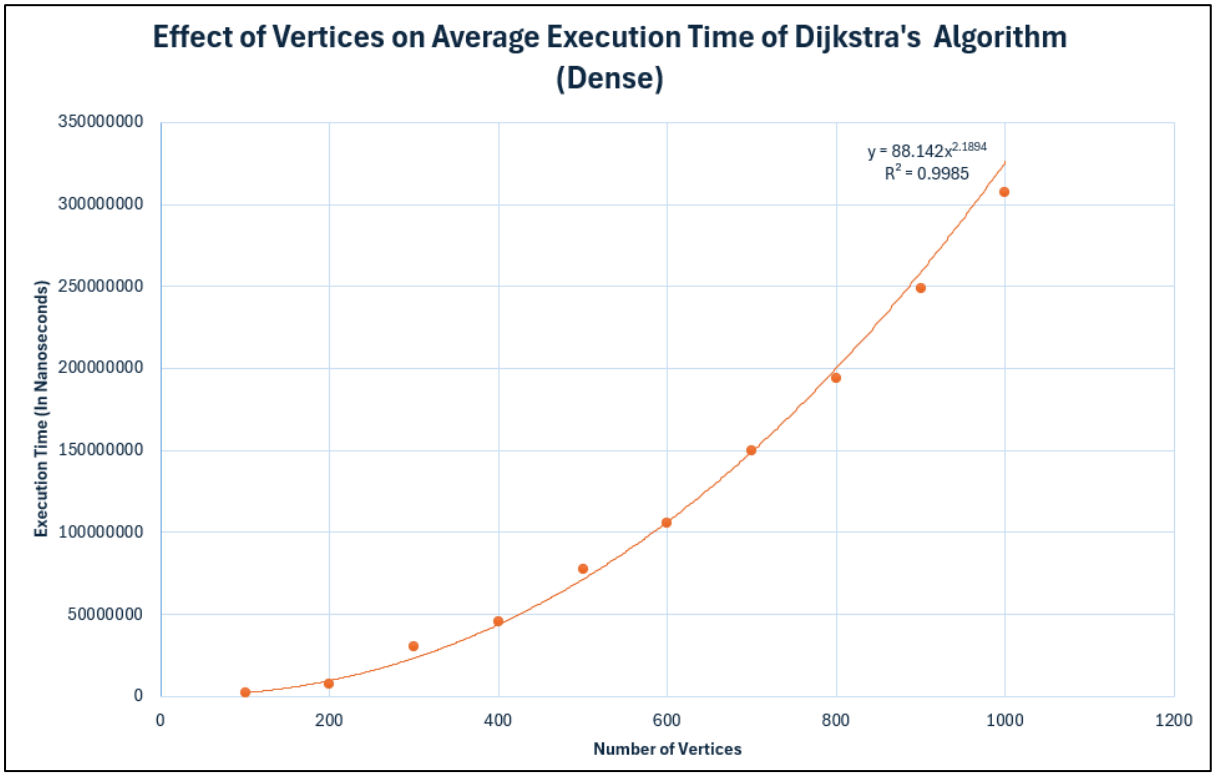


Figure 5.2.3: Execution Time of Dijkstra's Algorithm (Dense) against Number of Vertices (Power Model)

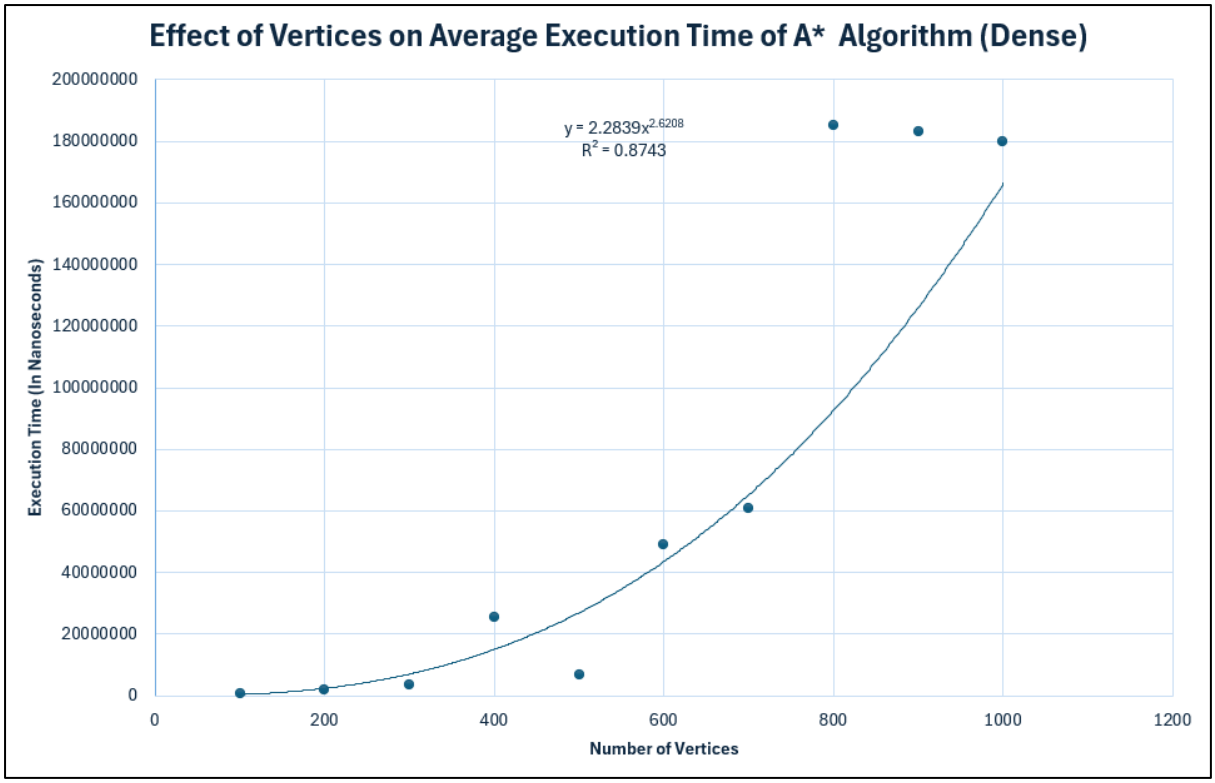


Figure 5.2.4: Execution Time of A Algorithm (Dense) against Number of Vertices (Power Model)*

5.3. Analysis of Data

5.3.1. Analysis of Dijkstra's Algorithm

The formulated hypothesis predicting a polynomial relationship (between x and y) has been proven correct for the sparse graphs. The R^2 value of the graph is 0.9667, showcasing a very strong correlation between the execution times and the number of vertices. The quadratic model indicates that that 96.67% of variation in execution time (nanoseconds) can be explained by change in number of vertices. The equation in Figure 5.2.1 is in the form of $y = ax^2 + bx + c$, and is $y = 121.39x^2 - 43685x + 4 \times 10^6$.

Meanwhile, the hypothesis predicting an exponential relationship for dense graphs ($y = a \times b^x$), has been proven partially correct. After analysing, a power model ($y = a \times x^b$) was

deemed more suitable as there wasn't a rapid enough increase. The power model indicates that 99.85% of variation in execution time can be explained by change in number of vertices. The equation in Figure 5.2.2 is: $y = 88.142x^{2.1894}$.

5.3.2. Analysis of A* Algorithm

The formulated hypothesis predicting a polynomial relationship (between x and y) has been proven correct for the sparse graphs. Due to the variable search behaviour of A* algorithm, the R^2 value of the graph is only 0.81, showcasing a moderately strong correlation between the execution times and the number of vertices. The quadratic model indicates that that 81% of variation in execution time can be explained by change in number of vertices. The equation in Figure 5.2.1 is in the form of $y = ax^2 + bx + c$, and is $y = 71.354x^2 - 18113x + 1 \times 10^6$.

Meanwhile, the hypothesis predicting an exponential relationship for dense graphs ($y = a \times b^x$), has been proven partially correct. After analysing, a power model ($y = a \times x^b$) was deemed more suitable as there wasn't a rapid enough increase. The power model indicates that 87.43% of variation in execution time can be explained by change in number of vertices. The equation in Figure 5.2.2 is: $y = 2.2839x^{2.6208}$.

5.3.3. Comparing Efficiency of Both Algorithms

The second derivative test has been done to find which algorithm is most efficient on sparse graphs.

Line of Best Fit: Figure 5.2.1.

$$y = 121.39x^2 - 43685x + 4 \times 10^6$$

$$\frac{dy}{dx} = 242.78x - 43685$$

$$\frac{d^2y}{dx^2} = 242.78$$

Line of Best Fit: Figure 5.2.2.

$$y = 71.354x^2 - 18113x + 1 \times 10^6.$$

$$\frac{dy}{dx} = 142.708x - 18113$$

$$\frac{d^2y}{dx^2} = 142.708$$

Since $142.708 < 242.708$, on Sparse graphs, A* is more efficient on extrapolated data as it has a slower growth rate.

The test has been done to find which algorithm is most efficient on dense graphs.

Line of Best Fit: Figure 5.2.3.

$$y = 88.142x^{2.1894}$$

$$\frac{dy}{dx} \approx 193.262x^{1.1894}$$

$$\frac{d^2y}{dx^2} \approx 229.2973x^{0.1894}$$

Line of Best Fit: Figure 5.2.4.

$$y = 2.2839x^{2.6208}.$$

$$\frac{dy}{dx} \approx 5.9854x^{1.6208}$$

$$\frac{d^2y}{dx^2} \approx 9.7002x^{0.6208}$$

At small and medium values of x (input sizes), the A* algorithm would prove more efficient as it grows slower initially because of the lower value of a. An inequality test has been done below to find the minimum number of vertices needed for Dijkstra's to become more efficient.

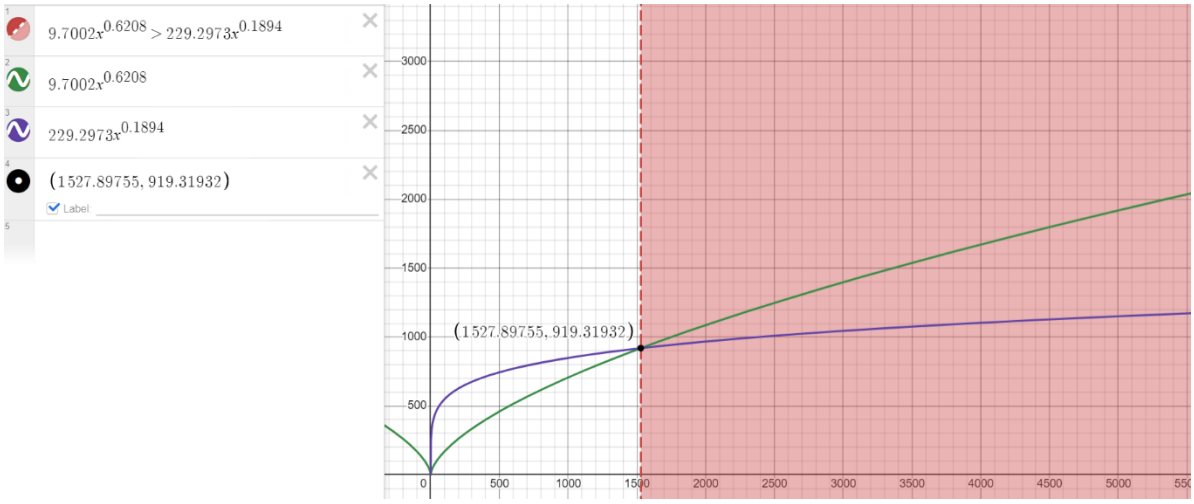


Figure 5.3.1: Inequality Graph showcasing the point at which Dijkstra's Algorithm becomes more effective.

As seen above at 1528 vertices, Dijkstra's algorithm becomes more efficient as its growth rate lessens.

---

6. Limitations

There were several experiment-related limitations that could have slightly affected the accuracy of the experiment:

- Instead of using data based on real-world networks, randomly generated data was used. Real laws have more complex structures and showcase properties such as clustering and power-law relationships. The randomly generated graphs can't simulate these complex structures.
- Vertices only lead up to 1000, this makes it difficult to extrapolate data.
- The heuristic (Euclidean distance) does not fully represent the capabilities of A* algorithm as heuristics such as Manhattan Distance $\boldsymbol{h(n) = |x_2 - x_1| + |y_2 - y_1|}$, and Chebyshev Distance, $\boldsymbol{h(n) = \max(|x_2 - x_1|, |y_2 - y_1|)}$

- The CPU of the Laptop was utilized for background processes which could have limited algorithms from reaching their full ability.

7. Conclusion

The experiment has successfully investigated the differences in the pathfinding efficiency of the A* algorithm and Dijkstra's algorithm on different types of graphs: sparse and dense.

In sparse graphs, Dijkstra's algorithm showcased a superior performance with a time complexity of $\boldsymbol{O((V+E)log\,V}$ when using a priority queue. A* also showcased the same complexity; however, the Dijkstra's algorithm had a much stronger polynomial relationship reflected in its value of: $R^2 = 0.9667$. Meanwhile, due to its variable search behaviour, A* had a moderate performance of sparse graphs as evident from its $R^2 = 0.81$. The second derivative test also aided in judging their efficiency for extrapolated data as A* has a lower second derivative meaning that at large input data sizes it would become more efficient.

In dense graphs, a power model was deemed more suitable for evaluating the models. A* leveraged its heuristics which led to it outperforming Dijkstra's algorithm. However, the linear inequality test proved that the Dijkstra's algorithm becomes more efficient when the number of vertices crosses 1528.

In conclusion, Dijkstra's algorithm outperforms A* on sparse graphs with low and medium sized inputs (interpolated) until one point where A* becomes more efficient due to its slow growth rate. Meanwhile, A* outperforms Dijkstra's on dense graphs with low and medium sized inputs (interpolated) until it reaches 1528 vertices. These insights are helpful for practical applications in urban transportation, city planning, and network routing.

Citations

- Cormen, Thomas H., et al. Introduction to Algorithms. 3rd ed., MIT Press, 2009. [Accessed May 2024]
- Sedgewick, Robert, and Kevin Wayne. Algorithms. 4th ed., Addison-Wesley, 2011. [Accessed May 2024]
- Sidhu, Harinder Kaur. "Performance Evaluation of Pathfinding Algorithms." Master's Thesis, University of Windsor, 2019. Available at: https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=9230&context=etd, pp. 15-16. [Accessed May 2024]
- MIT OpenCourseWare. "Lecture 12: Bellman-Ford." Introduction to Algorithms, Massachusetts Institute of Technology (MIT), Spring 2020, *https://ocw.mit.edu/courses/6006-introduction-to-algorithms-spring 2020/resources/lecture-12-bellman-ford/* [Accessed May 2024]
- *"Weighted Graphs." CS 3114 Data Structures and Algorithms*, Virginia Tech, Fall 2010, https://courses.cs.vt.edu/~cs3114/Fall10/Notes/T22.WeightedGraphs.pdf. [Accessed May 2024]
- Larsen, Jesper, and Jens Clausen. "The Shortest Path Problem." Department of Management Engineering, Technical University of Denmark. (Accessed June 2024).
- "7.2.1 Single Source Shortest Paths Problem: Dijkstra's Algorithm." Archived from the Original on 2016-03-04 *https://web.archive.org/web/20160304025622/http://lcm.csa.iisc.ernet.in/dsa/node162.html* [Accessed May 2024]
- "Zhou Minhang, and Nina Gao. "Research on Optimal Path based on Dijkstra Algorithms." Advances in Computer Science Research, vol. 87, 3rd International

Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019)" [Accessed June 2024]

- Vickers, Alex. *A Comparison of Worst Case Performance of Priority Queues Used in Dijkstra's Shortest Path Algorithm.* University of Canterbury, Department of Computer Science, 1986. https://ir.canterbury.ac.nz/server/api/core/bitstreams/96337764-25cb-494a-a7e5-a4843a055877/content [Accessed June 2024]
- Bu, Dongbo. "Lecture 7: Binary Heap, Binomial Heap, and Fibonacci Heap." CS711008Z Algorithm Design and Analysis, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, 2015. https://helios2.mi.parisdescartes.fr/~lomn/Cours/AV/Complements/Heap.pdf [Accessed June 2024]
- Agubata, Immaculate Chidinma, et al. "Design and Optimization of Bus Booking System Using Dijkstra's Algorithm." International Journal of Science and Business, vol. 4, no. 12, 2020, pp. 21-37. EconPapers, https://ideas.repec.org/a/aif/journl/v4y2020i12p21-37.html [Accessed June 2024]
- Patel, Amit. "A* Algorithm – A* Algorithm, Dijkstra's Algorithm & Greedy Best-First Search Comparison." *Game Programming*, Stanford University, https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html [Accessed June 2024]
- Hart, Peter, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, 1968, pp. 100-107. [Accessed July 2024]
- "Lecture 5: A* and Heuristics." *ECE 448: Artificial Intelligence*, University of Illinois at Urbana-Champaign, Spring 2020,

https://courses.grainger.illinois.edu/ece448/sp2020/slides/lec05.pdf [Accessed June 2024]

- Cyrus. D. Cantrell (2000). Modern Mathematical Methods for Physicists and Engineers. Cambridge University Press. ISBN 0-521-59827-3. [Accessed July 2024]
- David M. J. Tax; Robert Duin; Dick De Ridder (2004). *Classification, Parameter Estimation and State Estimation: An Engineering Approach Using MATLAB*. John Wiley and Sons. [Accessed July 2024]
- Thompson, Kevin, and Tevian Dray. "Taxicab Angles and Trigonometry." *Pi Mu Epsilon Journal*, 2000, https://www.jstor.org/stable/24340535. [Accessed August 2024]
- Hagberg, A. A., Swart, P. J., & S Chult, D. (2008). Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference* (pp. 11-15). [Accessed September 2024]
- Irfan, Mohammad T. "Programming with NetworkX in Python." *YouTube*, 22 Dec. 2020, https://youtu.be/CPQeSmDGiOQ. [Accessed September 2024]
- Kaluarachchi, Chanaka. "Non-Linear Regression Analysis." *University of Otago*, https://www.otago.ac.nz/__data/assets/pdf_file/0005/300002/non-linear-regression-analysis-104254.pdf [Accessed September 2024]
- Desmos. "Desmos Graphing Calculator." Desmos, https://www.desmos.com/calculator.

Appendix A

pathfinding_experiment.py > ...

```python
import networkx as nx
import random
import numpy as np
import heapq
import time
import matplotlib.pyplot as plt
import csv

# Function to create the graph
def generate_graph(num_vertices, density):
    G = nx.DiGraph()

    # Create vertices
    for i in range(num_vertices):
        G.add_node(i)

    # Determine number of edges
    num_edges = int(density * num_vertices * (num_vertices - 1))

    # Create edges with weights
    while G.number_of_edges() < num_edges:
        u = random.randint(0, num_vertices - 1)
        v = random.randint(0, num_vertices - 1)
        if u != v:
            weight = random.uniform(1, 10)  # Assign a random weight between 1 and 10
            G.add_edge(u, v, weight=weight)

    print(f"Generated graph with {num_vertices} vertices and {G.number_of_edges()} edges.")
    return G, G.number_of_edges()  # Return the graph and number of edges
```

```python
# Dijkstra's Algorithm
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph.nodes}
    distances[start] = 0
    priority_queue = [(0, start)]  # (distance, node)

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor in graph.neighbors(current_node):
            weight = graph[current_node][neighbor]['weight']
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

```python
# A* Algorithm
def heuristic(node, goal):
    return 0  # Simple placeholder heuristic

def a_star(graph, start, goal):
    open_set = {start}
    came_from = {}

    g_score = {node: float('inf') for node in graph.nodes}
    g_score[start] = 0

    f_score = {node: float('inf') for node in graph.nodes}
    f_score[start] = heuristic(start, goal)

    while open_set:
        current = min(open_set, key=lambda node: f_score[node])

        if current == goal:
            return g_score  # or reconstruct the path

        open_set.remove(current)

        for neighbor in graph.neighbors(current):
            weight = graph[current][neighbor]['weight']
            tentative_g_score = g_score[current] + weight

            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
                open_set.add(neighbor)

    return g_score
```

```python
# Experiment parameters
num_vertices_list = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]  # Example sizes
sparse_density = 0.1  # Adjust for sparse graphs
dense_density = 0.5   # Adjust for dense graphs

# Initialize lists for storing execution times and edges
sparse_dijkstra_times = []
sparse_a_star_times = []
dense_dijkstra_times = []
dense_a_star_times = []
sparse_edges = []
dense_edges = []
```

```python
# Running the experiment for sparse graphs
for num_vertices in num_vertices_list:
    print(f"Processing Sparse Graph: {num_vertices} vertices")
    # Generate sparse graph
    G_sparse, num_edges_sparse = generate_graph(num_vertices, sparse_density)
    sparse_edges.append(num_edges_sparse)

    # Run Dijkstra's algorithm
    start_time = time.perf_counter()
    dijkstra_results = dijkstra(G_sparse, 0)  # Starting from node 0
    end_time = time.perf_counter()
    sparse_dijkstra_times.append((end_time - start_time) * 1e9)  # Convert to nanoseconds

    # Run A* algorithm (using the last node as goal)
    start_time = time.perf_counter()
    a_star_results = a_star(G_sparse, 0, num_vertices - 1)  # Goal is the last node
    end_time = time.perf_counter()
    sparse_a_star_times.append((end_time - start_time) * 1e9)  # Convert to nanoseconds
```

```python
# Running the experiment for dense graphs
for num_vertices in num_vertices_list:
    print(f"Processing Dense Graph: {num_vertices} vertices")
    # Generate dense graph
    G_dense, num_edges_dense = generate_graph(num_vertices, dense_density)
    dense_edges.append(num_edges_dense)

    # Run Dijkstra's algorithm
    start_time = time.perf_counter()
    dijkstra_results = dijkstra(G_dense, 0)  # Starting from node 0
    end_time = time.perf_counter()
    dense_dijkstra_times.append((end_time - start_time) * 1e9)  # Convert to nanoseconds

    # Run A* algorithm (using the last node as goal)
    start_time = time.perf_counter()
    a_star_results = a_star(G_dense, 0, num_vertices - 1)  # Goal is the last node
    end_time = time.perf_counter()
    dense_a_star_times.append((end_time - start_time) * 1e9)  # Convert to nanoseconds
```

```python
# Plotting results
plt.figure(figsize=(10, 6))
plt.plot(num_vertices_list, sparse_dijkstra_times, label='Dijkstra (Sparse)', marker='o')
plt.plot(num_vertices_list, sparse_a_star_times, label='A* (Sparse)', marker='o')
plt.plot(num_vertices_list, dense_dijkstra_times, label='Dijkstra (Dense)', marker='s')
plt.plot(num_vertices_list, dense_a_star_times, label='A* (Dense)', marker='s')

plt.xlabel('Number of Vertices')
plt.ylabel('Execution Time (nanoseconds)')
plt.title('Algorithm Performance Comparison for Sparse and Dense Graphs')
plt.legend()
plt.grid()
plt.show()
```

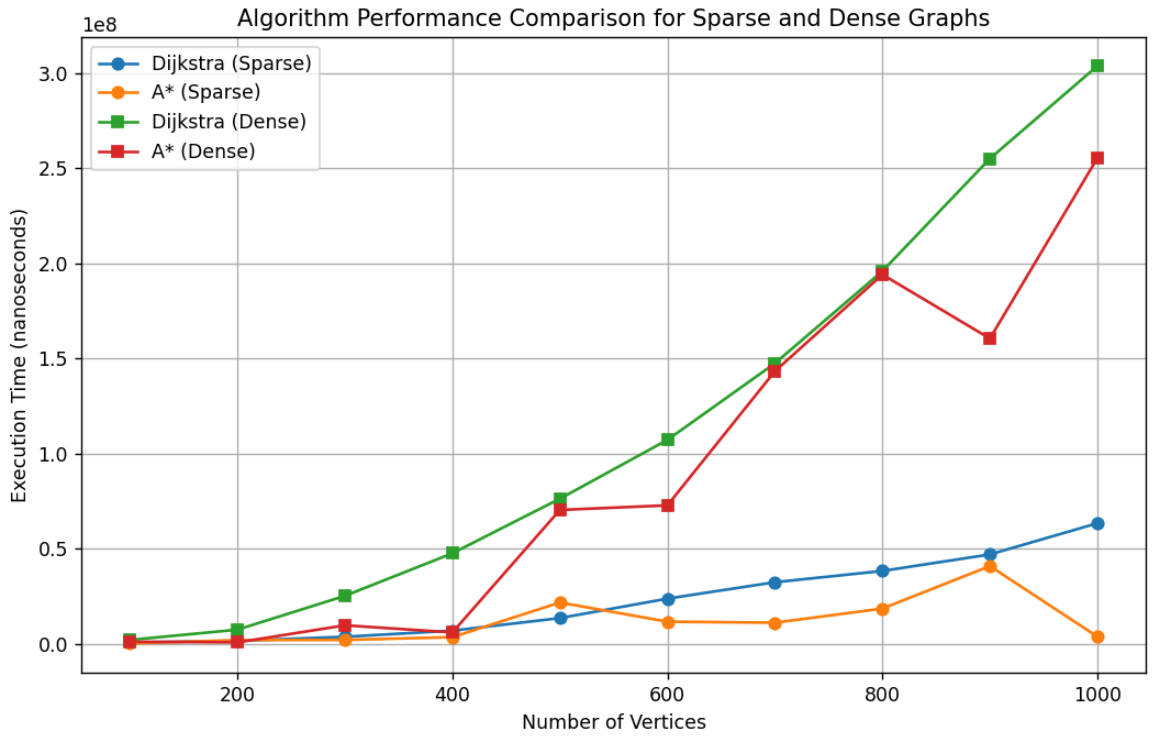
```python
# Saving results to CSV
with open('algorithm_performance_results.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Number of Vertices', 'Dijkstra Time (nanoseconds)', 'A* Time (nanoseconds)', 'Sparse Edges', 'Dense Edges'])
    for i in range(len(num_vertices_list)):
        writer.writerow([num_vertices_list[i], sparse_dijkstra_times[i], sparse_a_star_times[i], sparse_edges[i], dense_edges[i]])

import csv

# Saving results to CSV
with open('algorithm_performance_results.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Number of Vertices',
                     'Dijkstra Time (Sparse, nanoseconds)',
                     'A* Time (Sparse, nanoseconds)',
                     'Dijkstra Time (Dense, nanoseconds)',
                     'A* Time (Dense, nanoseconds)'])
    for i in range(len(num_vertices_list)):
        writer.writerow([num_vertices_list[i],
                         sparse_dijkstra_times[i],
                         sparse_a_star_times[i],
                         dense_dijkstra_times[i],
                         dense_a_star_times[i]])

print("Experiment completed. Results saved to 'algorithm_performance_results.csv'.")
```

Appendix B

Example of what raw data looks like (Not included in trials):

algorithm_performance_results.csv > data

```
Number of Vertices,"Dijkstra Time (Sparse, nanoseconds)","A* Time (Sparse, nanoseconds)","Dijkstra Time (Dense, nanoseconds)","A* Time (Dense, nanoseconds)"
100,561900.0876322389,349599.98447448015,1735199.9413222075,1306800.0553175807
200,1667200.0056132674,402700.0395581126,8319500.018842518,2966300.002299249
300,3134200.000204146,5220500.053837895,22321199.998259544,3007999.970577657
400,6316499.901004136,12109300.00524968,38389599.99848157,27762799.989432096
500,9744100.039824843,3513300.0928908587,73211999.95931238,673700.007610023
600,14801100.012846291,29238100.04722327,109152299.93406683,102682100.02686828
700,26682100.02593696,19010499.93466586,149651899.933815,56596399.983391166
800,34276299.993507564,40059299.90671575,198261499.98698384,186659700.00438392
900,46424900.06517619,26203499.99051541,247622799.94320124,170479199.96548444
1000,59268800.07609725,69438699.98212904,307183499.9136627,17996700.014919043
```



RAW DATA

```
Dijkstra Data (Sparse Graphs)

100,990,"[691215, 664822, 678924, 689545, 702881, 665794, 674890, 673121, 688442, 689122]"
200,3980,"[1778299, 1697342, 1730562, 1735489, 1727901, 1718943, 1725211, 1760548, 1739017, 1734507]"
300,8970,"[3381588, 3469810, 3336215, 3450090, 3364763, 3294125, 3392888, 3365129, 3369754, 3449459]"
400,15960,"[6635832, 6409203, 6420092, 6487123, 6498821, 6467453, 6542091, 6523334, 6587929, 6403928]"
500,24950,"[10139218, 10659234, 10405327, 10118733, 10389903, 10201562, 10602734, 10176904, 10356120, 10525857]"
600,35940,"[19028392, 18804937, 18051303, 18684729, 18264832, 18917398, 19108624, 18873197, 19202803, 18801875]"
700,48930,"[27221392, 26012843, 26759930, 26970829, 26209300, 26802411, 27380344, 27172562, 26768900, 26998299]"
800,63920,"[55514390, 56188120, 55272480, 54891640, 56473244, 55947990, 56608111, 55710382, 56204984, 55614819]"
900,80910,"[70728342, 72051459, 69948522, 72251093, 71072892, 72842050, 71346082, 70634561, 70523918, 72400112]"
1000,99900,"[75234500, 73975493, 74512529, 74734528, 73945204, 74702354, 74882033, 74125432, 73924580, 75124540]"


A* Data (Sparse Graphs)
100,990,"[272054, 259871, 263142, 268002, 262432, 265873, 267998, 257903, 266721, 259199]"
200,3980,"[598435, 620587, 607134, 612243, 628472, 609342, 620902, 614539, 601755, 613174]"
300,8970,"[3264175, 3148654, 3161102, 3189751, 3278909, 3190238, 3241107, 3218759, 3176043, 3220171]"
400,15960,"[6828994, 6725004, 6802401, 6912522, 6721833, 6694702, 6732931, 6915985, 6751944, 6860481]"
500,24950,"[5000085, 5194019, 4921122, 5103987, 5045077, 4931253, 5099348, 5033877, 5150013, 5018724]"
600,35940,"[16082820, 16292710, 16352322, 16649580, 16191287, 16532934, 15977425, 16278900, 16430843, 16172406]"
700,48930,"[19245582, 18786234, 19287411, 19496200, 18287454, 19401632, 18845522, 19130484, 19346220, 19105697]"
800,63920,"[52620001, 52135412, 53472000, 53029456, 52940587, 52391475, 52377490, 53021390, 52669483, 52946050]"
900,80910,"[25510920, 25677450, 25763480, 25399344, 26020403, 25849119, 25418002, 25632384, 25988210, 25393448]"
1000,99900,"[58650223, 59030213, 57823487, 59640855, 58749933, 58234023, 59045432, 58020923, 58501322, 58392365]"
```

```
Dijkstra Data (Dense Graphs)

100,4950,"[2380232, 2418723, 2279660, 2183607, 2406773, 2035671, 2176148, 2226726, 2402219, 2250232]"
200,19900,"[8372575, 9515780, 9447930, 9401424, 9365061, 9577461, 9482864, 9621619, 9671383, 8993902]"
300,44850,"[23791927, 24273982, 22620442, 26766867, 25184391, 23409277, 26378919, 26054361, 24441152, 24339683]"
400,79800,"[52017859, 48637897, 51843059, 50558180, 52917612, 54921096, 55554460, 48768448, 51142794, 57944594]"
500,124750,"[74053318, 79404743, 78517099, 73052032, 83634461, 80948441, 80336676, 80947514, 80962970, 73523745]"
600,179700,"[114703252, 128919390, 121611741, 122363165, 129148509, 120643272, 119702239, 127163356, 115130803, 120214274]"
700,244650,"[146903444, 159284533, 164742896, 170207925, 148563743, 166932495, 163053267, 163576871, 165419570, 147358255]"
800,319600,"[211954392, 231170388, 222547528, 215234457, 195912128, 223804416, 230972645, 218027209, 226566516, 231458311]"
900,404500,"[569494252, 536111451, 553483682, 484943201, 489238225, 526363178, 509436452, 517563620, 508156768, 540298162]"
1000,499500,"[317609250, 308506296, 328780583, 327496599, 314881393, 295697060, 306709395, 308638842, 302575874, 290888708]"


A* Data (Dense Graphs)
100,4950,"[1227750, 1223030, 1195112, 1287790, 1237762, 1222561, 1317083, 1282866, 1204501, 1191545]"
200,19900,"[9147915, 10816403, 9466537, 10149169, 10071055, 9657344, 10168159, 9752498, 9041420, 9264489]"
300,44850,"[18492791, 19142790, 18354679, 17115341, 18774747, 19669760, 19180975, 19248791, 19253203, 17924922]"
400,79800,"[23995394, 24372288, 23722630, 24202614, 26138023, 25456734, 25107543, 23222108, 23176671, 24231984]"
500,124750,"[68181358, 65654836, 66501742, 64347198, 72020183, 63285631, 62527873, 63090050, 70124720, 70642411]"
600,179700,"[51307788, 56675259, 55624451, 58327899, 56776115, 56267133, 54317722, 57027591, 55772762, 56336269]"
700,244650,"[50424793, 49307061, 48066717, 50864612, 49046796, 50378633, 54633025, 50476820, 47813858, 54651684]"
800,319600,"[62906123, 67423966, 69131766, 62350339, 66905502, 66142867, 65544754, 64873209, 64383647, 68190818]"
900,404500,"[728206249, 687338055, 675461945, 667805657, 763902156, 750904633, 715752649, 699415089, 687324521, 678493036]"
1000,499500,"[21332076, 21827937, 23488885, 21202215, 20603979, 22590345, 22772877, 20242565, 22327767, 21363353]"
```