CHMC Advanced Group:
Graph Algorithms

Apr. 13, 2024

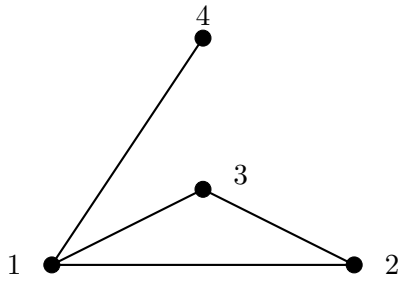
1 Introduction

Graph Theory, which is within the field of combinatorics, is an area of mathematics rich with theory and interesting problems. With natural connections to computer science, it has been the source of much research and is still an active field of mathematics and computer science. In this session, we will look at some classical algorithms used to tackle certain problems posed in graph theory.

2 Graphs

A **simple graph**, $G = (V, E)$ is a set V of distinct vertices and a set E of distinct edges between the vertices with certain restrictions. For our purposes, we will look at *finite graphs* (where both V and E are finite sets). Associated to each edge in E is an unordered pair of distinct vertices $\{v_i, v_j\}$ (i.e. we don't care which direction the edge travels, we only care about which vertices it connects). We also will only consider graphs where any pair of vertices is connected by at most one edge.

For example, the graph with vertices $V = \{1, 2, 3, 4\}$ and edges $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}\}$ can be represented by the following picture:



A **path** P is an ordered list of vertices $v_1 v_2 \cdots v_k$, where each pair of consecutive vertices are connected by an edge; i.e., the path consists of the edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$. Since edges are uniquely determined by a pair of vertices, there is only one such edge that can connect a pair of vertices. The **length** of a path is equal to the number of edges in the path. A graph is **connected** if there is a path between any two vertices in the graph. Today, we will only focus on connected graphs, (even though some of the definitions that we write do not require the graphs involved to be connected).

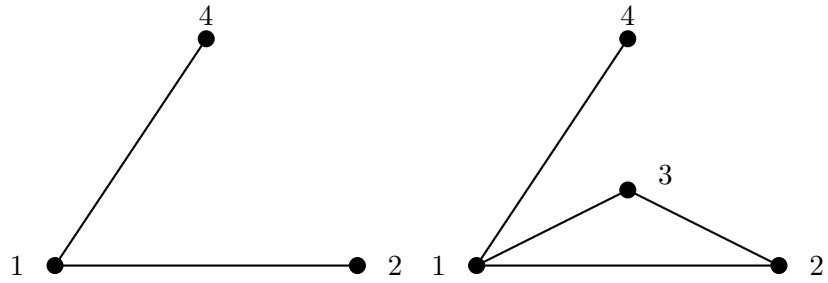
Let us look at a couple special types of graphs. First, a **cycle** C_n is a path where the first vertex and the last vertex are the same and the path has length $n \geq 3$. A **tree** is a graph that has no cycles. The **complete graph on n vertices** K_n is the graph consisting of n vertices and every pair of distinct vertices is connected by an edge.

Exercise 2.1. Draw a copy of the complete graphs K_3, K_4, K_5, and K_6. How many edges does each of these graphs have? Can you relate it to the number of vertices? How many edges does the graph K_n have for $n \geq 1$?

Exercise 2.2. Draw a tree with 3 vertices, a tree with 4 vertices, and a tree with 5 vertices. How many edges does each tree have? How many edges does a tree with n vertices have? What happens if we add one more edge to the tree without adding any new vertices?

Given a graph $G = (V, E)$, a **subgraph** G' of G is defined by $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$. That is, a subgraph consists of a collection of vertices from the original set of vertices and a collection of edges from the original set of edges such that for each edge $e \in E'$, the vertices at its endpoints are also in V'. We may think of a subgraph of G as a graph obtained by possibly deleting vertices and edges from G.

For example, the graph with vertices $V' = \{1, 2, 4\}$ and edges $E' = \{\{1, 4\}, \{1, 2\}\}$ is a subgraph of the graph with vertices $V = \{1, 2, 3, 4\}$ and edges $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}\}$:



Exercise 2.3. In the complete graphs K_3, K_4, find different subgraphs that are trees and are cycles. Notice, if $m \leq n$ then K_m is a subgraph of K_n. How many subgraphs of K_n can you find that look like K_m? Try this first for K_2 in K_4, K_2 in K_5, K_3 in K_4, and K_3 in K_5. Once you have an idea, conjecture on the number of subgraphs of K_n that look like K_m.

A **spanning tree** T of a graph G is a subgraph of G that contains all of the vertices of G and is also a tree. If G is already a tree, then it is also a spanning tree. However, if G contains a cycle, then there are multiple spanning trees of G.

Exercise 2.4. How many spanning trees are there in C_n for $n \geq 3$? Consider what a spanning tree in C_n looks like and how to delete edges from C_n to make such a spanning tree.

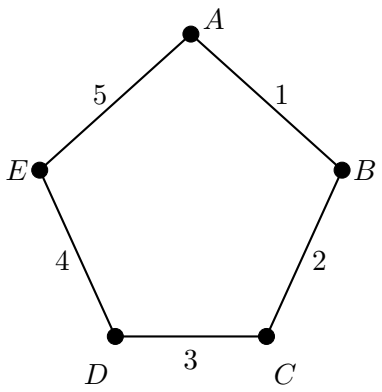
Exercise 2.5. How many spanning trees are there for K_n for $n \geq 2$? Doe you see a pattern in the number of trees and the values of n and $n-2$? (Note: this is a little harder to recognize and the numbers grow rather large rather quickly)

3 Weighted Graphs and Associated Problems

In the previous section, we introduced some basics of graphs and counting trees in two special types of graphs. We now introduce a new type of graph and see how it relates to this counting problem.

A **weighted graph** is a graph $G = (V, E, W)$ such that for each edge $e \in E$, there is an associated **weight** $w_e \in W$. We will only consider these weights to be positive numbers. One way to interpret these weights is to let every edge represent a city, each edge is a road between cities, and each weight is the distance of a particular edge. (Some other interpretations of these weights may involve time or cost associated with traveling a particular edge, but we can change our interpretation depending on the context of the original problem).
If we say the weight of a *path* is the sum of the weights of the edges in the path, one natural question we can ask is "given a weighted graph G, what is the minimal weighted path between a pair of distinct vertices?"

Exercise 3.1. Consider the cycle graph C_5 with the following weighted edges. Find the path from vertex A to vertex D which has the lowest total weight. Is this the same as the path from vertex A to vertex D which has minimal *length* (i.e. the fewest number of edges)?



Exercise 3.2. For the cycle C_n, denote the weight of the edge connecting v_i with v_{i+1} by w_i for $1 \leq i \leq n-1$ and w_n for the edge connecting v_n and v_1. Since creating a path between two vertices of C_n requires choosing a direction around the cycle to traverse from v_i to v_f, think of a way to find the minimum weight path from v_i to v_f.

One potential strategy to answering this question would be to record *all* possible paths from the initial vertex v_i to the final vertex v_f and among those paths, choosing the one with the minimal weight. This may be simple for something like a cycle, but for more complicated graphs, there may be far too many paths for us to check.

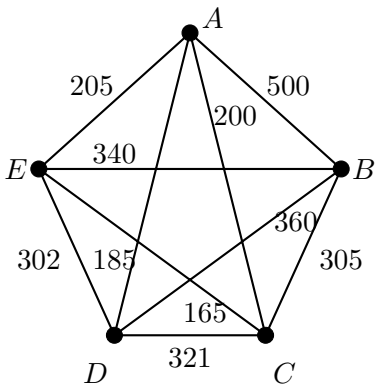
Another natural question we might ask is "Given a weighted graph G, which spanning tree T minimizes the total sum of edge weights included in the spanning tree?" We will call such a spanning tree a **minimal spanning tree**.

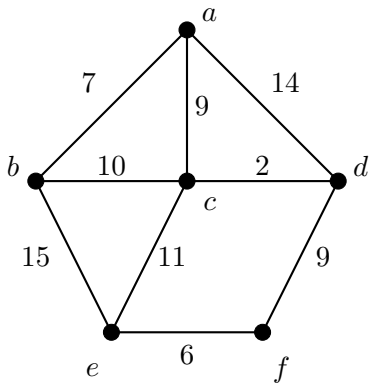
Exercise 3.3. For the graph given in 3.1, find a minimal spanning of the graph. In general, for the cycle C_n, denote the weight of the edge connecting v_i with v_{i+1} by w_i for $1 \leq i \leq n-1$ and w_n for the edge connecting v_n and v_1. Since creating a spanning tree for C_n requires removing a single edge, think of a way to find a minimal spanning tree.

One way to find a minimal spanning tree is to list out all the possible spanning trees and then choose the one with the lowest total weight. However, for a complete graph K_n with $n \geq 2$, the number of spanning trees is n^{n-2} which grows exponentially, so listing all possible spanning trees becomes rather unwieldy. Perhaps we can find a better way to approach this problem.

4 Algorithms for Weighted Graph Problems

In the previous section, we introduced some problems related to weighted graphs. In this section, we will learn different algorithms designed to provide answers to these problems. For each of the following algortihms, we will consider the same two graphs: G_1 will be the weighted K_5 graph with weights $W = \{w_{AB} = 500, w_{BC} = 305, w_{CD} = 321, w_{DE} = 302, w_{AE} = 205, w_{AC} = 200, w_{AD} = 185, w_{BD} = 360, w_{BE} = 340, w_{CE} = 165\}$, and G_2 will be the following weighted graph with 6 vertices $V = \{a, b, c, d, e, f\}$, edges $E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{b, e\}, \{c, d\}, \{c, e\}, \{d, f\}, \{e, f\}\}$ and weights $W = \{w_{ab} = 7, w_{ac} = 9, w_{ad} = 14, w_{bc} = 10, w_{be} = 15, w_{cd} = 2, w_{ce} = 11, w_{df} = 9, w_{ef} = 6\}$.





4.1 Dijkstra's Algorithm

Exercise 4.1. For the graph G_2, try to find a path from a to f that is of minimal weight. What values to you get? Now try to find a path from d to e that minimizes the weight of the path.

We will now introduce an algorithm called **Dijkstra's Algorithm** which is designed to give a minimal weight path between two given distinct vertices. (Note, there may be multiple paths between two vertices that may have the same weight, so we usually refer to *a* minimal weight path instead of *the* minimal weight path).

Dijkstra's algorithm works in the following way: suppose we want to find the minimum weight path between vertex a and vertex f in graph G_2. To run the algorithm, we will establish path weight values for each vertex. Initially, set the starting vertex path weight to 0 and the rest of the vertices to ∞. Denote by Q the set of vertices that are unprocessed; initially all of the vertices are in Q. For a given unprocessed vertex, look at all its neighbors that are also unprocessed. Add the weight of the edge connecting the current vertex to a neighbor to the path weight of the current vertex. If this value is less than the path weight of the neighbor vertex, update the path weight of the neighbor vertex with this value. Otherwise, check the next unprocessed neighbor. Once all unprocessed neighbors have been checked, consider the current vertex processed and remove it from Q. The new current vertex is the vertex among the unprocessed vertices that is of minimal path weight. Keep repeating this process until the destination vertex has smallest path weight among all of the unprocessed vertices.

That seems like a lot, so let's take this step by step:

Step One: We set the path weight of a to 0 and the path weights of the remaining vertices to be ∞. Update the path weight of b to 7, c to 9, and d to 14. Since all neighbors of a have been updated, we mark a as processed and remove it from Q.

Step Two: Now, the minimal path weight of unprocessed vertices belongs to b with 7. The neighbors of b that have not been processed are c and e. The path weight of c is 9 which is less than the path weight of b plus the edge weight of bc ($9 < 7 + 10$), so the path weight of c stays the same. The edge weight of be is 15, so the path weight of e is updated to $7 + 15 = 22$. Since all the unprocessed neighbors of b have been updated, we mark b as processed and remove it from Q.

Step Three: The minimum path weight of the unprocessed vertices belongs to c with 9. The edge weight of cd is 2, and $9 + 2 = 11 < 14$, so update the path weight at d to be 11. The edge weight of ce is 11 and $9 + 11 = 20 < 22$, so update the path weight of e to be 20. All of the unprocessed neighbors of c have been updated, so c is now processed and we remove it from Q.

Step Four: The remaining unprocessed vertices are d, e, f and d has the smallest path weight of 11. Its only neighbor is f with edge weight 9 so update the path weight of f to be $11 + 9 = 20$.

At this point, d is now processed, so the remaining unprocessed vertices are e, f. However, since f has a minimal path weight among all path weights of unprocessed vertices, we have completed the algorithm. The minimal weight path from a to f has weight 20 and travels from a to c to d to f.

Exercise 4.2. Try to calculate the minimal weight path from the vertex d to the vertex e using Dijkstra's Algorithm. Now do the same calculation but instead starting from d and finishing at e. What do you get?

4.2 Prim's Algorithm

We may also want to determine minimal spanning trees.

Exercise 4.3. For both graphs G_1 and G_2, construct a few different spanning trees. Try to minimize the sum of the edge weights of the tree. Can you think of a strategy for contructing a minimal spanning tree?

There are a few approaches to constructing a minimal weight spanning tree that we will consider. The first is called **Prim's Algorithm**, which involves building up a spanning tree.

Step One: Pick a vertex and look at all edges at that vertex. Pick the one with minimal weight. The original vertex, edge, and second vertex at the end of the edge are now part of the tree.

Step Two: Among the vertices in the tree constructed, look at all edges extending from these vertices that connect to a vertex not in the tree. Select the one with minimal weight and include this and the new vertex in the tree.

Step Three: Repeat Step Two until all vertices have been reached.

Exercise 4.4. Try running this algorithm on G_1 first starting with vertex A. Then run the algorithm again, this time starting with vertex C. Do you get the same minimal spanning tree?

Exercise 4.5. Try running this algorithm on G_2 first starting with vertex a. Then run the algorithm again, this time starting with vertex c. Do you get the same minimal spanning tree?

4.3 The Reduction Algorithm

Another algorithm to find a minimal spanning tree is known as the **Reduction Algorithm**, and it works as follows.

Step One: Select any cycle in the graph. From this cycle, remove the edge with the largest weight.

Step Two: Repeat step one until no cycles remain. The resulting graph will remain connected yet have no cycles, hence will be a tree.

Exercise 4.6. Try running the reduction algorithm several times on graphs G_1 and G_2, each time choosing a different cycle as your starting cycle. Do you get different minimal spanning trees?

4.4 Kruskal's Algorithm

The previous two algorithms create a tree either by connecting certain vertices to a graph or by removing edges from a graph. Now, we introduce a new algorithm called **Kruskal's Algorithm** which builds a tree by adding the best possible edges (even if they may not be connected at first).

Step One: Order *all* of the edges by their weights in increasing order, from smallest to largest. Add the edge with the smallest weight to the tree and cross this off the list.

Step Two: Continue going down the list of edge weights and add the minimal edge weight that has not been crossed of yet and that will **not** result in a cycle. If adding an edge to the graph will create a cycle, then cross it off the list but do not add it to the tree.

Step Three: Repeat this process until you have constructed a tree.

Exercise 4.7. Try running Kruskal's Algorithm on both graphs G_1 and G_2.

4.5 Adapted Dijkstra's Algorithm

There is an adaptation of Dijkstra's algorithm for finding a spanning tree at a particular vertex. This results in minimal weight paths from the rooted vertex to the other vertices of the graph.

Step One: Write down a 0 next to the starting vertex and select among all the edges coming from it the one with minimal weight. This edge and new vertex are now included in the tree. Record next to the new vertex the path weight.

Step Two: Look at all vertices you have not reached that are neighbors of the vertices you have reached. For each of these vertices, consider all possible paths from the tree constructed to the neighbor vertices obtained by adding one edge. Among these possible extensions, choose the edge that results in the minimal path weight from the initial vertex. Include this edge and vertex in the tree.

Step Three: Repeat step two until all vertices in the graph have been reached. The resulting paths will be of minimal weight from the rooted initial vertex to every other vertex, though again not necessarily unique.

Exercise 4.8. Run this algorithm on G_2 using vertex d as the root. Run it again using vertex f as the root. Compare the minimal paths you determined before between a and f and between d and e. Are they the same or different?

Exercise 4.9. Run this algortihm on G_1 using different vertices as the root.