CHMC Advanced Group: Graph Theory

07/27/2019

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 still is a very active field of mathematics and computer science. In this worksheet 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 with certain restrictions. More precisely, we will look at **finite graphs** for which V and E are finite. Associated to each edge $e \in E$ is an unordered pair of distinct vertices $\{v_i, v_j\}$; we do not care about the direction the edge travels, just the vertices it connects. We also will only consider graphs where any pair of vertices is connected by at most one edge.

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 $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{n-1}, v_n\}$. 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. We will focus our attention only on connected graphs in this worksheet; some of the definitions that follow do not require graphs to be connected, but we will not worry about that.

There are a few special types of graphs. A **cycle** C_n is a path where the first vertex is equal to the last vertex and is of length $n \geq 3$. A **tree** is a graph in which no cycle exists. The **complete graph on n vertices** K_n is the graph consisting of n vertices and an edge connecting every pair of distinct vertices.

Exercise 2.1 Draw a copy of K_3, K_4, K_5, and K_6. How many edges does each complete graph have? Can you relate it to the number of vertices? How many edges does 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 you add one more edge to the tree without adding any new vertices (think about the type of paths that exist in the tree)?

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 G' consists of a collection of vertices V' from the original set of vertices V in the graph, and a collection of edges E' from the original collection of edges E of the graph, such that for every edge $e' \in E'$ in the subgraph, the vertices which are its endpoints are also in V'. We may think of a subgraph G' of G as a graph obtained from G by possibly deleting vertices and edges from G.

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 a tree itself. 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 $n \geq 2$? Do you see a pattern in the number of trees and the value of n and $n-2$? This is a harder pattern to recognize and the numbers grow pretty large very quickly.

One interesting way to count the number of such spanning trees is given by the Matrix Tree Theorem, which I will state but not prove. For this, we need a little bit more terminology and information. To each vertex $v \in V$, we denote by $d(v)$ the **degree** of that vertex, which just counts the number of edges attached to the vertex. Since edges are given by distinct pairs of vertices, there are no loops in the graph, so we do not double count an edge when calculating $d(v)$.

The **adjacency matrix** A for a graph G of n vertices is a matrix $\{a_{i,j}\}_{i,j=1}^{n}$ such that

$$a_{i,j} = \begin{cases} 1 & \{v_i, v_j\} \in E \\ 0 & \{v_i, v_j\} \notin E \end{cases}.$$

That is, there is a 1 in the i^{th} row and j^{th} column if there is an edge connecting vertices v_i and v_j, and 0 otherwise. Notice that if $a_{i,j} = 1$ then $a_{j,i}$ also equals 1, and similarly if $a_{i,j} = 0$ then so too does $a_{j,i} = 0$.

The **degree matrix** D is a diagonal matrix with $a_{i,j} = 0$ for $i \neq j$ and $a_{i,i} = d(v_i)$. This records the degree of each vertex v_i in the i^{th} row and column.

Set $M = D - A$. The **Matrix Tree Theorem** states that picking $1 \leq j \leq n$, deleting the j^{th} row and column from M, and taking the determinant of the resulting matrix gives the number of spanning trees.

Exercise 2.6 Verify the Matrix Tree Theorem for C_3 and C_4. If you have questions regarding calculating the determinant of a matrix, we will assist you.

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 only consider positive numbers as weights. One way to consider this is setting each vertex to represent a city and the weights of edges between cities denoting the distance between the cities.

One natural question to ask is, "Given a weighted graph G, what is the minimal weighted path between a pair of distinct vertices?"

Exercise 3.1 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 distinct vertices of C_n requires choosing a direction around the cycle to traverse from v_0 to v_f, think of a way to find a minimum edge weight path.

One potential strategy to answering this question would be to record all possible paths from the initial vertex v_0 to the final vertex v_f and among those paths, chose one with the minimal edge weight. However, there may be many such paths to check and this would be inefficient.

Another natural question to ask is, "Given a weighted graph G, which spanning tree T (or trees if not unique) minimize the total sum of edge weights included in the spanning tree?" We call such a spanning tree a **minimum spanning tree**.

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 spanning tree for C_n requires removing a single edge, think of a way to find a minimum spanning tree.

One way to find a minimum spanning tree is to list out all the possible spanning trees with their associated total edge weight sums and pick one with lowest value. However, for a complete graph K_n for $n \geq 2$, the number of spanning trees is n^{n-2} which grows exponentially, so listing all the possible spanning trees becomes cumbersome very quickly. Perhaps there are better ways of approaching this problem.

4 Algorithms for solving weighted graph problems

In the previous section, you learned about weighted graphs and two problems associated to weighted graphs. In this section, you will learn different algorithms designed to provide answers to these problems.

The first one is called **Dijkstra's Algorithm**, which gives a minimal weight path between two given distinct vertices. A minimal weight path is path between two vertices such that the sum of the edge weights along the path is minimal among all such paths. Note, multiple minimal paths may exists so we refer to **a** minimal path rather than **the** minimal path.

Suppose we are given the weighted graph $G = (V, E, W)$ where $V = \{a, b, c, d, e, f\}$ $E = \{(a, b), (a, c), (a, d), (b, c), (b, e), (c, d), (c, e)(d, f), (e, f)\}$ and associated weights $W = \{w_{a,b} = 7, w_{a,c} = 9, w_{a,d} = 14, w_{b,c} = 10, w_{b,e} = 15, w_{c,d} = 2, w_{c,e} = 11, w_{d,f} = 9, w_{e,f} = 6\}$.

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

First, draw the graph labelling edges with their appropriate weights. Now, suppose we want to find the minimum edge weight path between vertex a and vertex f. To run the algorithm, we 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 of the graph are in Q. For a given unprocessed vertex, look at all of 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 path weight minimal among the path weights of all the unvisited vertices.

To illustrate this, consider the above weighted graph.

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, of c to 9, and of d to 14. Since all neighbors of a have been processed, we mark a as processed and remove it from Q.

Step Two: Now, the minimal path weight of the unprocessed vertices belongs to b with a value of 7. The neighbors of b that have not been processed are c and e. The path weight at c is 9 which is less than the path weight of b (7) plus the edge weight of (b, c) (10), so the path weight of c stays the same. The edge weight of (b, e) is 15, so the path weight of e

is updated to $7 + 15 = 22$. Since all the unprocessed neighbors of b have been checked, b is now considered process and removed from Q.

Step Three: The minimum path weight of the unprocessed vertices is 9 at c. The edge weight of (c, d) is 2, which is less than the path weight of d currently at 14, so update the path weight at d to be 11. The edge weight of (c, e) is 11, so $9 + 11 = 20 < 22$, so update the path weight of e to be 20. All of the unprocessed neighbors of c have been checked so c is now processed and removed from Q.

Step Four: The remaining unprocessed vertices are d, e, f and d has the smallest path weight of 11, so it becomes the current vertex. 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 path weight minimal among all of the path weights of the unprocessed vertices the algorithm stops. The minimal path weight from a to f has weight 20, consisting of traveling from a to c to d to f.

Exercise 4.2 Try to calculate the minimal path weight between vertices d and e in this graph, starting at d. What path do you get? Now do the same calculation, but start instead from e and work towards d. What do you get?

Similarly, we may want to determine a minimal spanning tree.

Exercise 4.3 For the same graph, 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 constructing 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 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 yet in the tree. Select 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.

This constructs a minimal spanning tree.

Exercise 4.4 Try running this algorithm on the graph given above. First start with vertex a. Run the algorithm again, this time starting with vertex c. Do you get different minimal spanning trees?

Another algorithm is the **Reduction Algorithm**.

Step One: Select any cycle in the graph. Remove from this cycle an edge with maximal edge weight.

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

Exercise 4.5 Try running the algorithm on the graph several times, choosing different cycles as your starting cycles. Do you get different minimal spanning trees?

Another algorithm for finding a minimal spanning tree is **Kruskal's algorithm**.

Step One: Order all of the edges by their weights in increasing order, from smallest to largest.

Step Two: Add the smallest weight to the tree. Cross this off the list.

Step Three: Continuing going down the list of edge weights (increasing edge weights as you advance along the list) and add the minimal edge weight that has not been checked yet and that will **not** result in a cycle.

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

Exercise 4.6 Try running the algorithm on the graph. Change the edge weights on the graph and try running it again.

There is an adaptation of Dijkstra's algorithm for finding a spanning tree rooted 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 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, 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 4.7 Run this algorithm using vertex d as the root. Run it again using vertex f as a root. Compare the minimal paths you determined before between a and f and d and e. Are they the same or different?

References

1. Bronstein, Alexander M., Bronstein, Michael M., Kimmel, Ron, *Numerical Geometry of Non-Rigid Shapes,* First Edition, Springer, New York, NY, 2008.

2. Cioabă, Sebastian M., Murty, M. Ram, *A First Course in Graph Theory and Combinatorics,* First Edition, Hindustan Book Agency, New Delhi, India, 2009.

3. Marcus, Daniel A., *Graph Theory, a problem oriented approach,* First Edition, The Mathematical Association of America, Washington, DC, 2008.