The little book of graph notes

Graph theory students

April 28, 2020

Contents

1 On Graphs **7**
- 1.1 What is a Graph? 7
- 1.2 Why do we care about Graphs? 7
- 1.3 On Different Types of Graphs 8
  - 1.3.1 Multigraphs 8
  - 1.3.2 Digraphs 8
- 1.4 On Vertices and Edges 9
- 1.5 On Isomorphisms 9
  - 1.5.1 The Handshaking Lemma 10
- 1.6 On Special Kinds of Graphs 11
- 1.7 On the Storage of Graphs 11

2 Forest for the Trees **13**
- 2.1 Definitions 13
- 2.2 Trailblazing 15
- 2.3 The Lorax 17
- 2.4 (Minimum) Spanning Trees 18
  - 2.4.1 Prim's Algorithm 19
  - 2.4.2 Kruskal's Algorithm 20
  - 2.4.3 Borůvka's Algorithm 21
- 2.5 Many Branching Theorems 22

3 Single Source Shortest Path **25**
- 3.1 Dijkstra 25
- 3.2 Bellman-Ford 29

4 Eulerian and Hamiltonian **35**
- 4.1 7 Bridges o of Königsberg 35
- 4.2 Eulerian 35
  - 4.2.1 Hierholzer's Algorithm 35
  - 4.2.2 Fleury's Algorithm 36
- 4.3 Hamiltonian 36
  - 4.3.1 Examples 36
- 4.4 Theorems 38

- 4.4.1 Ore's Theorem (1960) . . . 38
- 4.4.2 Las Vergnas (1971) . . . 38
- 4.4.3 Chvátal(1970) . . . 38
- 4.4.4 Bandy(1969) . . . 38
- 4.4.5 Pŏsa (1962) . . . 38
- 4.4.6 Dirac (1962) . . . 39
- 4.4.7 Bandy and Chvátal(1976) . . . 39

5 NP-Completeness 43
- 5.1 Definitions . . . 43
- 5.2 Founding Papers . . . 44
  - 5.2.1 Cook-Levin Theorem (1971) . . . 44
  - 5.2.2 Karp's 21 Problems (1972) . . . 45
- 5.3 Problems . . . 45
  - 5.3.1 Boolean Satisfiability . . . 45
  - 5.3.2 3-Satisfiability . . . 46
  - 5.3.3 Directed Hamiltonian Cycle . . . 46
  - 5.3.4 Clique . . . 50
  - 5.3.5 Vertex Cover . . . 51

6 Pigment of your Imagination 53
- 6.1 Coloring your Graphs . . . 53
- 6.2 On the Rules of Coloring . . . 53

7 Grin and Berg it 57
- 7.1 Grinberg's Theorm(1976) . . . 57
- 7.2 Diagonals . . . 57
- 7.3 Herschel Graph . . . 58

8 Plane and Simple 59
- 8.1 Euler's Formula (1750) . . . 59
- 8.2 Lemma . . . 59
- 8.3 Kuratowski's Theorm (1930) . . . 59
- 8.4 Wagner's Theorm(1937) . . . 59
- 8.5 Minors . . . 60

9 Graph Classes 61
- 9.1 Bipartite Graphs . . . 61
  - 9.1.1 Definitions . . . 61
  - 9.1.2 When Can We Use These? . . . 63
  - 9.1.3 How Does This Help Us? . . . 63

Preface

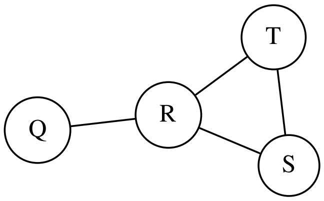
This compilation of notes exists as a project for students in the CMSC 420 Graph Theory course at Longwood University. These notes have been created by all the students of the class, and an attempt has been made to both categorize the notes in a logical, sequential manner, and to ensure that the information is clear yet concise.

Chapter 1

On Graphs

1.1 What is a Graph?

GRAPHS, in their elementary form, are a pair of sets: V (**Vertices**) and E (**Edges**) such that $V \neq \emptyset$ (V cannot be empty), and $\forall e \in E, e = \{v_i, v_j\}, v_i, v_j \in V$ (every edge in E is a pairwise subset of vertices v_i and v_j in V). Here is an example of a very simple graph:



In this graph, our vertex set is $V = \{Q, R, S, T\}$, and our edge set is $E = \big\{\{Q, R\}.\{R, S\}, \{R, T\}, \{S, T\}\big\}$.

Other names for Vertices are **nodes** or **points**. Other names for edges are **links** or **lines**. When we want to refer to the vertex set of a specific graph G, we will denote it as $V(G)$. Similarly, we will denote the edge set as $E(G)$.

1.2 Why do we care about Graphs?

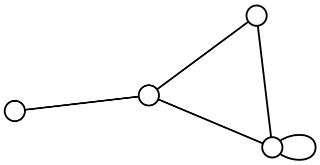
Graphs can be used to model many real world problems, or theoretical ideas. For example, graphs can be used to represent relationships between different objects or groups. They are useful for visualizing data, such as social media connections, or network topology. Additionally, a more practical use is for mapping routes from one location to another, much like the GPS on your phone.

As for more theoretical uses, graphs can represent other structural relationships. Tree structures are a type of graph, as are flow diagrams or DFAs/NFAs from Computation Theory. Perhaps one of their more important uses is in determining where problems fit in P vs. NP.

1.3 On Different Types of Graphs

1.3.1 Multigraphs

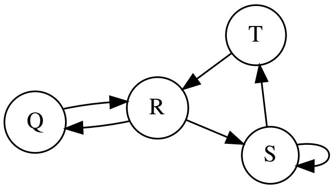
A **multigraph** is a pair of sets (V, E) such that $V \neq \emptyset$, and E is a finite family of unordered pairs of V. That is to say, edges are not necessarily unique subsets, and do not have order. Thus, multigraphs allow for loops from one vertex to itself. Here is an example of a multigraph:



Because multigraphs are unordered, then an edge (a, b) would be the same as (b, a). But what if we only want an edge in one direction but not the other?

1.3.2 Digraphs

A **digraph** is a pair of sets (V, E) such that $V \neq \emptyset$, and E is a set of ordered pairs of elements in V. Informally, we say that a digraph is a directed graph, where we annotate each edge with an arrow to mark the direction it flows. Here is an example of a digraph:



1.4 On Vertices and Edges

There is a lot to be said of the mathematics of graphs, but before we can establish such mathematics, we must first establish some essential definitions. Let G be a graph with vertex set V and edge set E. Let $u, v, w \in V$ be vertices of G.

Join: If $\{u, v\} \in E$, then u is said to **join** v.

Adjacent: u and v are said to be **adjacent** if $\{u, v\} \in E$. Similarly, if $\{u, v\}, \{u, w\} \in E$, then $\{u, v\}$ and $\{u, w\}$ are said to be **adjacent**.

Incident: If $\{u, v\} \in E$, then u and v are said to be **incident** to $\{u, v\}$. Similarly, if $\{u, v\}, \{u, w\} \in E$, then $\{u, v\}$ and $\{u, w\}$ are said to be **incident** to u.

Degree: The **degree** of a vertex, denoted $\deg(v)$ or $\rho(v)$, is the number of edges incident to v.

Isolated: If $\deg(v) = 0$, then we say v is an **isolated** vertex.

Pendant: If $\deg(v) = 1$, then we say v is a **pendant** vertex.

These are a lot of formal definitions, but they will be useful when classifying different graphs and discussing their structures and properties. Informally, we say two edges are adjacent if they share a vertex, and two vertices are adjacent if they share an edge. Similarly, we say that two edges are incident to a vertex if they share that vertex, and two vertices are incident to an edge if they share that edge.

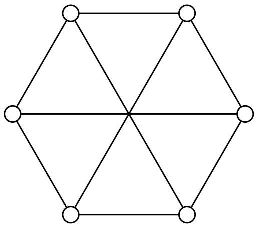
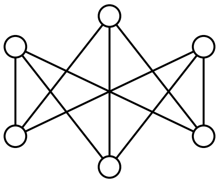
1.5 On Isomorphisms

The term *isomorphic* has its roots in Greek, *iso-* meaning same, and *morphic* referring to form or shape. We say that two graphs are **isomorphic** if they have this "same shape" property, but what does that mean?

Mathematically, we say that the graph G_1 is **isomorphic** to the graph G_2, denoted $G_1 \sim G_2$, if there is a one-to-one correspondence between the vertices V_1 of G_1 and the vertices V_2 of G_2 that preserves adjacencies. This is a mapping f from G_1 to G_2 defined as follows:

$$f : G_1 \mapsto G_2 \text{ such that } \forall \{v_i, v_j\} \in E(G_1), \{f(v_i), f(v_j)\} \in E(G_2)$$

This may look like a lot of intimidating gibberish, but it represents the idea of preserving the edges between two vertices in G_1 after mapping them to G_2. For example, the following graphs are isomorphic:

Isomorphisms are a mapping, and thus have certain propoerties that allow us to classify them with other mappings. Let G_1, G_2, and G_3 be graphs. Isomorphisms are what we call an equivalence relation, as the following three propoerties hold:

Reflexive: $G_1 \sim G_1$

Symmetric: If $G_1 \sim G_2$, then $G_2 \sim G_1$

Transitive: If $G_1 \sim G_2$ and $G_2 \sim G_3$, then $G_1 \sim G_3$

1.5.1 The Handshaking Lemma

We are now armed with enough information to decipher some information hidden in the graphs we deal with. The first of these is called the Handshaking Lemma. Let's tackle this proof:

Lemma 1.1 (The Handshaking Lemma)**.** *Let G be a graph. Then the number of vertices with odd degree is even.*

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the vertices of G. The the sum of the degrees,

$$\sum_{i=1}^{n} \deg(v_i) = 2m, m \in \mathbb{Z},$$

is even, since each edge is counted twice (convince yourself that this must be true, if you don't understand). Now, if we were to remove all vertices of even degree, then we would be removing an even number of edges from this sum. This leaves us with

$$\sum_{odd\ degrees}^{n} \deg(v_i) = 2k, k \in \mathbb{Z},$$

which is even. Now, the only way for the sum of odd numbers to be even is for there to be an even number of those odds. Thus, we must have an even number of odd degree vertices in G. ∎

1.6 On Special Kinds of Graphs

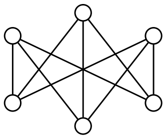
We've talked a lot about the different aspects of graphs, but now let's use this information to actually classify different kinds of graphs, starting with the **null graph**. the null graph is a graph with n vertices but no edges, and is denoted as N_n.

A **regular graph** is a graph G with n vertices whose vertices all have the same degree, and we say such a graph is regular of degree $r < n$.

A **complete graph** is a graph G with n vertices that is regular of degree $n-1$. We denote such a graph by K_n. The total number of edges in the complete graph K_n is given by summing the number of edges:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n-2)}{2}$$

A **bipartite graph** is a graph with vertex set $V = V_1 \cup V_2$ where $V_1 \cap V_2 = \emptyset$ and $\forall v_i, v_j \in V_1$ and $v_m, v_n \in V_2$, $(v_i, v_j), (v_m, v_n) \notin E$. This is another way to say that the vertices of a bipartite graph G can be divided into two sets there all edges in G only go between both sets - neither set of vertices has two vertices which make an edge. A complete bipartite graph is denoted $K_{m,n}$, with two sets containing m and n vertices, with all vertices in one set connecting to all the vertices of the other. Here is an example of a bipartite graph, $K_{3,3}$:

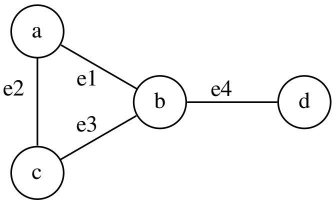


A **subgraph** of $G = (V, E)$ has the vertex set H and the edge set F, where $H \subseteq V$ and $F \subseteq E$, but only the edges of E which correspond to the vertices in H can be in F.

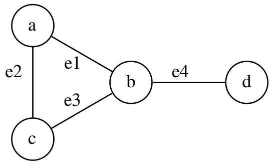
1.7 On the Storage of Graphs

We've talked a lot about graphs and their theories and applications, but how exactly do we represent graphs in a computer? The idea that a graph defines a set of relations in important, so the natural way to consider this is some sort of data structure with nodes which point to their adjacent neighbors. This certainly is an intuitive way to consider this, but examining them mathematically will make our lives easier in the future.

Consider the following graph:



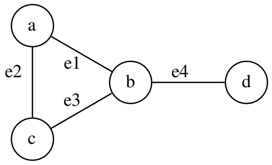
We can store this in what's called an **Adjacency Matrix**, a matrix where each row and column represents a particular vertex in the graph, and entries in the matrix signify that two vertices share an edge. The adjacency matrix for the given graph is:



$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

We usually denote an adjacency matrix $[A_{i,j}]$, where a 1 in entry $a_{i,j}$ identifies that vertices i and j share an edge, and a 0 means that they don't. While adjacency matrices for a graph with n vertices occupies n^2 space, they do provide constant lookup time to see if two vertices share an edge.

We also have an **Incidence Matrix**, which tells us if an edge and a vertex are incident. The incidence matrix for the graph above is:



$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

It might be preferable to use the adjacency matrix for the symmetry along the main diagonal of the matrix.

Chapter 2

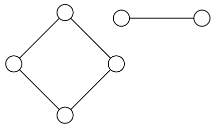
Forest for the Trees

TREES are another special kind of graph which we did not discuss in the previous chapter. Trees are somewhat unique compared to the other types of specialty graphs described before as they allow us certain algorithms to perform to solve different problems. Before we go in depth into trees, we must first define an operation and some other terms.

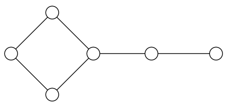
2.1 Definitions

Let G_1 and G_2 be graphs. Then $G_1 \cup G_2$ is the **union** of G_1 and G_2 with the vertex set $V = V(G_1) \cup V(G_2)$ and edge set $E = E(G_1) \cup E(G_2)$. Now, let's establish some more definitions:

Disconnected: A graph is **disconnected** if it is the union of two graphs. Here is an example of a disconnected graph:

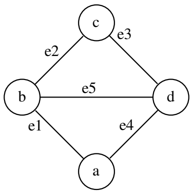


Connected: A graph is **connected** if it is not disconnected (wow). Here is an example of a connected graph:

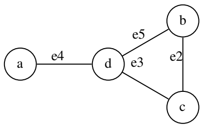


Component: A **component** graph is one graph which makes up part of a union. In $G = G_1 \cup G_2$, G_1 and G_2 are components.

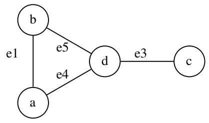
We can also define the **subtraction** of a set of edges F from a graph G. We say that $G - F$ is the graph resulting in the removal of the edges in F from G. Suppose we have the following graph G:



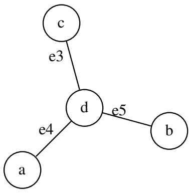
Then the following are different subtractions from G.



(a) $G - \{e1\}$

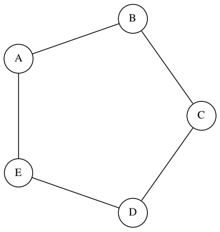


(b) $G - \{e2\}$

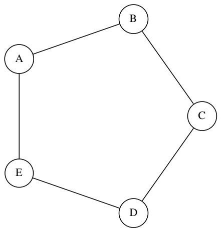


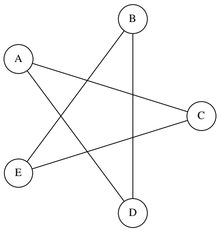
(c) $G - \{e1, e2\}$

Let's define a new type of graph. A connected graph that is regular of degree 2 is called a **cycle**, which we denote C_n. Here is an example of a cycle, C_5:



A useful operation on graphs is to take the complement. The **complement** of a graph G is the graph $\overline{G}$, with vertex set $V(\overline{G}) = V(G)$, with two vertices adjacent in $\overline{G}$ if and only if they are *not* adjacent in G. Given C_5, we can fine its complement, $\overline{C_5}$:

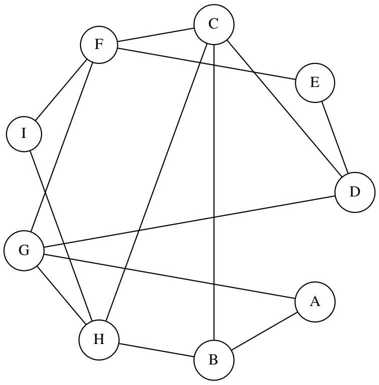




In this particular example, notice that $C_5 \sim \overline{C_5}$, so C_5 is what we call **self-complementary** - it is isomorphic to its own complement.

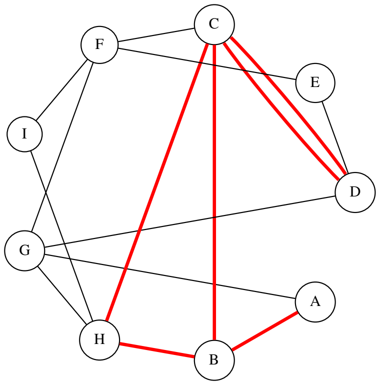
2.2 Trailblazing

We've defined what graphs are as a construct, but how do we actually navigate from one point to another? There are different ways to navigate from vertex to vertex, and each form of navigation takes on different attributes. Consider the following graph G_0:



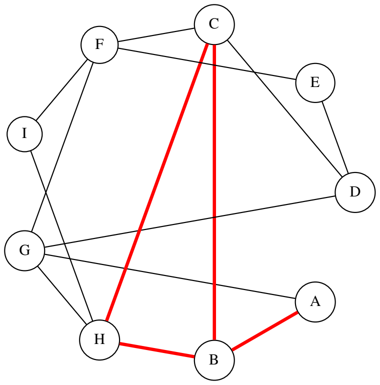
Let's define some of the different ways we can navigate through G_0:

Walk: A **walk** in a graph G is any finite sequence of edges, $v_0v_1, v_1v_2, v_2v_3, \ldots, v_{k-1}v_k$. The initial vertex is denoted v_0, and the final vertex is denoted v_k. The **length** of the walk is k. Here is an example of a walk in G_0:



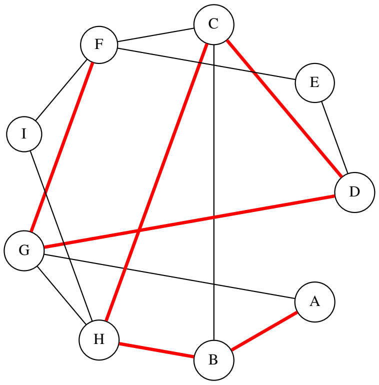
Here, the walk shown begins at node A. The walk is AB, BH, HC, CD, DC, CB. The length of this walk is 6.

Trail: A walk where all the *edges* are distinct is a **trail**. The walk observed above is not a trail, since the edge $\{C, D\}$ is repeated. Thus, a variation of this as a trail might be:



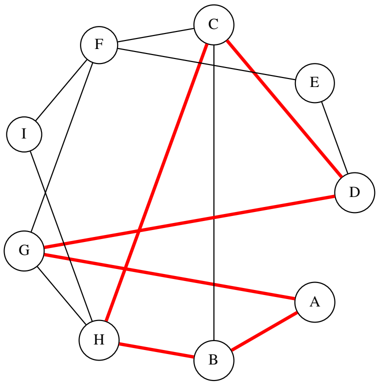
The trail here is AB, BH, HC, CB.

Path: A trail where all the *vertices* are unique is called a **path**. The trail above would not be a path, as the vertex B is visited twice. Here is an example of a path in G_0:



This path is AB, BH, HC, CD, DG, GF.

Cycle: A **closed** path (where $v_0 = v_k$) has at least one edge, then it is a **cycle**. Here is an example of a cycle in G_0:

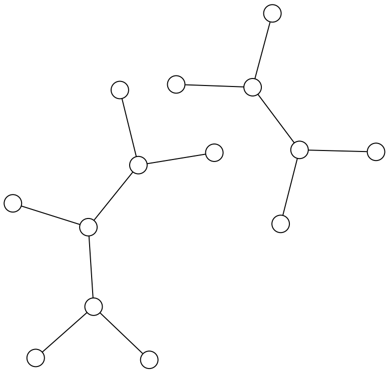


This cycle is AB, BH, HC, CD, DG, GA.

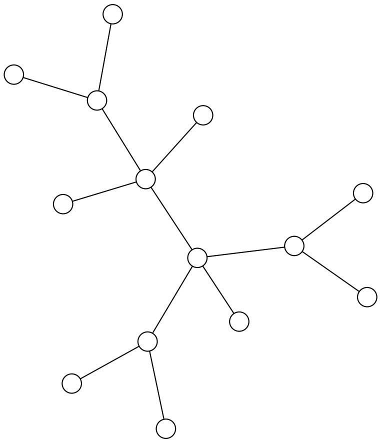
2.3 The Lorax

We have the terminology we need to finally describe the subjects of this chapter: trees and forests.

Forest: A **forest** is a graph with no cycles. It need not necessarily be connected, so long as there are no cycles. Here is an example of a forest:



Tree: A **tree** is a connected forest. Here is an example of a tree:



Trees come in many different shapes and sizes, the count of which grows quickly as more nodes are introduces. For example, there is only one tree of one, two, and three nodes. Once we introduce a fourth node, we see that there are two distinct trees of four nodes. The more nodes we introduce, the more rapidly the count of distinct isomorphisms increases.

2.4 (Minimum) Spanning Trees

While fundamentally trees and forests are just different kinds of graphs, they are useful tools when used in conjunction with graphs. One example of this is the **spanning tree** of a graph. The spanning tree of a graph G is a tree which includes all vertices of the graph G. Another way to imagine this is a non-cyclic representation of G. On top of spanning trees, we have **minimum spanning trees** (MSTs). The MST of a graph G is a spanning tree whose whose total weight is minimal compared to all spanning trees. First, let's define what a weight is.

A **weight** is a numerical value assigned to an edge. This number is representative of the total cost of traversing from one vertex to another along that particular edge - some edges may be cheaper, others may be longer. We assume that these weights are strictly non-negative (so for a weight $n, n \in \mathbb{Z}^*$). The total weight of a graph is the sum of the weights of all edges.

Again, the MST for a graph G is a spanning tree with minimal weight. How do we find the minimum spanning tree for a graph? There are three popular algorithms which solve this problem: Prim's Algorithm, Kruskal's Algorithm, and Borůvka's Algorithm.

2.4.1 Prim's Algorithm

Prim's Algorithm was originally developed by Czech mathematician Vojtěch Jarník in 1930, but was later published by Robert Prim and Edsger Dijkstra in 1959[3]. Prim's algorithm focuses on the vertices of a graph, and considers edges incident to the vertices in the MST as it grows. Below is a quick pseudocode representation of this algorithm[1].

Algorithm 1: Prim's Algorithm

```
Data: Weighted graph $G$, root vertex $v_0$
Result: Minimum spanning tree $T$
for $v \in V(G)$ do
    Weight($v$) $\leftarrow \infty$
    Parent($v$) $\leftarrow$ null
end
Weight($v_0$) $\leftarrow 0$
$Q \leftarrow V(G)$
while $Q \neq \emptyset$ do
    $u \leftarrow$ GetMinVertex($Q$)
    foreach $v \in$ AdjacentTo ($u$) do
        if $v \in Q$ and EdgeWeight($u, v$) $<$ Weight($v$) then
            Parent($v$) $\leftarrow u$
            Weight($v$) $\leftarrow$ EdgeWeight($u, v$)
        end
    end
end
```

Given a root node v_0 to begin the construction of the MST, the objective for the algorithm is to construct a tree based on weights from nodes already examined. The preliminaries of this algorithm look very much like Dijkstra's algorithm (discussed later), where we initialize the weights of all vertices to infinity, and the root node to zero. Additionally, we want to keep track of the edge through which we reach the vertex, so we initialize the parent to null.

We use Q as a minimum heap structure to grab the smallest weighted vertex, calling it u. The idea is to examine all adjacent vertices to u, and update their current weight if following the edge from u to said vertex is better than what was found previously. If so, we set the parent of that vertex to u, and update the weight.

Once Q is exhausted, we will have determined the minimal edges connecting each vertex in the graph, resulting in the MST. This formalization of the algorithm is a bit dense, so the simplified version of what was discussed in class is as follows:

First, grab a vertex v at random from a graph G, and initialize a tree T as empty, and a set P with v. P will be our set of visited vertices, and T will be the collection of edges which make the MST. Find the minimum weighted edge connecting a vertex in P to a vertex not in P, and add this edge to T and the vertex to P. Repeat until $P = V(G)$. The collection of edges T will be the MST.

This algorithm is greedy - at each decision point, we follow the smallest available edge available. This results in the minimum total weight of the spanning tree, and thus gives us the MST. If we use a min-heap to store the weights of the edges of vertices in the tree, then this algorithm achieves a complexity of $O(E\ log(V))$.

2.4.2 Kruskal's Algorithm

Kruskal's Algorithm was devised the mathematician and computer scientist Joseph Kruskal in 1956[5]. Unlike Prim's algorithm, which focused on the vertices of a graph, this algorithm focuses on aggregating edges in a graph until we achieve a MST. Below is the pseudocode for Kruskal's.

Like Prim's Algorithm, Kruskal's is greedy, examining the lowest weighted edges sequentially. The objective of this algorithm is to combine subtrees in the graph until one tree spans the totality of the graph, and through its construction, is minimally weighted.

Initially, we initialize all vertices to be their own subtrees. Additionally, we store the edges in a min-heap Q so that we can examine the lowest weighted edge. As we exhaust the list of edges, we want to examine each one and, if the edge joins two separate and disjoint subtrees, union them and store the edge in the MST A. Otherwise, if the vertices incident to the edge are from the same tree, we discard that edge. At the termination of the algorithm, we will have joined all subtrees together using the smallest weighted edges possible to do so, thus resulting in a MST.

The run time for this algorithm depends on the implementation of how we store the subtrees of the graph. Ideally, this will be some form of disjoint-set data structure, with good management for finding and taking the union of sets. A good implementation will result in a worst case runtime of $O(E\ log(V))$.

Algorithm 2: Kruskal's Algorithm

```
Data: Weighted graph $G$
Result: Minimum spanning tree $T$
$A \leftarrow \emptyset$
for $v \in V(G)$ do
    $A \leftarrow A \cup \{v\}$
end
$Q \leftarrow E(G)$
while $Q \neq \emptyset$ do
    $e(u, v) \leftarrow$ GetMinEdge($Q$)
    if TreeWith($u$) $\neq$ TreeWith($v$) then
        JoinSets($u, v$)
        $A \leftarrow A \cup e$
    end
end
```

2.4.3 Borůvka's Algorithm

Borůvka's Algorithm was perhaps the first MST algorithm, first published by Otakar Borůvka in 1926 for finding more efficient power line routes[2]. This algorithm looks very much like Kruskal's algorithm, focusing on edges joining different subtrees which are initially solely vertices. However, Kruskal's algorithm works very well in a parallel computing environment. The pseudocode is shown below.

Algorithm 3: Borůvka's Algorithm

```
Data: Weighted graph $G$
Result: Minimum spanning tree $T$
$A \leftarrow \emptyset$
for $v \in V(G)$ do
    $A \leftarrow A \cup \{v\}$
end
$Q \leftarrow E(G)$
while SizeOf($A$) $> 1$ do
    foreach $e(u, v) \in E(G)$ do
        if TreeWith($u$) $\neq$ TreeWith($v$) then
            JoinSets($u, v$)
            $A \leftarrow A \cup e$
        end
    end
end
```

The idea here is to repeatedly join subtrees until the MST remains. To accomplish this, the algorithm calls for repeatedly finding minimally weighted edges while more than one tree remains, in much the same way that Kruskal's algorithm did. However, in a parallel environment, we can delegate the task of joining multiple trees to different systems, so that the time of each iteration can be reduced. In a non-parallel environment, this algorithm is shown to have the same complexity as Kruskal's, $O(E\ log(V))$.

2.5 Many Branching Theorems

For a more theoretical approach to trees, we can start to approach the idea of equivalent criteria in which a graph is actually a tree. First, let's define what a bridge is. A **bridge** is an edge from a graph which, when removed, increases the number of components in a graph. Consider the following example graph, where removing the edge e makes this a disconnected graph

An important and useful theorem arises from introducing the idea of a bridge.

Theorem 2.1 (Graph Bridge Theorem). *Let G be a graph, and $e \in E(G)$. Then e is an edge of a circuit if and only if e is not a bridge.*

Proof. Let G be a graph and $e \in E(G)$.
($\rightarrow$) Assume that e is an edge in a circuit. We aim to show that e is not a bridge. If e is an edge between vertices u and v, then by definition of a path, e is a final edge in the circuit which closes the path from u to v. Then following the removal of e, there will still be a path from u to v. Thus the removal of e does not increase the count of components, and e can't be a bridge.
($\rightarrow$) Assume that e is not a bridge. Let u and v be the vertices incident to e. If e is not a bridge, then its removal will not increase the number of components in G. Then by definition of connectivity, there must still exist a path between u and v. Therefore, e must have been an edge in a circuit from u to v. This completes the proof. ■

Having proved this theorem, we are in a position to begin proving another, rather large, theorem about the equivalent conditions for a graph to be a tree.

Theorem 2.2. *Let T be a graph with n vertices. Then the following are equivalent:*

1. *T is a tree.*
2. *T has no circuits, but has $n - 1$ edges.*
3. *T is connected and has $n - 1$ edges.*
4. *T is connected, and every edge is a bridge.*
5. *Any given pair of vertices of T is connected by exactly one path.*
6. *T has no circuits, but introducing any one new edge joining two vertices of G creates exactly one new circuit.*

Proof. For proving a theorem of this form, the goal is to sequentially assume one statement, and use that assumption to prove the following. Let's begin:
($1 \to 2$) Assume that T is a tree. Then by definition, T is connected and has no circuits. We must now show that T has $n-1$ edges, and we will do this via induction. The goal will be to show that, by removal of one edge, we will create two components which satisfy the hypothesis, which will show that the original graph must have also satisfied the hypothesis.
For our base case, assume that T has two vertices, u and v, with one edge connecting them. If we remove this edge, then we have two components, each with 1 vertex, and $n-1=0$ edges and thus no circuits. This satisfies (2).
Let's assume that this holds for a graph of up to $n-1$ vertices. We must now show it for a graph with n vertices. If T has no cycles, then by the Graph Bridge Theorem, any edge e is a bridge. If we remove one of these edges, we create two components with p and q vertices, and $p-1$ and $q-1$ edges respectively (via our inductive hypothesis). Then the summation of edges in each component and the bridge is:

$$E(T) = (p-1) + (q-1) + 1 = (p+q) - 1 = n - 1$$

which satisfies the second statement. ■

Chapter 3

Single Source Shortest Path

Weights are a great way to convey additional data in a graph. Whether it is representing the distance between two cities, or the cost in seconds to transfer data via a certain path, connected and weighted graphs are a huge help. However, a new problem arises with the addition of weights: The Single Source Shortest Path Problem.

Definition 3.0.1. A *Single Source Shortest Path* is the shortest path from a starting node to any other node in the connected graph.

A fairly straightforward concept, but one that comes with its own troubles in trying to actually find such a path between any two nodes algorithmically. Below are two solutions to such a problem: Dijkstra's Algorithm and the Bellman-Ford Algorithm.

3.1 Dijkstra

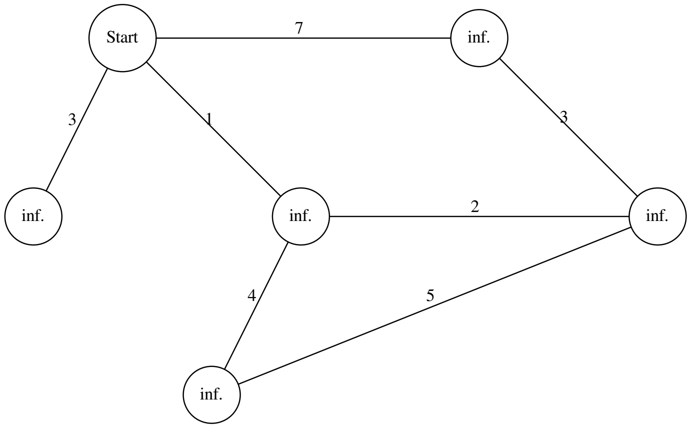
Dijkstra's algorithm starts at some decided source node, and then follows each edge from the source path out to the adjacent nodes. Each node is then given a "cost", which is the sum of all the edges' weights on the path from the source node to that node.

After finding the cost for each node adjacent to the source node, the algorithm repeats, taking each node adjacent to the source node, and finding the costs of thier adjacent nodes, until all nodes have a cost assigned to them. For example, the cost of a node that is two edges away from the source node would be the sum of the weights of those two edges.

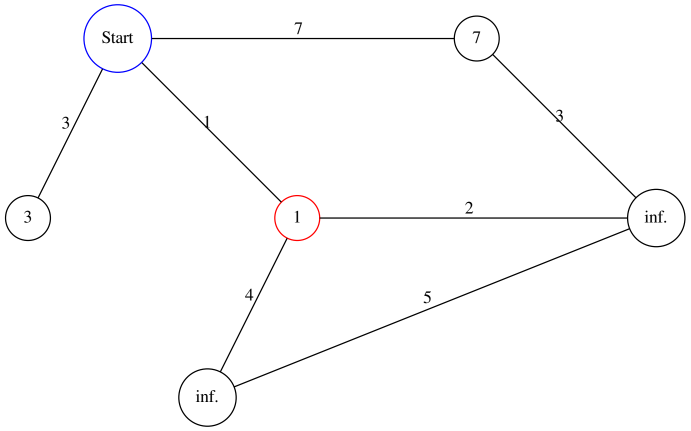
The algorithm does not end there, however. In a connected graph, there is a very real possibility of a cycle existing, which means there is more than one potential path from the source node to all other nodes. In this instance, the algorithm could work its way around a cycle, adding up the costs for each node, and then come across a node that was already reached by another path.

In that case, a simple comparison is made between the two costs, and the node keeps the "cheapest" or smallest of the two costs. This guarantees that the shortest path between the source node and any other node is preserved.

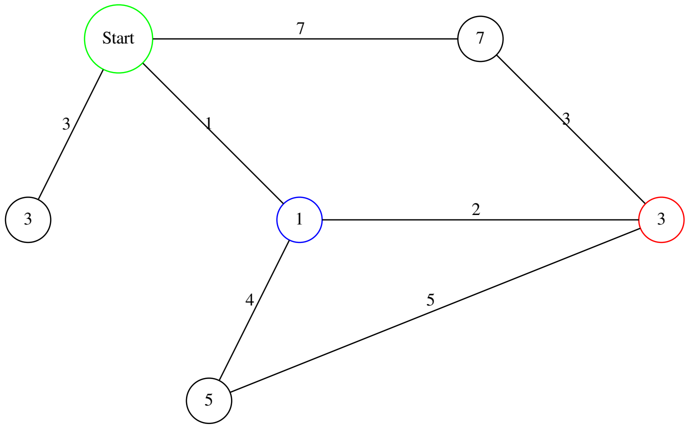
Below follows an example of how Dijkstra's Algorithm works through a connected graph.



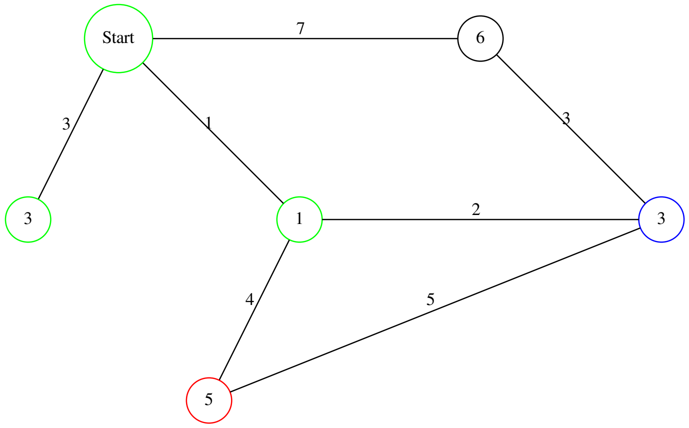
We start with a graph, where the Start vertex's cost is 0, and all other vertices have a cost of infinity.



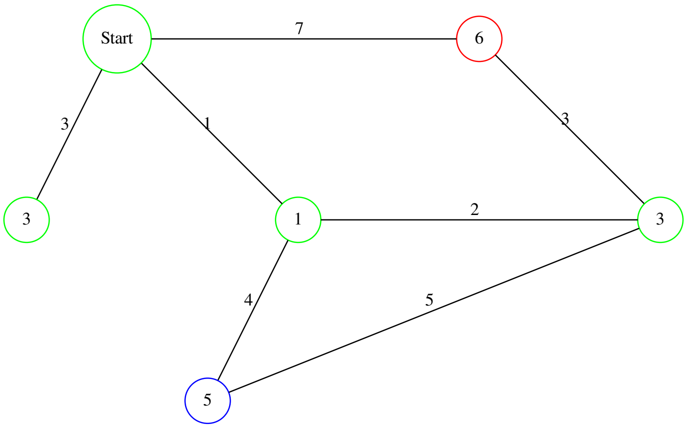
After checking and updating all of the adjacent vertices to Start, we move to the center node(in red) because it has the shortest edge.



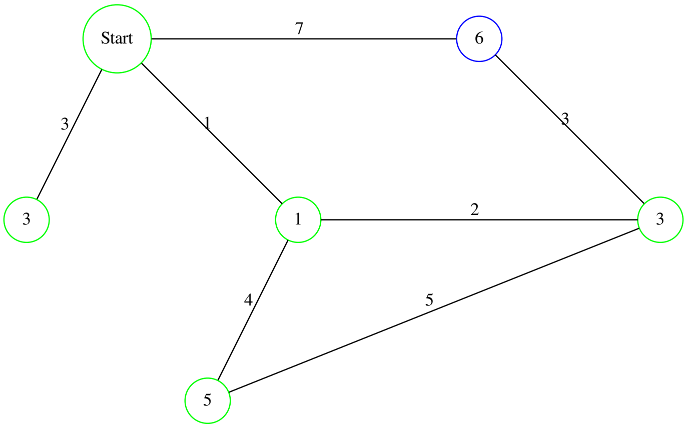
All adjacent vertices are updated, and the algorithm then moves along the lowest weighted edge to the far right vertex.



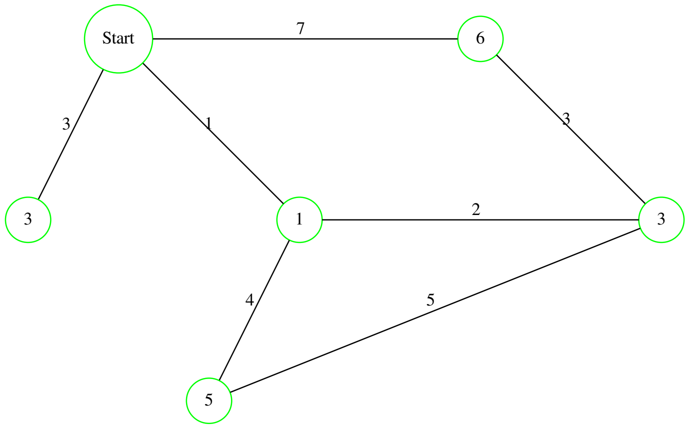
From the blue vertex, we check and see about updating its three adjacent vertices. The path from the blue vertex to the red is the shortest path from Start, so the algorithm moves to that node next. All vertices with their shortest path are now colored green.



From the blue vertex, the cost to travel to its adjacent vertices will not be less than their current weights, so their weights are not updated. We then move back to the previous node (far right) and follow to the final adjacent node at the top.



We continue on to the top node, and see that all of its paths lead to vertices that have their smallest weight. No more weights are updated.



Now, all nodes have their final weight, and the shortest path is established between the start vertex and any other vertex.

In the end, this algorithm has a run time of $O(ElogV)$, or the number of edges E multiplied by the natural log of the vertices V

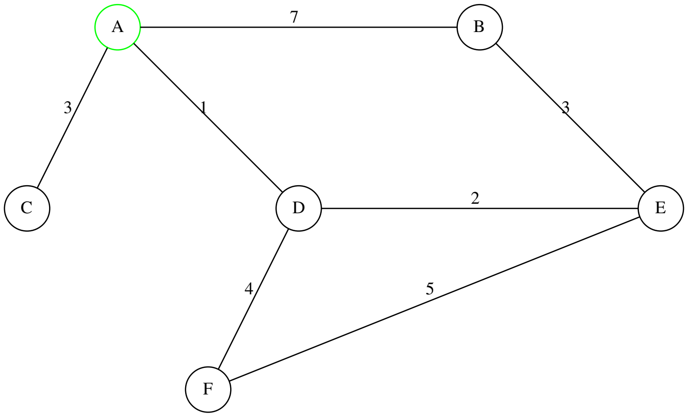
It is important to note that this algorithm will *not* work with negative weights on the edges. The negative weight will be continually added to the cost of a node, and skew the node's actual weight.

3.2 Bellman-Ford

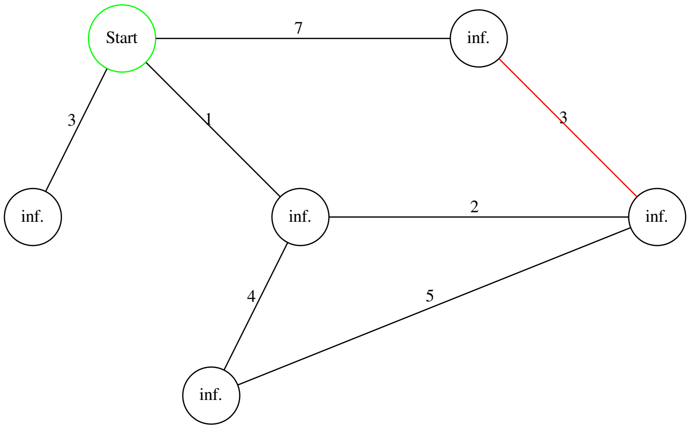
Bellman-Ford follows an overall similar process as Dijkstra's Algorithm, with one key change: Instead of running through the adjacent nodes, the algorithm loops through the list of edges V times (V being the number of vertices).

In this algorithm, each node starts off with a weight of inf. While looping through the list of edges, it updates the costs of the nodes adjacent to each edge, adding the weight both ways. In other words, it takes two nodes, and adds one node's cost to the edge's weight, and compares that sum to the cost of the other node. If the sum is less than the current cost, then the cost for the second node is updated. Doing this both ways between the two nodes ensures that the shortest path is maintained.

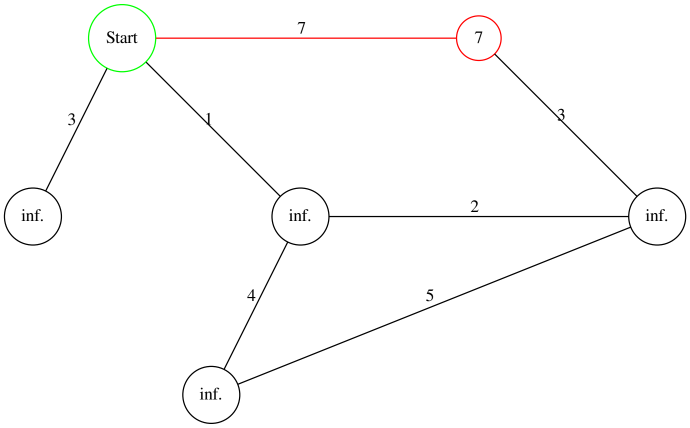
Below is an example of how one iteration of this algorithm would work thorugh the same graph. For the purpose of this example, the order of edges is $BE, AB, AC, AD, DE, DF, EF$, with the alphabetical labels assigned to the nodes in the following fashion.


A
B
C
D
E
F
7
3
1
3
2
4
5

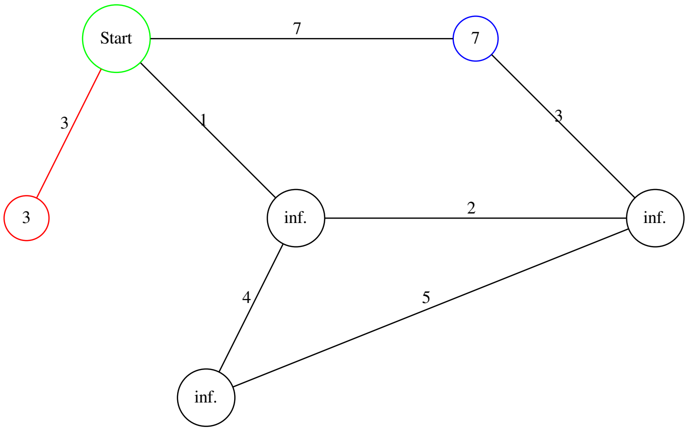
To start:


Start
inf.
inf.
inf.
inf.
inf.
7
3
1
3
2
4
5

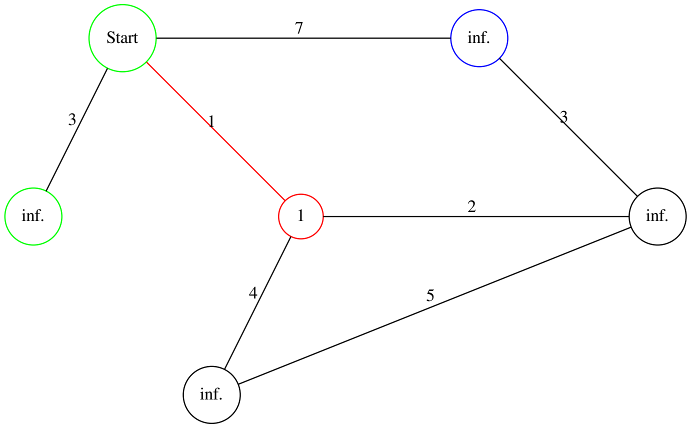
We start with edge BE, adding the weight to each of the adjacent node's cost. As the new calculated cost will not be less than their current one, neither node is updated.



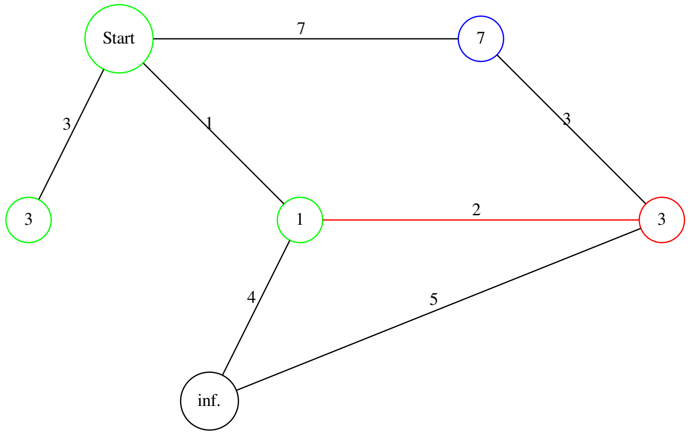
We continue to the next edge in the list. Here, the weight of 7 is added both ways, to account for how a path could move either way between the two nodes. The value for b changes, as $0 + 7 < inf.$. However, $inf. + 7 \not< 0$, so the Start node's cost is not updated (and never will be!).



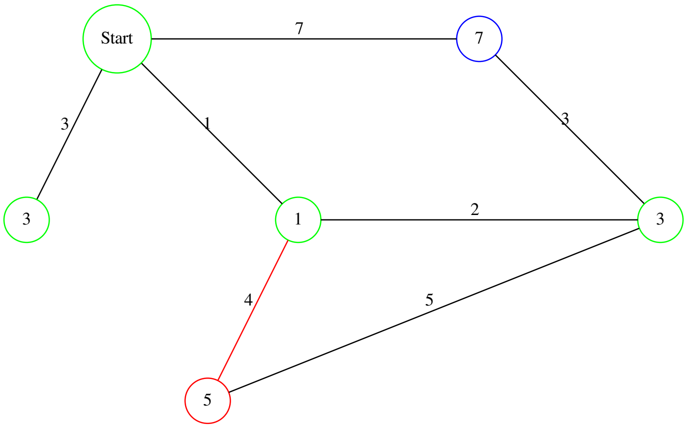
Moving to edge AC, node C's cost is updated as $0+3 < inf.$. This is the shortest possible value for that node's cost, so we will color it green, along with all other nodes with their shortest path for illustrative purposes, as the algorithm could only guarantee a shortest path after $V = 7$ iterations.


Start
inf.
inf.
1
inf.
inf.
7
3
1
3
2
4
5

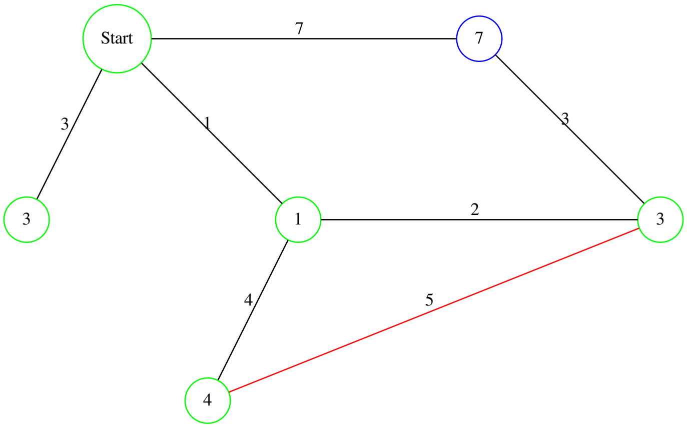
For edge AD, D's weight is updated, while Start's stays the same.


Start
7
3
1
3
inf.
7
3
1
3
2
4
5

Node E's cost can now be adjusted, as $1 + 2 < inf.$.



We then check the edge DF, updating F's cost to 5.



The final edge, EF, is checked. The algorithm calculates that $4+5 \not< 3$, and $3+5 \not< 4$, so neither node's cost is updated.

After this one iteration, it is clear to see that while some nodes had their final shortest weight (like C, nodes like B will need at least one more iteration before having its true final cost.

Unlike Dijkstra's algorithm, the Bellman-Ford algorithm does work with negative weights, with a slight caveat. In order for the negative weights to not artificially skew the cost of a vertex, the weight cannot be a part of a cycle. If it is, however, the algorithm can catch the existence of this negative weight at the end. That is because of the algorithm running V times. If there is not a negative cycle, it will take at most $V - 1$ times for the algorithm to compute the correct cost for each vertex. After the V iterations, if there is still a chance to update a vertex's cost, then there must be a negative cycle.

Chapter 4

Eulerian and Hamiltonian

4.1 7 Bridges o of Königsberg

The 7 Bridges of Königsberg is a problem that would eventually lay the foundation to graph theory asks if it is possible to walk across all 7 bridges in the city exactly once. First to prove that it was impossible to do was Leonhard Euler in 1736, who proposed an abstract look at the city in terms of vertices and edges. Each landmass of the city was turned into a vertex and the bridges were edges. This would be known as a graph.[7]

4.2 Eulerian

After Euler proved the 7 Bridges problem impossible he would consider a graph to have an Eulerian Trail (or Eulerian Circuit) if it is connected and the graph has a trail that includes every edge once. If the graph has an Eulerian Trail then the graph is Eulerian if the trail is closed. The graph is Semi-Eulerian (or Eulerian Path) if the trail is not closed. In other words if the graph has 0 vertices of odd degree the graph is Eulerian. If the graph has 2 vertices of odd degree the graph is Semi-Eulerian. Any other combination the graph is not Eulerian. This leads us to Euler's Theorem: A connected graph G is Eulerian if and only if every vertex has an even degree.

4.2.1 Hierholzer's Algorithm

```
Given a connected graph $G$ with every vertex with an even degree
    Start at vertex $v$
    Visit an edge (randomly)
    Add edges to trail (make sure there are no repeats)
    Until we get back to $v$
```

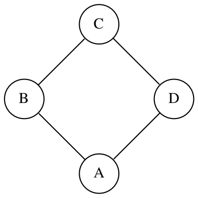


Figure 4.1: Eulerian and Hamiltonian

If all edges visited, we're done
Else recursively do it again at an edge we never visited
Add all edges at end to get Eulerian Circuit
Has run time of $O(E)$[4]

4.2.2 Fleury's Algorithm

Given a graph G with 0 or 2 vertices of odd degree
If G has 0 vertices of odd degree start at any vertex
If G has 2 vertices of odd degree start at a vertex with odd degree
Choose an edge whose deletion would not disconnect the graph
If that is not possible it picks the remaining edge left at the vertex
It moves to the other endpoint of the edge and deletes it
At the end of the algorithm there are no edges left
The sequence of edges chosen forms the Eulerian Circuit(or Trail)
Although the algorithm is linear we have to account for the complexity of detecting bridges so the run time is $O(E^2)$[4]

4.3 Hamiltonian

A Hamiltonian Circuit in a graph G is a circuit of length n. If G has a Hamiltonian Circuit then G is Hamiltonian. A Hamiltonian Path is a path of length $n-1$. If G has a Hamiltonian Path then G is Semi-Hamiltonian. If it is possible to visit every vertex once and end at the same vertex you started then G has a Hamiltonian Circuit. If the starting and ending vertex are different but the path still visits every vertex once, then G is Semi-Hamiltonian.

4.3.1 Examples

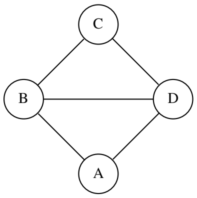


Figure 4.2: Eulerian but not Hamiltonian

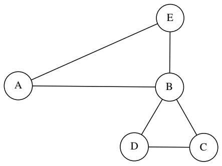


Figure 4.3: Hamiltonian but not Eulerian

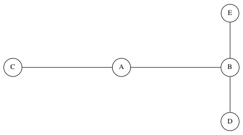


Figure 4.4: Neither Eulerian and Hamiltonian

4.4 Theorems

The theorems below present a hypothesis and prove if the graph satisfies the hypothesis then the graph is Hamiltonian. To prove the converse (so if a graph is Hamiltonian, then it has 'x' property) and produce and if and only if proof is a million dollar question. Finding if a graph is Hamiltonian is challenging, we will discuss next chapter how time consuming it can be for computers to solve the problem.

4.4.1 Ore's Theorem (1960)

If G is a graph with $n \geq 3$ vertices such that for all distinct non-adjacent vertices u and v, $deg(u) + deg(v) \geq n$, then G is Hamiltonian.

Proof. Suppose that there is a graph that is not Hamiltonian but satisfies the hypothesis of the theorem. Let $T = \{n \in \mathbb{N} :$ n is the number of vertices in the graph$\}$. Since T is a non-empty set of positive integers, then T has an element called n_0. Let G be a graph with n_0 vertices and as many edges as possible, but not Hamiltonian and satisfies the hypothesis. Now G is not complete since all complete graphs of more than 2 vertices are Hamiltonian. Let u and v be non-adjacent vertices in G. Note: $deg(u) + deg(v) \geq n$ and $G + uv$ is Hamiltonian.

If $uv_i \in E$, then $u_{i-1}v \notin E$. So, $deg(v) \leq (n-1) - deg(u)$. Therefore, $deg(v) + deg(u) \leq n - 1$, which is a contradiction to the hypothesis.[6] ■

4.4.2 Las Vergnas (1971)

A graph G with $n \geq 3$ vertices is Hamiltonian if the vertices of G can be labelled, $v_1, ..., v_n$ such that $j < k$, $j + k \geq n$, where $v_j, v_k \in E$ and $deg(v_j) \leq j, deg(v_k) \leq k - 1$ Note: this implies that $deg(v_j) + deg(v_k) \geq n$

4.4.3 Chvátal(1970)

Let G be a graph with $n \geq 3$ vertices and the degrees satisfy $d_1 \leq d_2 \leq .... \leq d_n$. If $d_j \leq j \leq \frac{n}{2}$ implies $d_{n-j} \geq n - j$ then G is Hamiltonian.

4.4.4 Bandy(1969)

Let G be a graph with $n \leq 3$ vertices and the degrees satisfy $d_1 \leq d_2 \leq .... \leq d_n$. If $d_j \leq j, d_k \leq k$ implies $d_j + d_k \geq n$ then G is Hamiltonian.

4.4.5 Pŏsa (1962)

Let G be a graph with $n \geq n$ vertices such that for each j, $1 \leq j \leq \frac{n}{2}$ the number of vertices of degree no longer than j is less than j, then G is Hamiltonian.

4.4.6 Dirac (1962)

If G is a graph with $n \geq 3$ vertices such that $deg(v) \geq \frac{n}{2}$ for every vertex of G, then G is Hamiltonian.

4.4.7 Bandy and Chvátal(1976)

First a definition: The closure of a graph G with n vertices, denoted $C(G)$ is the graph obtained from G by recursively joining pairs of non-adjacent vertices whose degree sum is at least n until no such pair remain.
Let G be a graph with $n \geq 3$ vertices if $C(G)$ is complete, then G is Hamiltonian.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2007.

[2] Wikipedia contributors. Borůvka's algorithm — Wikipedia, the free encyclopedia, 2019. [Online; accessed 10-March-2020].

[3] Wikipedia contributors. Prim's algorithm — Wikipedia, the free encyclopedia, 2019. [Online; accessed 10-March-2020].

[4] Wikipedia contributors. Eulerian path — Wikipedia, the free encyclopedia, 2020. [Online; accessed 18-April-2020].

[5] Wikipedia contributors. Kruskal's algorithm — Wikipedia, the free encyclopedia, 2020. [Online; accessed 10-March-2020].

[6] Wikipedia contributors. Ore's theorem — Wikipedia, the free encyclopedia, 2020. [Online; accessed 18-April-2020].

[7] Wikipedia contributors. Seven bridges of königsberg — Wikipedia, the free encyclopedia, 2020. [Online; accessed 18-April-2020].

Chapter 5

NP-Completeness

NP-Complete is a way of classifying the complexity of a problem in terms of how difficult the problem is, or how ‘long’ it takes to run. For our purposes, the problems will be limited to decision problems, where the solution is given as either a ‘yes’ or ‘no’. Before we can discuss NP-Complete problems, we should first go over some definitions.

5.1 Definitions

Deterministic: An algorithm is **deterministic** if it follows a single path from input to outcome.

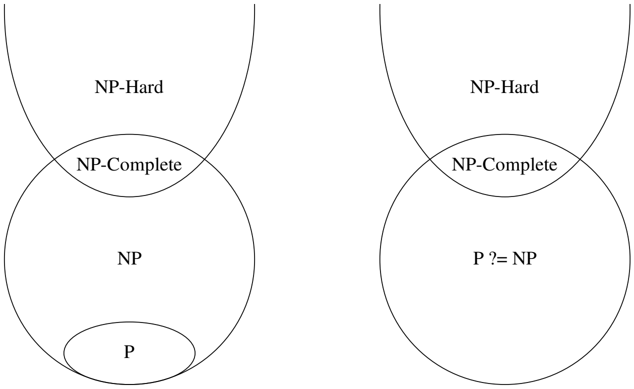
Nondeterministic: An algorithm is **nondeterministic** if it can follow a branching path that can go from one input to multiple outcomes. All of these paths are potentially traversed simultaneously.

Polynomial Time: An algorithm is said to be a **polynomial time** algorithm if its execution time scales by a polynomial. An example of this is a linear search algorithm, which can be run in $\mathrm{O}(n)$ time. **Polynomial time** algorithms could also be bounded above by a polynomial, meaning that an algorithm which runs in $\mathrm{O}(\log(n))$ time is also a polynomial time algorithm, as it is better than $\mathrm{O}(n)$. Examples of non-polynomial times include $\mathrm{O}(2^n)$, $\mathrm{O}(n!)$, or $\mathrm{O}(n^n)$.

P: A problem is classified as **P** if it can be solved by a deterministic turing machine in polynomial time.

NP: A problem is classified as **NP** if a potential solution can be verified by a deterministic turing machine in polynomial time. As each solution is verifiable in polynomial time, if we were to build a non-deterministic turing machine we could try every possible solution at the same time, verifying each of them. This is what it means to say that **NP** problems are decidable in non-deterministic polynomial time. The set of problems **NP** contains the set of problems **P**. There is also a very famous question to whether or not the entirety of **NP** lies in **P**.

NP-Hard: A problem is classified as **NP-Hard** if it can be reduced to any other **NP-Hard** problem in polynomial time.



NP-Complete: A problem is classified as **NP-Complete** if it is both **NP-Hard** and **NP**.

Reduction: A **reduction** is a method for using problems of known difficulty to show that another problem is at least as difficult. If one can perform a reduction in polynomial time from a known NP-Complete problem to some new problem, this new problem must be at least as complex as that NP-Complete problem.

Gadget: A **gadget** is a mapping from the input of one problem to the input of another problem. **Gadgets** are used in construction reductions.

5.2 Founding Papers

5.2.1 Cook-Levin Theorem (1971)

The first problem that was known to be NP-Complete is called the Boolean Satisfiability problem, or SAT. The theorem that declares SAT as NP-Complete is called the Cook-Levin Theorem, named after Steven Cook and Leonid Levin. The two worked in parallel writing their own papers, with Cook in the United States and Levin in the Soviet Union.

5.2.2 Karp's 21 Problems (1972)

In 1972, Richard Karp used the Cook-Levin theorem that SAT is NP-Complete to put together a list of 21 other problems that are also NP-Complete. He showed these problems as NP-Complete by performing polynomial time reductions from SAT to each of the 21 problems. Karp's 21 problems are as follows, with nesting indicating which reductions he used.

- SAT
  - 0-1 Integer Programming
  - Clique
    - Set Packing
    - Vertex Cover
      - Set Covering
      - Feedback Node Set
      - Feedback Arc Set
      - Directed Hamiltonian Circuit
        - Undirected Hamiltonian Circuit
  - 3-SAT
    - Chromatic Number
      - Clique Cover
      - Exact Cover
        - Hitting Set
        - Steiner Tree
        - 3-Dimensional Matching
        - Knapsack Problem
          - Job Sequencing
          - Partition
            - Max Cut

5.3 Problems

5.3.1 Boolean Satisfiability

The **Boolean Satisfiability** decision problem (**SAT**) is to determine if there is a set of values for which a given boolean formula evaluates to true. These formulae can contain the operations **and** ($x_1 \wedge x_2$), **or** ($x_1 \vee x_2$), or **not** ($\overline{x_1}$). An example of one such formula can be seen below. We will take the Cook-Levin theorem as our proof that **SAT** is **NP-Complete**.

$$(x_1 \vee x_2) \wedge (\overline{x_1}) \wedge (\overline{x_2} \vee x_3 \vee x_1)$$

5.3.2 3-Satisfiability

The **3-Satisfiability** decision problem (**3-SAT**) is similar to the **SAT** problem, but with the added restriction that the boolean formula must be in the form 3-conjunctive normal form (**3-CNF**). **CNF** states that clauses are sets of **or** operations separated by **and** operations. There is no limit on the number of clauses which the **and** operations operate on. **3-CNF** requires that each clause has two **or** operations performed on three variables. An example of a formula in **3-CNF** can be seen below.

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$$

Is it in NP?

To show that **3-SAT** is in **NP**, we must show that if given a potential solution to the problem we can verify the solution in polynomial time. A potential solution can be represented with a value for each of the variables in the formula. Plugging values into and verifying the formula is a polynomial time operation, so the **3-SAT** problem is in **NP**.

Is it in NP-Hard?

A boolean formula can be quickly converted to **3-CNF**. Clauses of fewer than three variables can be grown to contain three variables by simply adding a second copy of a variable already in the clause.

$$\begin{aligned}(\overline{x_1}) &= (\overline{x_1} \vee \overline{x_1} \vee \overline{x_1})\\ (x_1 \vee x_2) &= (x_1 \vee x_1 \vee x_2) = (x_1 \vee x_2 \vee x_2)\end{aligned}$$

Clauses of greater than three variables can be broken down into smaller clauses by introducing addition variables. The purpose of these new variables is to spread the one clause out without changing its meaning. An existing clause of n variables can be reduced to a collection of $n-2$ clauses in **3-CNF** with $n-3$ additional variables.

$$(x_1 \vee x_2 \vee ... \vee x_{n-1} \vee x_n) =$$
$$(x_1 \vee x_2 \vee z_1) \wedge (\overline{z_1} \vee x_3 \vee z_2) \wedge ... \wedge (\overline{z_{n-4}} \vee x_{n-2} \vee z_{n-3}) \wedge (\overline{z_{n-3}} \vee x_{n-1} \vee x_n)$$

This conversion to **3-CNF** can be done in polynomial time, meaning **SAT** is reducible to **3-SAT** in polynomial time. This means that **3-SAT** is in **NP-Hard**.

5.3.3 Directed Hamiltonian Cycle

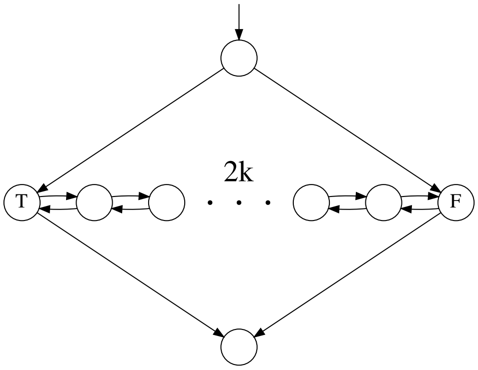
The **Directed Hamiltonian Cycle** decision problem is to determine if there is a Hamiltonian cycle in a given directed graph. That is to say, does there exist a cycle that passes through every vertex of the graph exactly once. We will now prove that the **Directed Hamiltonian Cycle** problem is an **NP-Complete** problem. To do this, we will first show that it is in **NP**. Then we will show that the problem is in **NP-Hard**.

Is it in NP?

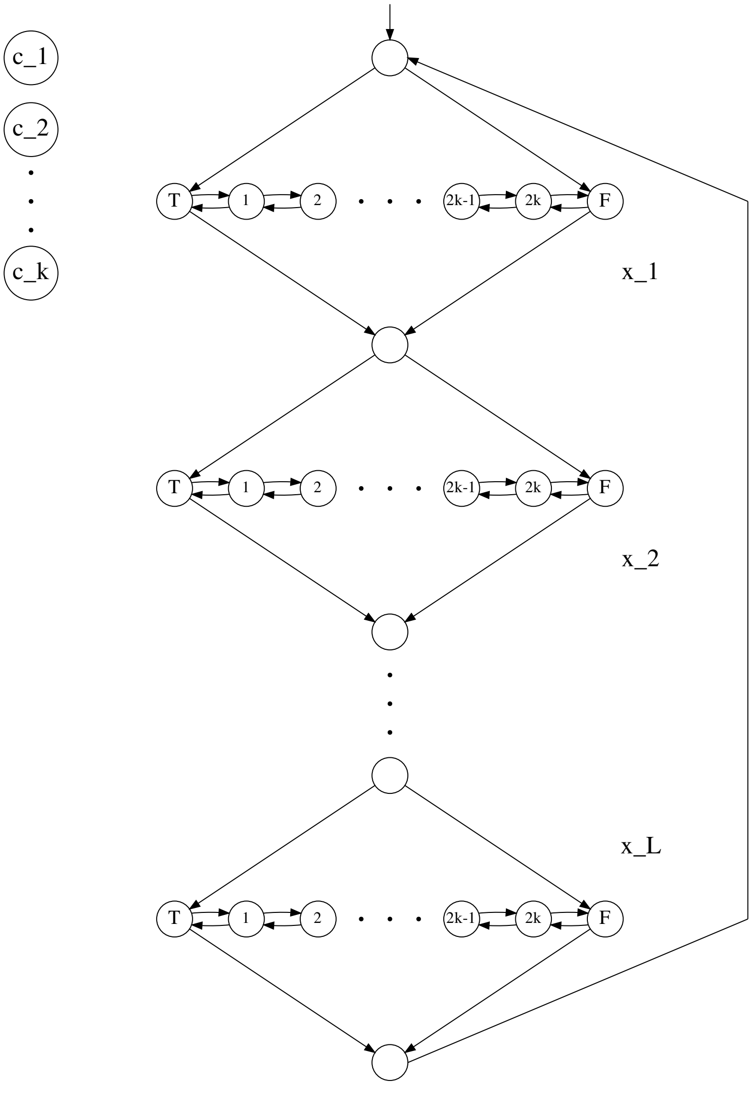
To show that the **Directed Hamiltonian Cycle** problem is in **NP**, we must show that if given a potential solution to the problem we can verify the solution in polynomial time. We will represent the solution with a list of vertices, representing the vertices of the cycle in the order that they occur. We first check that each vertex of the graph is in the list exactly once. Then we will ensure that each pair of vertices adjacent in the list are also adjacent in the graph. Both of these operations can be done in polynomial time. As we can verify a solution in polynomial time, the **Directed Hamiltonian Cycle** problem is in **NP**.

Is it in NP-Hard?

To show that the **Directed Hamiltonian Cycle** problem is in **NP-Hard**, we must show that we can reduce a known **NP-Hard** problem to the **Directed Hamiltonian Cycle** problem in polynomial time. We will be using **3-SAT** for this proof. Let the clauses be represented by $c_1, c_2, ..., c_k$ and the variables be represented by $x_1, x_2, ..., x_L$. For our gadget, we will build a directed graph based on the given clauses and variables. We will build L of the below structures, with $2 * k$ vertices in each of the crossbar.



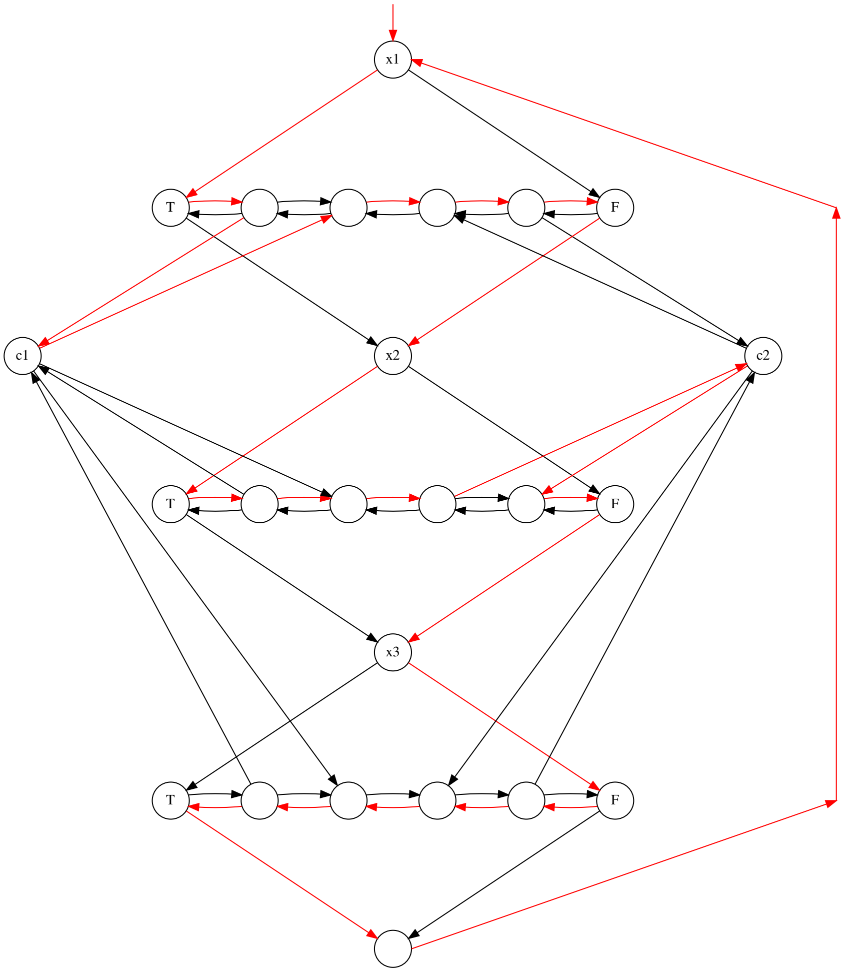
Each of the structures will represent one variable, attaching the bottom of one to the top of another. Each clause will be given a single vertex away from the main structure.



The final step is to connect each of the variables to a clause if and only if the variable occurs in the clause. We connect a variable to a clause by drawing edges from one of the variable's crossbar vertices to the clause vertex, then back to a vertex adjacent to the crossbar vertex. If the variable has a **not** operation performed on it in the clause, then the adjacenct vertex should be to the left of the initial. Otherwise, it should be to the right. Once all of the required edges have been connected from variables to clauses, we can perform our check. If there exists a Hamiltonian path on this graph, then the 3-SAT problem is also satisfiable. Take the 3-SAT problem below for example.

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$$

This 3-SAT problem can be turned into the below graph. The red edges show the path of the Hamiltonian cycle. As there exists a Hamiltonian cycle, the 3-SAT problem is known to be satisfiable, and we can read which values satisfy the 3-SAT problem by examining the graph. As the edges in x_1's crossbar point from left to right, we will take x_1's value as true. We can repeat this, finding a value of true for x_2 and false for x_3. With this set of values, we now know a set of values for which the above 3-SAT problem is satisfiable.



5.3.4 Clique

The **Clique** decision problem is to determine if there exists in a graph G a complete subgraph of size k.

Is it in NP?

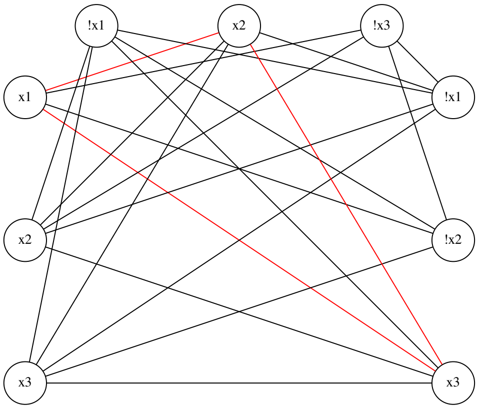
If given a subgraph of G, first determine if there are k vertices. If there are k vertices, then determine if they all share edges. This can be done in $O(k^2)$ time, which means this problem is polynomial verifiable.

Is it in NP-Hard?

To show that the **Clique** problem is **NP-Hard**, we will use a gadget to show that we can reduce **3-SAT** to **Clique** in polynomial time. We will denote each clause as a cluster of 3 vertices. For every pair of vertices, create an edge if the vertices are not in the same cluster and if the vertices are not complements of each other (ie. X and $\overline{X}$). For a boolean statement of k clauses, if the resulting graph contains a clique of size k, then the original 3-SAT problem is satisfiable. As an example, take the following 3-SAT problem.

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

In the graph we will be using the **!** symbol as our **not**. As this boolean statement contains three clauses, we are looking for a 3-clique from the resulting graph. As you can see below, there exists a clique of size three in the graph, so the original 3-SAT problem is satisfiable. By examining the graph we can see one set of values for which it is satisfiable. The connected vertices are x_1, x_2,and x_3; this means if we set those three values to true, it would satisfy the 3-SAT problem.



5.3.5 Vertex Cover

The **Vertex Cover** decision problem is to determine if there exists in a graph G a vertex cover of size $\leq k$. A vertex cover of an undirected graph is a subset of vertices S such that every edge in G is adjacent to at least one vertex in S.

Is it in NP?

If given a graph G, a max size k, and a potential vertex cover S, we can first check that $|S| \leq k$. This is likely a constant time operation. Next we can use the following algorithm:

```
• For each edge $(u, v) \in G$
    • If $u \notin S$ and $v \notin S$
        • Return $false$
• If not returned $false$, return $true$
```

This algorithm runs in $O(m)$ where m is the number of edges in G, meaning that the vertex cover is polynomial time verifiable.

Is it in NP-Hard?

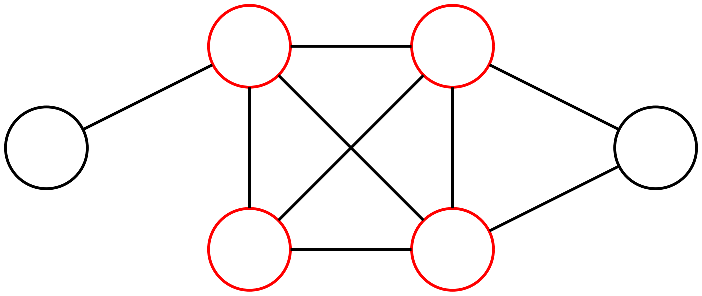
To show that the **Vertex Cover** problem is in **NP-Hard**, we will perform a reduction from the **Clique** problem. We can begin by looking at G', the complement of G. It turns out, the clique decision problem of size k in G is the same as the vertex cover of size $n - k$ in G'. This can be shown by assuming there is a clique of size k in G. We will call the set of vertices in the overall graph V and the set of vertices in the clique V'.

We will pick any edge (u, v) from the complement graph G'. Either u or v is a member of V', but not both. We can prove this by contradiction. Assume that both u and v are members of V', then there would be an edge between them in the clique. This would mean that there would not be an edge between them in G'. This is a contradiction. As our starting assumption was that u and v were both in V', at least one of the vertices must be outside of V'. This means that at least one of u or v must fall inside the set of vertices $V - V'$.

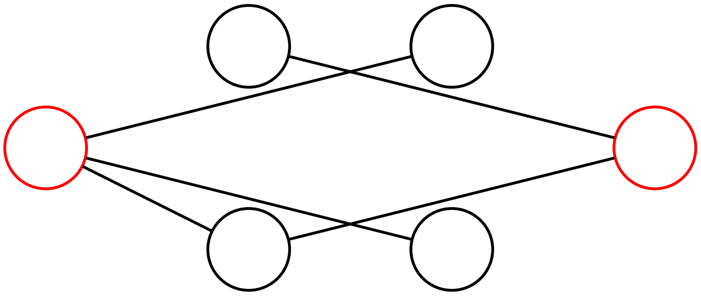
Now we will assume there is a vertex cover V'' of size $|V| - |V'|$, or $|V| - k$, in G'. If we pick any edge (u, v) from G', both u and v cannot be outside of V''. Thus, all edges (u, v) such that u and v are both outside of V'' are in G. These edges create a clique of size k.

This means that there is a clique of size k in graph G if and only if there is a vertex cover of size $n - k$ in G'. A graph can be converted to its complement in polynomial time, thus the **Clique** problem can be reduced to the **Vertex Cover** problem in polynomial time.

As an example, there is a clique of size four in the below graph.



As there is a clique of size 4 in the graph above, there will be a vertex cover of size $n-4$, or 2, in the complement below. As you can see, there is a vertex cover of size 2.



Chapter 6

Pigment of your Imagination

It might seem strange at first to think, “Why would we bother coloring our graphs?” While that’s certainly a fair question that applies to just about anything, ask yourself “Why am I reading these notes?” Then realize you’re now an everyday scholar of the graphs, and continue reading.

6.1 Coloring your Graphs

The notion of *graph coloring* is, fundamentally, no different than what we described back in Chapter 1 - it’s a labeling of the vertices of our graphs. Instead of using numbers, variables, or text, we ascribe to the vertex set a color set. But assigning colors to vertices allows us to observe and study graphs under an altogether different lense than what we have used previously. And *that* is why we would bother coloring our graphs.

Before diving into any rigorous terms, definitions, and theorems, we will first lay the ground rules for graph coloring. There is only one ground rule: no two vertices can share the same color.

6.2 On the Rules of Coloring

We will now begin a deep dive into graph coloring. First, we will define the formal definition of a graph being colorable.

Definition 6.2.1 (K-Colorable)**.** Given a graph G, we say that G is *k-colorable* if each vertex of G can be assigned one of k colors such that no two adjacent vertices receive the same color.

Below, we see two graphs which have their vertices colored. One is two-colorable, but the other is not.

We also introduce the idea of coloring edges, under a similar restriction:

Definition 6.2.2 (K-Edge-Colorable)**.** Given a graph G, we say that G is *k-edge-colorable* if each edge of G can be assigned one of k colors such that no two adjacent edges receive the same color.

Like before, we see two graphs which have their edges colored. One is two-edge-colorable, but the other is not.

Take some time to devise some graphs and color them with your favorite colors. You might quickly see that it's no easy task if you have a complicated graph with many vertices and edges. For now, it won't help us to know if a graph is *not* colorable from some k. This leads us to think, "What is the smallest k for which my favorite graph G is colorable?" We now introduce two new properties of graphs:

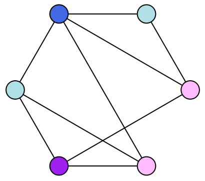
Definition 6.2.3 (K-Chromatic)**.** Given a graph G, we say that G is *k-chromatic* if G is *k-colorable*, but not *(k-1)-colorable.* We say that k is the *chromatic number* of G, and is denoted:

$$\chi(G) = k$$

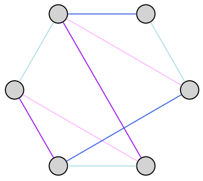
Definition 6.2.4 (K-Edge-Chromatic)**.** Given a graph G, we say that G is *k-edge-chromatic* if G is *k-edge-colorable*, but not *(k-1)-edge-colorable.* We say that k is the *edge-chromatic number* of G, and is denoted:

$$\chi\prime(G) = k$$

Here we show a graph which we have annotated with both the chromatic number and the edge-chromatic number:

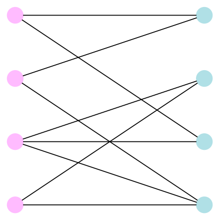


$\chi(G) = 4$



$\chi\prime(G) = 4$

An example of a class of graphs which we know to be two-chromatic is the set of bipartite graphs. You might imagine coloring all vertices in any one set the same color. As these vertices share no edges within the sets, this satisfies the rules for colorability. Here is an example of a two-coloring for a bipartite graph:



Recall the definitions of the maximal degree $\Delta(G)$ and the minimal degree $\delta(G)$ for a given graph G from Chapter 1. The maximal degree is relevant in this chapter! Here is a simple theorem:

Theorem 6.1. *Let G be a graph. Then $\chi(G) \leq \Delta(G) + 1$*

Proof. We will proceed with induction on the number of vertices. Our base case would be a graph with 1 vertex, whose maximal degree is 0. Then

$$\chi(G) = 1 \leq \Delta(G),$$

so this is satisfied. Let's assume this applies for any graph with up to n vertices, and suppose that G has n vertices. Let's consider the graph $G - v$ for some vertex v in G. Then $G - v$ has $n - 1$ vertices, and

$$\Delta(G - v) \leq \Delta(G)$$

. Therefore we have that $G - v$ is $\Delta(G) + 1$ colorable. ■

Now, we've seen planar graphs before. We have some interesting coloring theorems which apply to planar graphs, so let's visit some of them. We've seen that every planar graph contains a vertex whose degree is at most 5.

Theorem 6.2. *Every planar graph is 6-colorable*

Chapter 7

Grin and Berg it

7.1 Grinberg's Theorm(1976)

Let G be a plane graph with n vertices and a hamiltonian cycle C. let a_i be the number of faces of G in the interior of C whose boundary contains exactly i edges. Let b_i be the number of faces of G in the exterior of C whose boundary contains i edges. Then

$\sum_{i=3}(i-2)(a_i-b_i)=0.$

7.2 Diagonals

A diagonal of a cycle is an edge of the graph joining to non-adjacent vertices of the cycle.

Proof:

Let d denote the number of diagonals of G in the interior of C. The removal of any diagonal results in one less interior region. The removal of d diagonals results in one region and so there are $d-1$ interior faces. Hence:
$d+1=(\sum_{i=3}a_i)-1$
The sum of the number of edges bounding each region will count each diagonal twice.

$\sum_{i=3}ia_i=2d+n.$
$\sum_{i=3}ia_i=2(\sum_{i=3}a_i)-2+n$

$\sum_{i=3} ia_i = (\sum_{i=3} 2a_i) - 2 + n$
$\sum_{i=3} ia_i - \sum_{i=3} 2a_i = n - 2$
$\sum_{i=3}(i-2)a_i = n - 2 = \sum_{i=3}(i-2)b_i$
$\sum_{i=3}(i-2)a_i - \sum_{i=3}(i-2)b_i = 0$
$\sum_{i=3}(i-2)(a_i - b_i) = 0$

7.3 Herschel Graph

Planar,Connected,not Regular, and not Eulerian

Chapter 8

Plane and Simple

8.1 Euler's Formula (1750)

Let G be a connected plane graph and let n,m, and f denot the number of vertices, edges, and faces of G: $n - m + f = 1$

8.2 Lemma

If G is a connected planar graph with $n \leq 3$ vertices and m edges then $m \leq 3n-6$ Proof: Every face of G must have at least three sides and each edge is on at most two faces. $3f \leq 2m$ Euler's Formula: $n - m + f = 23n - 3m + 3f = 66 \leq 3n - 3m + 2mm \leq 3n - 6$ If G is a connected planar graph with $n \geq 3$ vertices and m edges and no triangles then $m \leq 2n - 4$ Proof: Euler's Formula: $n - m + f = 24n - 4m + 4f = 88 \leq 4n - 4m + 2m2m \leq 4n - 8m \leq 2n - 4$

8.3 Kuratowski's Theorm (1930)

A graph is planar if and only if it contains no subgraph homeomorphic to K_5 or $K_(3,3)$. Two graphs are homeomorphic (identical within vertices of degree two) if they both can be obtained from the same graph by inserting new vertices of degree two into the edges of G.

8.4 Wagner's Theorm(1937)

A graph is planar iff its minors include neither K_5 nor $K_(3,3)$

A graph is planar iff it contains no subgraph contractible to K_5 or $K_(3,3)$

8.5 Minors

H is called a minor of a graph G if H can be created from G by deleting edges, vertices, and contracting edges.

Chapter 9

Graph Classes

9.1 Bipartite Graphs

We've mentioned bipartite graphs several times in different chapters of this book. However, we've only explored this class of graph at a surface level. In this section, we'll take a deeper look into bipartite graph - what are the conditions for being bipartite, how do bipartite graphs allow us to more easily answer certain questions, and how are they relevant in our own real scenarios?

9.1.1 Definitions

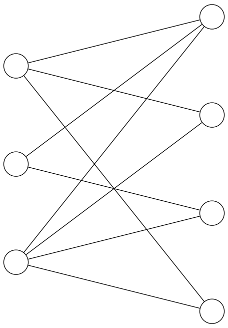
Let's give a formal definition for what it means for a graph to be bipartite. First, let's define what an independent vertex set is:

Definition 9.1.1 (Independent Vertex Set)**.** An *independent vertex set* is a set of vertices in a graph G, none of which are adjacent.

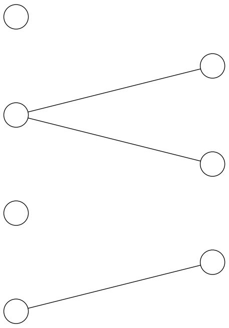
Using this, we can give a good definition for bipartite graphs:

Definition 9.1.2 (Bipartite Graph)**.** A graph G is *bipartite* if its vertex set can be partitioned into two independent vertex sets.

This is no different than how we've defined them before. Here are some examples of bipartite graphs:



Connected Bipartite Graph



Disconnected Bipartite Graph

Having defined bipartite, we can derive several other related classes of graphs, including complete bipartite, or tripartite. There are other classes as well which are equivalent to or are subsets of the set of bipartite graphs. For example, all trees are bipartite (this is easy to imagine, and there are some different proof strategies for this). Cycles of even length are bipartite as well, though cycles of odd length are not.

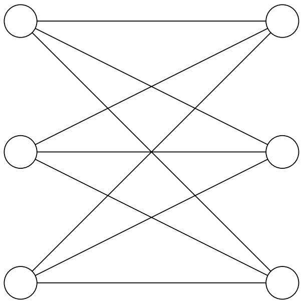
Complete bipartite graphs are an important subset of bipartite graphs. Here is the formal definition for complete bipartite:

Definition 9.1.3 (Complete Bipartite Graph)**.** A *complete bipartite* graph G is a graph whose vertex set can be partitioned into two independent sets U and V such that for any $u \in U$ and $v \in V$, (u, v) is an edge. Such a graph is denoted as

$$K_{m,n}$$

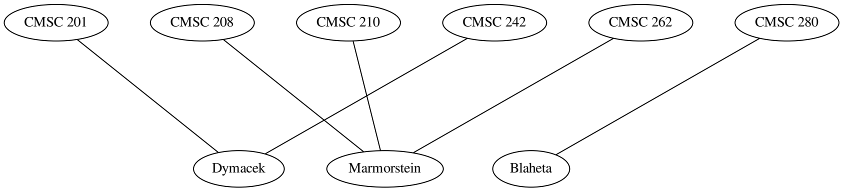
where m, n are the sizes of each independent set.

An example of a complete bipartite graph is $K_{3,3}$, known as the Utility Graph, shown below:



9.1.2 When Can We Use These?

As you might imagine, bipartite graphs might be useful when we want to take information in two distinct categories, and match them. For example, suppose we want to consider the relationships between professors and courses they teach. In our graph, we can separate these into two categories, ”professors” and ”courses”, and create an edge any time a particular professor teaches a course. By construction, we have that there would not be any edges between two professors (as a professor is not a course), nor would there be any edges between two courses (as a course cannot teach another course). Thus, we have a bipartite relationship by construction.



A classical example of this problem is something we’ve covered before - the Knight’s Tour. On a chessboard, the movement pattern of a knight allows us to separate the spaces on a chessboard into sets of light and dark spaces. When the knight moves, the space it lands on alternates color. Drawing edges between a particular space and all other spaces to which the knight can move, we construct a bipartite graph!

9.1.3 How Does This Help Us?

Given a bipartite graph, it becomes much easier to determine asnwers to certain historically difficult problems. For example, if G is bipartite with an odd number of vertices, then we know for certain that G is non-hamiltonian. This was proven in previous exercises, and can easily be found online.

Regarding cliques, we know that if G is bipartite, then its clique number is 2. If G has a clique number $c > 2$, then we could find a complete subgraph of size c, which would contain a triangle. However, to be bipartite, we cannot have a triangle, and thus G would not be bipartite.

Finally, regarding graph coloring, if G is a bipartite graph, then we know that the chromatic number of G is 2. We can achieve this by coloring each independent set one color (since no vertices in either set are adjacent to each other). Thus we have two colors covering all vertices, so $\chi(G) = 2$.

Index

3-Satisfiability, 46
6-Colorable, 55
7 Bridges of Königsberg, 35

Adjacency Matrix, 12
Adjacent, 9

Bandy, 38
Bipartite Graph, 11
Boolean Satisfiability, 45
Borůvka's Algorithm, 21
Bridge, 22

Chvátal, 38
Clique, 50
Complement, 14
Complete Graph, 11
Component, 14
Connected, 13
Cook-Levin Theorem, 44
Cycle, 14, 17

Degree, 9
Deterministic, 43
Diagonals, 57
Digraph, 8
Dirac, 39
Directed Hamiltonian Cycle, 46
Disconnected, 13

Edges, 7
Euler's Formula, 59
Eulerian, 35

Fleury, 36
Forest, 17

Gadget, 44
Graph Bridge Theorem, 22
Graph Coloring, 53
Graphs, 7
Grinberg's Theorm, 57

Hamiltonian, 36
Herschel Graph, 58
Hierholzer, 35

Incidence Matrix, 12
Incident, 9
Isolated, 9
Isomorphic, 9

Join, 9

K-Chromatic, 54
K-Colorable, 53
K-Edge-Chromatic, 54
K-Edge-Colorable, 54
Karp's 21 Problems, 45
Kruskal's Algorithm, 20
Kuratowski's Theorm, 59

Las Vergnas, 38
Lines, 7
Links, 7

Minimum Spanning Trees(MST), 19
Minors, 60
Multigraph, 8

Nodes, 7
Nondeterministic, 43
NP, 43
NP-Complete, 44
NP-Hard, 44
Null Graph, 11

Ore, 38

P, 43
Pŏsa, 38
Path, 16
Pendant, 9
Points, 7
Polynomial Time, 43
Prim's Algorithm, 19

Reduction, 44
Reflexive, 10
Regular Graph, 11

Self-Complementary, 15
Spanning Tree, 18
Subgraph, 11
Subtraction, 14
Symmetric, 10

The Handshaking Lemma, 10
Trail, 16
Transitive, 10
Tree, 13

Union, 13

Vertex Cover, 51
Vertices, 7

Wagner's Theorm, 59
Walk, 15
Weight, 19