

**LECTURE NOTES FOR CMSC 651
VARIANTS OF BORUVKA'S ALGORITHM**

1 Introduction

Boruvka's algorithm grows disjoint trees of low weight and then puts them together. It runs in time $O(E \log V)$. There are two bottlenecks:

1. Every time it starts a new phase it has to recompute all of the shortest distances from trees to trees.
2. There are $O(\log V)$ phases.

Yao's algorithm for MST is a variant of Boruvka's that does preprocessing so that computing the shortest distance between trees is shorter. Fredman-Tarjan algorithm is a variant of Boruvka's that has substantially less phases.

2 Yao's Algorithm

The input is a weighted graph $G = (V, E, w)$ where $w : E \rightarrow \mathbb{N}$. We assume all the weights are different. k is a parameter to be named later. We will use V for both V and $|V|$. We will use E for both E and $|E|$. We assume the vertices are $\{1, 2, \dots, |V|\}$.

We present the following:

1. INIT algorithm. Initializes the process and does some preprocessing.
2. MAIN ALGORITHM. This is Yao's Algorithm but it calls a procedure CHEAPT.
3. CHEAPT. On input i this procedure outputs the cheapest edge (v, u) such that $v \in T_i$ and $u \notin T_i$. Over the course of the MAIN ALGORITHM edges will be removed from the original graph. CHEAPT will output the cheapest edge that is still in the graph. This procedure calls a procedure called CHEAPV.
4. CHEAPV. On input v this procedure outputs the cheapest edge (v, u) such that u and v are in different trees.

INITIALIZATION

FOR ALL $v \in V$

Partition the edges with one endpoint in v into k sets E_1^v, \dots, E_k^v where, for all $a < b$, every edge in E_a^v has a lesser weight than every edge in E_b^v . All of the E_j^v 's are roughly the same size, $d(v)/k$. This step takes $O(E \log k)$ steps. All vertices know which E_j^v they are in.

Relink the adjacency list for v by having it first have vertices in E_1^v , then E_2^v , etc.

Also add to the end of the adjacency list for v a virtual vertex NULL which is in E_{k+1}^v and has $w(v, \text{NULL}) = \infty$.

ENDFOR

FOR $i = 1$ TO V let $T_i = \{i\}$. (These are our initial trees.)

$t = V$. (t will be the number of trees. Initially there are V trees.)

$MST = \emptyset$. (Initially the MST is empty.)

END INITIALIZATION

Running time: Initialization takes $O(E \log k)$ steps. (We did this in class.)

MAIN ALGORITHM

WHILE $t \geq 2$

FOR $i = 1$ TO t MARK(i) = FALSE ENDFOR (initially all of the T_i 's are unmarked)

FOR $i = 1$ TO t WITH MARK(i) = *FALSE*.

$(u, v, L) = \text{CHEAPT}(i)$ (Recall that (v, u) is cheapest edge with $v \in T_i$, $u \notin T_i$ and $u \in T_L$)

$MST = MST \cup \{(v, u)\}$.

MARK(L) = TRUE. (It's possible MARK(L) was already true. That's okay.)

UNION(T_i, T_L) (The smaller tree gets destroyed.)

ENDFOR

t gets the current number of trees.

Renumber trees so that they are numbered $\{1, \dots, t\}$.

ENDWHILE

END MAIN ALGORITHM

Running time: Initially $t = V$. Every time through the WHILE t gets reduced by at least half. Hence the number of times that CHEAPT is called in the L th time through the loop is bounded by roughly $V/2^L$. Hence the number of times CHEAPT is called is $O(V)$.

CHEAPT

Input is i but the values of G, T_1, \dots, T_t are known global variables.

MIN = ∞ . (Initially we don't have a cheapest cost edge.)

EDGE = *NULL* (Initially we don't know the cheapest edge out of T_i .)

ENDTREE = ∞ (The cheapest edge out of v will go to T_{ENDTREE} .)

FOR $v \in T_i$

$(u, v, L) = \text{CHEAPV}(v)$: (Recall that (v, u) is the cheapest edge with $u \notin T_i$, and $u \in T_L$. If there are no such vertices out of v then $u = \text{NULL}$ and $w(v, u) = \infty$.)

IF $w(v, u) < \text{MIN}$ then

MIN = $w(v, u)$

EDGE = (v, u)

ENDTREE = L

ENDIF

ENDFOR

RETURN(EDGE, L)

END OF CHEAPT

Now we need a procedure that will, given $v \in T_i$, will tell us what the cheapest edge with one endpoint in v and the other endpoint not in T_i .

CHEAPV

Input is v but algorithm knows global variables G, T_1, \dots, T_t, i .

FOUNDJ = FALSE (Look for least j , E_j^v has (v, u) where $u \notin T_i$).

$u = \text{FIRST}(v)$. The first element on the ADJ list for v .

WHILE FOUNDJ = FALSE

$L = \text{FIND}(u)$

 IF $L = i$ then remove edge (v, u) from graph ENDIF

 ELSE (so $L \neq i$)

 IF $L = k + 1$ THEN RETURN(NULL) ENDIF

 ELSE

$j = \text{INDEX}(u)$ (This is the j such that $u \in E_j^v$.)

 FOUNDJ = TRUE

 SMALL = $w(v, u)$ ((v, u) is the cheapest edge out of v so far.)

 RETURN u

 ENDELSE

 ENDELSE

 IF FOUNDJ = FALSE THEN $u = \text{NEXT}(u)$ ENDIF

END WHILE

WHILE $\text{INDEX}(u) = j$

$u = \text{NEXT}(u)$

$f = \text{FIND}(u)$

 IF $w(v, u) < \text{SMALL}$ and $L \neq i$ THEN

 SMALL = $w(v, u)$

$L = f$

 RETURN u

 ENDIF

ENDWHILE

RETURN(RETURNU, v , L)

END CHEAPV

We figure the running time for the entire algorithm.

The initialization takes, $O(E \log k)$.

During a phase every tree gets UNIONED with another tree. Hence the number of trees gets halved. So there are $O(\log V)$ phases.

We count the number of times FIND is called. There are two kinds of calls to FIND:

1. $v \in T_i$ and (v, u) is an edge and $\text{FIND}(u) = i$, and this happens before you get to the correct j . When this happens the edge (v, u) is removed. Hence this kind of FIND happens at most $O(E)$ times.
2. $v \in T_i$ and you have determined the proper j such that (v, u) is an edge, $\text{FIND}(u) \neq i$, and $u \in E_j^v$. In the worst case you will do FIND to every element in E_j^v (this includes the one to u that alerted you to the fact that j is the correct segment to look at). Hence you will do $d(v)/k$ FIND's. During one phase this is $\sum_v d(v)/k = O(E/k)$ FIND's. Since there are $\log V$ phases you have $O(\frac{E \log V}{k})$ FIND's.

Hence the total number of FINDS is $O(E + (E/k) \log V)$ which takes

$$O((E + (E/k) \log V) \log^* V) = O(E \log^* V + (E/k)(\log V) \log^* V).$$

The number of UNION is clearly $O(V)$ which takes $O(V)$. Since this is dominated by $O(E \log^* V)$ (even by $O(E)$) we ignore this cost.

How much time do we spend finding $\text{SMALL}(v)$ after we find the j ? There are only $d(v)/k$ elements to look at. Hence during one phase we make $O(\sum_v d(v)/k) = O(E/k)$ comparisons. There are $\log V$ phases. Hence the total number of comparisons is

$$O((E/k) \log V).$$

Thus the total is

$$O(E \log k + E \log^* V + (E/k)(\log V) \log^* V + (E/k) \log V).$$

We take $k = \log V$ to obtain

$$O(E \log \log V + E \log^* V + E \log^* V + E) = O(E \log \log V).$$

Hence this algorithm takes $O(E \log \log V)$ steps. Recall that PRIM's algorithm takes $O(V \log V + E)$ steps. Hence Yao's algorithm is better than PRIM's when $E \ll V \frac{\log V}{\log \log V}$.

3 Fredman-Tarjan Algorithm

In Bourvka's and Yao's algorithm we have a set of trees and merge them. In Fredman-Tarjan we may merge far more than two into one tree. This will cut down on the number of phases. The idea is that we view the trees as vertices and do PRIM's algorithm several times with different starting points; however, we will always stop it when either k (a parameter that will change with the phase) vertices are in the the Fib Heap OR we have merged with a tree that has already been through the process.

When we say we view trees as vertices we really mean it— there will be a cleanup phase where we shrink all of the trees into vertices. The new graph will be denoted $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$. The weight of an edge between these new vertices will be the cheapest edge from tree to tree. Computing this will be expensive ($O(E)$ per phase) but there will be so few phases that this will be okay.

Formally we define the following procedures.

We present the following:

1. INIT algorithm. Initializes the process and does some preprocessing.
2. MAIN ALGORITHM. This is Fredman-Tarjan Algorithm but it calls a procedures PRIMK and CLEANUP. The main algorithm has a WHILE loop. Every time through it we set a parameter k . We will tell you what k is later— but be warned— it will change in every iteration! It will depend on t , the number of trees still left.
3. PRIMK(k, i): When this is called there are t trees T_1, \dots, T_t that we will really regard as vertices. They will already have edges between them. T_1, \dots, T_{i-1} have already been dealt with. PRIMK(k, i) will do PRIM's algorithm starting with vertex T_i but with the following modifications.
 - (a) If T_i is marked (meaning that some earlier tree merged with it) then we do not do this.
 - (b) There are a few variants of PRIM's algorithm so we need to be clear which one we are using. In some versions one INSERTS all of the vertices into the F-HEAP at the beginning with d -values ∞ . We will not do this. We will insert elements into the F-HEAP when there d -value is first defined (and is finite).

- (c) The algorithm will STOP if the F-HEAP has k elements.
 - (d) The algorithm will STOP if the tree you are building merges with some T_j with $j < i$ (a T_j that has already been processed or merged.)
4. CLEANUP: We have a set of trees but we want to regard them as vertices. This operation finds the cheapest edge from tree to tree. After CLEANUP the cheap edges found knows BOTH which two trees it connects AND which two of the original vertices it connected.

INITIALIZATION

FOR $i = 1$ TO V

$T_i = \{i\}$ (These are our initial trees.)

MARK(i) = FALSE

$t = V$. (t will be the number of trees. Initially there are V trees.)

$MST = \emptyset$. (Initially the MST is empty.)

$\mathcal{G} = G$. (Initially we will just use the original graph.)

END INITIALIZATION

Running time: Initialization takes $O(V)$ steps.

MAIN ALGORITHM

WHILE $t \geq 2$

$k =$ (We will fill this in later but it will be a function of t)

FOR $i = 1$ TO t PRIMK(k, i) ENDFOR

CLEANUP

ENDWHILE

END MAIN ALGORITHM

Let $t_1 = t$ and let $t_2, t_3, \dots, t_L \leq 1$ be the values of t going into the WHILE loop. Let k_i be the value when $t = t_i$. Then the number of steps this takes is

$$\sum_{j=1}^L \sum_{i=1}^{t_j} (\text{time PRIMK}(k_j, i) \text{ takes}) .$$

INITPRIMK

Input is (k, i) but the algorithm knows global variables $T_1, \dots, T_t, G, \mathcal{G}$.

FOR $j = 1$ TO t

$d(T_j) = \infty$

$e(T_j) = NULL$

ENDFOR

$F = \emptyset$ (F is an F-HEAP)

MERGED = FALSE

NUMF = 0 (The number of elements in the F-heap.)

FOR $S \in \text{ADJ}_{\mathcal{G}}(T_i)$ (Neighbors of T_i in \mathcal{G} .)

IF NUMF < k THEN

$d(S) = \mathcal{W}(S, T_i)$

$e(S) = (T, S)$

INSERT(F, S)

NUMF = NUMF + 1

ENDIF

ENDFOR

END INITPRIMK

PRIMK

INITPRIMK(k, i). (F , MERGED, NUMF, some d -values are initialized.)

WHILE (NUMF < k AND MERGED = FALSE)

$T_j = \text{FINDMIN}(F)$

IF MARK(j) = *TRUE* THEN MERGED = TRUE

FOR $S \in \text{ADJ}_{\mathcal{G}}(T_j)$ (Neighbors of T_j in \mathcal{G} .)

IF $S \notin F$ and NUMF < k THEN

INSERT(S)

NUMF = NUMF + 1

ENDIF

IF $\mathcal{W}(T_j, S) < d(S)$ THEN

$d(S) = \mathcal{W}(T_j, S)$

$e(S) = (T_j, S)$

ENDIF

ENDFOR

$U = e(T_j)$

u, v are vertices of U, T_j that cause $\mathcal{W}(U, T_j) = w(u, v)$.

$MST = MST \cup \{(u, v)\}$.

DELETEMIN (this removes T_j from F)

NUMF = NUMF - 1

MARK(j) = *TRUE*

ENDWHILE

MARK(i) = TRUE

END OF PRIMK

At most t DELETEMINS are performed. Each one occurs in an F-heap with $\leq k$ elements. Hence the time spend on DELETEMIN is $O(t \log k)$. Similarly, the time spend on INSERT is $O(t \log k)$. Every time a DECKEY is called it gets traced back to some edge in the original graph. Hence there are (by the usual arguments) $O(E)$ DECKEYS throughout all calls to *PRIMK*.

Using the definitions of t_i and k_i mentioned above, we have that the number of operations taken by all calls to PRIMK is

$$O(E + \sum_{i=1}^L t_i \log k_i)$$

The procedure CLEANUP sets up the graph \mathcal{G} which will be worked on in the next phase of the MAIN ALGORITHM. We leave as an exercise that this take $O(E)$.

Time analysis. Recall that we are assuming there are L phases and the value of t are t_1, \dots, t_L and for k are k_1, \dots, k_L . We get to pick the k_i 's. Putting together the above we get that the time is

$$O(LE \log^* V + \sum_{i=1}^L t_i \log k_i).$$

We pick $k_i = 2^{2E/t_i}$. Hence we obtain

$$O(LE \log^* V + \sum_{i=1}^L E) = O(LE \log^* V).$$

So now the question is, how many phases does the algorithm go through? For this we need to upper bound on the t_i 's.

Note that stage i begins with t_i trees which are regarded as one point each (we refer to such as VERTS) At the end of stage i each VERT has at least k_i edges incident on it. this is because either the VERT stopped growing because it had k_i VERTS incident or it stopped growing because it merged with another VERT that stopped growing. Eventually it goes back to a VERT that stopped growing because it had k_i VERTS incident on it. So to every VERT at the end of stage i we associate the set of edges coming out of it that go to other VERTS.

Map every VERT to the set of $\geq k_i$ edges coming out of it. Take the image of that mapping and count the number of edges. It is at most $2m$. Hence the number of VERTS is at most $2m/k_i$.

Hence we define

$$\begin{aligned} t_1 &= V \\ k_1 &= 2^{2E/t_1} \\ t_i &= 2E/k_{i-1} \\ k_i &= 2^{2E/t_i} \end{aligned}$$

We seek upper bounds on t_i and lower bounds on k_i . For both we need a definition.

Def 3.1 Let $\text{TOW} : \mathbb{N} \rightarrow \mathbb{N}$ be defined as follows.

$$\begin{aligned} \text{TOW}(0) &= 1 \\ \text{TOW}(i+1) &= 2^{\text{TOW}(i)} \end{aligned}$$

We show by induction that

$$\begin{aligned} t_i &\leq 2E/\text{TOW}(i-1) \\ k_i &\geq \text{TOW}(i) \end{aligned}$$

Base Case:

$$t_1 = V = V/1 = V/\text{TOW}(0) = V/\text{TOW}(1-1) \leq 2E/\text{TOW}(i-1).$$

$$k_1 = 2^{E/V} \geq 2 = \text{TOW}(2).$$

Assume true for $i-1$.

$$t_i = 2E/k_{i-1} \leq 2E/\text{TOW}(i-1).$$

$$k_i = 2^{2E/t_i} \geq 2^{\text{TOW}(i-1)} = \text{TOW}(i).$$

(Actually much better bounds are known.)

Hence $L \leq \log^* V$ and the algorithm runs in time

$$O(E(\log^* V)).$$

Hence this algorithm takes $O(E \log^* V)$ steps.

Recall that PRIM's algorithm takes $O(V \log V + E)$ steps. Hence the Fredman-Tarjan algorithm is better than PRIM's when $E \ll \frac{V \log V}{\log^* V}$, of the average degree $d(v) \ll \frac{\log V}{\log^* V}$.