CS 5800 Assignment 3 Solutions

Course Staff

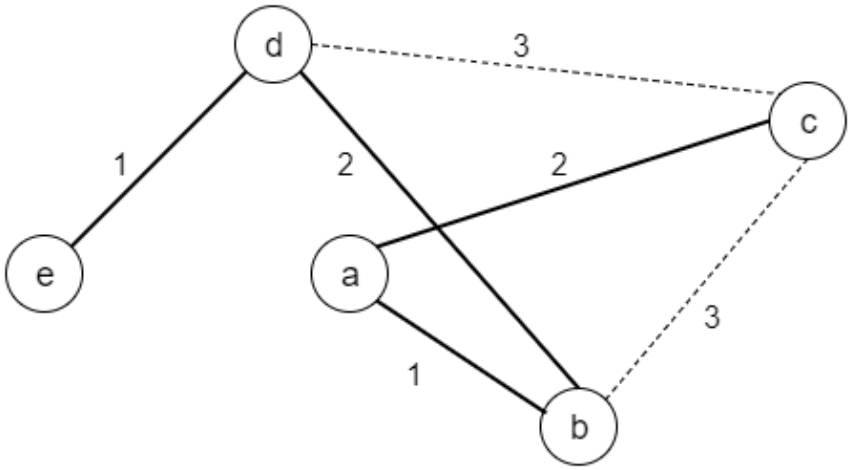
August 6, 2017

1 Prim's example

In Prim's algorithm, at each step we hold a tree $T_k = (V_k, E_k)$ where $V_k = \{v_1, ..., v_k\} \subseteq V$ and $E_k = \{e_1, ...e_{k-1}\} \subseteq E$ and a heap H_k whose nodes are the edges of the cut $(V_k, V - V_k)$ ordered by the weights of edges in the graph G. We start from a random vertex and create a heap corresponding to it along with an empty edge set for the MST we will build. We select edge with the minimal weight available at the root of the heap and then update the heap. We keep on repeating this process until we get our MST.

1. Initializing $V_1 = \{a\}, \quad E_1 = \phi, \quad H_1 = \{e_{ab}, e_{ac}\}$
2. $e_1 = e_{ab} \quad v_2 = b$
   $V_2 = \{a, b\} \quad E_2 = \{e_{ab}\} \quad H_2 = \{e_{ac}, e_{bd}, e_{bc}\}$
3. $e_2 = e_{ac} \quad v_3 = c$
   $V_3 = \{a, b, c\} \quad E_3 = \{e_{ab}, e_{ac}\} \quad H_3 = \{e_{bd}, e_{cd}\}$
4. $e_3 = e_{bd} \quad v_4 = d$
   $V_4 = \{a, b, c, d\} \quad E_4 = \{e_{ab}, e_{ac}, e_{bd}\} \quad H_4 = \{e_{de}\}$
5. $e_4 = e_{de} \quad v_5 = e$
   $V_5 = \{a, b, c, d, e\} \quad E_5 = \{e_{ab}, e_{ac}, e_{bd}, e_{de}\} \quad H_5 = \phi$

Hence our MST will have total weight of 6 with edges $\{e_{ab}, e_{ac}, e_{bd}, e_{de}\}$.



2 BFS example

For this example, we assume an adjacency-list representation in which each vertex's adjacency list is sorted alphabetically by label. (Different orderings would mean different intermediate state and back-pointers, but they would produce the same distances.)

After initialization, we have state as follows:

a	b	c	d	e	f	g
0	∞	∞	∞	∞	∞	∞
-	-	-	-	-	-	-

$Q = \{a\}$

In the first iteration of the loop body, we dequeue a and process its neighbors before coloring a black.

a	b	c	d	e	f	g
0	*1*	*1*	∞	∞	*1*	∞
-	a	a	-	-	a	-

$Q = \{b, c, f\}$

Next we dequeue b and process its neighbors. Since we have already seen both a and c, there is nothing we need to enqueue.

a	b	c	d	e	f	g
0	*1*	*1*	∞	∞	*1*	∞
-	a	a	-	-	a	-

$Q = \{c, f\}$

Then we dequeue c and process its neighbors. Since we have already seen a and b, we add d and g to the queue.

a	b	c	d	e	f	g
0	*1*	*1*	*2*	∞	*1*	*2*
-	a	a	c	-	a	c

$Q = \{f, d, g\}$

Then we dequeue f and process its neighbors, stolidly ignoring a's cries for additional attention.

a	b	c	d	e	f	g
0	*1*	*1*	*2*	*2*	*1*	*2*
-	a	a	c	f	a	c

$Q = \{d, g, e\}$

The rest of the example

a	b	c	d	e	f	g

$Q = \{g, e\}$

is pretty

a	b	c	d	e	f	g

$Q = \{e\}$

uninteresting.

a	b	c	d	e	f	g

$Q = \{\}$

3 Computing d_3

We will use the following example to illustrate the idea of the solution.
Let G = (V,E) be an undirected graph such that

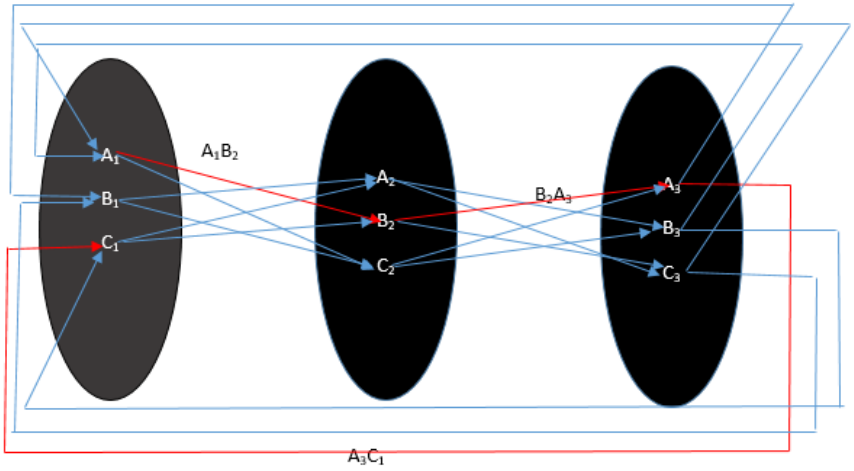
- V = $\{A, B, C\}$
- E = $\{AB, BC, CA\}$

1. We construct a new directed graph G'=(V',E') such that:-

   - V' has three copies of each vertex in V:
     V' = $\{A_1, B_1, C_1, A_2, B_2, C_2, A_3, B_3, C_3\}$

   - We define E' in the following way. For each edge in $\{u, v\} \in E$ we will define 6 different edges in E'. For the edge (A, B) (for example) we will have the edges (A_1, B_2), (A_2, B_3),(A_3, B_1) and (B_1, A_2),(B_2, A_3),(B_3, A_1). In this manner we will get the following edges:

$$\begin{aligned} E' = \{&(A_1, B_2), (A_1, C_2), (B_1, A_2), (B_1, C_2), (C_1, A_2), (C_1, B_2), \\ &(A_2, B_3), (A_2, C_3), (B_2, A_3), (B_2, C_3), (C_2, A_3), (C_2, B_3), \\ &(A_3, B_1), (A_3, C_1), (B_3, A_1), (B_3, C_1), (C_3, A_1), (C_3, B_1)\} \end{aligned}$$



2. To find $d_3(v_0 = A, v \in V)$, we run the BFS algorithm on G' with A_1 as the source. The distances calculated by the BFS algorithm for the first copy of V in the graph G', are the values we are looking for. That is:

$$d_{G'}(A_1, B_1) = d_3(A, B)$$

$$d_{G'}(A_1, C_1) = d_3(A, C)$$

To see that this is correct note that there is a correspondance between paths of length $3k$ in G to paths in G'. For example the path (of length 3) $A \to B \to A \to C$ in G, corresponds to the path $A_1 \to B_2 \to A_3 \to C_1$ (See the red path in the picture)

4 DFS forests

4.a

Let (u, v) be an edge in G. It is clear from the given inequalities that only in the case of the back edge do we have $f(u) < f(v)$, since in every other case $f(v) < f(u)$. This follows from the fact that a back edge by definition is an edge that connects a vertex u to its ancestor. Since v is u's ancestor in a DFS tree, it must have been discovered before u ($d(v) < d(u)$) and it can only finish after u and its descendants ($f(u) < f(v)$). Finishing time for u has to be greater than its discover time, so $d(u) < f(u)$. When we put all the inequalities together, we get the result: $d(v) < d(u) < f(u) < f(v)$

4.b

Part 1- (G has back edge $\implies$ G has a cycle)
Let $e = (u, v)$ be a back edge in G. It follows that e connects a node u to its

ancestor v, so there is a path $p(v,u) = v \to v_1 \to ... \to u$. Adding the edge (u,v) to this path creates a cycle:

$$v \to v_1 \to ... \to u \to v$$

Part 2- (G has a cycle $\implies$ G has a back edge)
Let $v_1 \to v_2 \to ... \to v_n \to v_1$ be a simple cycle in G. Consider the finishing times $f(v_1), f(v_2), ..., f(v_n)$. If we have $f(v_1) < f(v_2)$ then it follows from part (a) that (v_1, v_2) is a back edge. Therefore, if (v_1, v_2) is not a back edge, we must have $f(v_1) > f(v_2)$. Similarly, if there is no backedge in the cycle we get:

$$f(v_1) > f(v_2) > ... > f(v_n) > f(v_1)$$

This is clearly a contradiction.

4.c

We can use our results from parts (a) and (b) to suggest an algorithm.

Given a directed, graph $G = (V, E)$, we need to determine if its acyclic or not. We begin by applying the DFS algorithm on G. After the DFS algorithm finishes, we check for each edge $(u, v) \in E$, whether $f(u) < f(v)$. If there is no edge $(u, v) \in E$ for which $f(u) < f(v)$, then the graph is acyclic and we return true. Otherwise there is a back edge in G and hence there is a cycle in G, so it is not acyclic.

5 MakeHeap

(a) The HeapSort(A,n) algorithm is:

1. MakeHeap(A,n). //This turns A into a heap.
2. For $i = n$ down to 2
3. Exchange $A[1]$ and $A[i]$ //We move the smallest element to the ith place
4. Heapfy(A[1],...,A[i-1]) //We turn $A[1, ..., i-1]$ back into a heap.

Note that we sort the array such that $A[1] \geq A[2] \geq ... \geq A[n]$ (assuming that in our heap each node is smaller than its children).

(b) The fact that the MakeHeap(A,n) algorithm performs $O(n)$ comparisons implies that there exists a constant $c > 0$ such that the comparisons tree of the algorithm have at most 2^{cn} leafs.

Given some input array A of length n, the MakeHeap(A,n) algorithm performs at most cn comparisons and produces some permutation π which turns the input array into a heap. Since (as we said) there are at most 2^{cn} leafs, it

follows that there are at most 2^{cn} possible permutations that the algorithm can produce.

Clearly there are $n!$ possible inputs for the algoritm. The algorithm matches each of the $n!$ possible input arrays to one of the 2^{cn} permutations. It follows that at least one permutation corresponds to at least $\frac{n!}{2^{cn}}$ different input arrays. Applying this permutation on each of these (more than) $\frac{n!}{2^{cn}}$ arrays will give a different heap. It follows that there exist at least $\frac{n!}{2^{cn}}$ different heaps.

(c)As explained in the question, we have:

$$\sum_{k=0}^{n} kx^k = x \cdot \left(\frac{1-x^{n+1}}{1-x}\right)' =$$

$$x \cdot \frac{(1-x^{n+1}) - (n+1)x^n(1-x)}{(1-x)^2} =$$

It follows that for $x = \frac{1}{2}$ we have:

$$\sum_{k=0}^{n} k\left(\frac{1}{2}\right)^k = \frac{1}{2} \cdot \frac{\left(1-\frac{1}{2}^{n+1}\right) - (n+1)\left(\frac{1}{2}\right)^n\left(1-\frac{1}{2}\right)}{\left(1-\frac{1}{2}\right)^2} \underset{n\to\infty}{\longrightarrow} \frac{1}{2} \cdot \frac{1}{\frac{1}{4}} = 2$$

To compute the number of operations of the MakeHeap algorithm we need to sum up the operations of preformed by heapfy algorithms from the bottom of the tree to the top. Overall the tree has approximately $\log_2 n$ levels. At level i we have (at most) 2^i nodes and each heapfy call will perform $\log_2 n - i$ operations. Summing the operations of all the levels together we get:

$$\sum_{k=\lceil \log_2 n\rceil - 1 \text{ down to } 1} (2^k \cdot (\lceil \log_2 n\rceil - k)) =$$

$$2^{\lceil \log_2 n\rceil - 1}(\lceil \log_2 n\rceil - (\lceil \log_2 n\rceil - 1)) + 2^{\lceil \log_2 n\rceil - 2}(\lceil \log_2 n\rceil - (\lceil \log_2 n\rceil - 2)) ... + 2^1(\lceil \log_2 n\rceil - (1)) =$$

$$= \sum_{k=1}^{\lceil \log_2 n\rceil - 1} (2^{\lceil \log_2 n\rceil - k} \cdot k)$$

$$= 2^{\lceil \log_2 n\rceil} \sum_{k=1}^{\lceil \log_2 n\rceil - 1} (2^{-k} \cdot k) =$$

$$\leq 2^{\lceil \log_2 n\rceil} \cdot \sum_{k=1}^{\infty} (2^{-k} \cdot k) = \leq 2^{\lceil \log_2 n\rceil + 1} = O(n)$$