Navigating the Network:

Real-Life Applications of Graph Traversal Algorithms

Jonathan Nguyen and Emmanuel Mateo

May 22, 2023

Abstract

In this paper, we will introduce the history, fundamentals, and real-life applications of graph theory. After beginning with an introduction of the history of the field through the Königsberg Bridge Problem, we will first cover the traversability of graphs—namely Eulerian and Hamiltonian graphs—and introduce important algorithms with real-life applications. We then shift away from traversability to focus on a specific type of algorithm that finds the shortest path between two points—Dijkstra's algorithm—and discuss how this algorithm can be used for cost efficiency and route optimization. Finally, we narrow our focus to tree graphs to tackle the Minimum Spanning Tree problem and present the two main algorithms used to solve it.

1 Introduction to Graph Theory

Graph theory is a mathematical field that deals with the study of relationships and networks of connections between objects. With the countless real-life applications, having an understanding of graph theory is essential to grasp the infinite relationships in this world. From computer science to traffic networks, to GPS systems and the Internet, graph theory helps us to analyze and solve complex problems by providing a powerful tool to model and represent relationships between objects. In this paper, we will discuss some of the real-world applications of graph theory by focusing on how to find the most efficient and effective routes between locations

To motivate the study of graph theory, Section 1 will summarize the history of the field by introducing the Königsberg Bridge Problem. In addition, we will introduce fundamental definitions and types of graphs. In Section 2, we will focus on how to traverse different types of graphs. In particular, we will explore Eulerian and Hamiltonian graphs and contrast different algorithms used to find the optimal routes to traverse the graph.

In Section 3, we shift away from traversability to focus on a specific type of algorithm that finds the shortest path between two vertices: Dijkstra's algorithm. We then discuss how this algorithm can be used for cost efficiency and calculating the shortest distance in real-life situations. In Section 4, we will focus specifically on tree graphs and introduce the Minimum Spanning Tree problem, which seeks to find a tree that spans all the vertices in a connected, weighted graph while minimizing the sum of the weights of the edges. In particular, we will introduce two specific algorithms used to solve it – Kruskal's Algorithm and Prim's Algorithm – and their various applications in real-life scenarios.

1.1 History of Graph Theory

The Königsberg Bridge Problem laid the foundations for graph theory. The city Königsberg, located in Prussia, was separated by the River Pregel. This caused the city to be split into four different land masses with seven bridges connecting them, as depicted in Figure 1 below. In the early 18th century, the people of Königsberg wondered whether it would be possible to take a route that crosses over all of the bridges exactly once.

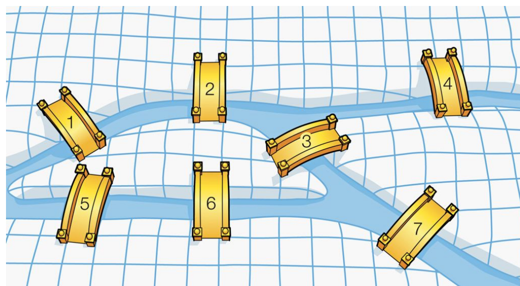


Figure 1: The city of Königsberg and its bridges.

This became known as the Königsberg Bridge Problem, and stayed unsolved for many years. This problem then caught the attention of Leonhard Euler, a Swiss mathematician who was believed to be close by. Euler represented this problem in a new way, as shown in Figure 2. He used letters to represent the landmasses and the lines connecting them to represent the bridges.

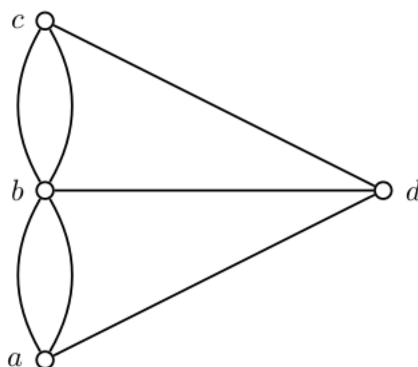


Figure 2: Simplified model of Königsberg represented by a graph

After studying his simplified model, Euler reached a few notable conclusions. First, he noticed that if more than two of the landmasses had an odd number of bridges leading to them, then it was always impossible to complete the walk crossing each bridge only once. Secondly, Euler observed that if there were exactly two landmasses with an odd number of bridges leading to them, the walk would be possible if it began in either of these two locations. Finally, Euler concluded that if there were no landmasses with an odd number of bridges leading to them, then the walk was always possible.

Therefore, such a walk was impossible in Königsberg. Although the immediate application of Euler's work only related to the trivial matter of walking across bridges in a city, his revolutionary thinking led to the foundation of modern graph theory. We continue the discussion of Eulerian graphs in Section 2.

1.2 General Definitions

Graphs are made to represent relationships between objects. These connections are shown using vertices and edges. Vertices are the basic building block of graphs, and it is the most important component of a graph. Edges are the other fundamental piece that connects vertices to each other or themselves.

Definition 1.1. *A **graph** G consists of a finite nonempty set V of objects called **vertices** (the singular is vertex) and a set E of 2-element subsets of V called **edges**.*

It can be difficult to interpret the graphs without quantifying them. This is why finding the order and size of graphs are important, as they give us a way to compare other graphs.

Definition 1.2. *The **order** of a graph is the number of vertices in a graph. This is usually denoted by n.*

Definition 1.3. *The **size** of a graph is the number of edges in a graph. This is usually denoted by m.*

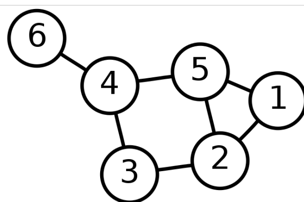


Figure 3: A graph with an order of $n = 6$ and a size of $m = 7$

One of the most common relationships between vertices is being neighbors. This is a fundamental way of comparing vertices, and having an edge between vertices is the best way to visualize how they're connected.

Definition 1.4. *Vertices that are connected together by an edge are called **neighbors**.*

Definition 1.5. *The number of edges connecting to a vertex is called the **degree** of the vertex.*

The degree of a vertex can also be thought of as the number of neighbors it has. Sometimes it is important to only analyze a part of a graph instead of the whole graph. With subgraphs, certain pieces of the graph are taken out to simplify the graph.

Definition 1.6. *A **subgraph** is a graph that is a smaller part of another graph.*

Definition 1.7. *A **spanning subgraph** is a subgraph that contains all of the vertices of the original graph.*

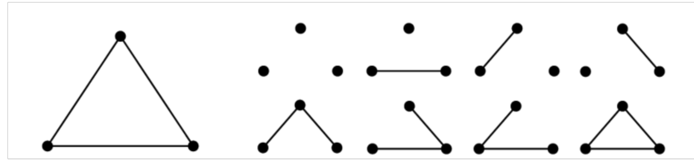


Figure 4: A graph G and the spanning subgraphs of G

Definition 1.8. *A* ***Vertex-Induced Subgraph*** *is a subgraph that consists of some of the vertices in the original graph, and all of the edges connecting them.*

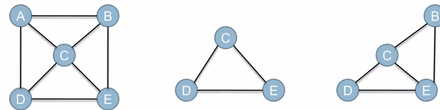


Figure 5: A graph G and three of its vertex-induced subgraphs.

Definition 1.9. *An* ***Edge-Induced Subgraph*** *is a subgraph that contains some of the edges of the original graph, and all of the vertices connecting them.*

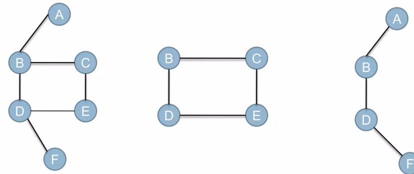


Figure 6: A graph G and the edge-induced subgraphs of G.

Another fundamental result in graph theory relates the degree of the vertices in a graph to the number of edges. The following theorem is often referred to as the First Theorem of Graph Theory.

Theorem 1 (**The First Theorem of Graph Theory**)**.** *In a graph G, the sum of the degrees of the vertices is equal to twice the number of edges.*

Proof. When summing the degrees of the vertices of G, each edge of G is counted twice, once for each of its two incident vertices. [1, Theorem 2.1] □

1.3 Graph Traversal

After constructing a graph, it is natural to consider different ways of moving around it. In this section, we define some key terms to describe the types of ways in which a graph can be traversed along certain edges.

Definition 1.10. *A **walk** is a list of adjacent vertices in a graph G.*

Definition 1.11. *A **trail** is a walk where no edge is crossed more than once.*

Definition 1.12. *A **path** is a walk where no vertices are crossed more than once.*

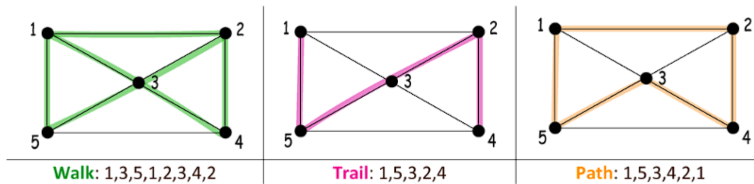


Figure 7: The difference between walks, trails, and paths.

Definition 1.13. *A **circuit** is a trail that ends at the same vertex that has at least a length of three.*

Definition 1.14. *A **cycle** is a circuit that doesn't repeat any vertices.*

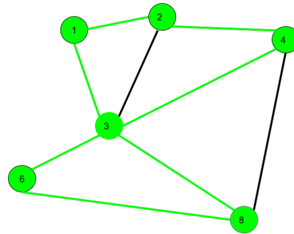


Figure 8: A graph that contains a cycle and a circuit. The circuit is the whole green area. This is because it goes over each green edge once, but it repeats a vertex. This is what separates a circuit from a cycle.

1.4 Connected Graphs

Understanding the connectedness of graphs is another fundamental concept in graph theory.

Definition 1.15. *A graph G is **connected** if every two vertices of G are connected, that is, if G contains a u–v path for every pair u, v of vertices of G.*

If a graph is not connected, we say it is *disconnected*. A connected subgraph of G that is not a proper subgraph of any other connected subgraph of G is a *component* of G. In other words, components are the number of separate parts that make up a graph. The number of components of a graph G is denoted by $k(G)$. There are several ways to produce a new graph from a given pair of graphs.

Definition 1.16. *The **union** of two graphs G and H is a new graph that contains all the vertices and edges of both G and H. The union of G and H is denoted by $G \cup H$ and is defined as a graph with vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$.*

In the union graph, if two vertices are present in both G and H, then they are represented only once, and the edges are combined. Similarly, if an edge is present in both G and H, then it appears only once in the union graph. Another useful relation in graph theory is the join of two graphs, which forms a new graph that contains all the vertices of G and H along with all possible edges between them.

Definition 1.17. *The **join** $G + H$ consists of $G \cup H$ and all edges joining a vertex of G and a vertex of H.*

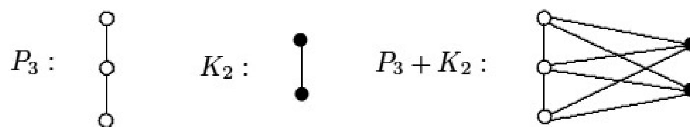


Figure 9: The join of two graphs

Given two graphs, a new graph can also be formed by taking the Cartesian Product.

Definition 1.18. *The **Cartesian Product**, $G \times H$, is a join of 2 graphs where all vertices of a graph G create new vertices with each vertex of H. The graph $G \times H$ has vertex set $V(G \times H) = V(G) \times V(H)$, that is, every vertex of $G \times H$ is an ordered pair (u, v), where $u \in V(G)$ and $v \in V(H)$.*

Example. $V(G) = (1, 2, 3)$ and $V(H) = (4, 5)$. So $G \times H$ creates the new vertices: $\{1, 4\}, \{2, 4\}, \{3, 4\}, \{1, 5\}, \{2, 5\},$ and$\{3, 5\}$. It would then create edges $(\{1, 4\}, \{2, 4\})$, $(\{2, 4\}, \{3, 4\})$, etc...

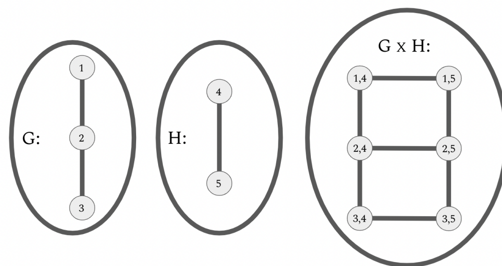
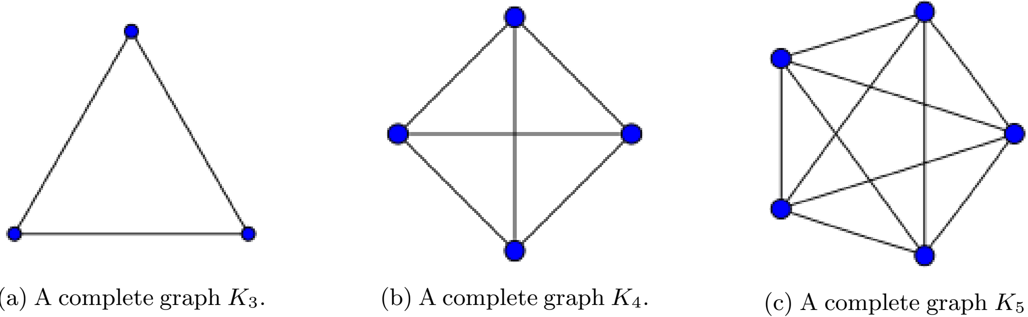


Figure 10: The Cartesian Product of two graphs

1.5 Common Classes of Graphs

After covering connected graphs, one could come to the conclusion that connected graphs are the limit of graph theory. Fortunately, there are many more types of graphs, all with different characteristics. In this section, we will go over these graphs and their different traits.

Definition 1.19. *A* ***complete graph*** *is a connected graph of any order of at least 3 has all vertices containing an edge between each other. Complete Graphs are also denoted as* K_n.



(a) A complete graph K_3. (b) A complete graph K_4. (c) A complete graph K_5.

Figure 11: Examples of complete graphs. As you can see, each vertex is connected to every other vertex.

Although complete graphs illustrate many important concepts of graph theory, there are more applicable and common graphs. Bipartite graphs are able to be visualized in a more creative way compared to complete graphs.

Definition 1.20. *A* ***bipartite graph*** *is a graph with two sets of vertices that make up the graph, but a vertex cannot connect to other vertices if they are both part of the same set.*

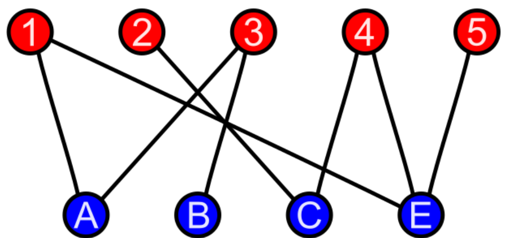


Figure 12: An example of a bipartite graph. As shown, there are only 2 different sets, and none of the vertices in the same set have an edge between them.

An even more common graph is a multigraph. They are essentially any ordinary graph but have the potential be a little different.

Definition 1.21. *A* ***Multigraph*** *is a graph where every two vertices can be connected by no edge, one edge, or multiple edges as long as it stays at a finite value of edges.*

A pseudograph is a little less common. They are multigraphs with one extra trait.

Definition 1.22. *A* ***Pseudograph*** *is a graph that allows parallel edges and vertices to join themselves.*
Note: ***Parallel Edges*** *are multiple edges that lay on the same vertices.*

Definition 1.23. *A **loop** is a edge that connects a vertex to itself.*

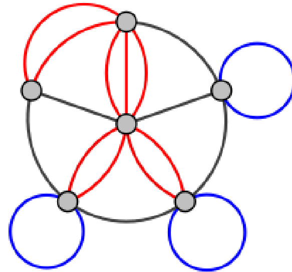


Figure 13: A pseudograph. The blue edges are loops, which make this a pseudograph. If we were to remove those loops, this graph would be a multigraph.

Now that we have introduced these many graphs, we will now go over a few special types of graphs. These graphs are specific, but are extremely common in the real world.

Definition 1.24. *A **Weighted Graph** is a graph that has one or more edges with an n value.*

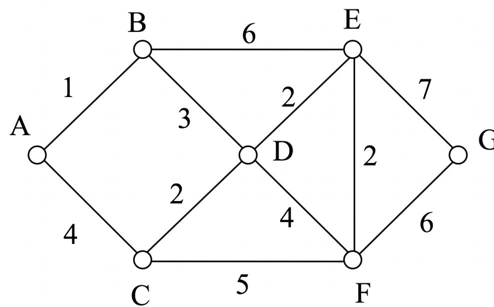


Figure 14: An example of a weighted graph. Each edge has its own specific weight on it.

Definition 1.25. *A **Digraph** is a graph that has edges directed by arrows to certain vertices meaning it can only go in certain directions.*

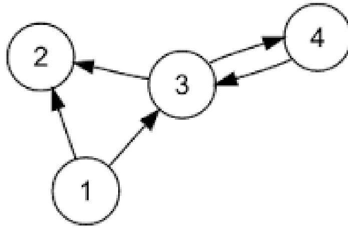


Figure 15: An example of a digraph. Every edge has a specific direction that dictates how one can move across the graph.

Definition 1.26. *A* ***Tree*** *is a connected acyclic graph.*
Note: ***Acyclic*** *means a graph that does not have any cycles.*

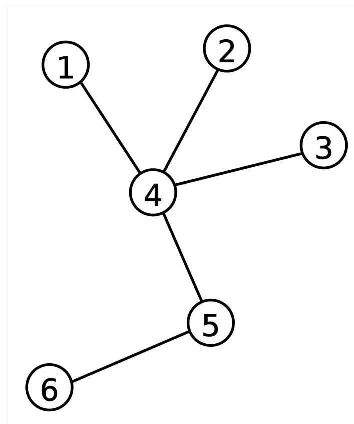


Figure 16: A tree. One can tell by seeing that there are no cycles in the graph.

Trees are an especially important part of graph theory, and even more important in the next section, graph traversability. Having no cycles allows for efficient traversal and streamlining any possible paths one could take, which will be covered in a later section.

2 Graph Traversability

As previously stated in Section 1.1, graph traversability goes back to the Königsberg Bridge Problem and how Euler discovered how it isn't possible to cross every bridge once without repeating one. This section goes more in depth on how this works, the fundamentals, and the concepts that derive from this idea.

2.1 Eulerian Graphs

In the introduction, we modeled Königsberg bridges with a graph to tackle the question of whether it was possible to walk across every bridge once. Since Euler was the first mathematician to study this question, the types of graphs which allow such a walk to be

possible are named after him. In other words, Eulerian Graphs and properties follow the same rules as the Königsberg Bridge Problem: they can't repeat any edges and have to go over every edge of a graph.

Definition 2.1. *A* ***Euler Path*** *is a path that contains all edges of a graph.*

Definition 2.2. *A* ***Euler Circuit*** *is a circuit in a graph G that contains every edge of G.*

Definition 2.3. *A* ***Euler Graph*** *is a graph that contains a Euler circuit.*

Naturally, not every graph has an Euler path or circuit. Thus, a natural question arises about how one could change a graph to ensure that it is an Euler graph. In particular, we introduce the process of Eulerization [2].

Definition 2.4. ***Eulerization*** *is the process of duplicating edges on a graph to create an Euler graph.*

Note that in this process we can only duplicate edges, not create edges where there were not any before. In order to explain the process of eulerizing a graph, there are a few important rules that can help identify an Euler path or circuit:

1. If every vertex has an even degree then the graph has an Eulerian Circuit.
2. If either every vertex has an even degree or exactly two vertices have odd degrees then the graph has an Eulerian path.

Thus, to successfully eulerize a graph, various edges are duplicated in order to connect pairs of vertices with odd degree. By connecting two odd degree vertices, it would thereby increase the degree of each by one, giving them both an even degree. When two odd degree vertices are not directly connected, we can duplicate all edges in a path connecting the two. An example of the Eulerization process is shown in the figure below.

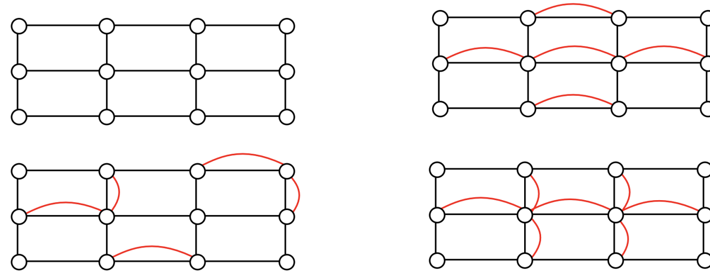


Figure 17: A rectangular graph with three possible eulerizations.

Note that in all three cases, the vertices that started with odd degrees have even degrees after the process, which allows for an Euler circuit.

2.2 Hamiltonian Graphs

Hamiltonian graphs and properties follow similar rules as Eulerian graphs and properties the only differences are: Every vertex has to be crossed and no vertex can be repeated. To define a Hamiltonian Graph, we first need to define a Hamiltonian circuit.

Definition 2.5. *A **Hamiltonian Circuit** is a circuit that crosses over every vertex of a graph with no repeats.*

Now, we can give the following definition for a Hamiltonian Graph:

Definition 2.6. *A **Hamiltonian Graph** is graph which contains a Hamiltonian circuit.*

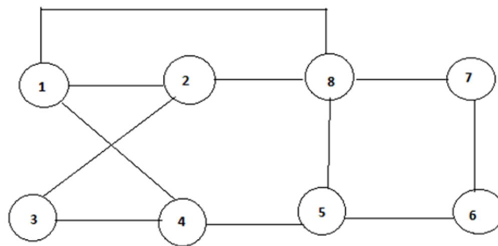


Figure 18: This graph is Hamiltonian because it contains a Hamiltonian circuit.

Furthermore, we can define another term related to traversing Hamiltonian Graphs:

Definition 2.7. *A **Hamiltonian Path** is a path where every vertex of a graph is crossed but does not have to start and end at the same vertex.*

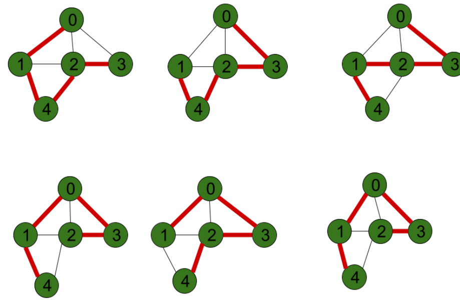


Figure 19: These are the different possible Hamiltonian paths in a graph. As shown, the path does not have to start and end at the same vertex, which makes it not a circuit, but a path.

With Hamiltonian circuits, our focus will not be on existence, but on the question of optimization. Given a weighted graph, we want to find the optimal Hamiltonian circuit:

the one with the lowest total weight. To do so, we will introduce some of the most popular algorithms to find the Hamiltonian Circuit with the least weight. The first and most simple is the Brute Force Algorithm, which is a *greedy algorithm*. A greedy algorithm means it only looks at the immediate decision without considering the consequences in the future [2].

2.2.1 The Brute Force Algorithm

1. Start with a weighted graph.

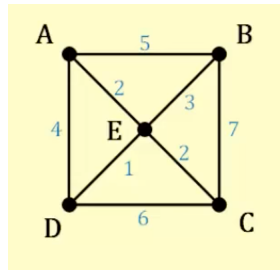
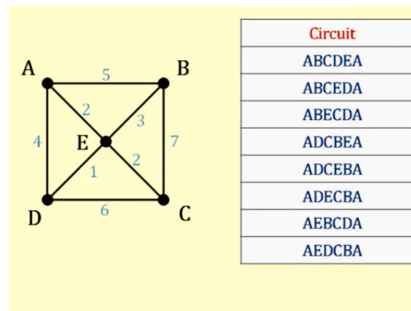


Figure 20: A weighted graph.

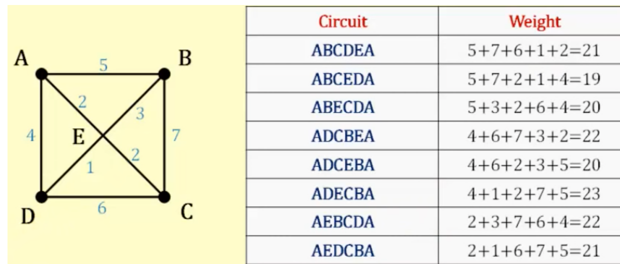
2. Find all of the possible Hamiltonian Circuits in the graph.



Circuit
ABCDEA
ABCEDA
ABECDA
ADCBEA
ADCEBA
ADECBA
AEBCDA
AEDCBA

Figure 21: All of the possible Hamiltonian Circuits of this graph listed in the table.

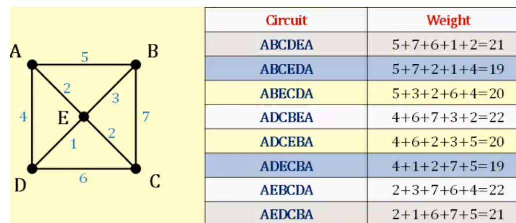
3. Find the length of each circuit by adding the edge weights.



Circuit	Weight
ABCDEA	5+7+6+1+2=21
ABCEDA	5+7+2+1+4=19
ABECDA	5+3+2+6+4=20
ADCBEA	4+6+7+3+2=22
ADCEBA	4+6+2+3+5=20
ADECBA	4+1+2+7+5=23
AEBCDA	2+3+7+6+4=22
AEDCBA	2+1+6+7+5=21

Figure 22: All of the weights of the circuits added up in the table.

4. Choose the Hamiltonian Circuit with the least weight.



Circuit	Weight
ABCDEA	5+7+6+1+2=21
ABCEDA	5+7+2+1+4=19
ABECDA	5+3+2+6+4=20
ADCBEA	4+6+7+3+2=22
ADCEBA	4+6+2+3+5=20
ADECBA	4+1+2+7+5=19
AEBCDA	2+3+7+6+4=22
AEDCBA	2+1+6+7+5=21

Figure 23: We now know that the Hamiltonian Circuits with the least weight in this graph are ABCEDA and ADECBA.

This algorithm is very simple to understand and easy to apply. Obviously, there are many drawbacks, the biggest one being that it is inefficient. This next algorithm helps a little with that issue.

2.2.2 The Nearest Neighbor Algorithm

1. Start with a weighted graph and choose a starting vertex.

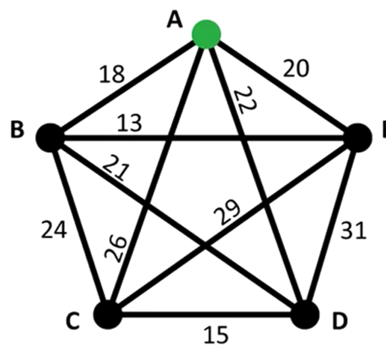


Figure 24: A weighted graph. In this situation we are starting at vertex A.

2. Move to the neared unvisited vertex with the smallest weight.

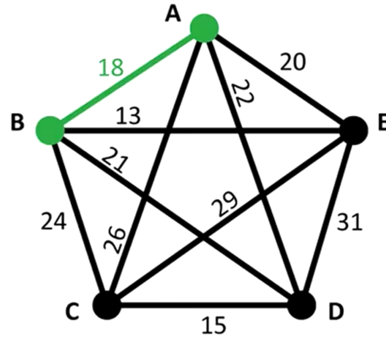


Figure 25: Because the weight of the edge going to vertex B is less than the weight of the edge going to vertex E, we go to vertex B and add it to the circuit.

3. Go to the next nearest unvisited vertex with the smallest weight.

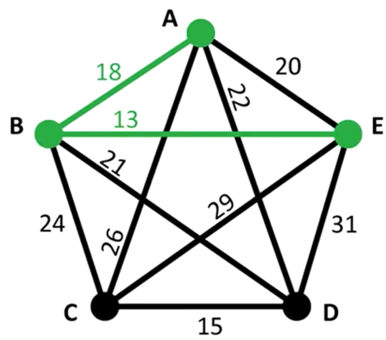


Figure 26: Now, we go to vertex E instead of vertex C or D for the same reason.

4. Repeat until a Hamiltonian Circuit is found.

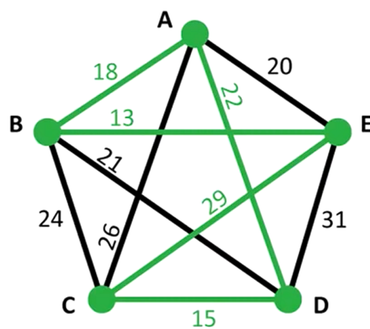


Figure 27: After repeating the prior steps, we now have the Hamiltonian Circuit ABECDA.

This algorithm solves some of the repetitive steps of the Brute Force Algorithm. The only problem is that it may not result in the Hamiltonian Circuit with the least weight. This is because we only start at one vertex, without any knowledge of what Hamiltonian Circuit may be created from starting at another vertex. This next algorithm is extremely similar to the Nearest Neighbor Algorithm, but just repeated.

2.2.3 The Repeated Nearest Neighbor Algorithm

1. Start with a weighted graph.

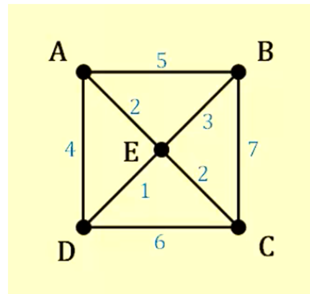


Figure 28: A weighted graph

2. Use the NNA for every possible Hamiltonian Circuit.

RNNA	Weight
AEDCBA	21
BEADCB	22
BECDAB	20
CEDABC	19
DEABCD	21
DECBAD	19
EDABCE	19

Figure 29: Every possible Hamiltonian Circuit in this graph and their weights.

3. Choose the Hamiltonian Circuit with the least weight.

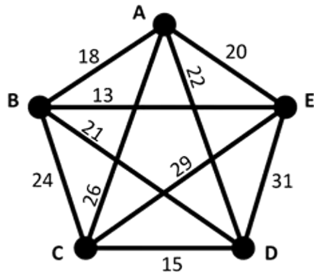
RNNA	Weight
AEDCBA	21
BEADCB	22
BECDAB	20
CEDABC	19
DEABCD	21
DECBAD	19
EDABCE	19

Figure 30: As shown in the table, there are 3 possible Hamiltonian Circuits to choose from.

The Repeated Nearest Neighbor Algorithm solves the issue of NNA of seeing every possible Hamiltonian Circuit and choosing the one with the least weight. But, this reverts back to the issue of the Brute Force Algorithm, meaning that it is inefficient and tedious. The Sorted Edges Algorithm uses a different approach to find a more efficient way.

2.2.4 The Sorted Edges Algorithm (a.k.a Cheapest Link Algorithm)

1. Start with a weighted graph and list out its edges.



(a) A weighted graph.

B-E 13
C-D 15
A-B 18
A-E 20
B-D 21
A-D 22
B-C 24
A-C 26
C-E 29
D-E 31

(b) The weights of all of the edges listed out.

2. Select the smallest unused edge in the graph.

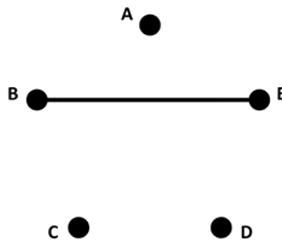


Figure 32: Because edge BC has the least weight, we add it to the circuit first.

3. Choose the next smallest edge in the graph and add it to the circuit. Be careful not to add any edges that will create another circuit that doesn't contain all of the vertices or give any vertex a degree of 3.

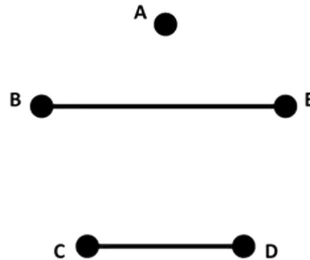
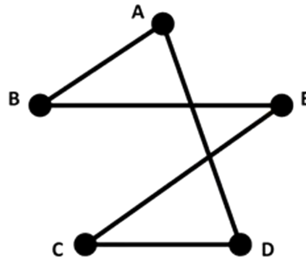


Figure 33: The next edge is added to the circuit.

4. Repeat until a Hamiltonian Circuit is created.



Here, we have the final Hamiltonian Circuit. As you can see, no vertex has a degree of 3, and there are no other circuits within the Hamiltonian Circuit we made.

The many algorithms that can be used to find the Hamiltonian Circuit are all different in their own ways. The Brute Force Algorithm, which is a greedy algorithm, is true to its name in the way that it finds all possible combinations and compares them, which is very inefficient. The Nearest Neighbor Algorithm uses a more structured strategy to find the circuit, which is a little better. The Repeated Nearest Neighbor Algorithm just repeats the Nearest Neighbor Algorithm, which is more accurate but less efficient. The Sorted Edge Algorithm takes a completely different approach, using edges as a way to find the Hamiltonian Circuit. It may seem like the best method, but the Sorted Edge Algorithm has a few downsides. It can be hard to keep track of larger graphs, and it can be quite tedious depending on the graph being solved.

3 The Shortest Path Problem

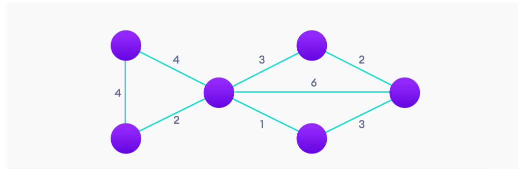
The shortest path problem is a prominent issue in graph theory. As the name states, it seeks to find the shortest path from one vertex to another. It is important to find the

shortest path to maximize efficiency when traveling to another vertex. One of the main ways to find the shortest path is with the use of Dijkstra's Algorithm.

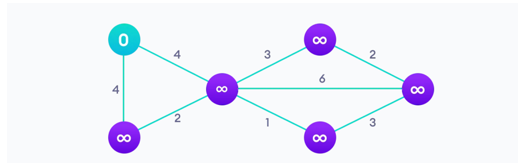
3.1 Dijkstra's Algorithm

Describe how Dijkstra's algorithm is used to find solutions to the shortest-path problem Dijkstra's Algorithm is one of the most popular methods of finding the shortest path, and it consists of many intuitive steps.

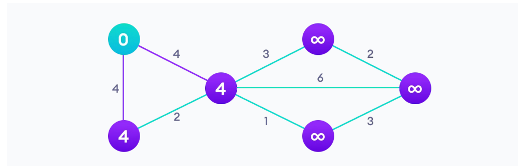
1. Start with a weighted graph.



2. Choose a starting vertex and assign infinite path values to all other vertices.

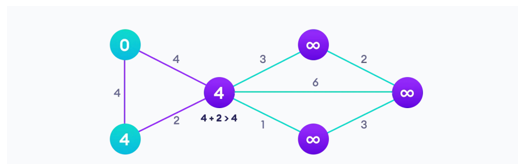


3. Go to the connected edges and update the path length.



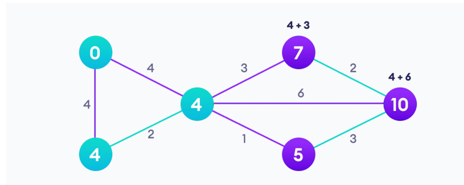
   In this example, the connected edges have a weight of 4, so the path length to both neighboring vertices is 4.

4. If the path length of the adjacent vertex is lesser than the new path length, don't update it.



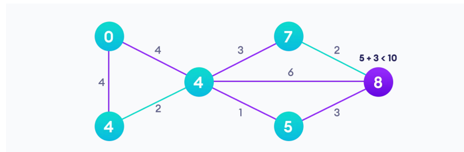
As shown in the figure, there is a lesser weight path to the vertex, so we choose the one with less weight.

5. Do not update any path lengths for vertices you have already visited.



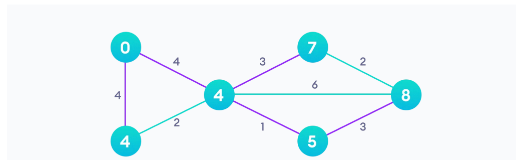
The blue vertices have already been visited, so we do not update their path lengths anymore.

6. When going to another vertex, go to the one with the least path length.



As shown above, since 5 has less weight than 7, we go to 5 first.

7. Repeat until every vertex has been visited.



From the starting vertex, the purple edges represent the shortest path to every other vertex in the graph. The edges that are still blue represent the unvisited ones.

This algorithm can seem very complex, but the most important aspect of this algorithm is to keep track of the updated shortest paths. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

3.2 Real Life Applications

The shortest path problem makes very common appearances in computer science and coding. The main issue that this deals with in the real world is finding the shortest or fastest way to travel from one place to another. For example, this could be finding the shortest way to travel from Lyon to Berlin.

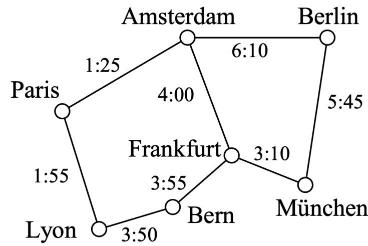


Figure 34: An example of a situation where Dijkstra's Algorithm would be useful.

Usually, taking the most efficient route is the best route, and Dijkstra's Algorithm does a great job of finding the shortest path.
Not only is Dijkstra's Algorithm good for finding the shortest physical distance, it can also effectively help with social networking. Social media apps often suggest other people to follow, and they use the help of the shortest path to recommend people who you would most likely have a connection with. In this case, it is not a physical distance separating people, but people who share mutual connections. Dijkstra's Algorithm allows people to effectively network with new people and play a huge role in the growth of social media apps.

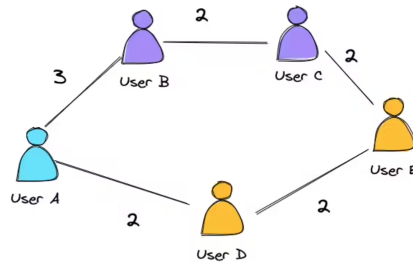


Figure 35: A social network and how Dijkstra's Algorithm can be applied to it.

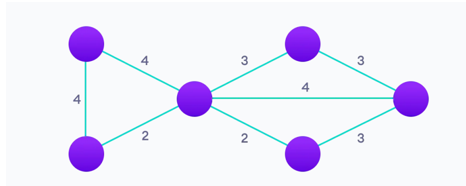
4 The Minimum Spanning Tree Problem

After exploring Dijkstra's Algorithm and its applications in finding the shortest path in a graph, it is natural to consider its relation to another important concept in graph theory: tree graphs. While Dijkstra's Algorithm is a powerful tool in its own right, it is not directly applicable to find the *minimum spanning tree* in a graph. A *tree* is a graph without any *cycles*. A *minimum spanning tree* is a tree that contains all of the vertices of the graph *and* minimizes that total weight of the edges included in the graph. In the following paragraphs, we will explore the concept of minimum spanning trees and 2 algorithms used to solve this problem.

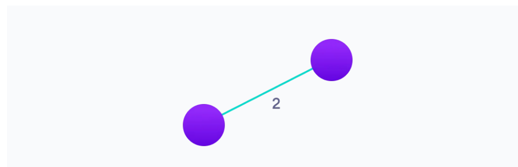
4.1 Kruskal's Algorithm

Kruskal's Algorithm uses edges to find the minimum spanning tree. The steps are intuitive and relatively easy to follow.

1. Start with a weighted graph.

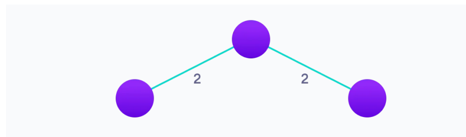


2. Choose the edge with the least weight and add it. If there are multiple, choose any.



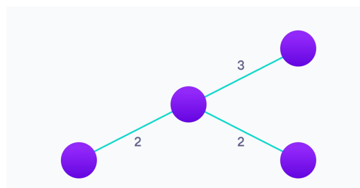
   Since the edge with the lowest weight has a weight of 2, we add it to the tree.

3. Choose the next edge of minimum weight and add it.

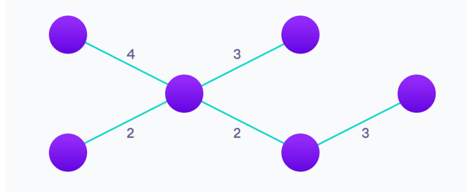


   The next smallest edge also has a weight of 2, so we add it to the tree.

4. Choose the next smallest edge except if that edge creates a cycle, in which case you choose the



5. Repeat until the minimum spanning tree is created.

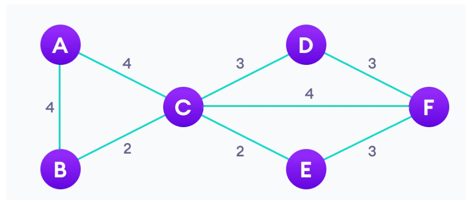


This algorithm is effective in finding the minimum spanning tree is most graphs. But, the biggest drawback is that it becomes hard to keep track of every edge when analyzing a graph with many edges. This is where Prim's Algorithm is stronger.

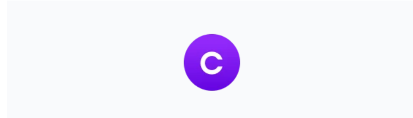
4.2 Prim's Algorithm

Prim's Algorithm uses vertices to find the minimum spanning tree. The approach is a little different from Kruskal's Algorithm, but they both build on similar concepts.

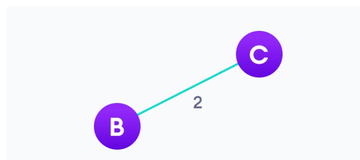
1. Start with a weighted graph.



2. Choose a random vertex.

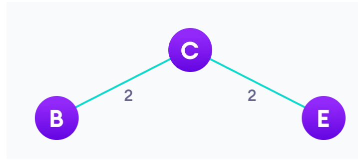


3. Choose the edge of minimum weight of this vertex and add it to the tree.



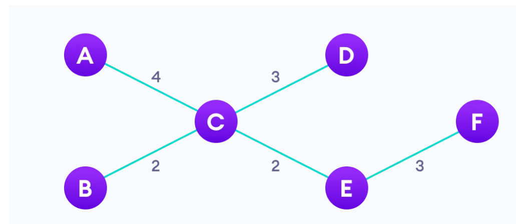
   This edge was chosen because it has the least weight out of all the edges of vertex C.

4. Choose the nearest vertex not yet in the tree.



This edge was chosen because it has the minimum weight.

5. Repeat until you have the minimum spanning tree



Prim's and Kruskal's Algorithms are very similar in the way that they find the minimum spanning tree. The biggest difference between them is that Kruskal's Algorithm starts the tree from 2 vertices, while Prim's Algorithm starts with 1. As a result of this, Prim's Algorithm is much better suited to find the minimum spanning tree in larger graphs because the actual steps are more complex. However, Kruskal's Algorithm is stronger in finding the minimum spanning trees in smaller graphs because of its simplicity.

4.3 Real Life Applications

Now that we have covered the two main ways to find the minimum spanning tree, we will go over the main uses of minimum spanning tree. The most prominent use of finding the minimum spanning tree is through networks. The original use of finding the minimum spanning tree was to make the most efficient electrical grid. It is important to minimize the costs of supplying electricity to every house and finding the minimum spanning tree solves that problem. Another example of a network is a telecommunication network. These also use the help of minimum spanning trees to allow for the fastest and lowest cost communication between two locations.

References

[1] Gary Chartrand and Ping Zhang. *A First Course in Graph Theory.* McGraw-Hill Higher Education, 2012. ISBN: 9780486483689.

[2] David Lippman. *Math in Society.* CreateSpace Independent Publishing Platform, 2012. Chap. 6. ISBN: 9781479276530.