

Lecture notes for “Advanced Graph Algorithms”: Verification of Minimum Spanning Trees

Lecturer: *Uri Zwick* *

November 18, 2009

Abstract

We present a deterministic linear time algorithm for the *Tree Path Maxima* problem. As a corollary, we obtain a deterministic linear time algorithm for checking whether a given spanning tree T of a weighted graph is a minimum spanning tree.

1 Tree Path maxima

The (off-line version) of the *Tree Path Maxima* is defined as follows. Given a weighted tree $T = (V, E_T, w)$ and a collection $E = \{(u_1, v_1), \dots, (u_m, v_m)\}$ of vertex pairs, find for every $1 \leq i \leq m$, an edge of maximal weight on the unique path in T from u_i to v_i . For $u, v \in V$, we let $w_T(u, v)$ be the maximal weight of an edge on the path from u to v in T .

If $G = (V, E, w)$ is a weighted graph and T is a spanning tree of G , then T is a minimum spanning tree of G iff $w(u, v) \geq w_T(u, v)$, for every $(u, v) \in E$. Thus, a linear solution for the Tree Path Maxima problem immediately supplies an algorithm for verifying minimum spanning trees.

Komlós [Kom85] showed that $w_T(u, v)$, for every $(u, v) \in E$ can be found using $O(m + n)$ comparisons. Dixon, Rauch and Tarjan [DRT92] were the first to show that his algorithm can actually be realized in $O(m + n)$ time. Together, they therefore obtained the following theorem:

Theorem 1.1 *There is a deterministic $O(m + n)$ time algorithm that given a weighted undirected graph $G = (V, E, w)$, where $|V| = n$ and $|E| = m$, and a spanning tree T of G , computes $w_T(e)$ for every $e \notin T$.*

King [Kin97] and Hagerup [Hag09] presented simplified linear time algorithms. Our presentation is based on their results.

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: zwick@tau.ac.il

2 Reduction to fully branching trees

The tree T above is arbitrary. We first show that it is sufficient to be able to answer path maxima queries on a particular family of well behaved trees.

Definition 2.1 (Fully branching) *A rooted tree T is said to be fully branching if and only if (i) each internal vertex has at least two children, and (ii) all leaves are at the same depth.*

Let $T = (V, E_T, w)$ be a weighted tree. For simplicity, we assume that all edge weights are distinct. We do the seemingly stupid thing of running Borůvka's algorithm on T . As the number of edges in each iteration is at most the number of vertices, and as the number of vertices in each iteration decreases by at least a factor of 2, this requires only $O(n)$ time, where $n = |V|$ is the number of vertices in T .

Borůvka's algorithm uses a sequence of at most $\log n$ phases to contract T into a single vertex. We construct a weighted rooted tree T' whose vertices are all the super-vertices generated during the operation of Borůvka's algorithm. The leaves of T' are the original vertices of T . The root of T' is the super-vertex obtained by contracting the whole of T into a single vertex. Other vertices of T' correspond to super-vertices obtained during intermediate stages of the algorithm. The parent of a super-vertex u in T' is the super-vertex v of the next phase into which u is contracted. We let $w'(u, v)$, the weight attached to the edge (u, v) of T' be the weight of the edge chosen by u during the iteration of Borůvka's algorithm that contracts it. Note that each internal vertex of T' has at least two children and that all the leaves of T' are at the same depth, hence, T' is fully branching.

Lemma 2.2 (King [Kin97]) *Let $T = (V, E_T, w)$ be an arbitrary tree and let $T' = (V', E', w')$ be the tree obtained by running Borůvka's algorithm on T , as described above. Then, for every $u, v \in V$ we have $w_T(u, v) = w'_{T'}(u, v)$.*

3 Reduction to ancestor-descendant queries

We saw above that we may assume that the tree $T = (V, E_T, w)$ given to us is fully branching. In fact, we may also assume that all queries start and end at leaves.

For $u, v \in V$, we let $LCA(u, v)$ be their *lowest common ancestor* (LCA), i.e., the lowest vertex in T which is an ancestor of both u and v .

We replace each query (u, v) by the two queries $(u, LCA(u, v))$ and $(v, LCA(u, v))$. As the path from u to v in T is the concatenation of the paths from u to $LCA(u, v)$ and from $LCA(u, v)$ to v , it is clear that $w_T(u, v) = \max\{w_T(u, LCA(u, v)), w_T(v, LCA(u, v))\}$.

To carry out this simple reduction, we need to find $LCA(u, v)$ for every query $(u, v) \in E$. This can be done in $O(m + n)$ time using the following result:

Theorem 3.1 *There is an $O(n)$ -time algorithm that can preprocess a rooted tree T on n vertices such that subsequently any $LCA(u, v)$ query can be answered in $O(1)$ time.*

Theorem 3.1 was first proved by Harel and Tarjan [HT84]. For a simple linear time LCA algorithm, see Bender *et al.* [BFP⁺05] (or [BF00]). For a survey, see Alstrup *et al.* [AGKR02]

Note that Theorem 3.1 is in fact a bit stronger than what we need here. It would have been enough for us to specify all the LCA queries in advance, together with the tree.

4 Komlós’ algorithm for fully branching trees

Let T be a fully branching tree. We let $r = \text{root}[T]$ be the *root* of T . For every $v \in T$, we let $d[v]$ be the *depth* of v in T , where the depth of the root is 0. If $v \neq r$, then $p[v]$ is the *parent* of v in T , and $w[v] = w(v, p[v])$ is the weight of the edge connecting v with its parent.

Let $E = \{(u_1, v_1), \dots, (u_m, v_m)\}$ be the queries. We assume that u_1, u_2, \dots, u_m are leaves, and that v_i is an ancestor of u_i , for every $1 \leq i \leq m$. Let u be a leaf and let $(u, v_1), (u, v_2), \dots, (u, v_k)$ be the queries that involve u such that $d[v_1] < d[v_2] < \dots < d[v_k]$. We let $Q[u] = \langle d[v_1], d[v_2], \dots, d[v_k] \rangle$ be a *sequence* whose elements are the depths of the ancestors of u to which there is a query from u . (Note that a depth uniquely identifies an ancestor of u .)

If v is a vertex of T and $i \leq d[v]$, we let $LA(v, i)$ be depth i ancestor of v . (Note that if u is an ancestor of v , then $LA(v, d[u]) = u$.)

Our goal is to compute for every leaf u of T with $Q[u] = \langle d_1, d_2, \dots, d_k \rangle$ a sequence $A[u] = \langle a_1, a_2, \dots, a_k \rangle$ such that for every $1 \leq i \leq k$, $LA(u, a_i)$ is the lower endpoint of the heaviest edge on the path from u to $LA(u, d_i)$. This is done by algorithm `TreePathMaxima` given in Figure 1.

Algorithm `TreePathMaxima` uses the following operations on sequences:

`unite(A, B)` – Unite the sequences A and B . Return the elements of the united sequence in sorted order. Remove duplicates. (For example `unite`($\langle 0, 3, 4, 5 \rangle, \langle 1, 3, 8 \rangle$) returns the sequence $\langle 0.1, 3, 4, 5, 8 \rangle$.)

`remove(A, a)` – Return the sequence obtained from A by removing the element a . (If A does not contain a , then A is returned.)

`append(A, a)` – Return the sequence obtained by appending the element a to the sequence A . It is assumed that a is larger than all the elements of A .

`extract(A, j)` – Extract the j -th element of A .

`replace-suffix(A, a, j)` – Returns a sequence obtained by replacing all the elements in A at position j and beyond by a . (For example, `replace-suffix`($\langle 2, 4, 5, 5, 7, 8 \rangle, 9, 4$) returns the sequence $\langle 2, 4, 5, 9, 9, 9 \rangle$. If j is greater than the length of A , the sequence returned is A .)

`subseq(A, B1, B2)` – Find the positions of the elements of B_1 that are also contained in B_2 and return the elements of A in these positions. (For example, `subseq`($\langle 2, 4, 5, 5, 7, 8 \rangle, \langle 0, 2, 4, 5, 7, 9 \rangle, \langle 0, 4, 7 \rangle$)

Function TreePathMaxima(T, Q)

```

 $r \leftarrow \text{root}[T]$ 
for  $v$  not a leaf do
   $Q[v] \leftarrow \langle \rangle$ 
for  $v \neq r$  (from bottom to top) do
   $Q[p[v]] \leftarrow \text{unite}(Q[p[v]], \text{remove}(Q[v], d[v] - 1))$ 
 $A[r] \leftarrow \langle \rangle$ 
foreach  $v \neq r$  (from top to bottom) do
   $A[v] \leftarrow \text{subseq}(A[p[v]], Q[p[v]], Q[v])$ 
   $j \leftarrow \text{binary-search}(v)$ 
  replace-suffix( $A[v], d[v], j$ )
  if  $d[v] - 1 \in Q[v]$  then
     $A[v] \leftarrow \text{append}(A[v], d[v])$ 
return  $A$ 

```

Figure 1: An algorithm for answering tree path maxima queries.

returns $\langle 2, 5, 7 \rangle$.) It is assumed that A and B_1 are of the same length.

The algorithm is composed of two phases. The first phase proceeds in a bottom to top version. It assigns sequences $Q[v]$ to all non-leaf vertices. If v is a non-leaf, the $Q[v]$ contains the depth of the strict ancestors of v to which there is a query from a descendant of v . No comparisons are performed in this first phase. Using a slight abuse of notation, treating the $Q[v]$'s as sets rather than sequences, it is easy to see that

$$Q[u] = \left(\bigcup_{v:p[v]=u} Q[v] \right) - \{d[u]\}.$$

This is essentially the relation used by the algorithm.

The interesting part of the algorithm is its second phase. This part proceeds in a top down fashion. We initialize $A[r] = \langle \rangle$. Suppose now that $p[v] = u$ and that $A[u]$ was already computed. Note that $Q[v] \subseteq Q[u] \cup \{d[v] - 1\}$. For every $d \in Q[v]$, the heaviest edge on the path from v to $u' = LA(v, d)$ is either the heaviest edge on the path from u to u' , or the edge (v, u) . The identity of the heaviest edge from u to u' can be extracted from $A[u]$. We thus start by extracting from $A[u]$ the answer to the queries in $Q[u]$ that are also present in $Q[v]$. This is done by the command $A[v] \leftarrow \text{subseq}(A[p[v]], Q[p[v]], Q[v])$.) Let $A'[v]$ be the sequence returned by this operation. We next have to determined which of the answers currently in $A'[v]$ should be changed to $d[v]$, indicating the edge (v, u) . A naive approach would be to compare $w[v]$, the weight of (v, u) , with the weights of the 'winners' in $A'[v]$. This may require, however, $|A'[v]| \geq |A[v]| - 1$ comparisons, which is too expensive. Luckily, if $A'[v] = \langle d_1, d_2, \dots, d_k \rangle$, and $u_1 = LA[v, d_1], u_2 = LA[v, d_2], \dots, u_k = LA[v, d_k]$ then we have

$$w[u_1] \geq w[u_2] \geq \dots \geq w[u_k].$$

```

Function binary-search( $v$ )


---


if  $w[v] > \text{weight}(v, 1)$  then
   $\sqsubset$  return 1
if  $w[v] \leq \text{weight}(v, |A[v]|)$  then
   $\sqsubset$  return  $|A[v]| + 1$ 

 $p \leftarrow 1$  ;  $q \leftarrow |A[v]|$ 
while  $p < q$  do
   $r \leftarrow \lfloor (p + q) / 2 \rfloor$ 
  if  $w[v] \leq \text{weight}(v, r)$  then
     $\mid$   $p \leftarrow r$ 
  else
     $\sqsubset$   $q \leftarrow r$ 
return  $q$ 

```

Figure 2: The binary search algorithm used by `TreePathMaxima`.

```

Function weight( $v, r$ )


---


 $a \leftarrow \text{extract}(A[v], r)$ 
 $u \leftarrow LA(v, a)$ 
return  $w[u]$ 

```

Figure 3: Extracting the weight of an edge.

This follows from the fact that the path from v to u_1 contains the path from v to u_2 , etc. Instead of comparing $w[v]$ with each of $w[u_1], w[u_2], \dots, w[u_k]$ we can use a binary search that requires at most $\lceil \log_2(|A[v]| + 1) \rceil$ comparisons. This binary search is performed by the call `binary-search(v)`. The function `binary-search` uses `weight(v, r)` to extract $w[u_r] = w[LA[v, d_r]]$. The binary search returns an index j such that $w[u_{j-1}] \leq w[u] < w[u_j]$. (We assume here that $w[u_0] = +\infty$, and $w[u_{k+1}] = -\infty$.) All elements in $A'[v]$ at and beyond position j should therefore be replaced by $d[v]$. Finally, if $d[v] - 1 \in Q[v]$, then $d[v]$ is appended to $A[v]$.

5 Analysis of Komlós' algorithm

6 Hagerup's implementation

In the preceding two sections we were mainly interested in bounding the number of comparisons performed by the algorithm. We showed that Komlós' algorithm performs only $O(m + n)$ comparisons, but can we implement it in $O(m + n)$ time?

The main feature of Komlós' algorithm is the computation of $A[v]$, given $A[p[v]]$, using only

$\lceil \log_2(|A[v]| + 1) \rceil$ comparisons. If we implement Komlós' algorithm naively, the construction of $A[v]$ would require however $|A[v]|$ time. This will result with a non-linear total running time.

As the trees we are dealing with are fully branching, their depth is at most $\log_2 n$. Thus, each sequence $Q[v]$ represents a subset of $\{0, 1, \dots, \lceil \log_2 n \rceil\}$. This naturally suggests¹ representing each sequence $Q[v]$ is a single $\lceil \log_2 n \rceil$ -bit word. More specifically, if $Q[v] = \langle d_1, d_2, \dots, d_k \rangle$, then we represent it using a $\lceil \log_2 n \rceil$ -bit word $q[v]$ that has 1s exactly in positions d_1, d_2, \dots, d_k , and 0s in all other positions. (For concreteness, let us say that position 0 is the least significant position of a word.)

We similarly represent each sequence $A[v]$ as a $(\log_2 n)$ -bit word. If $A[v] = \langle a_1, a_2, \dots, a_k \rangle$, then $a_1 \leq a_2 \leq \dots \leq a_k$, but the a_i 's are not necessarily distinct. (This problem did not arise with the $Q[v]$'s.) How do we deal with multiplicities? We simply ignore them! We let $a[v]$ be a $(\log_2 n)$ -bit word with 1s exactly in positions a_1, a_2, \dots, a_k . If some of the elements in $A[v]$ are equal, then $a[v]$ will contain less than k 1s.

Representing $Q[v]$ and $A[v]$ as the bit vectors $q[v]$ and $a[v]$, is equivalent to viewing them as sets. We adopt this notation and write $a[v] = \{a_1, a_2, \dots, a_k\}$ to indicate that $a[v]$ has 1s exactly in positions a_1, a_2, \dots, a_k , which are now assumed to be distinct.

Given $q[v]$ and $a[v]$ we can recover $A[v]$ as follows. Suppose that $q[v] = \{d_1, d_2, \dots, d_k\}$, where $d_1 < d_2 < \dots < d_k$, and that $a[v] = \{a_1, a_2, \dots, a_{k'}\}$, where $a_1 < a_2 < \dots < a_{k'}$. (We may have $k' < k$.) Then, $A[v] = \langle a'_1, a'_2, \dots, a'_k \rangle$, where a'_i is the smallest a_j greater than d_i . This motivates the definition of the following *up* operation. If A is a set and $d < \max(A)$, we let

$$d \uparrow A = \min\{a \in A \mid a > d\}.$$

Note that with this notation $a'_i = d_i \uparrow a[v]$.

We next extend the *up* operation to sets.² If Q is a set and $\max(Q) < \max(A)$, we let

$$Q \uparrow A = \{d \uparrow A \mid d \in Q\}.$$

A moments' reflection shows that the `subseq`($A[p[v]]$, $Q[p[v]]$, $Q[v]$) operation used by `TreePathMaxima` can be simply replaced by $q[v] \uparrow a[p[v]]$.

With this representation, all operations on the sequences $Q[v]$ and $A[v]$ are replaced by $O(1)$ -time operations on the $(\log_2 n)$ -bit words $q[v]$ and $a[v]$.

To implement some of the operations required on two $(\log_2 n)$ -bit words we use *table lookup*. For any dyadic operation $op(a, b)$ on two $(\log_2 n)$ -bit words, we can precompute a table $OP[a, b]$. We can then find $op(a, b)$ in $O(1)$ time simply by looking it up in the table. Unfortunately, the size of this table is $2^{2 \log_2 n} = n^2$ which is too much. We can, however, apply this idea on operations on two $(\frac{1}{3} \log_2 n)$ -bits words. The space required now is only $n^{2/3}$, and the time required to prepare the table would be $o(n)$, provided that $op(a, b)$, for any given two words a and b can be computed in

¹It was not so obvious before Hagerup [Hag09] suggested it.

²*up* is an analog of Hagerup's *down* [Hag09] which fits nicer with our presentation.

$o(n^{1/3})$ time. All the operations $op(a, b)$ that we need on two $(\log_2 n)$ -bit words are *decomposable*, in the sense that we can quickly compute $op(a, b)$ given the results of $op_1(a_1, b_1)$, $op_2(a_2, b_2)$ and $op_3(a_3, b_3)$, where $a = a_1a_2a_3$, $b = b_1b_2b_3$, and op_1, op_2, op_3 are suitably defined operations. Thus all required operations can indeed be computed in $O(1)$ time.

References

- [AGKR02] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. of 14th SPAA*, pages 258–264, 2002.
- [BF00] M.A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. of 4th LATIN*, pages 88–94, 2000.
- [BFP⁺05] M.A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.
- [DRT92] B. Dixon, M. Rauch, and R.E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.
- [Hag09] T. Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *Proc. of 35th WG*, 2009.
- [HT84] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
- [Kin97] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.
- [Kom85] J. Komlós. Linear verification of spanning trees. *Combinatorica*, 5(1):57–65, 1985.