4041 Homework 7

Fletcher Gornick

December 3, 2021

23.1

23.1-3

Show that if an edge $(\mathbf{u}, \mathbf{v})$ is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

Let's first start with the definition of a minimum weight spanning tree. Minimum spanning trees have two properties; first, they must not contain any cycles, this comes from the definition of a tree. Second, the sum of all the weights of the edges must the smallest possible, hence the "minimum weight".

Let's call our minimum spanning tree T, with it's correspond set of edges A. Now let's assume, to the contrary, that our edge (u, v) is contained in T, but is not a light edge. Since this is a tree, if we remove the edge (u, v), we'll have two disjoint subtrees, we'll call them T_1' and T_2'. Since these subtrees are disjoint, there exists a cut (T_1', T_2') that respects $A \setminus \{(u, v)\}$.

Now, in order for us to obtain a minimum spanning tree, we must choose an edge that crosses our cut, connecting the two disjoint subtrees, and creating a new minimum weight spanning tree. We will call this edge (x, y), and since our tree must have a minimum weight over all, it must be the case that (x, y) is a light edge.

Now if we have our new tree T' containing minimum weight edge (x, y), we can now re-introduce our edge (u, v), but since both (x, y) and (u, v) cross the cut previously mentioned, then we have a cylce. Therefore, we can exclude the edge with a larger weight, which would be (u, v), thus giving us a new minimum weight spanning tree without (u, v), which contradicts our original claim.

23.2

23.2-2

Suppose that we represent the graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $\mathbf{O(V^2)}$ time.

This question asks us to implement our Prim's algorithm with an adjacency matrix and without using a priority queue. If we want our algorithm to run in $O(V^2)$ time then we can just create an array of key values indexed by their vertice number. So node i would have a key of key[i], where key is our array.

This key array would be initialized with all ∞'s, then key[*source node id*] would be set to 0. Then, every time we "extract" the minimum vertex, we just go through and find the minimum, then once we find it, update it's value to ∞. So our implementation would look like this ...

```
PRIM-V2(G, W, r)
 1  i = 0
 2  array keys = new length-|V| array
 3  for each u ∈ G.V
 4       u.key = ∞
 5       u.π = NIL
 6       keys[i] = ∞
 7       u.id = i++
 8  r.keys[i] = 0
 9  keys[r.id] = 0
10  while true
11       double min = ∞
12       int min_idx = 0
13       for j = 0 to keys.length − 1
14            if min > keys[j]  &  keys[j] ≠ −1
15                 min = keys[j]
16                 min_idx = j
17                 keys[j] = −1
18       if min < ∞
19            for each k ∈ G.Adj[min_idx]
20                 if −1 < keys[k] < ∞  &  w(min_idx, k) < keys[k]
21                      k.π = min_idx
22                      k.key = w(min_idx, k)
23                      keys[k] = w(min_idx, k)
24       else break
```

Setting keys[*node*] equal to -1 is the equivalent of taking it out of the array, because we are assuming no negative wieghts.

24.3

24.3-2

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

Take the graph of nodes a, b, c, where our adjacency list / matrix contains 1 adjacent node for each node, giving us $Adj[a] = b, Adj[b] = c, Adj[c] = a$. Let's give each of these edges the following weights, $\ldots w(a, b) = 1, w(b, c) = 1, w(c, a) = -3$.

Say we want to use Dijkstra's algorithm to find the shortest path from node a to the others. We would first initialize nodes with a as the source, meaning that $a.d = 0$ while the rest are ∞. Then we would create our set S of nodes representing out growing tree of nodes with a minimum distance from a. We would also have a priority queue Q with all the nodes.

First we pop off a, because it has the only 0 weight, then union it with S. Now we call relex function on all adjacent nodes to a, which is just b, so $b.d$ is upadated to $a.d + w(a, b) = 0 + 1 = 1$. And we get $b.\pi = a$. Next, we pop b from Q, and union it with S. Then go through the adjacent nodes, which is just c. Now calling relax, we get $c.d = b.d + w(b, c) = 1 + 1 = 2$, and $c.\pi = b$. Finally we do one last loop, popping off c from Q, unioning it with S. Now, since we have negative weight edges, $a.d$ gets updated from 0 to $1+1-3 = -1$, and we're done because Q is now empty.

So with Dijkstra's algorithm, we have the shortest path from a to b, c and itself are of weight $1, 2, -1$ respectively. But the true shortest paths from a to any other node, including itself is $-\infty$, because we could just loop the whole cycle to reduce a path length by one every time, and do it infinite times.

The reason the theorem falls through for negative-weight edges is because we check for minimum weight assuming edges are always positive, so once we've called the relax function for every node in the Q, there'd be no point to go any further, because any further iterations would find that the minimum couldn't be reduced. This changes if instead we were to use negative-weight edges.

24.3-6

We are given a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ on which each edge $(\mathbf{u}, \mathbf{v}) \in \mathbf{E}$ has an associated value $\mathbf{r}(\mathbf{u}, \mathbf{v})$ which is a real number in the range $\mathbf{0} \leq \mathbf{r}(\mathbf{u}, \mathbf{v}) \leq \mathbf{1}$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $\mathbf{r}(\mathbf{u}, \mathbf{v})$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

We can use a variation of Dijkstra's algorithm for this, but instead of comparing weight in the relax function, we can compare our reliability constant. So our new relax function would look like this...

RELAX_R(u , v , r)
  if $v.d < u.d * r(u, v)$
    $v.d = u.v * r(u, v)$
    $v.\pi = u$

In this case, for a given node v, $v.d$ represents the overall reliability from the source node to v, which is also in between 0 and 1. And for this type of question, we're looking for nodes with a larger reliability value, whereas in the original implementation we were looking for smaller weights. Since our d values are between 0 and 1 though, and they mean different things, we will also need to change our initialize single source function like so ...

INITIALIZE_SINGLE_SOURCE_R(G, s)
  for each vertex $v \in G.V$
    $v.d = 0$
    $v.\pi = \text{NIL}$
  $s.d = 1$

Since we don't yet have any paths from s to another node, the reliability is 0. And since s is our source, it starts at 100% reliability, meaning $s.d = 1$.

Now we can just used Dijkstra's algorithm like normal, except with these algorithms called instead. One more caveat is that is that our priority queue Q should actually be a max-heap, because we want higher reliability values. So instead of EXTRACT-MIN, we would call EXTRACT-MAX. Other than that, we don't need to change anything else. This algorithm runs just like Dijkstra's in $O(E \lg(V))$ time, assuming we're using a priority queue.

25.1

25.1-3

What does the matrix

$$\mathbf{L}^{(\mathbf{0})} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

used in the shortest-paths algorithms correspond to in regular matrix multiplication?

This matrix represents the identity matrix for shortest-path algorithms. Basically, each i, j position on the matrix represents the length of the shortest 0-length path from i to j. And since the only occasion where a 0-length path from i to j exists, is when $i = j$, the entries where $i \neq j$ contain ∞ because you can't travel from a node to a different node with 0 steps. The diagonal consists of 0's because the length of a path from one node to the same node is 0.

25.1-4

Show that matrix multiplication defined by `EXTEND-SHORTEST-PATHS` is associative.

We know that ordinary matrix multiplication is associative. And if we look at the algorithm for `EXTEND-SHORTEST-PATHS`, we see that by simply replacing "min" with "+" and "+" with "·", we get the actual algorithm for matrix multiplication. So if we can show that this tropical semiring also obeys the associative property, then it suffices to show that our matrix multiplication defined by `EXTEND-SHORTEST-PATHS` is also associative.

Let's take 3 $n \times n$ matrices X, Y, and Z. We will show that $(XYZ)_{ij} = (XY)_{il}Z_{lj} = X_{ik}(YZ)_{kj}$.

$$(XY)_{ik}Z_{kj} = \min_{1 \le a \le n}(XY)_{ia} + Z_{aj} = \min_{1 \le a \le 1}\left(\min_{1 \le b \le n} X_{ib} + Y_{ba}\right) + Z_{aj}$$

Now for any given a where $1 \le a \le n$, there exists a b where $1 \le b \le n$ such that $X_{ib} + Y_{ba}$ is a minimum. Then you can take all the minimum values of $X_{ib} + Y_{ba}$ for each a, and add that with Z_{aj} for every possible a. Then, taking the minimum of those, you get the smalles possible value for $(XYZ)_{ij}$. Thus, $(XY)_{ik}Z_{kj} = \min_{1 \le a \le 1}\min_{1 \le b \le n} X_{ib} + Y_{ba} + Z_{aj} = (XYZ)_{ij}$. We also have

$$X_{il}(YZ)_{lj} = \min_{1 \le a \le n} X_{ia} + (YZ)_{aj} = \min_{1 \le a \le 1} X_i a + \left(\min_{1 \le b \le n} Y_{ab} + Z_{bj}\right)$$

. Without loss of generality, we also get that this equates to $(XYZ)_{ij}$. So we've shown that

$$(XYZ)_{ij} = (XY)_{il}Z_{lj} = X_{ik}(YZ)_{kj}$$

Therefore, it must be the case that this new form of matrix multiplication is also associative.

25.1-6

Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $\mathbf{O(n^3)}$ time.

We're givin matrix L where l_{ij} represents the weight of the shortest path from node i to j. So if we want to find the penultimate vertex representing the last node visited on the path from i to j, we must find a vertex k such that $l_{ik} + w(k, j) = l_{ij}$. In other words, the shortest length path from i to k plus the weight of the edge from k to j should be the shortest length path from i to j.

So for our algorithm, we can just denote our π_{ij} value as the number that corresponds to the penultimate node on the path from i to j. Assuming L (and consequently Π) is an $n \times n$ matrix, the values in π_{ij} would range from 1 to n. Also 0 corresponds to no penultimate node existing, i.e. the path from i to j doesn't exist, or $i = j$, because if $i = j$ then we don't need to know what came before. Here's what our algorithm looks like...

```
FIND-PREDECESSOR-MATRIX(L, W)
1  let Π be a new n × n matrix of all zeros
2  for i = 1 to n
3       for j = 1 to n
4            for k = 1 to n
5                 if l_ik + w_kj = l_ij
6                      π_ij = k
7  return Π
```

This algorithm has 3 for loops going through n elements each, making the running time $O(n^3)$.

25.2

25.2-4

As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \ldots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

```
FLOYD-WARSHALL'(W)
1  n = W.rows
2  D = W
3  for k = 1 to n
4       for i = 1 to n
5            for j = 1 to n
6                 d_ij = min(d_ij, d_ik + d_kj)
7  return D
```

We know by definition, that if $C = A \cdot B$, then $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. And we also know that $D^k = D^{k-1} D$, so we can rewrite $d_{ij}^{(k)} = \sum_{l=1}^{n} d_{il}^{(k-1)} d_{lj}$. And using the tropical analogue by treating "+" as "min" and "x" as "+", we get $d_{ij}^{(k)} = \min_{1 \le l \le n} d_{il}^{(k-1)} + d_{lj}$. This formula would give us the shortest k-length path from i to j, but it doesn't take into account the fact that there could possibly be shorter paths of length less than k.

To fix this, if we simply start with $k = 1$ and work up to $k = n$, we can make sure we have the smallest path from i to j by taking $\min(d_{ij}, d_{ik} + d_{kj})$, where d_{ij} is the shortest $k - 1$ or shorter length path. We also initialize D to the identity matrix in question 25.1-3 to cover the zero-length case. And through induction, by the time we've accounted for all paths of lenth $1 \le k \le n$, d_{ij} will have the smallest weight. This algorithm I described is exactly what's written above.

Also, If you look at the original `FLOYD-WARSHALL` algorithm in the textbook, you see that the values for D^k are solely dependent on D^{k-1}, so if for every iteration we find a new value for $d_{ij}^{(k)}$, we know that the next $d_{ij}^{(k+1)}$ value will only be dependent on D^k, therefore, inductively, we only need to keep track of the $k - 1$ matrix to solve for the k matrix.

Technically, if you wanted to make sure the algorithm does exactly what you want it to, you could just have two matrices, one for the $k-1$ case, and one for the k case. This would still only be $O(n^2)$ space, but it's pretty clear that this is unnecessary.

The reason having two matrices is unnecessary is because every time we update a value in the D^k matrix, it's a new minimum, so calling $d_{ij} = \min(d_{ij}, d_{ik} + d_{kj})$, even if $d_{ik} + d_{kj}$ might actually correlate with the D^k matrix and not the D^{k-1} matrix, it's still finding the smallest possible value for either $d_{ij}^{(k)}$ or $d_{ij}^{(k+1)}$. And since k is increasing to n, and any matrix D^m where $m > n$ will have all the same values as D^n (because adding an edge that's already been used doesn't increase efficiency), we don't actually care if our value isn't 100% reliant on D^{k-1}. Actually, not only is this way more space efficient, it's also faster, because on the cases where d_{ik} and d_{kj} were already found for D^k, d_{ij} will approach it's answer even faster, because it's basically skipping the $k-1$ step for free.