CIS 575. Introduction to Algorithm Analysis
Material for April 10, 2024

Minimum-Cost Spanning Trees: Prim's Algorithm

The topic of this note is presented in *Cormen*'s Section 21.2.

1 Prim's Algorithm

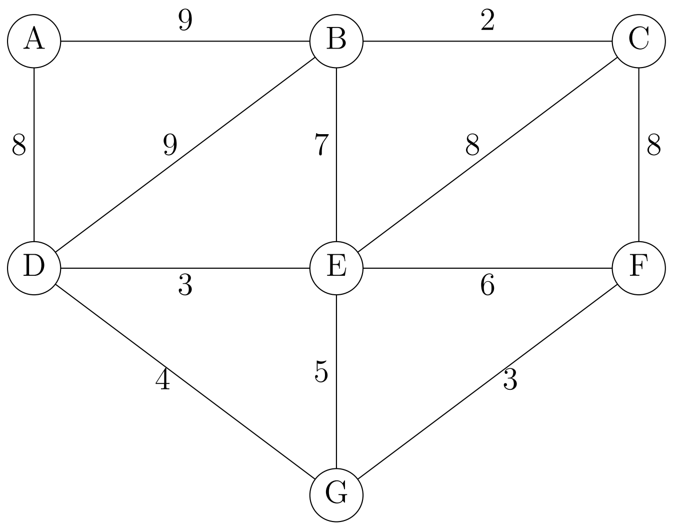
The main idea behind this algorithm is to let the minimum spanning tree T grow from a *single* node (selected in advance). We then repeatedly pick a minimum-weight edge between a node in T and a node not in T; we know that this is safe from a result stated and proved in a previous note (applicable with the nodes in T forming one component and all the remaining nodes forming another component). To facilitate this procedure, we maintain a data structure that to each node u that is *not yet* in T, but which has a neighbor in T, associates an edge $e(u)$ with the property:

$e(u)$ is between u and a node in T, and has minimal weight among such edges.

We then repeatedly

1. find a node w such that the weight of $e(w)$ is at least as low as the weight of any $e(u)$
2. examine the other edges from w; when v neighbors w we may have to update $e(v)$.

Example Recall our running example:



When starting from node A, Prim's algorithm would take the actions

nodes in T	add to T	e(B)	e(C)	e(D)	e(E)	e(F)	e(G)
A		BA(9)	None	DA(8)	None	None	None
AD	(D,A)	BA(9)	None	—	ED(3)	None	GD(4)
ADE	(E,D)	BE(7)	CE(8)	—	—	FE(6)	GD(4)
ADEG	(G,D)	BE(7)	CE(8)	—	—	FG(3)	—
ADEFG	(F,G)	BE(7)	CE(8)	—	—	—	—
ABDEFG	(B,E)	—	CB(2)	—	—	—	—
ABCDEFG	(C,B)	—	—	—	—	—	—

For example, in the first iteration, we select the edge DA since its weight 8 is smaller than the weight of BA; thus D is added to the spanning tree and we update e(E) and e(G), but do not need to update e(B) since BD has the same weight as BA.

We see that we get the *same* minimum spanning tree as produced by Kruskal's algorithm, though the edges are added to the tree in quite a different order. In general, there could be several different minimum spanning trees (but they will all have the same number of edges).

Implementation & Running Time Recall that Prim's algorithm is based on repeatedly finding the node that is closest to the current tree, and next updating that information for the remaining nodes. This can be accomplished in at least two ways:

- **Linear Representation**: keep an array that for each node u contains $e(u)$. Then

  1. to find a w with $e(w)$ having minimal weight, we cannot do better than a linear search which will take time in $\Theta(n)$
  2. to update $e(v)$ for the v that are neighbors of w will take time in $O(n)$ (for a sparse graph, represented by adjacency lists, it will take constant time).

  As we do the above $\Theta(n)$ times, we conclude that with **linear representation**, the **running time of Prim's algorithm** is in $\mathbf{\Theta(n^2)}$.

- **Priority Queues**: we may organize the edges as a *priority queue*, which as described previously in this course allows us in logarithmic time to *(i)* retrieve an edge with minimum weight and subsequently delete it *(ii)* replace an edge $e(u)$ with one that has smaller weight. Then

  1. we will $\Theta(n)$ times have to retrieve a w with $e(w)$ having minimal weight, and subsequently remove it from the queue; in total this contributes $\Theta(n \lg(n))$ to the overall running time
  2. we will $\Theta(a)$ times have to process an edge from the retrieved nodes and potentially spend time in $\Theta(\lg(n))$ to insert it into the priority queue (replacing an edge with higher weight); in total this contributes $\Theta(a \lg(n))$ to the overall running time. (This assumes an adjacency list representation of the graph; with an adjacency matrix representation, we must add $\Theta(n^2)$ which is worse for sparse graphs.)

  Since we assume the graph to be connected, and thus $a \in \Omega(n)$, we conclude that with **priority queue representation**, the **running time of Prim's algorithm** is in $\mathbf{\Theta(a \lg(n))}$.