The little book of graph notes

Graph theory students

March 22, 2020

Contents

0.1 Preface

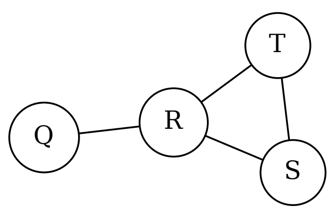
This compilation of notes exists as a project for students in the CMSC 420 Graph Theory course at Longwood University. These notes have been created by all the students of the class, and an attempt has been made to both categorize the notes in a logical, sequential manner, and to ensure that the information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

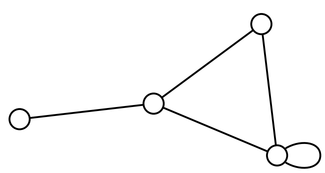
Graphs can be used to model many real world problems, or theoretical ideas. For example, graphs can be used to represent relationships between different objects or groups. They are useful for visualizing data, such as social media connections, or network topology. Additionally, a more practical use is 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

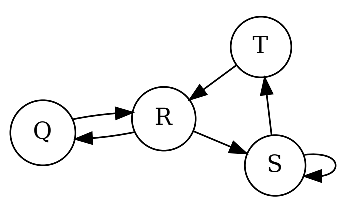
A **multigraph** is a pair of sets (V, E) such that $V \neq \emptyset$, and E is a finite family of unordered pairs of V. That is to say, edges are not necessarily unique subsets, and do not have order. Thus, multigraphs allow for loops from one vertex to itself. Here is an example of a multigraph:



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.

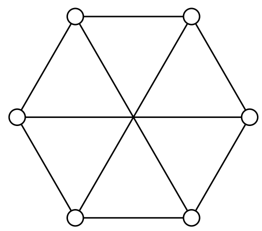
1.5 On Isomorphisms

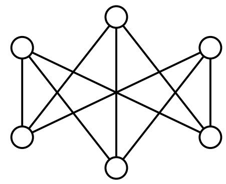
The term *isomorphic* has its roots in Greek, *iso-* meaning same, and *morphic* referring to form or shape. We say that two graphs are **isomorphic** if they have this "same shape" property, but what does that mean?

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. $\square$

1.6 On Special Kinds of Graphs

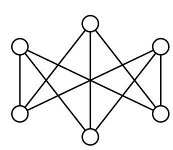
We've talked a lot about the different aspects of graphs, but now let's use this information to actually classify different kinds of graphs, starting with the **null graph**. the null graph is a graph with n vertices but no edges, and is denoted as N_n.

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, $K_{3,3}$:

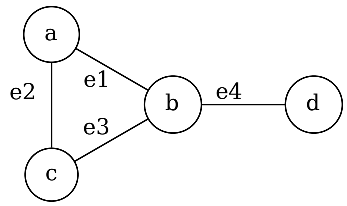


A **subgraph** of $G = (V, E)$ has the vertex set H and the edge set F, where $H \subseteq V$ and $F \subseteq E$, but only the edges of E which correspond to the vertices in H can be in F.

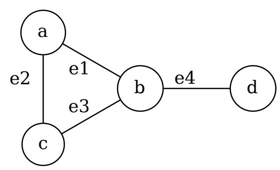
1.7 On the Storage of Graphs

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:



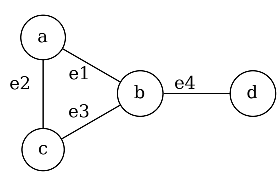
We can store this in what's called an **Adjacency Matrix**, a matrix where each row and column represents a particular vertex in the graph, and entries in the matrix signify that two vertices share an edge. The adjacency matrix for the given graph is:



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

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:



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

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

Chapter 2

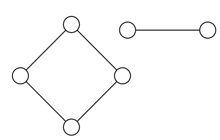
Forest for the Trees

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.

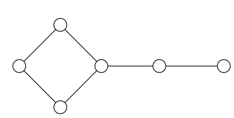
2.1 Definitions

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:

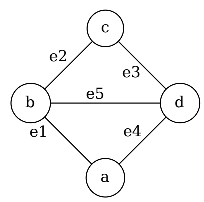


Connected: A graph is **connected** if it is not disconnected (wow). Here is an example of a connected graph:

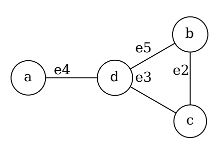


Component: A **component** graph is one graph which makes up part of a union. In $G = G_1 \cup G_2$, G_1 and G_2 are components.

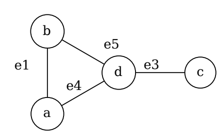
We can also define the **subtraction** of a set of edges F from a graph G. We say that $G - F$ is the graph resulting in the removal of the edges in F from G. Suppose we have the following graph G:



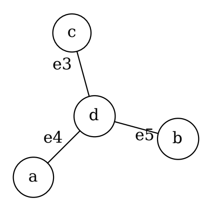
Then the following are different subtractions from G.



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

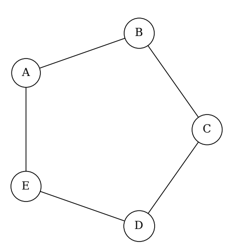


(b) $G - \{e2\}$

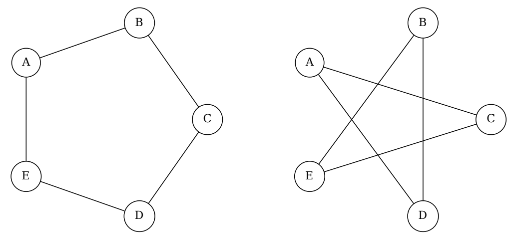


(c) $G - \{e1, e2\}$

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:



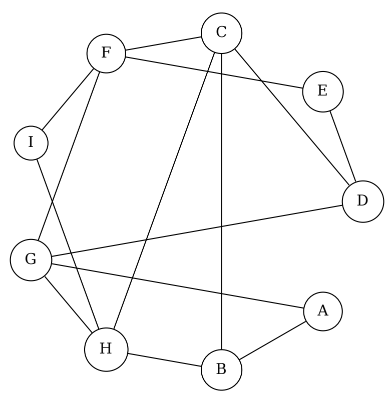
A useful operation on graphs is to take the complement. The **complement** of a graph G is the graph $\overline{G}$, with vertex set $V(\overline{G}) = V(G)$, with two vertices adjacent in $\overline{G}$ if and only if they are *not* adjacent in G. Given C_5, we can fine its complement, $\overline{C_5}$:



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.

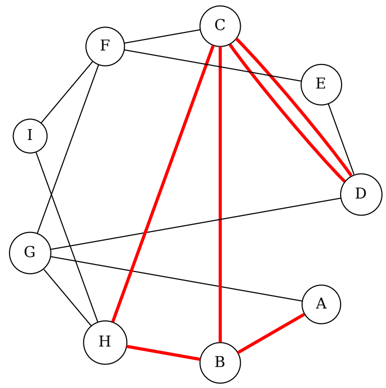
2.2 Trailblazing

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:



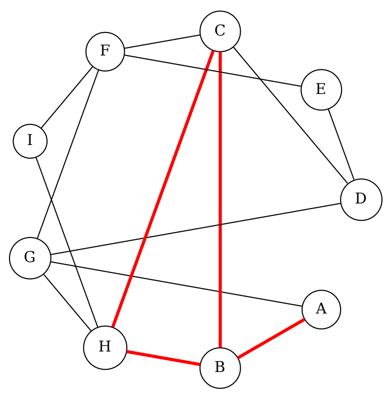
Let's define some of the different ways we can navigate through G_0:

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:



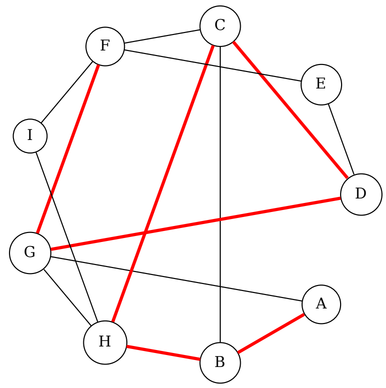
Here, the walk shown begins at node A. The walk is AB, BH, HC, CD, DC, CB. The length of this walk is 6.

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:



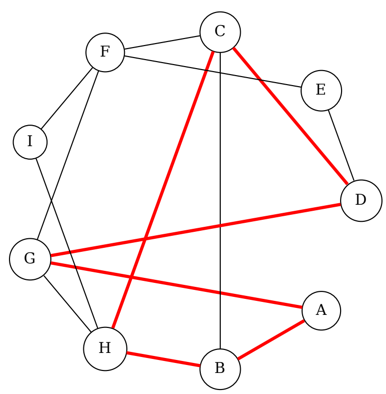
The trail here is AB, BH, HC, CB.

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:

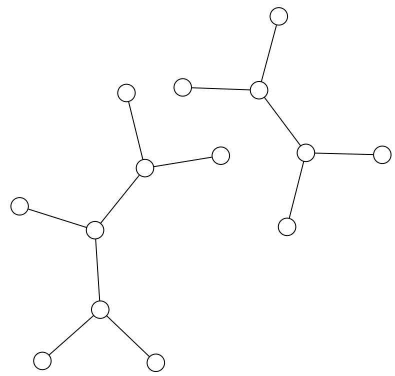


This cycle is AB, BH, HC, CD, DG, GA.

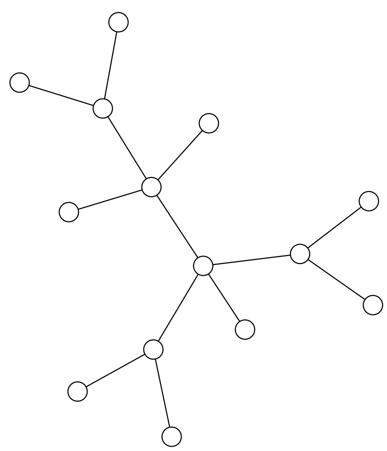
2.3 The Lorax

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 ∈ V(G) do
    Weight(v) ← ∞
    Parent(v) ← null
end
Weight(v_0) ← 0
Q ← V(G)
while Q ≠ ∅ do
    u ← GetMinVertex(Q)
    foreach v ∈ AdjacentTo (u) do
        if v ∈ Q and EdgeWeight(u,v) < Weight(v) then
            Parent(v) ← u
            Weight(v) ← 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[4]. 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 ← ∅
for v ∈ V(G) do
  | A ← A ∪ {v}
end
Q ← E(G)
while Q ≠ ∅ do
  | e(u, v) ← GetMinEdge(Q)
  | if TreeWith(u) ≠ TreeWith(v) then
  |   | JoinSets(u, v)
  |   | A ← A ∪ 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 ← ∅
for v ∈ V(G) do
  | A ← A ∪ {v}
end
Q ← E(G)
while SizeOf(A) > 1 do
  | foreach e(u, v) ∈ E(G) do
  |   | if TreeWith(u) ≠ TreeWith(v) then
  |   |   | JoinSets(u, v)
  |   |   | A ← A ∪ 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. $\square$

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

Chapter 3

NP-Completeness

3.1 What are NP-Complete Problems?

NP-Complete is a way of classifying the complexity of a problem in terms of how ”long” it takes to run. The set of problems that are NP-Complete is the intersection of the set of problems that are NP and the set of problems that are NP-Hard, as shown below.

3.2 NP Problems

3.3 P Problems

3.4 NP Hard Problems

3.5 NP Complete Problems

Chapter 4

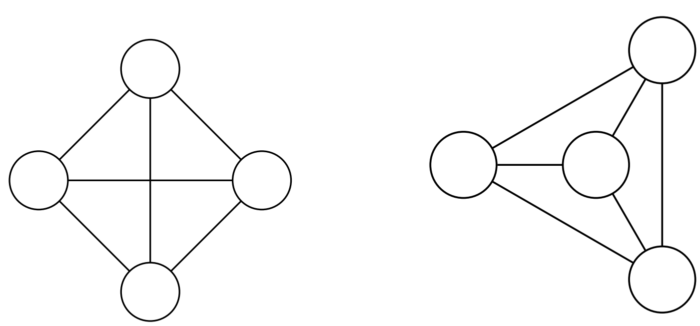
Planarity and more

4.1 The Plane

THE PLANE, is simply two-dimensional space. You may also see it refered to as $\mathbb{R}^2$ or the Euclidean Plane.

4.1.1 Plane and Simple

A graph is **planar** if it can be embedded in the plane without without any edges crossing.
A **plane graph** is a graph that has been drawn without any edges crossing.



(a) Planar (b) Plane

4.2 Don't cross me

The **Crossing Number** of a graph G, denoted "cr(G)", is the smallest number of edge crossings needed to draw the graph in the plane.
All planar graphs have a crossing number of 0. In the figure below, you can see that both K_5 and $K_{3,3}$ have crossing numbers of 1. (there isn't a figure yet get *allll* the way off my back about it)

4.3 Jordan Curves

4.4 Faces

4.4.1 Euler's formula

Let **G** be a connected plane graph and let n, m, and f be the number of vertices, edges, and faces of **G** (respectively).

$$n - m + f = 2$$

4.5 We're just Platonic

The **Platonic Solids** are the five 3-dimensional solids whose faces consist only of regular congruent polygons.
These solids are (in order of face count increasing): the Tetrahedron, the Cube, the Octahedron, the Dodecahedron, and the Icosahedron.
The **Platonic Graphs** are the graphs of the platonic solids. They are all planar.