Extended Essay

Mathematics

How do we compare graph theory algorithms such as brute force and Dijkstra's to improve data transmission speeds across the Internet?

Word count: 3930

Table of Contents

Comparing Internet Data Transmission Speeds of Graph Theory Algorithms

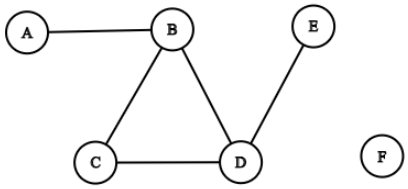
Introduction

Many people enjoy playing online multiplayer video games because of the competitive nature of competing against others around the world or because of the replayability that comes with playing with the community as opposed to computer-controlled characters. The primary disadvantage of playing online is one's ping, the delay in milliseconds between sending and retrieving data to a server, can spike at times ("Ping"). This renders the player unable to synchronize their game quickly enough with others to have a fun experience.

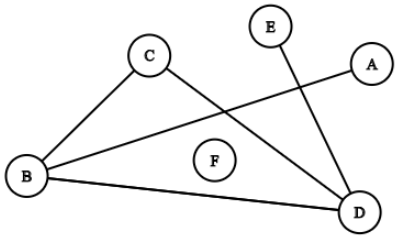
This year, I noticed I have usually been automatically connected to North American servers when I join online game lobbies, decreasing ping by reducing the physical distance between my computer and the server. I live in the United States, but I never specify this information to the games I play. During peak Internet usage hours in North America (7-11 pm EDT), I would sometimes be assigned to servers in Europe or Asia (Anderson). I recognized my games were programmed to minimize ping to their servers, but how were they able to do it? One branch of mathematics, called graph theory, is substantially associated with computer networking for purposes such as these. In this essay, I will be answering the research question "How do we compare graph theory algorithms such as brute force and Dijkstra's to improve data transmission speeds across the Internet?"

What is a graph?

Unless otherwise stated, the graphs in this essay will not refer to functions plotted on a coordinate system. Instead, my focus will be on the structure used to organize data. Graphs consist of edges connecting pairs of vertices together (“Graphs”). Below is a simple illustration:



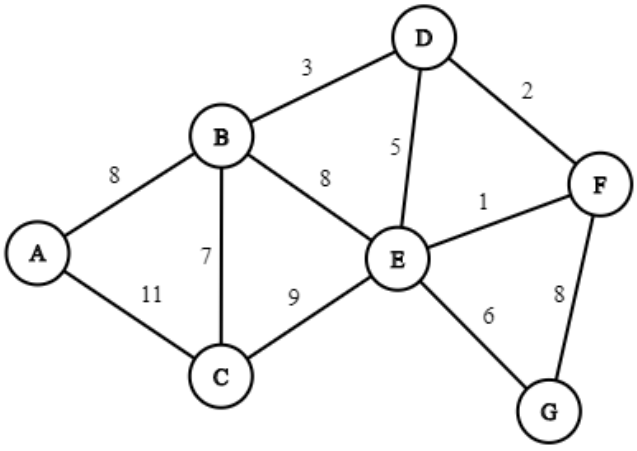
I labeled the vertices of the graph with the letters from *A* to *F*, and the edges are denoted by lines linking those vertices. Any particular vertex has its adjacent (or neighboring) vertices connected by the same edge. For example, Vertex B is adjacent to Vertices A, C, and D, but Vertex F has no neighbors. Note that we are only interested in which vertices exist and are adjacent in our graph and we disregard the exact locations of vertices and lengths of edges. Therefore, the graph below is equivalent (although messier) to the first graph since the same six vertices are present in it with edges connecting the same vertex pairs as before.



Shortest path problem overview

Computer networks send packets of data between individual computers. Network protocols such as Open Shortest Path First send these packets to intermediate routers along the way to reduce the time needed to transport the data (“Dijkstra’s”). The type of network, the current network traffic, and the location of the client computers also impact the speed of this transmission. We want to minimize the time it takes for one computer (the sender) to send information to another (the receiver).

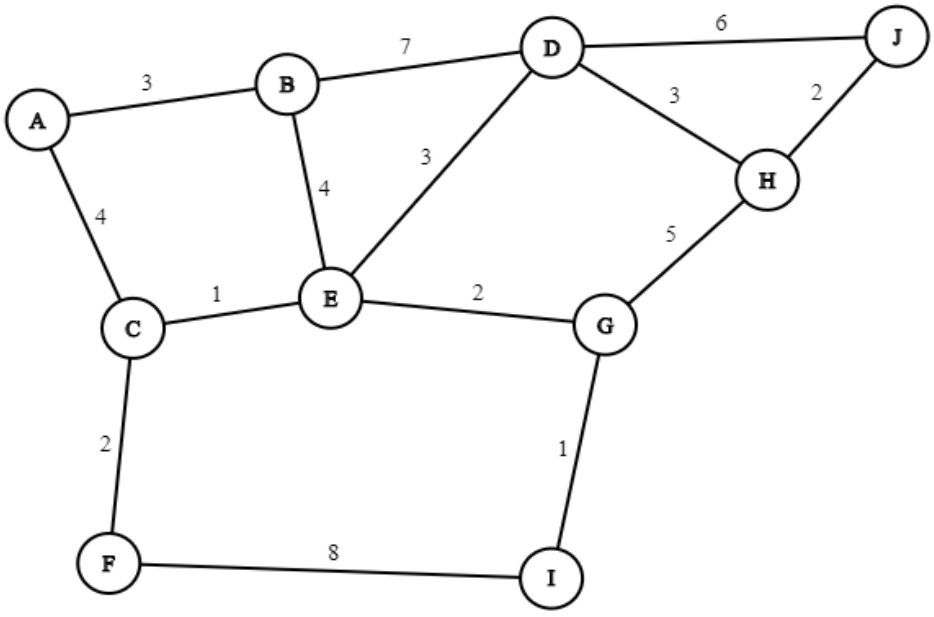
We could start this problem by representing the computers and routers as vertices on a graph, with edges showing the network connections between these nodes. However, we can not overlook one important detail—the speed of the network transportation. The fastest route for the data is not determined by the least number of edges the data will travel in the graph but how much time it takes for the sender to pass data to the receiver. Thus we will need to assign a time to each of the graph’s edges. A graph with values (or weights) on its edges is called a weighted graph, as is shown below (“Dijkstra’s”).



Using our new graph, let the sender be Vertex A and the receiver be Vertex G. The problem now becomes finding the minimum sum of weights of a route from Vertex A to G. This is called the “shortest path problem,” if we imagine the weights being road lengths between intersections and a driver wants to travel the least amount of distance to reach another intersection (“Dijkstra’s”).

Returning to the router problem, imagine I (the sender) live in Chicago, Illinois, and I am sending an email to my friend (the receiver) in Boston, Massachusetts. For the email to reach the receiver, the data travels through a path between the two cities, which may include intermediate router ports in other cities. Below is a map displaying most of the Northeastern United States and the cities we will consider.

If we convert this map to a graph and add edges for the connections between the router ports, we can start determining the fastest path the data can travel.



The weights of this graph denote the times, in seconds, for the data to move between any two vertices. We want to send the data from Vertex A (Chicago) to Vertex J (Boston). We want to minimize this send time, but how?

The first detail we notice is the data should never travel to a point twice. Why? Suppose the data goes from Vertex D, then B, then E, and back to D. I will write this subpath as D→B→E→D. The data, now located at Vertex D, can travel to the same vertices as when the data first reached Vertex D, making the cycle between the three vertices cost extra time.

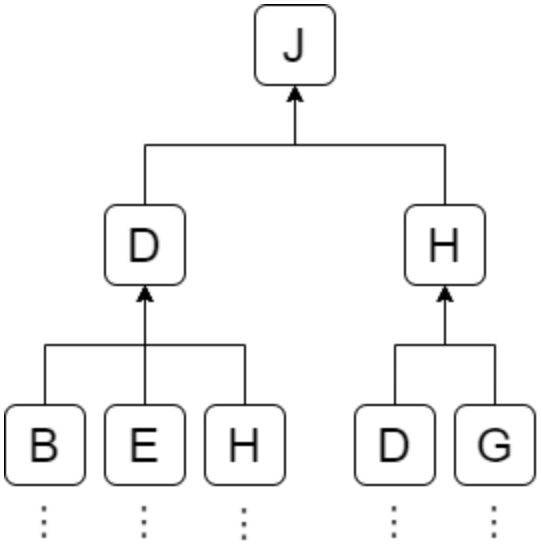
Algorithm 1: brute force

We can begin by using a brute force algorithm, which calculates the time of every permutation of vertices starting from Vertex A and finishing at Vertex J. There are 10 vertices in the graph, so we will test all paths from 0 vertices in between (i.e. A→J) to paths with 8 vertices in between (e.g. A→B→D→E→C→F→I→G→H→J). Note that impossible paths are included in this count, such as A→J or A→C→I→D→J, where an edge between two consecutive vertices in our "path" does not exist. This is because we do not want to miss a path in our count. We will discard any impossible paths we encounter when calculating their times. This algorithm *will* find the minimal time and its corresponding path, but it is terribly inefficient. The algorithm will examine

$$\sum_{i=0}^{8} \frac{8!}{i!} = \frac{8!}{0!} + \frac{8!}{1!} + \frac{8!}{2!} + \quad \cdots \quad + \frac{8!}{7!} + \frac{8!}{8!} = 109601$$

unique paths (including discarded impossible paths).

To arrive at this total, we first consider the paths with 5 vertices. We know we start at Vertex A and end at Vertex J, so that leaves 8 additional vertices to choose from, without replacement, to fill the 3 middle positions. There are 8 vertices to select from for the second vertex in the path, leaving 7 vertices for the third vertex, which then leaves 6 vertices for the fourth vertex. Thus there are $8 \cdot 7 \cdot 6$ paths for paths with 5 vertices in our graph. If we used a path with more vertices, then we would continue multiplying by numbers one less than before to represent the vertices available to select. Therefore, the number of paths with *n* vertices excluding Vertex A and J is equivalent to $\frac{8!}{(8-n)!}$ for our graph, and there are $\sum_{i=0}^{8} \frac{8!}{(8-i)!} = \sum_{i=0}^{8} \frac{8!}{i!} = 109601$ paths in total. (For a graph with *V* vertices including the starting and ending vertices, there would be $\sum_{i=2}^{V} \frac{(V-2)!}{(V-i)!}$ paths.) If the algorithm spent 100 milliseconds computing the time for each path, it would take 183 minutes to determine the fastest path! My friend would not like emailing me if this brute force algorithm was used! Of course, we could eliminate all the impossible paths by starting at Vertex J and working backwards, as displayed by the chart below.



(The chart indicates proper paths must end with any of B→D→J, E→D→J, H→D→J, D→H→J, or G→H→J.) However, as the number of edges increases in a graph, or, in our case, connections between router ports, the computational time of our improved algorithm increases to the old brute force method. What can we do instead?

Algorithm 2: Dijkstra's

Edsger Dijkstra, a Dutch computer scientist, invented an algorithm in 1956 to evaluate the shortest train route between cities in a graph-like network ("Edsger"). Unlike brute force, Dijkstra's algorithm records and updates the minimum cost of all vertices, or the sum of weights to reach a vertex, from any particular starting vertex ("Dijkstra's"). We will start by creating a table with the minimum time for the data to travel from Vertex A to each other vertex and the previous vertex from which the data departed from to reach the current vertex with the prior time; we will call this vertex the parent vertex, and record if the vertex has been visited. (In programming, the table

might be implemented using variable fields on each vertex object.) We will also need a list of vertices to explore from, sorted by the minimum time (in parentheses, equal time ties do not matter) to travel to that vertex, and we will call this list the queue. This will make more sense when we go through our example.

First, we add Vertex A to our queue and table:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	No
All others	∞		No

Queue
A (0)

(The minimum time to get the other vertices is infinity because we do not know whether a path exists to them yet.) Since Vertex A is at the top of the queue, we can “visit” it by adding its adjacent vertices to the table and queue, with the weights of the edges from Vertex A to those vertices being the minimum time for the data to reach those new vertices. We also remove Vertex A from the queue once we visit it, and sort the queue with the lowest time at the top.

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	No
C	4	A	No
All others	∞		No

Queue
B (3)
C (4)

We repeat this process by visiting Vertex B and adding D and E to the queue and table and then sorting the queue. We do not add Vertex A because it has already been visited and cannot have a lower time than 0. Since the parent vertex of these new vertices is Vertex B, we add the cost to reach Vertex B, which is 3, to the weights connecting Vertex B to the new vertices.

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	No
D	10	B	No
E	7	B	No
All others	∞		No

Queue
C (4)
E (7)
D (10)

We now visit Vertex C. We notice Vertex E is unvisited and adjacent to Vertex C, but it already has a minimum time of 7 going from Vertex B. When we exit from Vertex C, this minimum time reduces to $4 + 1 = 5$, meaning the path A→C→E is faster than A→B→E! Hence, we must update Vertex E in the table with Vertex C as its new parent vertex. Vertex E will have its minimum time finalized only when it is visited; when it reaches the top of the queue, any vertices with shorter minimum times would have already had their opportunities to set a lower minimum time. Since A→C had a time of 4 and Vertex E had a minimum time of 7, a shorter path to Vertex E through C could have existed, but if A→C had a time greater than or equal to 7, there would exist no

possibility of a shorter path through Vertex C. This property of Dijkstra's algorithm assumes that negative edge weights do not exist and applies to all vertices in the graph. We also add Vertex F as an adjacent vertex, which moves above Vertex D in the queue since it has a lower minimum time.

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	10	B	No
E	5	C	No
F	6	C	No
All others	∞		No

Queue
E (5)
F (6)
D (10)

Visiting Vertex E:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	No
E	5	C	Yes
F	6	C	No
G	7	E	No
All others	∞		No

Queue
F (6)
G (7)
D (8)

Visiting Vertex F:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	No
E	5	C	Yes
F	6	C	Yes
G	7	E	No
I	14	F	No
All others	∞		No

Queue
G (7)
D (8)
I (14)

Visiting Vertex G:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	No
E	5	C	Yes
F	6	C	Yes
G	7	E	Yes
H	12	G	No

Queue
D (8)
I (8)
H (12)

I	8	G	No
J	∞		No

Visiting Vertex D:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	Yes
E	5	C	Yes
F	6	C	Yes
G	7	E	Yes
H	11	D	No
I	8	G	No
J	14	D	No

Queue
I (8)
H (11)
J (14)

Note that we have reached Vertex J, but we continue going through the algorithm because a faster path may exist to Vertex J.

Visiting Vertex I:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes

Queue
H (11)

B	3	A	Yes
C	4	A	Yes
D	8	E	Yes
E	5	C	Yes
F	6	C	Yes
G	7	E	Yes
H	11	D	No
I	8	G	Yes
J	14	D	No

Queue
J (14)

Visiting Vertex H:

Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	Yes
E	5	C	Yes
F	6	C	Yes
G	7	E	Yes
H	11	D	Yes
I	8	G	Yes
J	13	H	No

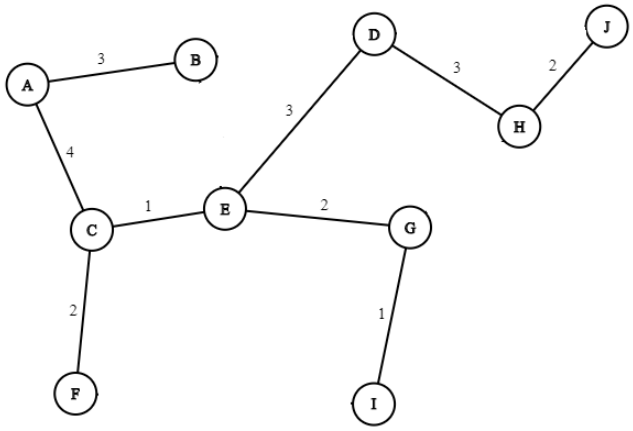
Queue
J (13)

Visiting Vertex J:

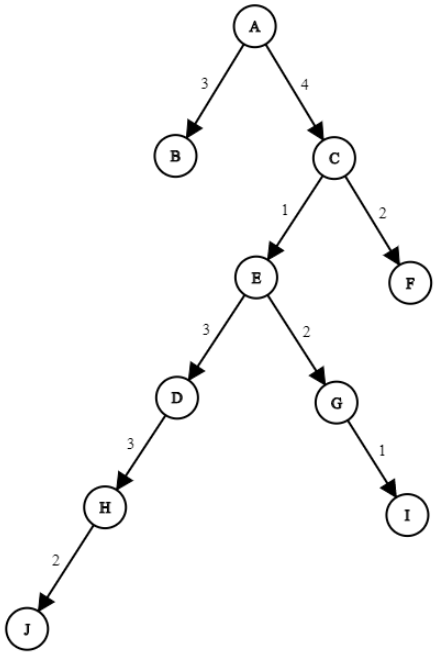
Vertex	Minimum time	Parent vertex	Visited?
A	0	none	Yes
B	3	A	Yes
C	4	A	Yes
D	8	E	Yes
E	5	C	Yes
F	6	C	Yes
G	7	E	Yes
H	11	D	Yes
I	8	G	Yes
J	13	H	Yes

Queue

And we are done! We now look at Vertex J in the table backtrack through parent vertices until we get to Vertex A. (Vertex H is J's parent, D is H's parent…) This results in the path A→C→E→D→H→J, with a 13 second data transmission time. Another benefit of using Dijkstra's algorithm is once we finish visiting all the vertices, we have a spanning tree of our graph. A tree is a graph containing no cycles, and our tree is considered spanning because the tree includes every vertex ("Dijkstra's").



I will reformat the tree with Vertex A at the top and add arrows to set the direction traveled along an edge, making the tree entirely directed, meaning each edge is unidirectional (“Dijkstra’s”). Our tree shows the time needed to travel from Vertex A to any other vertex by following the edges to that other vertex and adding the wedge weights.



I derived a proof by contradiction demonstrating all edges had to be directed for any given positive-time spanning tree produced by Dijkstra's algorithm. Let the starting vertex be Vertex A and $t(a, b)$ be the minimum time traveling from Vertex *a* to Vertex *b*. Assume the edge between Vertices X and Y is undirected. If X→Y and Y→X are both valid in the spanning tree, the fastest path to Vertex X must end with Y→X and the fastest path to Vertex Y must end with X→Y. Thus the minimum time to travel from Vertex A to X, or $t(A, X)$, equals $t(A, Y) + t(Y, X)$. Similarly $t(A, Y) = t(A, X) + t(X, Y)$. We now substitute this expression for $t(A, Y)$ into the first equation:

$$t(A, X) = t(A, Y) + t(Y, X)$$

$$t(A, X) = t(A, X) + t(X, Y) + t(Y, X)$$

$$0 = t(X, Y) + t(Y, X)$$

The times $t(X, Y)$ and $t(Y, X)$ must be positive, so the last equation cannot be true and all edges in the tree must be directed.

Time complexity

With nearly 60,000 public routers on the Internet as of 2018, efficient algorithms must be used to decide how data is sent ("CIDR"). In theoretical computer science, the time complexity of an algorithm details the time an algorithm takes to run depending on the number of elementary operations executed (with each operation taking one time unit) (Huang). We will only consider addition as one of these operations in our shortest-path algorithms, so calculating $1 + 2$ counts as 1 operation, $1 + 2 + 3 = (1 + 2) + 3$ as 2 operations, and so on. We will also express the time

complexities in terms of the number of vertices in a complete graph (*V*), which has edges connecting every pair of vertices (“Graphs”). A complete graph is used because more edges need to be checked when running either algorithm than with any other graph with the same number of vertices; the worst-case (maximum) time complexity for an input variable—the vertex count for our graphs—is commonly used in computer science.

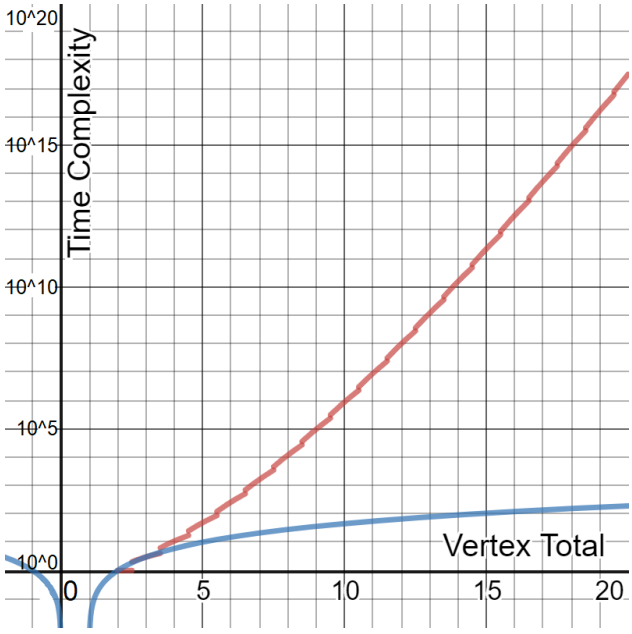
Earlier, we mentioned the brute force algorithm searches through $\sum_{i=2}^{V} \frac{(V-2)!}{(V-i)!}$ paths. In the summation, the number of elementary operations performed equals $i - 1$, where *i* is the summation index, representing the number of vertices tested in each path. For example, when $i = 5$, each path with 5 vertices is checked and 4 adding operations are needed on each path. Note that each path needs to be fully checked since complete graphs have no impossible paths with “missing” edges. Therefore, the time complexity of this algorithm is $\sum_{i=2}^{V} \frac{(i-1)(V-2)!}{(V-i)!}$.

In Dijkstra’s algorithm, each edge value is checked up to one time, as this value only tests whether a path to a new vertex is shorter than any previous path to that same vertex. Thus we will need the total number of edges of a complete graph to compute the time complexity. In a *V*-vertex graph, each vertex has edges connecting it to the other $V - 1$ vertices, appearing as if there are $V(V - 1)$ total edges. However, each edge is counted twice from using either vertex endpoint to form edges with the other $V - 1$ vertices, so there are $\frac{V(V-1)}{2}$ edges in a complete graph. Exactly 1 addition is performed

to calculate a path time to when a new edge is explored. To see why, let the new edge connect Vertices X and Y, where Vertex A is the starting vertex and Vertex X is being visited. The only operation computed is $t(A,Y) = t(A,X) + t(X,Y)$, $t(a,b)$ being the path time function from before. Therefore, the time complexity of Dijkstra's algorithm is $\frac{V(V-1)}{2}$.

Growth rate comparisons

Below is a coordinate graph of both time complexities plotted against the vertex total of each graph structure, with the brute force algorithm in red and Dijkstra's in blue.



(Note: The Gamma function (outside the scope of this exploration) is used in place of factorials to show the increase rate over all rational numbers. See the table of values in the appendix for the time complexity at each integer vertex total.) I used a logarithmic

scale because brute force is extremely slow compared to Dijkstra's. At $V = 10$, brute force does 878,809 operations whereas Dijkstra's does only 55!

But what is the mathematical reason behind the brute force's rapid increase as the vertex count grows? When describing time complexity in computer science, a function grows faster than another when the limit of the ratio of the functions as the input variable approaches infinity also tends toward infinity (Huang). Using mathematical notation, $f(n)$ grows faster than $g(n)$ when $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ goes to infinity (or $\lim_{n \to \infty} \frac{g(n)}{f(n)}$ goes to zero) where *n* is the input and $f(n)$ and $g(n)$ are two arbitrary algorithms. This method of calculating function growth implies n^2 grows faster than n, but $2n + 100$ does not grow faster than n. In calculus, L'Hôpital's rule states that when $\lim_{x \to \infty} \frac{f(x)}{g(x)}$ is of an indeterminate form, such as $\frac{0}{0}$ or $\frac{\infty}{\infty}$, then the limit equals $\lim_{x \to \infty} \frac{f'(x)}{g'(x)}$. Thus we use the derivatives of n^2 and n with respect to *n* to evaluate the limit, using n^2 in the numerator:

$$\lim_{n \to \infty} \frac{n^2}{n} \text{ (Indeterminate form } \frac{\infty}{\infty})$$

$$\lim_{n \to \infty} \frac{(n^2)'}{n'}$$

$$\lim_{n \to \infty} \frac{2n}{1}$$

$$2 \cdot \lim_{n \to \infty} n$$

Since $\lim_{n \to \infty} n$ tends toward infinity, n^2 grows faster than n. Similarly, we can apply L'Hôpital's rule on $2n + 100$ and n:

$$\lim_{n \to \infty} \frac{2n+100}{n} \text{ (Indeterminate form } \frac{\infty}{\infty}\text{)}$$

$$\lim_{n \to \infty} \frac{(2n+100)'}{n'}$$

$$\lim_{n \to \infty} \frac{2}{1}$$

$$2$$

As 2 is positive and finite, $2n + 100$ grows at the same rate as n.

At first, I was confused when I saw this result. How could $2n + 100$, having 2 times the slope and always being greater than n, have the same growth rates as each other? To start, I recognized only the fastest-growing term in each function needed to be compared in the limit because that term would consequently have the fastest-growing or greatest derivative, making all other terms negligible. For polynomials, only the term with the highest degree would affect the growth, making the constant 100 term increasingly insignificant as *n* approaches infinity. I then realized the limit has to be infinite for the function in the numerator to grow faster than the one in the denominator, so multiplying by a constant factor in terms of the same growth rate is also negligible since both functions would be uniformly increased when *n* also increases. In *n* triples in both $2n$ and n, then both functions would increase by a factor of 3, which is not affected by the constant factor of 2 in the first function.

We can use these properties to compare the time complexities of our shortest-path algorithms, $\sum_{i=2}^{V} \frac{(i-1)(V-2)!}{(V-i)!}$ and $V(V-1)$. For the brute force time complexity, as the summation index *i* increases to *V*, the numerator increases because of the increases $(i-1)$ factor and the denominator decreases because $(V-i)$ decreases and the factorial function is strictly increasing for positive integers. Thus the largest term in the summation is when $i = V$, which reduces the time complexity as *V* goes to infinity to $\frac{(V-1)(V-2)!}{(V-V)!}$. Simplifying this expression yields

$$\frac{(V-1)(V-2)!}{(V-V)!}$$

$$\frac{(V-1)(V-2)!}{0!}$$

$$\frac{(V-1)!}{1} \text{ (Combined } (V-1) \text{ into factorial)}$$

$$(V-1)!$$

In the time complexity of Dijkstra's algorithm, we know $V(V-1)$ expands to $V^2 - V$, which becomes V^2 after removing the term with a lower degree than the quadratic.

Since brute force seems to grow faster than Dijkstra's, we will check if $\lim_{V \to \infty} \frac{(V-1)!}{V^2}$ is infinite. Because the factorial function is discrete, I could not differentiate the numerator and L'Hôpital's rule could not be used like before. Using my knowledge from my IB Mathematics course, I analyzed a new approach to this problem. I knew the Ratio Test determines whether an infinite series, the sum of all the terms in an infinite sequence, converges or diverges by taking the ratio of consecutive terms as the

number of terms went toward infinity. This expression, $\lim_{n \to \infty} \left| \frac{a_{n+1}}{a_n} \right|$, with a_n being the *n*th term in the sequence, would indicate the series would converge when the absolute value of expression is less than 1 and diverge when greater than 1 (the test is inconclusive when exactly 1). I reasoned this was because I could model the terms similar to a geometric series (a series in the form of $\sum_{n=0}^{\infty} a_0 r^n$, converging to $\frac{a_0}{1-r}$ when $|r| < 1$) as *n* becomes very large, since the common ratio of a geometric series matches the criteria of convergence or divergence for the Ratio Test.

Although the Ratio Test assesses the behavior of a series, it can also be used in a similar manner with repeated multiplication to determine if one of our functions grows faster than the other. Let $h(V) = \frac{(V-1)!}{V^2}$. As stated before, if $\lim_{V \to \infty} h(V)$ tends toward infinity, then the brute force time complexity grows faster than that of Dijkstra's (and vice versa if the expression equals zero). As *V* increases, the ratio between consecutive values of $h(V)$ approaches some number. This number can be repeatedly multiplied to $h(k)$ for some large *k* to demonstrate the ratio between the time complexities as *V* goes to infinity. Hence $\lim_{V \to \infty} h(V) = \lim_{V \to \infty} h(k) \cdot \left(\frac{h(V+1)}{h(V)}\right)^V$.

Before we evaluate this limit, I want to show how the value of the fraction in the previous expression affects the result. In the following example, assume *R* is any positive number and *L* is the limit we are calculating:

$$L = \lim_{r \to R,\, V \to \infty} r^V$$

The right-hand side of the equation is representative of $\lim_{V \to \infty} (\frac{h(V+1)}{h(V)})^V$, both being in the same format. Notice that I have *r* approach *R*—but not equal to *R*—since $\frac{h(V+1)}{h(V)}$ is the ratio of the time complexities as *V* increases toward infinity but never reaches infinity. Taking the natural logarithm of both sides to "bring down" the exponent *V*:

$$ln(L) = \lim_{r \to R, V \to \infty} ln(r^V)$$

$$ln(L) = \lim_{r \to R, V \to \infty} V \cdot ln(r)$$

The value of *L* can now be used in a similar way as the limit in the Ratio Test. If $R > 1$, then $ln(r)$ is positive because e must be raised to some positive exponent to be greater than 1. Thus $\lim_{r \to R, V \to \infty} V \cdot ln(r)$ approaches infinity as the limit of infinity times any positive number is infinity, and L must go to infinity as well. Similarly, if $R < 1$, then $ln(r)$ is negative, $\lim_{r \to R, V \to \infty} V \cdot ln(r)$ approaches negative infinity, and L must tend towards 0. However, if $R = 1$, $ln(r)$ is positive when the limit is approached from the right (decreasing to 1) but negative when approached from the left (increasing to 1), so the value of L cannot be found this way.

Now we can determine if $\lim_{V \to \infty} \frac{h(V+1)}{h(V)}$ is greater or less than 1. Since $h(k)$ is positive and finite, we only need to evaluate $\lim_{V \to \infty} \frac{h(V+1)}{h(V)}$ as the value of $h(k)$ is insignificant if $\lim_{V \to \infty} (\frac{h(V+1)}{h(V)})^V$ is 0 or approaches infinity.

$$\lim_{V \to \infty} \frac{h(V+1)}{h(V)}$$

$$\lim_{V \to \infty} \frac{\frac{((V+1)-1)!}{(V+1)^2}}{\frac{(V-1)!}{V^2}}$$ (Substituted $h(V)$ with $\frac{(V-1)!}{V^2}$)

$$\lim_{V \to \infty} \frac{V!}{(V+1)^2} \cdot \frac{V^2}{(V-1)!}$$

$$\lim_{V \to \infty} \frac{V \cdot (V-1)! \cdot V^2}{(V-1)! \cdot (V+1)^2}$$

$$\lim_{V \to \infty} \frac{V^3}{(V+1)^2}$$

Before we continue, this $\frac{V^3}{(V+1)^2}$ conveys the ratio between the time complexities for consecutive values of V follows $h(V+1) = \frac{V^3}{(V+1)^2} h(V)$. Now we apply L'Hôpital's rule twice:

$$\lim_{V \to \infty} \frac{V^3}{(V+1)^2}$$ (Indeterminate form $\frac{\infty}{\infty}$)

$$\lim_{V \to \infty} \frac{(V^3)'}{((V+1)^2)'}$$

$$\lim_{V \to \infty} \frac{3V^2}{2(V+1)}$$

$$\lim_{V \to \infty} \frac{3V^2}{2V+2}$$ (Indeterminate form $\frac{\infty}{\infty}$)

$$\lim_{V \to \infty} \frac{6V}{2}$$

$$3 \cdot \lim_{V \to \infty} V$$

This $3V$ (removed duplicate limit notation) can be substituted into r in our previous example where $L = \lim_{r \to R, V \to \infty} r^V$. Because $3V$ approaches infinity, which is greater than 1, r^V also tends toward infinity. r^V corresponds to $(\frac{h(V+1)}{h(V)})^V$ in $\lim_{V \to \infty} h(V) = \lim_{V \to \infty} h(k) \cdot (\frac{h(V+1)}{h(V)})^V$, meaning $\lim_{V \to \infty} h(V)$ approaches infinity as well. Therefore, we can finally be confident the time complexity of the brute force algorithm grows faster than that of Dijkstra's algorithm. This signifies brute force takes increasingly longer times to run when compared to Dijkstra's as the number of vertices increases.

Conclusion

Through my research, one explanation I thought of to my introductory example on how video games minimize ping is through using Dijkstra's algorithm to generate a spanning tree with all of the game's servers. The minimum time to send data to each of their servers could easily be compared to find the optimal server.

There are many other shortest-path algorithms with different advantages and disadvantages. For example, the A* ("A-star") algorithm usually performs better than Dijkstra's with network routing because it can take in physical distance to factor in physical distance and direction, but it does not always calculate *the* fastest path (but often still *a* fast path) ("Dijkstra's"). Graph theory is not just for networking in computer science: it is also useful for machine learning such as in Amazon.com's recommendation system and storing social graphs such as Facebook's friendship

database (Bronson). Many questions remain unsolved and many algorithms have yet to be developed in graph theory, presenting a promising future for the field.

Word count: 3930

Works Cited

Anderson, Nate. "Does less evening Internet mean Europeans lead better lives?" *Ars Technica*, 1 Sept. 2009, arstechnica.com/tech-policy/2009/09/does-less-evening-internet-mean-europeans-lead-better-lives/. Accessed 1 Oct. 2020.

Bronson, Nathan. "TAO: Facebook's Distributed Data Store for the Social Graph." *USENIX*, www.usenix.org/system/files/conference/atc13/atc13-bronson.pdf. Accessed 1 Oct. 2020.

"CIDR Report for 22 Feb 21." *BGP Routing Table Analysis Reports*, www.cidr-report.org/as2.0/. Accessed 1 Oct. 2020.

"Dijkstra's Algorithm." *Rosetta Code*, rosettacode.org/wiki/Dijkstra%27s_algorithm. Accessed 1 Oct. 2020.

"Edsger Wybe Dijkstra." *Association for Computing Machinery*, amturing.acm.org/award_winners/dijkstra_1053701.cfm. Accessed 1 Oct. 2020.

"Graphs and Networks." *Encyclopaedia Britannica*, www.britannica.com/topic/number-game/Graphs-and-networks. Accessed 1 Oct. 2020.

Huang, Shen. "What Is Big O Notation Explained: Space and Time Complexity." *Free Code Camp*, 16 Jan. 2020, www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/. Accessed 1 Oct. 2020.

"Ping, Latency and Lag: What You Need to Know." *British Esports Association*, 1 Mar. 2017, britishesports.org/news/ping-latency-and-lag-what-you-need-to-know/. Accessed 1 Oct. 2020.

Appendix

Table of Values for Vertex Total vs. Time Complexities Graph:

Vertex Total	Brute force operations	Dijkstra's algorithm operations
2	1	1
3	3	3
4	11	6
5	49	10
6	261	15
7	1631	21
8	11743	28
9	95901	36
10	876809	45
11	$8.878 \cdot 10^6$	55
12	$9.864 \cdot 10^7$	66
13	$1.194 \cdot 10^9$	78
14	$1.562 \cdot 10^{10}$	91
15	$2.200 \cdot 10^{11}$	105
16	$3.318 \cdot 10^{12}$	120
17	$5.332 \cdot 10^{13}$	136
18	$9.100 \cdot 10^{14}$	153
19	$1.644 \cdot 10^{16}$	171
20	$3.313 \cdot 10^{17}$	190