Computer Science Extended Essay

Investigating the efficiency of pathfinding algorithms for solving the shortest path problem based on a geographical map.

Research question:

How far does Dijkstra's search algorithm compare to A* search algorithm for finding the shortest path in a graph as the size of the graph increases by number of vertices?

CS EE World
https://cseeworld.wixsite.com/home
May 2021
22/34
B
Submitter Info: Discord: sid_#6681

Essay word count: 3994

Table of Contents

1) Introduction

Pathfinding is an important aspect of Computer Science, where it's a solution that aims to solve the shortest path problem in a graph data structure. Pathfinding is the plotting of the shortest route between two points. This has various **applications** in fields such as **Artificial Intelligence, Network Theory and Game Development.** For instance, pathfinding can be used in mapping applications like **Google Maps** to find the **best route between origin and destination** or it can used to send packets across a computer network by finding the shortest route between start and end.

1.1) Personal interest in this essay

I'm interested in this particular aspect because during my **Personal project journey** at Middle Years Programme, I **developed a game**, which utilized artificial intelligence. However, the AI in my **game** wasn't complex because it **didn't have proper pathfinding** and I wasn't able to implement pathfinding because of my limited understanding then. Therefore, I'm **using this essay to develop my understanding of pathfinding**.

1.2) Aims of this essay

Moreover, I explored this essay by **investigating how the algorithms perform (efficiency) when finding routes on maps**. For example, finding the shortest distance between two cities. This topic also links to IB **Computer Science as it utilized abstract data structures and computational thinking**. Moreover, I applied my knowledge from my course to this essay for instance I used **lists and queues** in my implementation which we learnt in IB Computer Science.

1.3) The need for research in pathfinding algorithms

This essay explored the efficiency of the algorithms behind pathfinding by looking at certain algorithms which are the **Dijkstra's algorithm** and **A* algorithm.** The algorithms

were implemented because I want to investigate these particular algorithms in a practical environment. Consequently, the research question is: *How far does Dijkstra's search algorithm compare to A* search algorithm for finding the shortest path in a graph as the size of the graph increases by number of vertices?*

1.4) Need for efficient algorithms and algorithmic complexity

This essay **investigated the relationship between time taken for an entire operation with respect to the increase in size of the relevant dataset.** There is a need for this research and an important part of algorithms is efficiency. Algorithms are solved by programmers and understanding the efficiency of algorithms is important in programming as it allows for growth. Since programmers think in terms of maintaining code in the long term. Therefore, **creating efficient algorithms is about decreasing the amount of recursive operations** that is required to complete the task with respect to the size of dataset.[1] Furthermore, **this implicates to pathfinding as it's important to quickly calculate the shortest path with larger datasets**.

1.5) Use of pathfinding algorithms in satellite navigation and others

In the real world, pathfinding algorithms have various uses. For instance, take the internet and a pathfinding algorithm could be **used in finding the shortest path between a server and a node.** Another real-world application is in **satellite navigation**, in which pathfinding algorithms could be used to calculate the shortest route from current location to the desired destination. Google implements a **pathfinding in some form by turning a geographical map into a mathematical graph and using Dijkstra's algorithm using map data to find the shortest distance between two places**.[2]

[1] Choudry, Humzah., 2017. *Understanding Algorithm Efficiency And Why It'S Important*. [online] Medium.

[2] Lanning, D. R., Harrell, G. K., & Wang, J. (2014). *Dijkstra's algorithm and Google maps.*

1.6) Primary research being used

For the research I collated **data** from **primarily** from my own **experiment**[3] because it would lead to authentic results. The sources for my primary research was a **subject expert** who is a software engineer with 15 years of experience with algorithms in general. The subject expert served as guidance[4] and consultation for the direction of my research. Additionally, I gathered data from a **survey** to establish the need and verify it.[5]

1.7) Secondary research in the essay

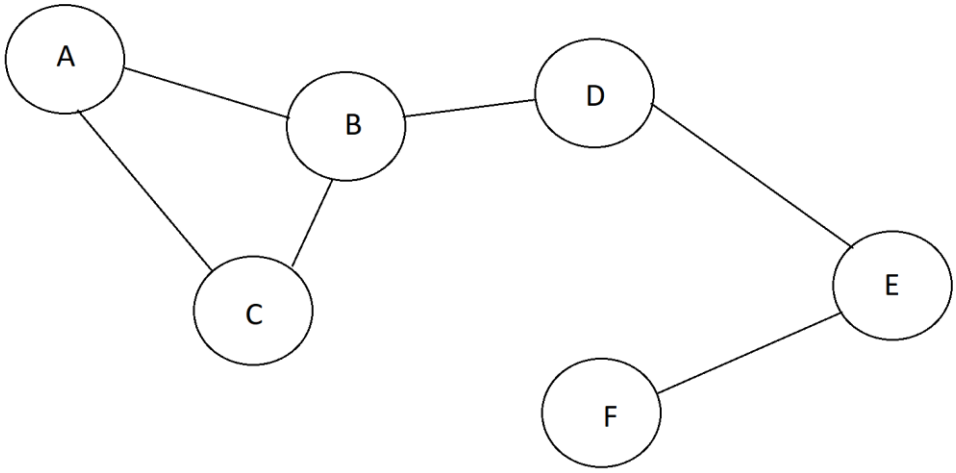
The **secondary research** which is a range of sources includes **books, research articles, lecture notes** and **web articles** was there to help my understanding and give direction to my experiment. Additionally, **completing a course on graph theory** to build my foundational understanding.[6] With these many sources, I **minimized bias** in my research, and it gave a balanced review.

2) Research findings and analysis of both algorithms

2.1) Graph Theory – Theory behind route planning in satellite navigation[7][8]

A **graph** G = {V, E} is a data structure which is a set of points called **vertices** (V) (vertices are also known as **nodes**) which are connected by a set of lines called **edges** (E). In essence it is a set of objects (vertices) where there are relationships between pairs of objects, for instance a graph of road networks could represent cities (as nodes) and show the roads/highways between cities (as edges). Example of a graph is shown below:

[3] See section 5.1 and appendix D
[4] See appendix F
[5] See appendix G
[6] Refer to appendix H
[7] Kulikov, Alexander S. “Introduction to Graph Theory.” University of California San Diego, National Research University Higher School of Economics.
[8] Quinn, Catherine, et al. *Mathematics for the international student: mathematics HL (option)*



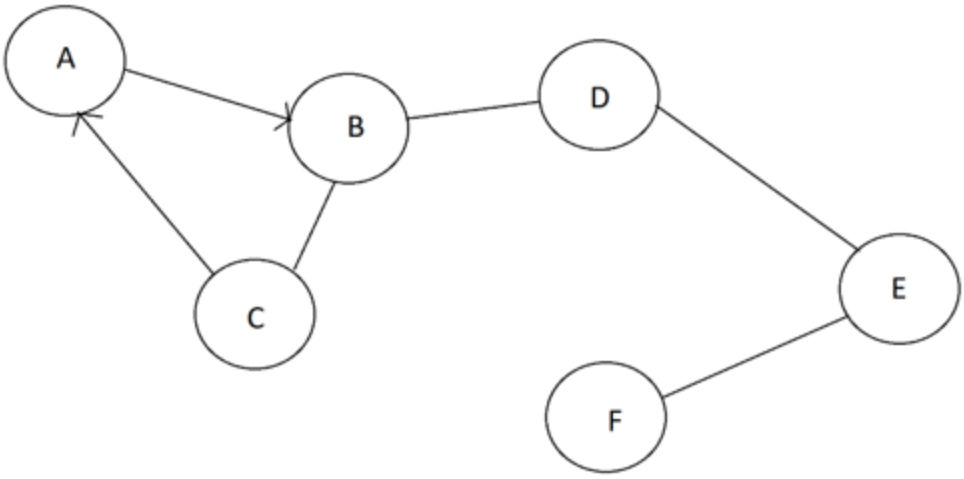
This is a graph that I made, where the circles (A, B, C, D, E, F) are the vertices and the lines that connect them are the edges. This shows how the objects relate to each other. For example, node A is directly connected (**adjacent**) to node B and node C but there is no direct connection with node F.

The edges can be **weighted**, meaning it can be assigned a value, for example the length of a road. The graph above shows an unweighted graph.

Definitions:[9]

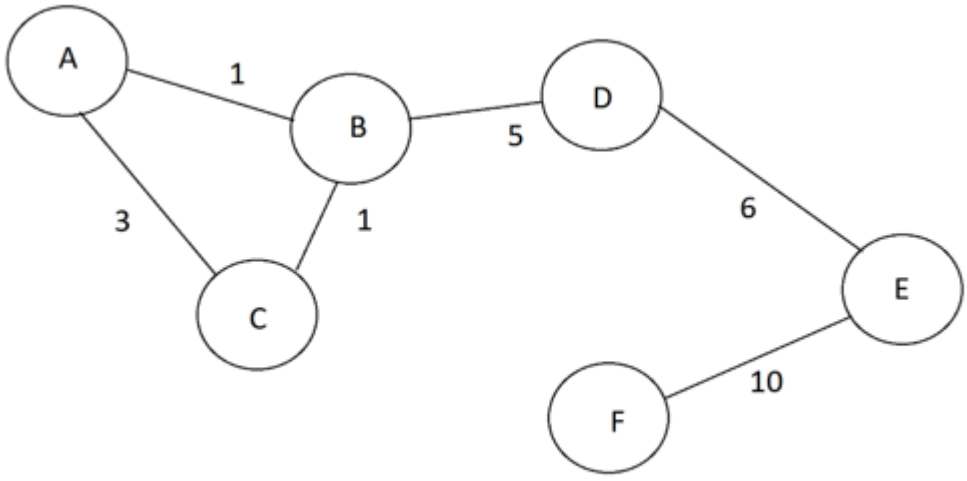
Path: A path is a **series** of **distinct edges** such that each edge (other than initial edge) starts with a vertex where the prior edge ended. A **simple path** is when all vertices are distinct.

[9] Appendix A for additional definitions



(Self-made) This is a directed graph, where the incident edge of node B adjacent to node A is directed as it shows a direction arrow, where you can traverse from A to B but not the other way. Similarly, you can traverse from C to A but not from A to C. The incident edge on node B and D are said to be undirected and are bidirectional.

Weight: The weight is a number that's associated with an edge. The quantity can represent anything such as distance, cost and etc. The weight of a path is sum of all edge weights from start vertex to end vertex. The shortest path is path with minimum weight between two vertices.



This is a weighted graph, with value assigned next to their respective edges

2.1.1) Implementation of my data set (graph)

For my experiment I created graphs using adjacency tables and implemented the graphs in the code and calculated heuristics.[10] This served as the dataset for my experiment. I trained the data to verify if everything in the graph is correct.

2.2) Shortest Path Problem – Theory behind route planning in satellite navigation[11]

The shortest path problem in graph theory is finding a path between two vertices such that the weight of the path is minimized. As said earlier the shortest path is the path where the sum of all edges between the two vertices are minimum. To find the shortest path, a search algorithm needs to be implemented, there are two kinds of search algorithm. Uninformed search is when the program has no information about cost from start node to goal node. An example of uninformed search algorithm is Dijkstra's algorithm created by Edsger Dijkstra in 1959. Informed search also known as heuristics search are when algorithms use heuristics (estimate) using given information in order to find the shortest path.

2.3) Algorithm Analysis and complexity

When analyzing algorithms, an important aspect that is to be considered is the computational complexity. Computational complexity has two parts, **time complexity** and space complexity. Time complexity is a measure of runtime as the input increases and space complexity is measure of memory usage as the input increases.[12] This essay will only focus on time complexity. Each function in programming has a runtime. Time complexity determines how the runtime of the function grows as the number of elements in that function grows. For example, if an 1D array is given and a function related to that

[10] See appendix A
[11] Russell, Stuart J, and Peter Norvig. *Artificial intelligence: a modern approach*.
[12] Sharma, Akash. "Time and Space Complexity Tutorials & Notes | Basic Programming." *HackerEarth*

array is given. Thus, the time complexity is the runtime of the function as the number of elements in that array increases.

2.3.1) Need for analyzing algorithms

The most important reason for analyzing algorithms is to find characteristics in order to evaluate the suitability for many applications or in order to compare to a similar algorithm used in the same program. The primary goal for analyzing algorithms is to accurately predict the performance of algorithms when implemented in programming in order to determine the resource usage, set parameters and compare similar algorithms. We use the analysis to build mathematical models to describe the performance of real-world implementations of algorithms.[13] This leads to us having an informed understanding of the particular algorithm and suggest informed improvements. In the real world, programmers have to deal with the limitations of resources such as memory and time, there is a need for efficient algorithms because programmers keep scalability in mind when implementing algorithms into their programs. For instance, in satellite navigation implementing pathfinding algorithms is increasing complex compared to a simple graph with a few nodes.

2.3.2) Big O notation and its relation to time complexity

The big O is a mathematical notation to express time complexity of a function in worst case. For example, if a function shows a linear relation between number of elements and runtime [f(n)=mn+c] then the big O notation for the time complexity is O(n). O(1) implies that there is no change in runtime as number of elements increases (constant time). The big O notation only uses terms with the highest degree ignoring any constants. For

[13] Sedgewick, Robert and Philippe Flajolet. *An introduction to the analysis of algorithms Second Edition.*

instance, if the function has a time complexity of $3n^2+100n+5$ then the time complexity in terms of big O is $O(n^2)$.[14]

2.4) Explanation and use of Dijkstra's algorithm in pathfinding

Dijkstra's search algorithm is one of the search algorithms that has applications in finding the shortest path in a graph. The general algorithm is[15][16][17]:

1. Let d(v) be the path cost of neighboring vertices *v* from starting node *s*.

2. d(v) = ∞ for a neighboring node as we don't know the value of it yet, moreover d(s) = 0 as it is the starting node.

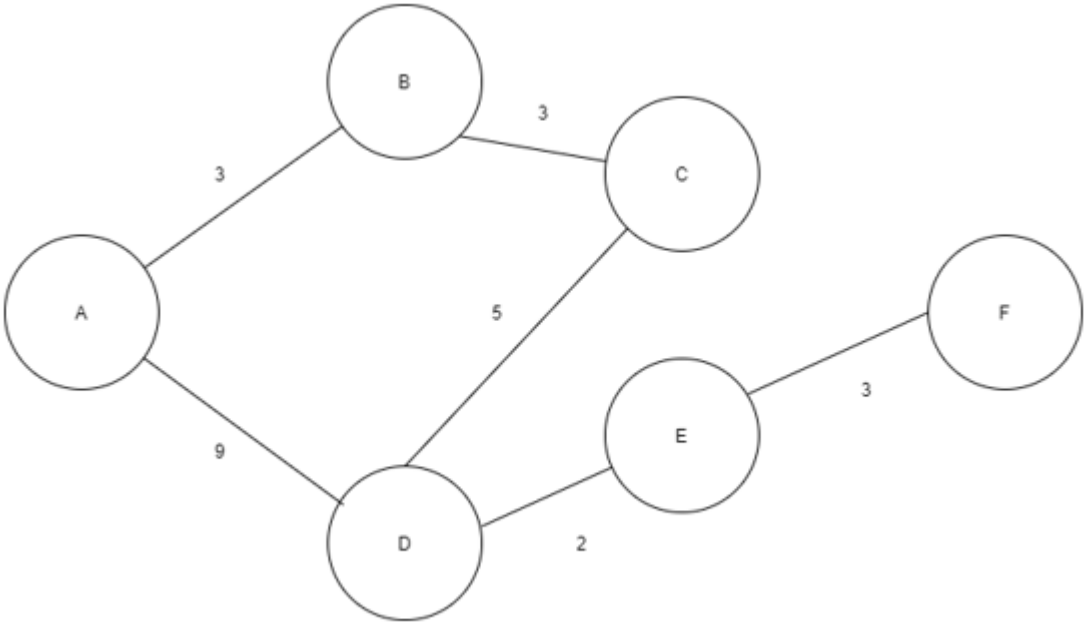
3. From the current vertex check the weight of neighboring nodes and calculate a temporary value of d(v). Compare all the values and assign the smallest value. For instance, if our current node is node N and with distance of 4 from starting node and the edge connecting its unvisited neighbor M has value of 2. Then d(v) will be 4+2 = 6. If M was marked with a value greater than 6 then assign it to 6.

4. After checking all the neighboring nodes from the current node, mark them as visited such that those vertices won't be checked again.

5. If the goal node has been marked visited or if the smallest d(v) = ∞, then end algorithm.

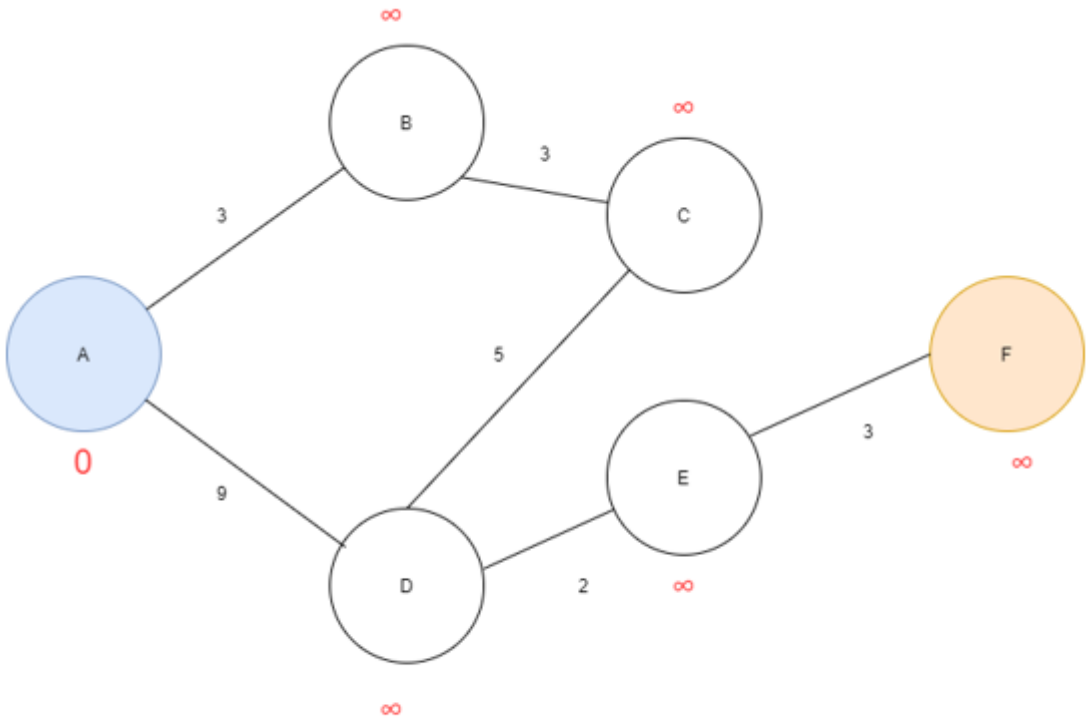
Explaining this with an example. Assuming a weighted graph with the starting vertex as A and let the goal, vertex be F:

[14] Adamchik, Victor S. *Algorithmic Complexity,* Carnegie Mellon University, 2009
[15] Cormen, Thomas H. et al. *Introduction to Algorithms*, 3rd ed.
[16] Pound, Mike. "Dijkstra's Algorithm - Computerphile."
[17] Quinn, Catherine, et al. *Mathematics for the international student: mathematics HL (option)*



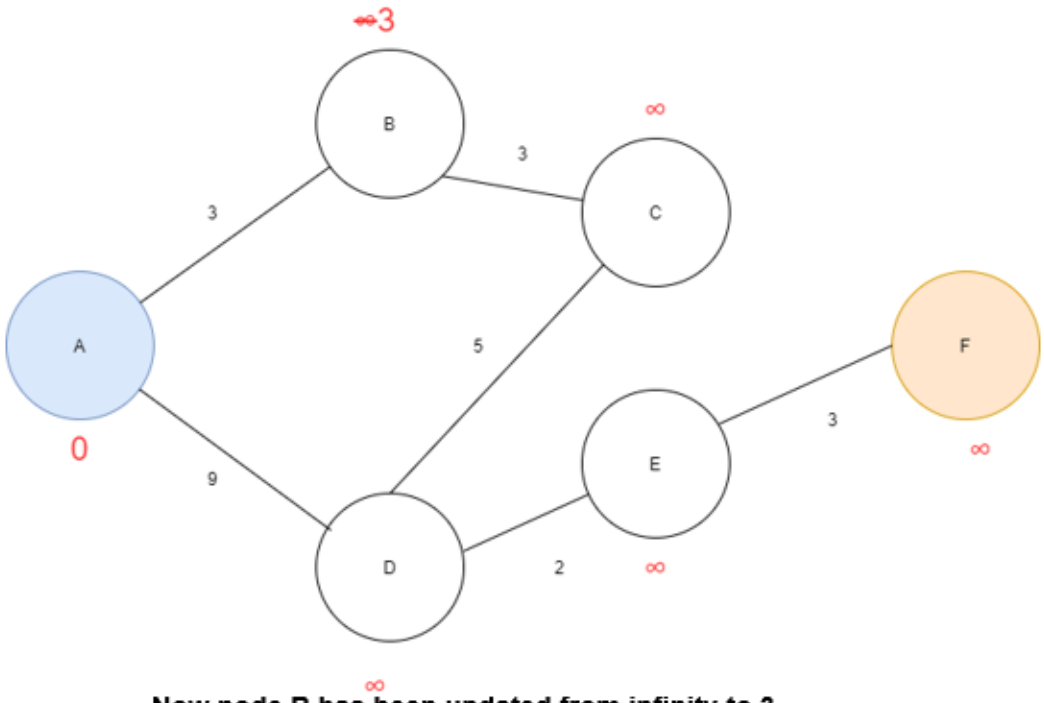
This graph with 6 nodes is weighted and undirected.

Set each node to infinity for the total distance as they haven't been visited yet. Initialize the starting distance from start node as 0.



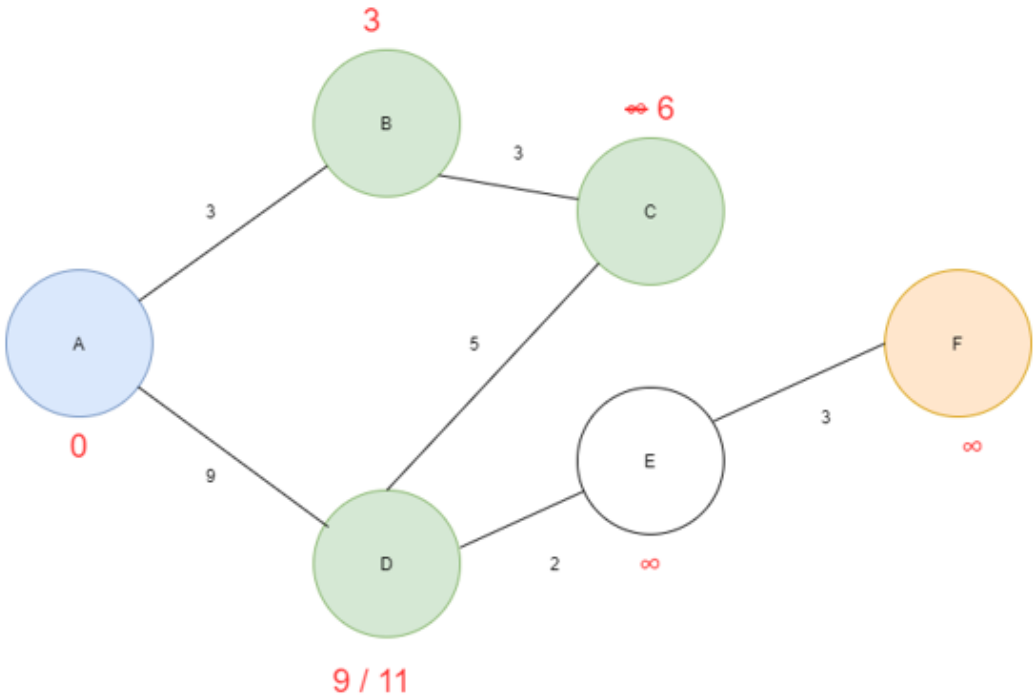
Node A has been marked as blue for start and node F has been marked yellow for end. The numbers in red signify the value of the node that is cumulative from its path. For example, A doesn't have any previous connections thus it will be 0.

With the current node check its adjacent nodes, if the current value of node (it will be infinity at first) is higher than value at current node plus the value of the connecting edge, then update the value of adjacent with current plus weight of the connecting edge and add neighbour with the minimum value to become part of shortest distance list.



Now node B has been updated from infinity to 3.

~~9~~ **is A -> B as 3 > 9.**

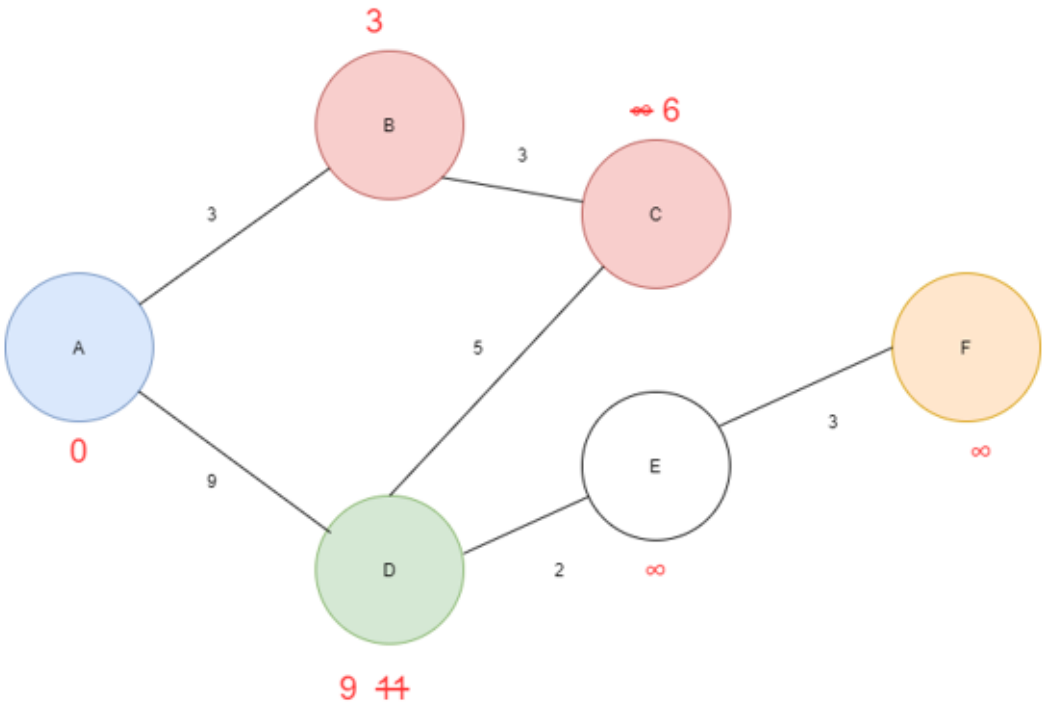


9 / 11

Node D has two paths A -> B -> C -> D at 11 cost and A -> D at 9 cost.

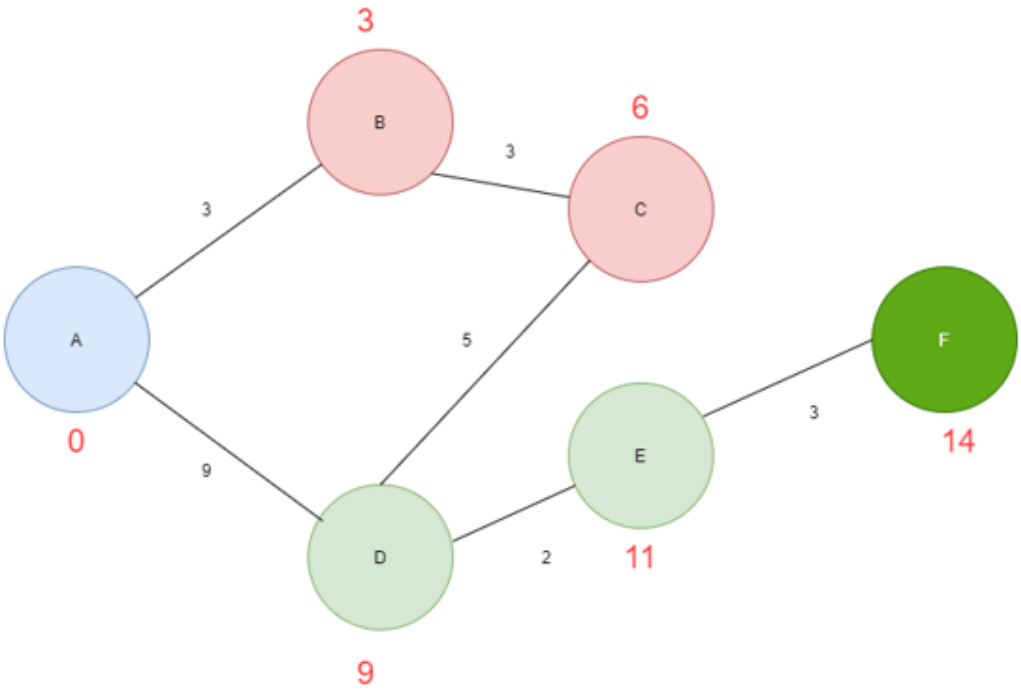
In the current node choose the neighbour with the minimum distance and set it as current node.

Keep iterating this process until all nodes have been marked visited.



The algorithm has selected the path with the lower cost (A -> D) after iterating and updated the cost of D to 9.

After all nodes have been visited and the target has been reached, we will get the shortest distance from A to F. The shortest distance here is 14.



The process keeps repeating and all the nodes are known. The algorithm finds Node F as highlighted by dark green and completes the program.

Below is a pseudocode[18] of the Dijkstra search algorithm using this pseudocode and the information above I was able to implement this algorithm in python 3.8.[19]

```
 1.  dist[S] ← 0                                          // The distance to source
    vertex is set to 0
 2.
 3.  Π[S] ← NIL                                           // The predecessor of source
    vertex is set as NIL
 4.
 5.  for all v ∈ V - {S}                                  // For all other vertices
 6.
 7.          do dist[v] ← ∞                               // All other distances are set
    to ∞
 8.
 9.          Π[v] ← NIL                                   // The predecessor of all other
    vertices is set as NIL
10.
11.  S ← Ø                                                // The set of vertices that
    have been visited 'S' is initially empty
12.
13.  Q ← V                                                // The queue 'Q' initially
    contains all the vertices
14.
15.  while Q ≠ Ø                                          // While loop executes till the
    queue is not empty
16.
17.          do u ← mindistance (Q, dist)                 // A vertex from Q with the
    least distance is selected
18.
19.          S ← S ∪ {u}                                  // Vertex 'u' is added to 'S'
    list of vertices that have been visited
20.
21.  for all v ∈ neighbors[u]                             // For all the neighboring
    vertices of vertex 'u'
22.
23.          do if dist[v] > dist[u] + w(u,v)              // if any new shortest path is
    discovered
24.
25.                  then dist[v] ← dist[u] + w(u,v)       // The new value of the
    shortest path is selected
26.
27.  return dist
```

2.5) Explanation and use of A* algorithm in pathfinding

A* algorithm works on the same principles as Dijkstra's algorithm.[20] Except it utilizes heuristics to make decisions when choosing a path.[21] Therefore A* search is considered to be an informed search algorithm. Although it tends to use more memory per node

[18] "Dijkstra Algorithm: Example: Time Complexity." *Gate Vidyalay*, 2020
[19] See Appendix B which has comments that explain each part of the code
[20] Pound, Mike. "A* (A Star) Search Algorithm - Computerphile."
[21] Russell, Stuart J, and Peter Norvig. *Artificial intelligence: a modern approach*.

compared to Dijkstra's. In A* we assign f(n) a cost evaluation function to all the nodes, thus[22]:

$$f(n) = g(n) + h(n)$$

Where f(n) is the total cost of the neighboring node, g(n) is the actual cost to travel from current node to the adjacent node and h(n) is a heuristics value from that node to the goal node. So, referring to graph in the previous section going from A to B or A to D will cost the weight of the connecting edge plus some assigned heuristics. This would lead to the algorithm going down path (A->D->E->F) first before exploring (A->B->C->D) because the heuristic value for B would likely be high. This will lead the algorithm to find the path faster. For my experimentation I borrowed the code with permission for the A* search algorithm.[23]

2.5.1) Heuristics in A* algorithm

The heuristics value is mainly **arbitrary and dependent on context**. For instance, if the problem was based on geographical map then a **heuristic could be calculated by finding the straight-line between two points**. A heuristic is considered admissible when the estimated heuristics cost is never higher than the actual cost from the current node to the goal node. Moreover, if heuristics always underestimate, then A* is guaranteed to find a solution.[24]

3) Conducting the experiment

3.1) Aim of experiment

This experiment aimed to **investigate the efficiency of Dijkstra's search algorithm and A* search algorithm in terms of time complexity by finding the shortest path**

[22] Hart, P., Nilsson, N., & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.*
[23] Refer to appendix C
[24] Kask, Kalev. *Lecture 4: Optimal and heuristic search, 2016*

between real life cities in a geographical region. The cities were represented as an undirected weighted graph with the weight representing the quantity of distance. The algorithms were executed in repeated trials and it complete the solution and return the shortest path between two cities in a given time.

3.2) Hypothesis

My first hypothesis for Dijkstra's algorithm was as the number of nodes in graph increases then the time taken for executing the algorithm will increase by polynomial order of 2 because the theoretical time complexity for Dijkstra using a simple list or array is $O(n^2 log(n))$ where n is the number of nodes . My **hypothesis for A* search algorithm was as the number of nodes in graph increases then the time taken would increase however it would increase at a slower rate compared to Dijkstra's because the theoretical time complexity which is dependent on optimal heuristics will be O(log(n)) meaning it would be linear.**

4) Experiment Methodology

4.1) Independent variable

In the experiment for both algorithms the independent variable that was being changed to produce a result was the number of nodes (vertices) in graph. I chose to use real life cities to be represented as nodes in a graph data structure. Moreover, I **used a weighted graph** where the edge weight was representing the distances between the cities in kilometers. After repeated executions of the algorithms, I would **adjust the graph** by adding cities in increments of one.

4.2) Dependent variable

The dependent variable that was measured is the **time taken for the particular algorithm to run in nanoseconds**. I chose this particular unit of time because modern computers

are extremely fast at making calculations, so to observe the changes I needed to use a precise unit of time. This was done to operationalize to see the time complexity of the algorithms empirically.

4.3) Control Variables

Variable	Description	Justification
Computer system	The computer system and the OS for each algorithm program remained same. Specs: - 8th gen i7 6 core processor - 16GB DDR4 3000Mhz RAM - Windows 10 OS ver. 10.0.18362	The system remained same for algorithm operation time to be not affected by hardware such as primary memory.
Same IDE (Integrated Development Environment)	The programs ran in the same IDE.	Using a different IDE for program might affect runtimes of the program
Same graph type	The graph characteristics remained same, meaning it'll be undirected, weighted, and finite graphs.	Features of the graph must remain same to give fairness to the algorithms.
Same start and end nodes	The graphs that will be traversed by the algorithms will have the same start and end node in every case (Berlin and Paris).	To keep consistency between graphs because changing the start and end node might affect times to calculate path.

4.4) Method

1. Implemented the Dijkstra's algorithm and the a* search algorithms in separate .py files.

2. Took two European cities, Berlin and Paris as the start and goal node respectively.

3. Using mapping software pinned the cities between Berlin and Paris and found distance from particular city to Paris (for heuristics values in A*) using longitudes and latitudes.[25]

4. Using the mapping software found the actual distances between cities and using this data created 20 different graphs[26] in excel as a list of pair of cities (one column is the node and the second column is the child node and third column is cost between cities).

5. Configured both algorithms for the particular graph, then ran each algorithm 10 times and recorded the execution timings each time it ran for both algorithms.

6. Repeated step 5 for all 20 graphs (from 4 nodes to 23 nodes that's 20 datapoints in total).

7. Collated data into excel and calculated averages and made graphs.

5) Findings of the experiment

5.1) Results

Table i. These are the results of mean execution time of **Dijkstra's algorithm** in nanoseconds against the number of vertices in graph.

No. of nodes	Average time
4	31000
5	32880
6	32220
7	33380
8	36540
9	37920
10	44470
11	44160
12	43930
13	57850
14	51310
15	51300
16	52160
17	103300
18	65540
19	71080
20	122610
21	164910
22	250790
23	346860

Table i. shows the number of nodes in the graph in the left column and the amount of time taken to complete search in the right column in nanoseconds. For example, in the 1st row it says the graph has 4 nodes and **Dijkstra's search** took **31000 nanoseconds.**

[25] Appendix A
[26] Refer to Appendix A

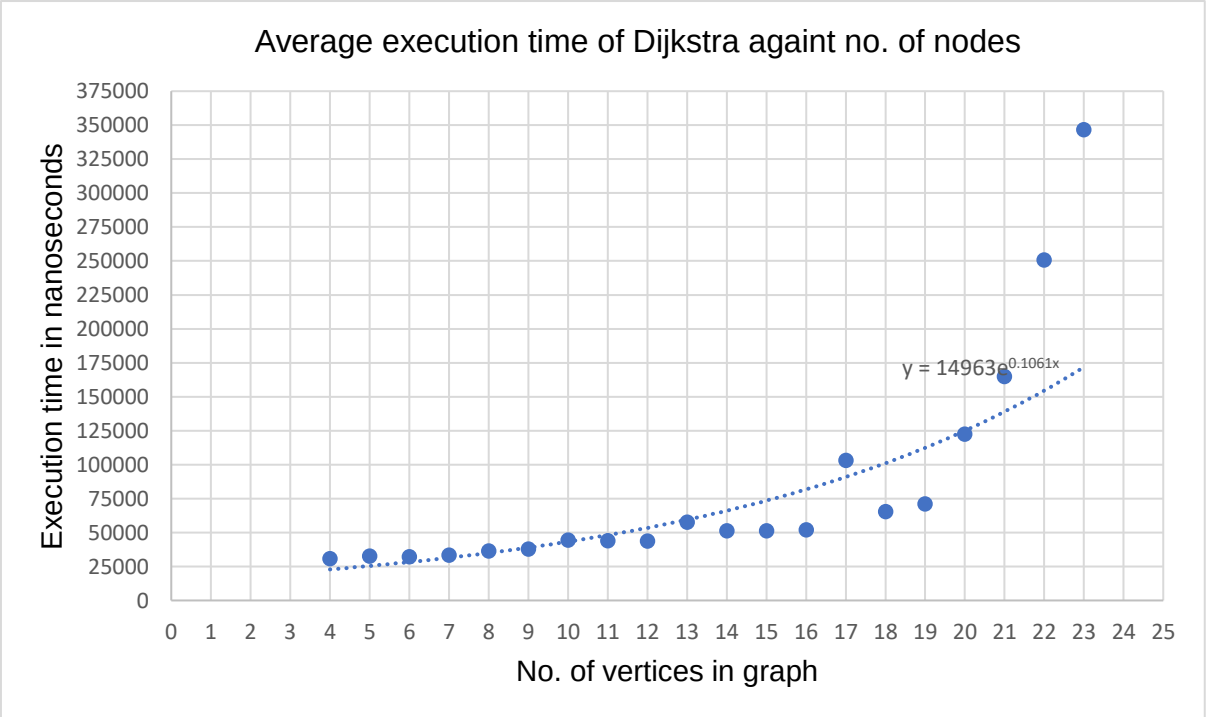
Table ii. These are the results of mean execution time of **A* algorithm** in nanoseconds against the number of vertices in graph.

No. of nodes	Average time
4	43980
5	47330
6	49750
7	42270
8	52240
9	56900
10	67360
11	80040
12	89450
13	89240
14	96820
15	103400
16	110410
17	75840
18	75940
19	168970
20	210560
21	205950
22	202690
23	215310

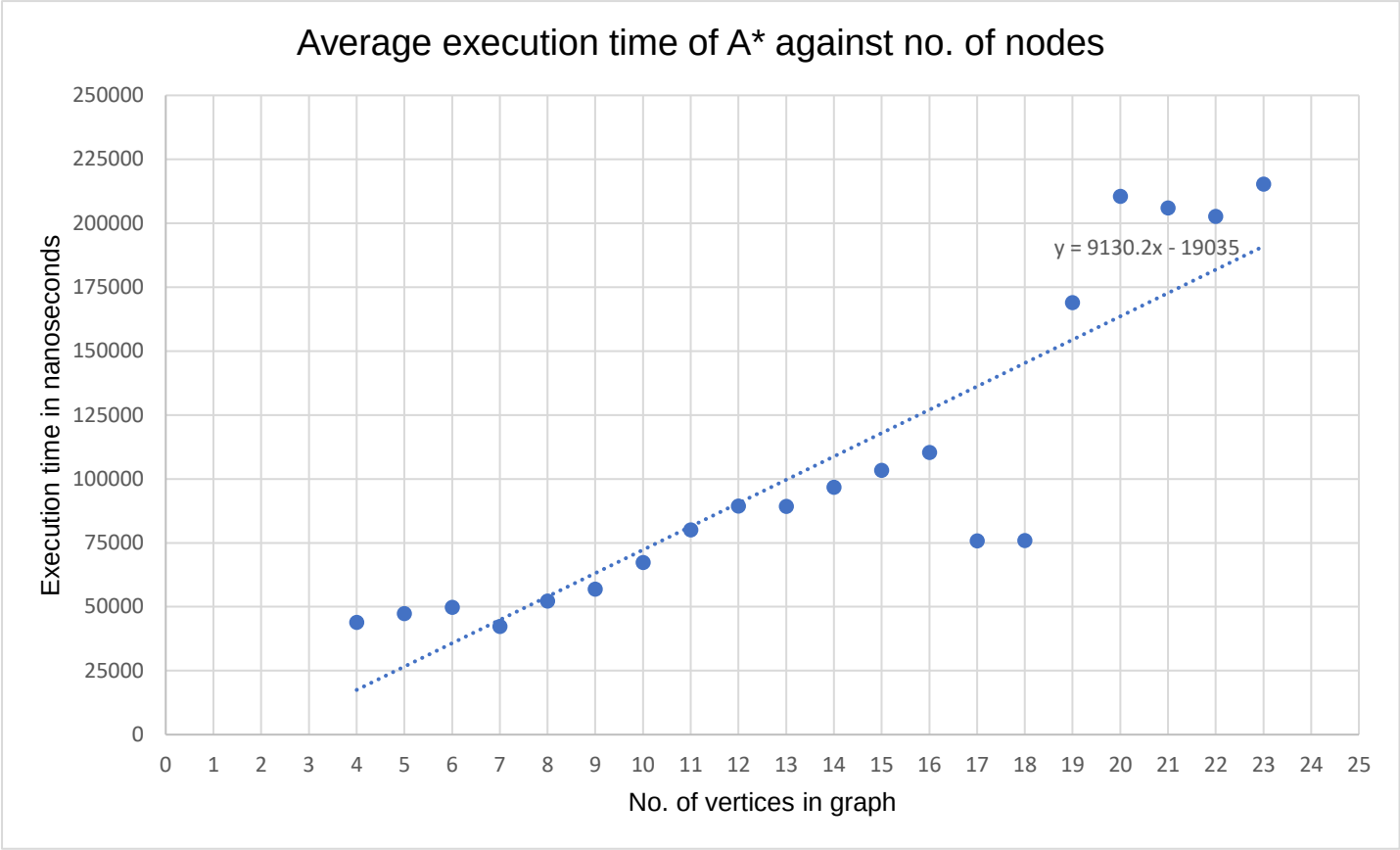
Table ii. shows the number of nodes in the graph in the left column and the amount of time taken to complete the search algorithm in the right column in nanoseconds. For example, in the 1st row it says the graph has **4 nodes** and **A* search** took **43980 nanoseconds** to complete.

5.2) Graphs of the results

Graph i. Line graph of **table i.**

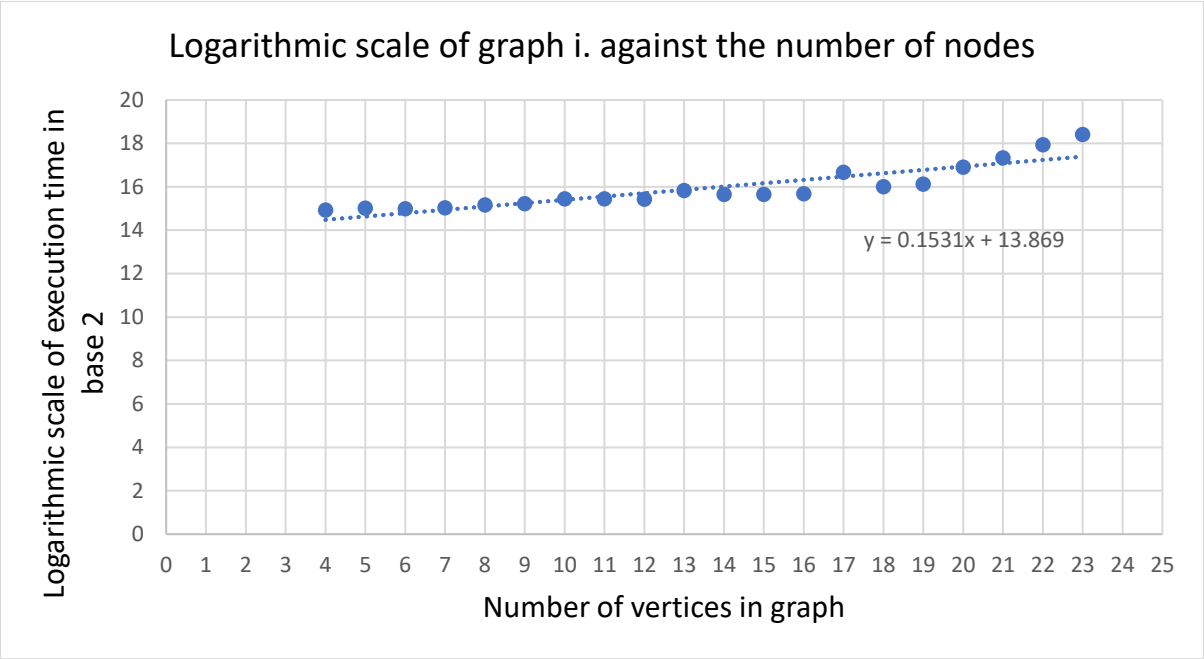


Graph ii. Line graph of **table ii**



6) Analyzing the experiment data

From the data above we can observe that the data from Dijkstra's algorithm shows a positive correlation between the execution timings and the no. of vertices in a graph, same with A* algorithm. Initially, from the data I can observe that the performance of A* is somewhat identical to Dijkstra's. However, after 19 nodes the difference in performance is observed and very noticeable and Dijkstra's algorithm search time is growing very quickly in comparison to A* algorithm. I believe that more data points would be needed to observe this quick change more precisely. I applied linear regression model for data in graph ii. to check if it fits the linear model. Moreover, I tested the **graph i**. for exponential growth. Additionally, put the values in y-axis on a logarithmic scale (in base 2):



7) Evaluating the experiment

7.1) Strengths of the experiments

A strength of this experiment was the **accuracy of the data** that is collected. As for each value of the independent variable, there were **10 repeated measurements** taken (trials). This helps with the accuracy of the data because the unit of measurements in nanoseconds, which is very small and therefore sensitive, so data would be less concise. Thus, it's balanced out by the repeated trials measurements. Another strength of this experiment was **the method of measurement** for the execution time, a built-in function called perf_counter_ns() was used. This is **more precise than using the system clock** to measure time and it returns integer values in nanoseconds.

7.2) Limitations of the experiments

I believe that there were limitations to my experiment were that in the first instance, I believe the **data wasn't large enough to show the full extent of the algorithms** and I should've used graphs with 100 or even 1000 nodes. But I wasn't able to do that because I didn't know how to implement and generate graphs this large. Another limitation, albeit a

minor one is that both algorithms don't have the same implementation of the graph data structure. In Dijkstra's algorithm an **adjacency list is used** to generate the graph, whereas in the A* algorithm the graph is created as an object from the **class graph**. This **shouldn't affect the results as much since the time measured is starting from when the search algorithm in the main method is executed** and not when the graph is generated however there should be a level of consistency.

7.3) Improvements to possible future experiments

To make improvements for further experimentation. I could start by learning how to **implement very large graphs** using databases, I could use a **map application programming interface** for my database and use it to automatically generate. Although it would be very challenging, it's feasible. The second improvement to experiment would be much easier and it would be to implement the algorithms using a **consistent data type like an adjacency list using a list data type.**

8) Conclusion

8.1) Summary of the essay

To conclude this essay, the overall aim of this experiment was to measure the efficiency of algorithms. Efficiency of algorithms is measured by analyzing algorithm complexities. This essay focused on time complexity as an aspect of algorithm complexities. **Time complexity is measure of an algorithm where it shows how the runtime increases with respect to the sample size of data that it's being applied on**. In this essay I choose **pathfinding algorithms as a basis for this** because they were complex and had many real-life applications. Pathfinding algorithms are a building block of machine learning and AI. Within this essay I explored pathfinding in relation to the shortest path problem Then I

recontextualized the shortest path problem in terms of navigation and mapping where in my experiment I found the shortest route between two real cities.

For my experiment I chose **Dijkstra's algorithm** which is one of the more popular one in pathfinding. Then I chose **A* algorithm** which is an extension to Dijkstra except it's a informed search which uses heuristics. In fact, as stated earlier Dijkstra is A* with heuristics value of 0 for all nodes. For my heuristics I took the straight-line distance between a node to the goal node. My overall hypothesis was that Dijkstra would increase at a faster rate in execution time as the size of the graph increases than A*.

8.2) Conclusion for the findings

The data from experiment shows them near identical for many graphs however **there is a noticeable difference for the final few datasets (graphs) and it shows Dijkstra quickly growing compared to A*.** Thus, to answer the research question there is no noticeable difference between the two algorithms in smaller graphs but A* is generally more efficient in terms of timing.

8.3) Way forward

This experiment should be **replicated for very large graphs** to observe the difference in more clarity. A future scope for this investigation could be **explore the facet of space complexity** which is another metric for algorithmic complexity which measures how much memory is used rather than execution time of an algorithm. Pathfinding is fundamental part of AI and it has potential to be utilized in many facets such as **machine learning, robotics, using AI for mathematical problems, computational biology and medicine**.

Appendix A – Graphs and implementation

These graphs serve as the datasets for my experiment

Additional definitions:

Adjacent: Adjacent means that when a node is connected to another node by an edge, it is said to adjacent to that particular node.

Degree: Degree of a vertex is the number of nodes connected through incident edges.

Direction: A graph is said to be directed when the edges have direction to them, to represent a one-way relationship. An undirected graph is when the edge incident to two adjacent vertices goes both ways.

Heuristics table of values (values are meant to be representative of kilometers):

For the heuristics in my experiment I calculated heuristics between Paris and a city using the straight-line distance using coordinates in terms longitude and latitude.

Nodes	Heuristics value
Berlin	881
Leipzig	768
Hamburg	745
Brunswick	692
Hanover	650
Kassel	578
Munster	509
Stuttgart	500
Frankfurt	480
Amsterdam	430
Cologne	405
Strasbourg	399
Eindhoven	364
Antwerp	300
Luxembourg	288
Nancy	282
Metz	281
Brussels	266
Namur	256
Verdun	225
Reims	130
Amiens	115
Paris	0

Graph 1 (4 nodes)

Current node	Child node	Cost (actual)
Berlin	Cologne	559
Berlin	Brussels	775
Cologne	Brussels	225
Cologne	Paris	495
Brussels	Paris	327

The table can interpret as showing adjacent nodes to a node and the cost (which is in kilometers). For example, Berlin is connected to Cologne by a cost of 559 and it is also other way around since it is an undirected graph.

Graph 2 (5 nodes)

Current node	Child node	Cost (actual)
Berlin	Leipzig	181
Berlin	Brussels	775
Leipzig	Cologne	497
Cologne	Brussels	225
Cologne	Paris	495
Brussels	Paris	327

Graph 3 (6)

Current node	Child node	Cost (actual)
Berlin	Leipzig	181
Berlin	Cologne	559
Leipzig	Cologne	497
Leipzig	Luxembourg	631
Cologne	Brussels	225
Cologne	Luxembourg	207
Brussels	Paris	327
Luxembourg	Paris	366

Graph 4 (7)

Current node	Child node	Cost (actual)
Berlin	Leipzig	181
Berlin	Cologne	559
Leipzig	Stuttgart	474
Leipzig	Cologne	497
Cologne	Luxembourg	207
Cologne	Brussels	225
Cologne	Stuttgart	373
Brussels	Paris	327
Luxembourg	Paris	366
Stuttgart	Paris	611

Graph 5 (8)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Leipzig	Cologne	497
Leipzig	Stuttgart	474
Cologne	Stuttgart	373
Cologne	Brussels	225
Cologne	Luxembourg	207
Stuttgart	Paris	611
Brussels	Paris	327
Luxembourg	Paris	366

Graph 6 (9)

Current node	Child node	Cost (actual)
Berlin	Leipzig	181
Berlin	Hanover	286
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Hanover	Cologne	293
Cologne	Brussels	225
Cologne	Luxembourg	207
Frankfurt	Luxembourg	239
Brussels	Paris	327
Luxembourg	Paris	366
Stuttgart	Paris	611

Graph 7 (10)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Cologne	Luxembourg	207
Cologne	Brussels	225
Frankfurt	Luxembourg	239
Amsterdam	Brussels	203
Stuttgart	Paris	611
Luxembourg	Paris	366
Brussels	Paris	327

Graph 8 (11)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Metz	285
Cologne	Luxembourg	207
Cologne	Brussels	225
Amsterdam	Brussels	203
Metz	Paris	330
Luxembourg	Paris	366
Brussels	Paris	327

Graph 9 (12)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Metz	285
Hanover	Amsterdam	375
Hanover	Cologne	293
Cologne	Luxembourg	207
Cologne	Brussels	225
Amsterdam	Brussels	203
Metz	Verdun	70
Luxembourg	Verdun	89
Verdun	Paris	260
Brussels	Paris	315

Graph 10 (13)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Metz	247
Frankfurt	Luxembourg	239
Luxembourg	Verdun	89
Metz	Verdun	70
Stuttgart	Strasbourg	147
Hanover	Cologne	293
Hanover	Amsterdam	375
Amsterdam	Brussels	203
Cologne	Brussels	225
Cologne	Luxembourg	207
Verdun	Paris	260
Strasbourg	Paris	491
Brussels	Paris	315

Graph 11 (14)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Amsterdam	Antwerp	158
Cologne	Brussels	225
Cologne	Luxembourg	207
Antwerp	Brussels	45
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Brussels	Paris	315

Graph 12 (15)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Eindhoven	364
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Eindhoven	Brussels	129
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Brussels	Paris	315

Graph 13 (16)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Hanover	Eindhoven	364
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Eindhoven	Brussels	129
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Brussels	Paris	315

Graph 14 (17)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Eindhoven	364
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Brussels	Paris	315
Namur	Paris	321

Graph 15 (18)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Cologne	293
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Eindhoven	364
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Amiens	Paris	142
Namur	Paris	321

Graph 16 (19)

Current node	Child node	Cost (actual)
Berlin	Hanover	286
Berlin	Leipzig	181
Hanover	Kassel	168
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Eindhoven	364
Kassel	Cologne	243
Kassel	Frankfurt	185
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Amiens	Paris	142
Namur	Paris	321

Graph 17 (20)

Current node	Child node	Cost (actual)
Berlin	Brusnwick	235
Berlin	Leipzig	181
Berlin	Hanover	286
Brusnwick	Kassel	154
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Eindhoven	364
Kassel	Cologne	243
Kassel	Frankfurt	185
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Amiens	Paris	142
Namur	Paris	321

Graph 18 (21)

Current node	Child node	Cost (actual)
Berlin	Brusnwick	235
Berlin	Leipzig	181
Berlin	Hanover	286
Brusnwick	Kassel	154
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Munster	193
Munster	Eindhoven	203
Kassel	Cologne	243
Kassel	Frankfurt	185
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Paris	491
Verdun	Paris	260
Amiens	Paris	142
Namur	Paris	321

Graph 19 (22)

Current node	Child node	Cost (actual)
Berlin	Brusnwick	235
Berlin	Leipzig	181
Berlin	Hanover	286
Brusnwick	Kassel	154
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Munster	193
Munster	Eindhoven	203
Kassel	Cologne	243
Kassel	Frankfurt	185
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Strasbourg	Nancy	149
Nancy	Paris	336
Verdun	Paris	260
Amiens	Paris	142
Namur	Paris	321

Graph 20 (23)

Current node	Child node	Cost (actual)
Berlin	Brusnwick	235
Berlin	Leipzig	181
Berlin	Hanover	286
Brusnwick	Kassel	154
Hanover	Amsterdam	375
Hanover	Hamburg	151
Hanover	Munster	193
Munster	Eindhoven	203
Kassel	Cologne	243
Kassel	Frankfurt	185
Amsterdam	Antwerp	158
Antwerp	Brussels	45
Cologne	Eindhoven	146
Cologne	Luxembourg	207
Cologne	Namur	184
Eindhoven	Brussels	129
Brussels	Amiens	229
Leipzig	Frankfurt	396
Leipzig	Stuttgart	474
Frankfurt	Luxembourg	239
Frankfurt	Metz	247
Stuttgart	Strasbourg	147
Metz	Verdun	70
Luxembourg	Verdun	89
Luxembourg	Reims	218
Namur	Reims	189
Strasbourg	Nancy	149
Nancy	Paris	336
Verdun	Paris	260
Amiens	Paris	142
Reims	Paris	144

Appendix B – Dijkstra's algorithm code

```python
from time import perf_counter_ns
from math import inf


def dijkstra_search(graph, src, target): #graph is our adj list, src is start node which is string and target is end

    unknown_V = graph #all nodes which haven't been visted yet so the entire graph at this point
    shortest_distance = {} #stores the weight of the shortest distance from one node to another
    path = [] #stores the path between src and target
    precursor = {} #stores the previous nodes of the current node

    for vertices in unknown_V:
        shortest_distance[vertices] = inf #initialize weight as infinite for all nodes

    shortest_distance[src] = 0 #distance to beginning is 0

    while (unknown_V):
        min_V = None #initialize the min value of nodes

        for current_V in unknown_V: #iterate through all unvisted vertices

            if min_V is None: #intially this loop will be called

                min_V = current_V #set the minimum value node as the current node only for initial condition

            #if the total path cost for current node is less than the value stored in min_V
            elif shortest_distance[min_V] > shortest_distance[current_V]:
                #if TRUE then update the min_V to current node
                min_V = current_V

        #iterating through all the neighbours of the current node
        for child_V, val in unknown_V[min_V].items():
            #checking if the value of current node + value of edge connecting to neighbor is less than
            #distance between current visited nodes and their connections
            if val + shortest_distance[min_V] < shortest_distance[child_V]:
                #if true then set the new value as minimum distance of that connection
                shortest_distance[child_V] = val + shortest_distance[min_V]
                #add the current node as the predecessor of the child node
                precursor[child_V] = min_V
        #after the node has been visited then remove it from the list of unvisted
        unknown_V.pop(min_V)

    V = target #when the shortest distance after iterating from start node to end node has been found,
    # set current node to the target node

    while V != src: #this while loop is the backtrack where we been from to get path

        try:  #a graph might not have a path so we enclose it in this try block
            path.insert(0, V)
            V = precursor[V]
        except Exception:
            print('Path not reachable')
            break

    path.insert(0, src) #include the start node in the path

    if shortest_distance[target] != inf: #if all nodes been visited print shortest distance value and the path
        print('Shortest distance is ' + str(shortest_distance[target]))
        print('The retraced path is ' + str(path))
```

```
def main():

    graph = {
        #this graph represents the neighbors and cost, for example Berlin is connected to Leipzig at a cost of 181
        'Berlin':{'Leipzig':181,'Hanover':286, 'Brunswick':235},
        'Brunswick':{'Berlin':235,'Kassel':154},
        'Hanover':{'Berlin':286,'Munster':193,'Amsterdam':375, 'Hamburg':151},
        'Hamburg':{'Hanover':151},
        'Munster':{'Hanover':193,'Eindhoven':203},
        'Kassel':{'Brunswick':154,'Cologne':243,'Frankfurt':185},
        'Leipzig':{'Berlin':181,'Frankfurt':396, 'Stuttgart':474},
        'Cologne':{'Kassel':243,'Eindhoven':146,'Luxembourg':207,'Namur':184},
        'Eindhoven':{'Munster':203,'Brussels':129},
        'Frankfurt':{'Kassel':185,'Leipzig':396, 'Luxembourg':239, 'Metz':247},
        'Stuttgart':{'Leipzig':474,'Strasbourg':147},
        'Strasbourg':{'Stuttgart':147,'Nancy':149},
        'Amsterdam':{'Hanover':375,'Antwerp':158},
        'Antwerp':{'Amsterdam':158,'Brussels':45},
        'Metz':{'Frankfurt':247,'Stuttgart':285,'Verdun':70},
        'Brussels':{'Antwerp':45,'Eindhoven':129,'Amiens':229},
        'Luxembourg':{'Cologne':207,'Verdun':89,'Reims':218},
        'Verdun':{'Luxembourg':89,'Metz':70,'Paris':260},
        'Namur':{'Cologne':184,'Reims':189},
        'Reims':{'Luxembourg':218,'Namur':189,'Paris':144},
        'Amiens':{'Brussels':229,'Paris':142},
        'Nancy':{'Strasbourg':149,'Paris':336},
        'Paris':{'Verdun':260,'Amiens':142,'Nancy':336,'Reims':144}
    }
    start = perf_counter_ns() #this is where we start measuring time
    dijkstra_search(graph, 'Berlin', 'Paris') #this is when the algo is executed
    end = perf_counter_ns() #this is where we measure the last instance of time
    print("--- %s nanoseconds ---" % (end - start))

if __name__ == "__main__": #calling the main method
    main()
```

Appendix C – A* algorithm code[27]

Note: I replaced the graphs and heuristics in the main() method with my own data.

[27] Code Taken from AnnyTab; refer to appendix E for permission of code

```python
from time import perf_counter_ns

class Graph:
    # Initialize the class
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    # Create an undirected graph by adding symmetric edges
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist

    # Add a link from A and B of given distance, and also add the inverse link if the graph is undirected
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    # Get neighbors or a neighbor
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

    # Return a list of nodes in the graph
    def nodes(self):
        s1 = set([k for k in self.graph_dict.keys()])
        s2 = set([k2 for v in self.graph_dict.values() for k2, v2 in v.items()])
        nodes = s1.union(s2)
        return list(nodes)


# This class represent a node
class Node:
    # Initialize the class
    def __init__(self, name: str, parent:str):
        self.name = name
        self.parent = parent
        self.g = 0  # Distance to start node
        self.h = 0  # Distance to goal node
        self.f = 0  # Total cost

    # Compare nodes
    def __eq__(self, other):
        return self.name == other.name

    # Sort nodes
    def __lt__(self, other):
        return self.f < other.f

    # Print node
    def __repr__(self):
        return ('({0},{1})'.format(self.name, self.f))


```

```python
# A* search
def astar_search(graph, heuristics, start, end):
    # Create lists for open nodes and closed nodes
    open = []
    closed = []
    # Create a start node and an goal node
    start_node = Node(start, None)
    goal_node = Node(end, None)
    # Add the start node
    open.append(start_node)

    # Loop until the open list is empty
    while len(open) > 0:
        # Sort the open list to get the node with the lowest cost first
        open.sort()
        # Get the node with the lowest cost
        current_node = open.pop(0)
        # Add the current node to the closed list
        closed.append(current_node)

        # Check if we have reached the goal, return the path
        if current_node == goal_node:
            path = []
            while current_node != start_node:
                path.append(current_node.name + ': ' + str(current_node.g))
                current_node = current_node.parent
            path.append(start_node.name + ': ' + str(start_node.g))
            # Return reversed path
            return path[::-1]
        # Get neighbours
        neighbors = graph.get(current_node.name)
        # Loop neighbors
        for key, value in neighbors.items(): #this should return true but returns -1
            # Create a neighbor node
            neighbor = Node(key, current_node)
            # Check if the neighbor is in the closed list
            if (neighbor in closed):
                continue
            # Calculate full path cost
            neighbor.g = current_node.g + graph.get(current_node.name, neighbor.name)
            neighbor.h = heuristics.get(neighbor.name)
            neighbor.f = neighbor.g + neighbor.h
            # Check if neighbor is in open list and if it has a lower f value
            if (add_to_open(open, neighbor) == True):
                # Everything is green, add neighbor to open list
                open.append(neighbor)
    # Return None, no path is found
    return None


# Check if a neighbor should be added to open list
def add_to_open(open, neighbor):
    for node in open:
        if (neighbor == node and neighbor.f > node.f):
            return False
    return True

```

```
# The main entry point for this module
def main():

    # Create a graph
    graph = Graph()
    # Create graph connections (Actual distance)
    graph.connect('Berlin', 'Brunswick', 235)
    graph.connect('Berlin','Hanover',286)
    graph.connect('Berlin','Leipzig',181)
    graph.connect('Brunswick','Kassel',154)
    graph.connect('Hanover','Amsterdam',375)
    graph.connect('Hanover','Hamburg',151)
    graph.connect('Hanover','Munster',193)
    graph.connect('Munster','Eindhoven',203)
    graph.connect('Kassel','Cologne',243)
    graph.connect('Kassel','Frankfurt',185)
    graph.connect('Amsterdam','Antwerp',158)
    graph.connect('Antwerp','Brussels',45)
    graph.connect('Cologne','Eindhoven',146)
    graph.connect('Cologne','Luxembourg',207)
    graph.connect('Cologne','Namur',184)
    graph.connect('Eindhoven','Brussels',129)
    graph.connect('Brussels','Amiens',229)
    graph.connect('Leipzig','Frankfurt',396)
    graph.connect('Leipzig','Stuttgart',474)
    graph.connect('Frankfurt','Luxembourg',239)
    graph.connect('Frankfurt','Metz',247)
    graph.connect('Stuttgart','Strasbourg',147)
    graph.connect('Metz','Verdun',70)
    graph.connect('Luxembourg','Verdun',89)
    graph.connect('Strasbourg','Nancy',149)
    graph.connect('Namur','Reims',189)
    graph.connect('Verdun','Paris',260)
    graph.connect('Amiens','Paris',142)
    graph.connect('Reims','Paris',144)
    graph.connect('Nancy','Paris',336)

    # Make graph undirected, create symmetric connections
    graph.make_undirected()
    # Create heuristics (straight-line distance, air-travel distance)
    heuristics = {}
    heuristics['Berlin'] = 881
    heuristics['Leipzig'] = 768
    heuristics['Hamburg'] = 745
    heuristics['Brunswick'] = 692
    heuristics['Hanover'] = 650
    heuristics['Kassel'] = 578
    heuristics['Munster'] = 509
    heuristics['Stuttgart'] = 500
    heuristics['Frankfurt'] = 480
    heuristics['Amsterdam'] = 430
    heuristics['Cologne'] = 405
    heuristics['Strasbourg'] = 399
    heuristics['Eindhoven'] = 364
    heuristics['Antwerp'] = 300
    heuristics['Luxembourg'] = 288
    heuristics['Nancy'] = 282
    heuristics['Metz'] = 281
    heuristics['Brussels'] = 266
    heuristics['Namur'] = 256
    heuristics['Verdun'] = 225
    heuristics['Reims'] = 130
```

```
    heuristics['Amiens'] = 115
    heuristics['Paris'] = 0


    # Run the search algorithm
    start = perf_counter_ns()
    path = astar_search(graph, heuristics, 'Berlin', 'Paris')
    end = perf_counter_ns()
    print(path)
    print("--- %s nanoseconds ---" % (end - start))
    print()
    #print("[Berlin: 0','Hanover: 286', 'Cologne: 579', 'Namur: 763', 'Paris: 1084']")
    #print("--- 76600 nanoseconds ---")


# Tell python to run main method
if __name__ == "__main__": main()
```

Appendix D – Raw data of experiment

Values in trials columns are in nanoseconds

Dijkstra's Algorithm:

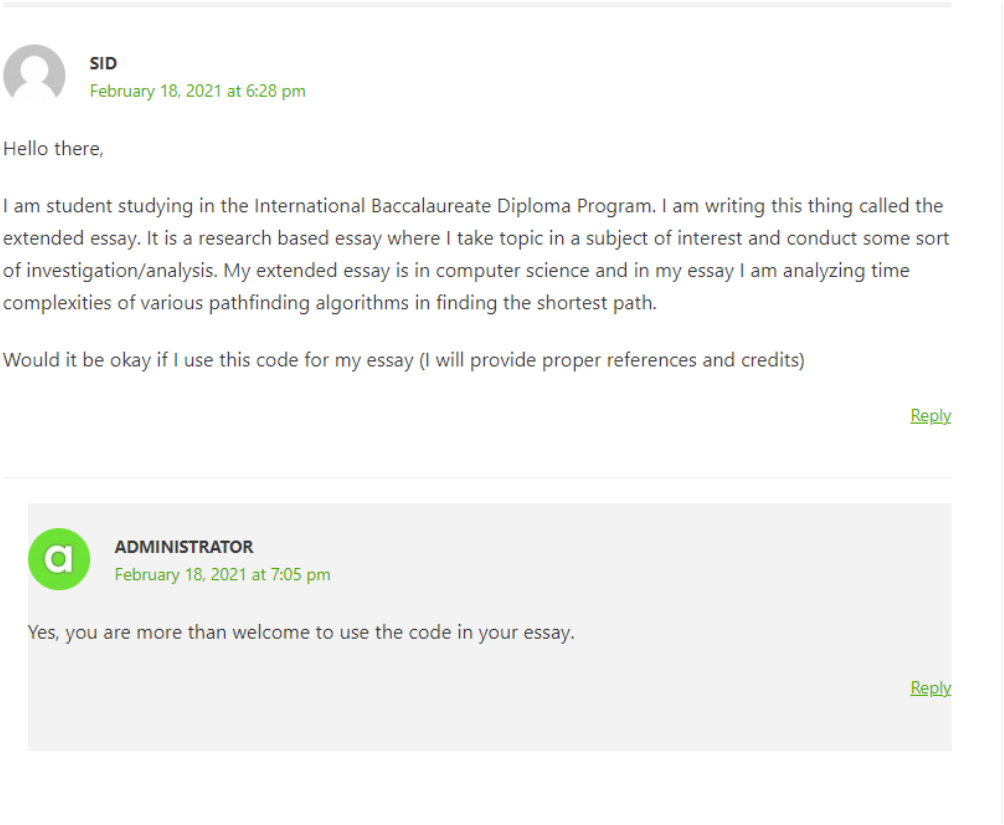
Vertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
4	48000	28900	27100	30000	28300	28800	28800	28400	32900	28800
5	32700	37700	32400	30000	33900	30000	30500	31200	33100	37300
6	31100	31900	31700	32600	34800	30300	31900	30600	32500	34800
7	33500	34300	31600	31700	34200	32900	33700	32000	34000	35900
8	34600	45100	36100	35900	36500	35700	35300	34900	36000	35300
9	38000	39300	35600	35800	37100	36000	36600	47500	36500	36800
10	39500	38700	38400	40100	40000	40900	38000	89800	40100	39200
11	54000	45100	41000	45000	41700	41100	40100	43400	43200	47000
12	43000	42600	43000	42900	43100	44200	44100	43900	44100	48400
13	47500	44900	47700	43600	46400	45600	46100	46600	51100	159000
14	46600	47900	47700	87000	46700	46800	47100	48900	46000	48400
15	49900	50300	48800	50500	51400	53400	51400	51200	57700	48400
16	52600	51200	52000	50900	52100	54700	52100	51500	53100	51400
17	99400	108200	110700	98500	97900	102600	104100	109500	99700	102400
18	68500	67200	56700	59900	68000	66400	66900	58400	80900	62500
19	58100	71100	83000	83200	62000	59500	83900	71900	62400	75700
20	104200	84900	95400	117700	125300	172100	107200	122200	172100	125000
21	174200	173800	171100	186400	105500	133400	174600	177900	177000	175200
22	208000	267000	280000	195000	281800	269400	281300	278000	250800	196600
23	402700	420700	311700	306400	305700	311600	321400	371900	411800	304700

A* algorithm:

Vertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
4	44500	43700	46200	43100	46500	44200	44600	41900	42300	42800
5	49200	46300	44500	47200	49300	47600	46500	48300	47600	46800
6	49400	50800	50000	50000	49500	49100	49900	49100	48600	51100
7	45900	46100	46300	47200	46300	45800	4700	45800	48800	45800
8	52200	51100	51700	52200	52600	52800	52300	52200	52800	52500
9	56000	56200	57600	56800	56200	57300	57200	56300	57000	58400
10	65600	68000	66300	66700	68600	67400	68100	68600	66500	67800
11	80000	79700	81000	78800	79900	79000	81800	81100	79400	79700
12	89000	90000	89500	89900	89200	89500	89400	88900	91000	88100
13	91300	85700	86100	86900	86100	85200	115500	84500	84800	86300
14	97400	96500	96400	98000	96700	94400	100200	95900	96400	96300
15	104100	103500	102800	102700	102100	104800	102900	103100	105200	102800
16	111300	100100	108600	112300	112300	111600	114100	111300	112200	110300
17	76200	76300	75400	77100	74600	76300	75800	74700	75400	76600
18	76300	73800	75500	77800	77200	75100	76300	74500	76500	76400
19	179200	155800	165600	174600	168700	167000	169300	167500	169600	172400
20	199300	207600	208500	247700	206600	200100	207200	213500	211100	204000
21	203300	224500	207000	202900	204000	204100	207100	205700	202300	198600
22	205600	201100	206000	198300	201100	207300	202000	205900	198700	200900
23	213200	219200	212300	211800	216700	215900	219300	215200	214000	215500

Appendix E – Permission for the A* code[28]

I communicated with the owner of this code remotely.



SID
February 18, 2021 at 6:28 pm

Hello there,

I am student studying in the International Baccalaureate Diploma Program. I am writing this thing called the extended essay. It is a research based essay where I take topic in a subject of interest and conduct some sort of investigation/analysis. My extended essay is in computer science and in my essay I am analyzing time complexities of various pathfinding algorithms in finding the shortest path.

Would it be okay if I use this code for my essay (I will provide proper references and credits)

Reply

ADMINISTRATOR
February 18, 2021 at 7:05 pm

Yes, you are more than welcome to use the code in your essay.

Reply

[28] "A* Search Algorithm in Python." *A Name Not Yet Taken AB*, 22 Jan. 2020,

Appendix F – Evidence of subject expert[29]

Hello sir,

This is a formal email, with questions I have regarding path finding algorithms and time complexity. You can copy these questions and write your response and respond to this email.

1. Please state who you are and your area of expertise (i.e name, profesion, degree & etc)

Biswajit: this is Biswajit Paria having 15+ yrs experience in advancedJava programming.

2. What does it exactly mean by time complexity and algorithm complexity in general?

Biswajit: Time complexity is the part of algorithm complexity. Time Complexity is basically to determine the total unit of time required to execute an algorithm.

3. Furthermore, to what extent is time complexity representative of efficiency?

Biswajit:To maximize the efficiency we need to minimize the use of the resource with the help of measuring time complexity.

4. How can implementation of an algorithm affect its efficiency?

Biswajit: it's true that way of implementing algorithm may impact is efficiency. Binary search Tree vs Red Black tree are the great examples of how to make efficient binary search in case when search becomes a linear search.

5. What are real life applications of path finding algorithms in shortest path problems?

Biswajit: Google map direction from one place to target location is the real example of shortest path algorithm.

6. In your opinion, what do you think about the performance of Dijkstra's algorithm in comparison to A* algorithm in finding the shortest? Is there a noticeable difference? Is A* better than Dijkstra because it is an informed search?

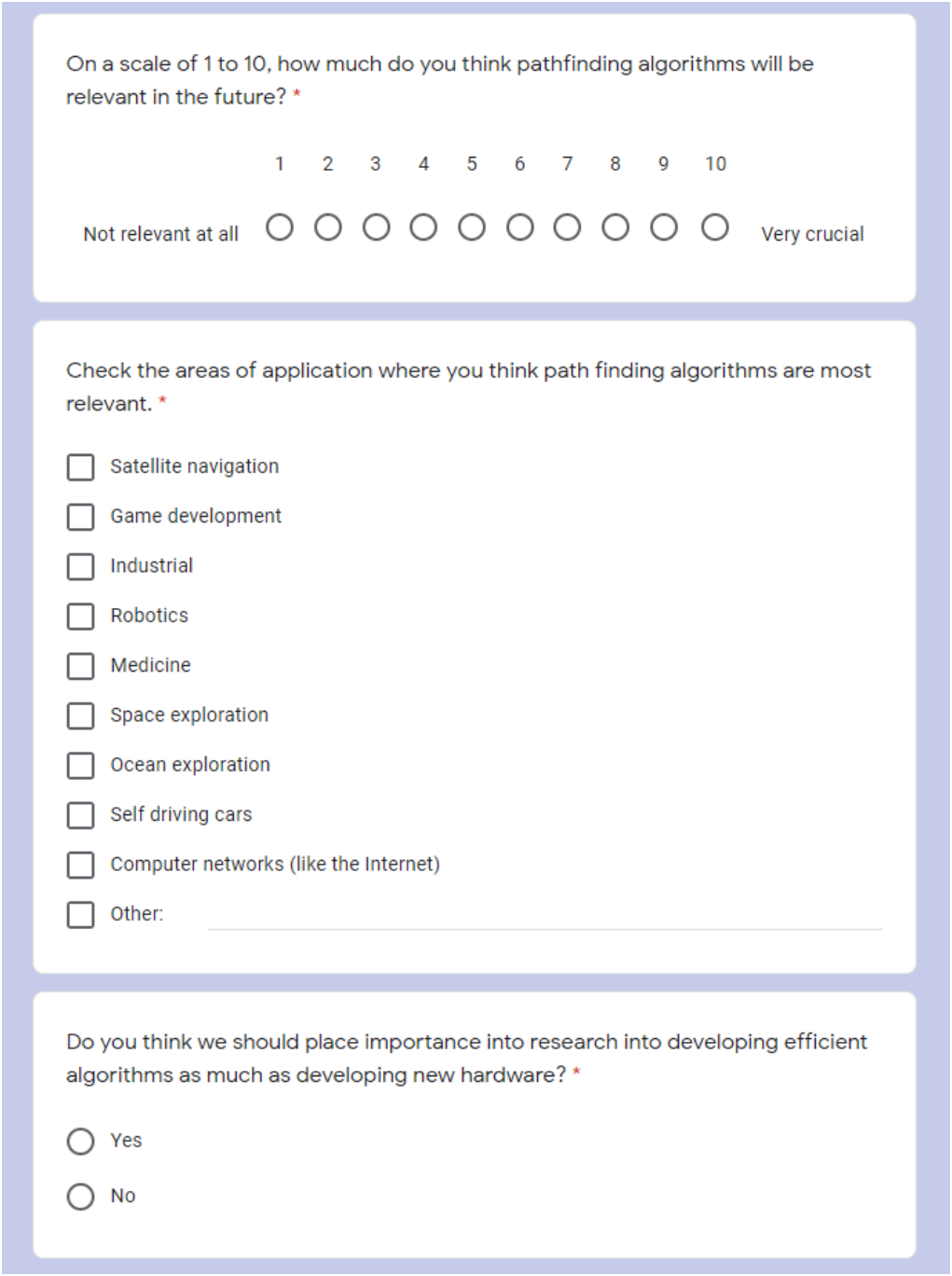
Biswajit: Yes true. A* tries to look for a better path by using a heuristic function, however it consumes more memory as it does more operations per node.

Due to COVID-19 pandemic I couldn't meet the subject expert in person. Therefore, we communicated remotely.

[29] Paria, Biswajit. Subject Expert, 2021

Appendix G – Survey and results

Survey questions



On a scale of 1 to 10, how much do you think pathfinding algorithms will be relevant in the future? *

	1	2	3	4	5	6	7	8	9	10	
Not relevant at all	○	○	○	○	○	○	○	○	○	○	Very crucial

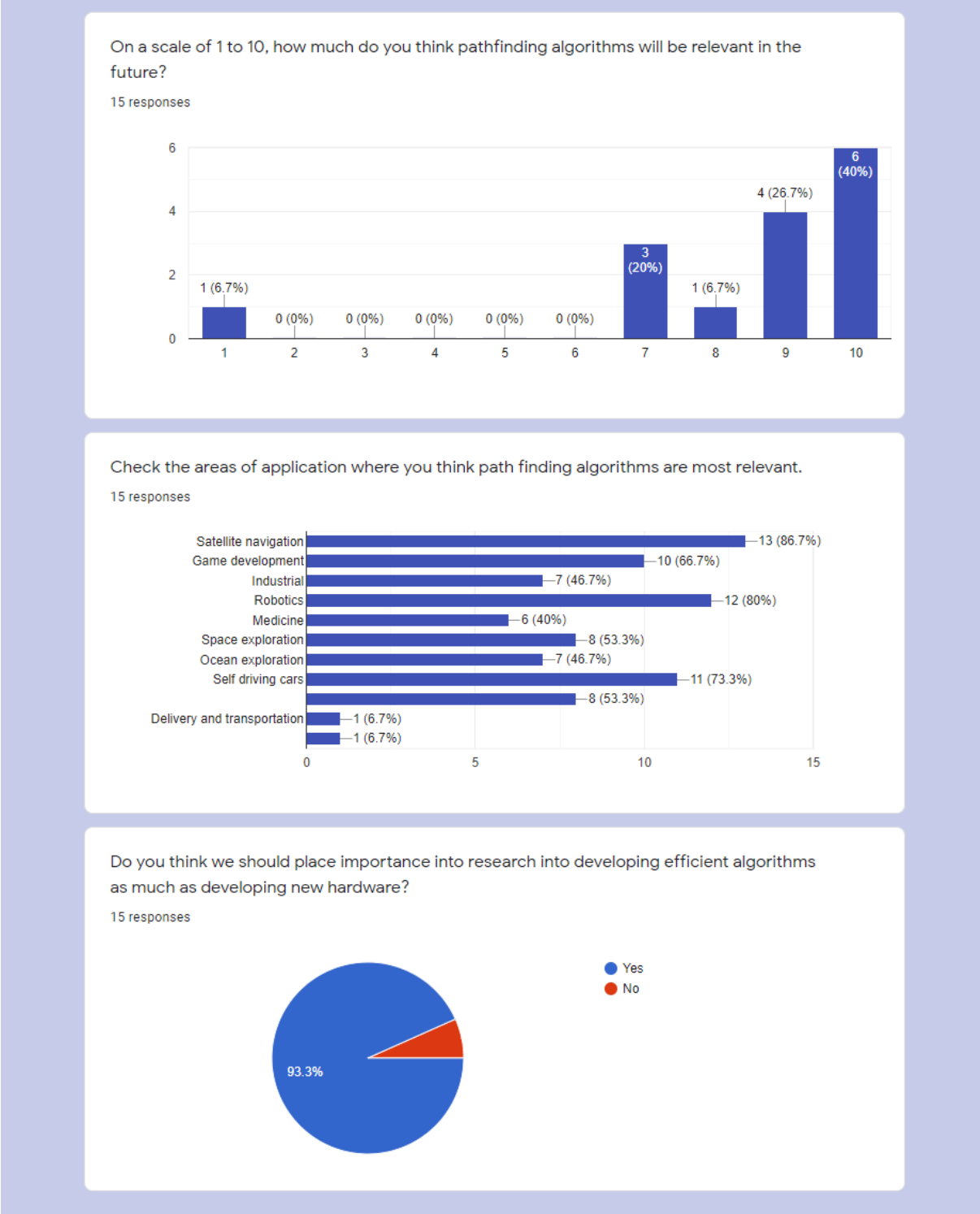
Check the areas of application where you think path finding algorithms are most relevant. *

- ☐ Satellite navigation
- ☐ Game development
- ☐ Industrial
- ☐ Robotics
- ☐ Medicine
- ☐ Space exploration
- ☐ Ocean exploration
- ☐ Self driving cars
- ☐ Computer networks (like the Internet)
- ☐ Other: _____

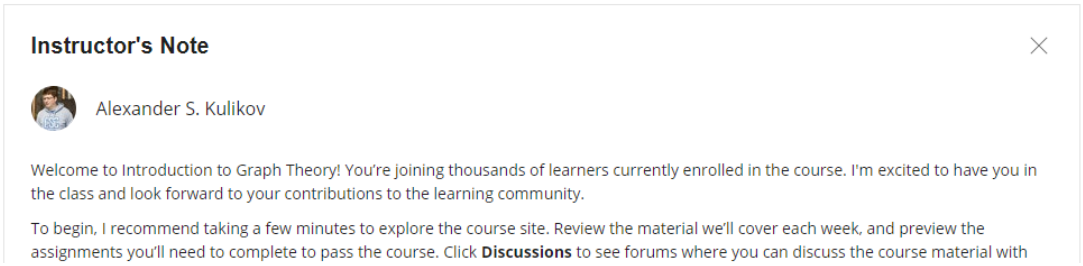
Do you think we should place importance into research into developing efficient algorithms as much as developing new hardware? *

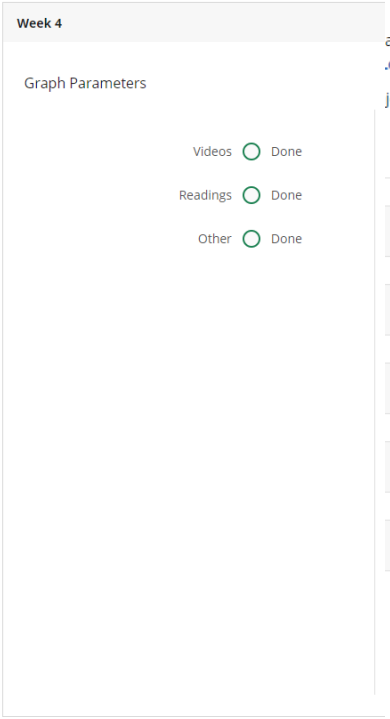
- ○ Yes
- ○ No

Survey results (15 responses):

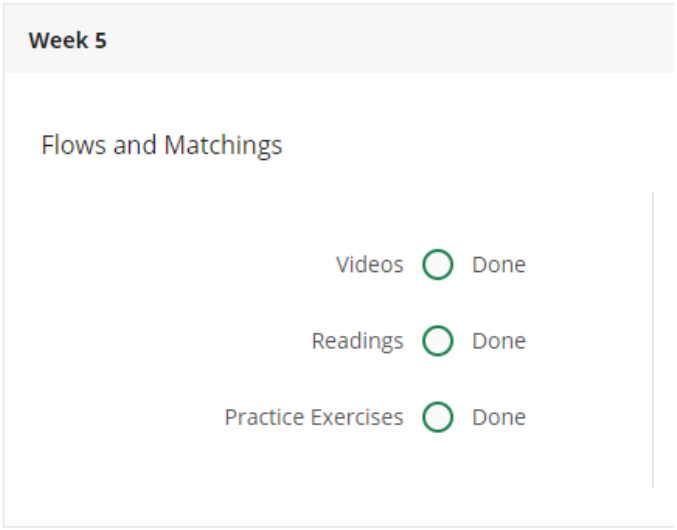


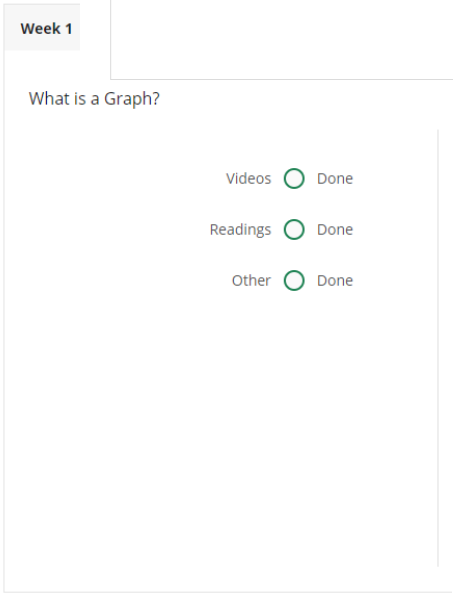
Appendix H – Course completion evidence

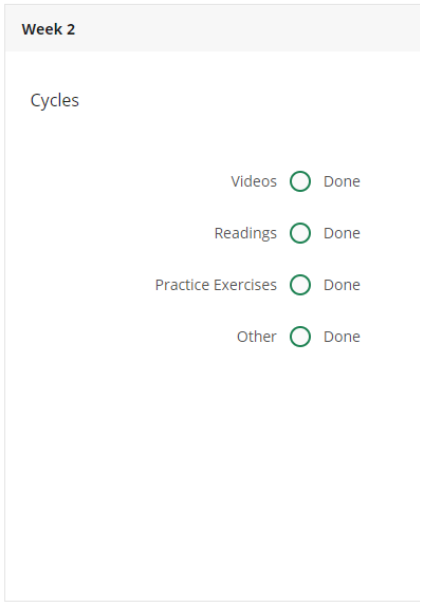


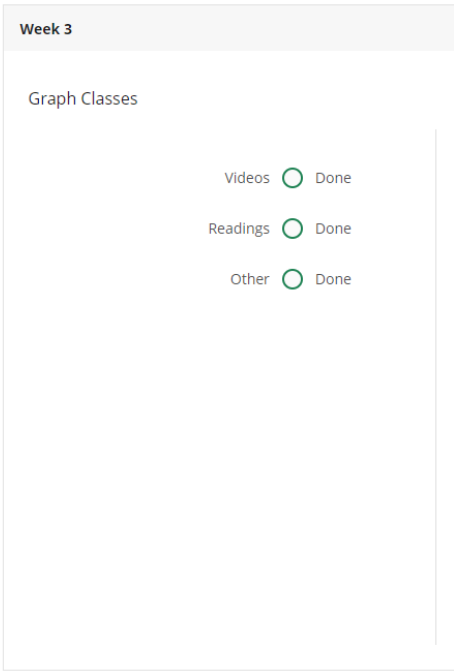


ase post them
.earner Help C
joy the course!









Works Cited

"A* Search Algorithm in Python." *A Name Not Yet Taken AB*, 22 Jan. 2020, www.annytab.com/a-star-search-algorithm-in-python.

Choudry, Humzah., 2017. *Understanding Algorithm Efficiency And Why Its's Important*. [online] Medium. <https://medium.com/@humzah.choudry/understand-algorithm-efficiency-and-why-its-important-89df0d5dfb64>.

Cormen, Thomas H. et al. *Introduction to Algorithms*, 3rd ed., MIT Press, 2009, pp. 589–658.

"Dijkstra Algorithm: Example: Time Complexity." *Gate Vidyalay*, 2020, www.gatevidyalay.com/tag/dijkstra-algorithm-pseudocode/.

Hart, P., Nilsson, N., & Raphael, B. (1968). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107.* doi:10.1109/tssc.1968.300136

Kask, Kalev. *Lecture 4: Optimal and heuristic search*, lecture notes, ICS 271, University of California, Irvine, 2016.

Kulikov, Alexander S. "Introduction to Graph Theory." University of California San Diego, National Research University Higher School of Economics. *Coursera.* Online Lecture <https://www.coursera.org/learn/graphs>

Lanning, D. R., Harrell, G. K., & Wang, J. (2014). *Dijkstra's algorithm and Google maps. Proceedings of the 2014 ACM Southeast Regional Conference on - ACM SE '14.* doi:10.1145/2638404.2638494

Paria, Biswajit. Subject Expert, 2021

Pound, Mike. “A* (A Star) Search Algorithm - Computerphile.” *YouTube*, uploaded by Computerphile, 15 Feb. 2017, www.youtube.com/watch?v=ySN5Wnu88nE.

Pound, Mike. “Dijkstra’s Algorithm - Computerphile.” *YouTube*, uploaded by Computerphile, 4 Jan. 2017, www.youtube.com/watch?v=GazC3A4OQTE.

Russell, Stuart J, and Peter Norvig. *Artificial intelligence: a modern approach*. Englewood Cliffs, N.J: Prentice Hall, 1995. Print.

Adamchik, Victor S. *Algorithmic Complexity,* Carnegie Mellon University, 2009

Sedgewick, Robert and Philippe Flajolet. *An introduction to the analysis of algorithms Second Edition.* Addision-Wesley, 2013.

Sharma, Akash. “Time and Space Complexity Tutorials & Notes | Basic Programming.” *HackerEarth*, 30 Aug. 2016, www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial.

Quinn, Catherine, et al. *Mathematics for the international student : mathematics HL (option) : discrete mathematics, HL topic 10, FM topic 6, for use with IB diploma programme*. Marleston, SA: Haese Mathematics, 2014. Print.