These notes cover a tree-packing algorithm due to Gabow [Gab95], which was used to compute all small $s$-$t$ connectivities by Bhalgat, Hariharan, Kavitha and Panigrahi [HKPB07]. These notes closely follow the presentation by Hariharan [Har12].

The problem is to find, in a directed graph with a special vertex $r$, either

1. $c$ edge-disjoint directionless trees such that the total in-degree of any vertex other than $r$ is $c$.

2. A certificate of infeasibility: a cut $(S, V \setminus S)$ such that less than $c$ edges leave $V$

This is a weaker condition than trees rooted at $r$: we no longer need all edges in each tree being directed away from $r$, which is equivalent to all non-root vertices having in-degree exactly 1 (instead of summing to $c$ across the $c$ trees). A bit surprisingly, these turn out to be equivalent (via their dual certificates).

This algorithm immediately gives a way to check whether an undirected graph has edge-connectivity at least $c$: we ‘directify’ the graph by putting in each edge in both directions, and then ask for $c$ spanning trees:

1. For any cut $\overline{S}$ that does not contain $r$, the total in-degree in $\overline{S}$ must be $c|\overline{S}|$. However, each tree can have at most $|\overline{S}| - 1$ edges contained in $\overline{S}$ (or we’ll have a cycle inside $\overline{S}$). So at least $c$ of those in-edges must be coming from outside.

2. If a cut is returned, because the edges are ‘symmetric’, each edge leaving corresponds to an original edge, so we get a cut.

A bit surprisingly, this algorithm for the relaxed version of the problem can be implemented in $O(cm \log n)$ time.

1 Overall Schema

The plan is to run the above scheme where we grow $T_k$ in parallel over multiple components, with one root per component. Then we ensure that the component sizes are doubling (in a way analogous to Brouvka’s algorithm), which implies a total round count of $O(\log n)$.

The main difference in this algorithm is that it grows the trees backward. This can be done in several rounds, and the top-level invariants are:

1. Indegree of $r$ is 0 at any time.
2. In degree of all non-root vertices in the component of \( r \) (in undirected sense) in \( T_k \) have in-degree \( k \) over \( T_1 \ldots T_k \).

3. Each component \( \text{Comp} \) in \( T_k \) not containing \( r \) has at exactly 1 vertex of in-degree \( k - 1 \). Call it the deficient vertex of \( \text{Comp} \). This can be viewed as a the ‘root’ of the tree corresponding to \( \text{Comp} \), but we can’t formalize it as so because we no longer guarantee all edges pointing away.

The goal of each round is to connect each component to some other component. Thus, we can focus on only one component, \( \text{Comp} \). Let \( v \) be its deficient vertex. Note that \( v \) must have an unused edge into it, because its in-degree is at least \( k \). Let this edge be \( e = u \rightarrow v \).

The simple case is if \( u \notin \text{Comp} \). Then we can just add this edge, and maintain the invariant. So it suffices to consider the case where \( u \in \text{Comp} \).

Now consider the path between \( v \) and \( u \) in any \( T_i \). Any edge on this cycle can be exchanged with \( e \) while still retraining the property that the tree is still an undirected spanning tree. Removing such an edge in turn changes the in-degree of some other vertex:

1. If this edge points outside of \( \text{Comp} \), we’re done by adding this edge to \( T_k \) instead.
2. Otherwise, we can remove that edge, and thus change the deficient vertex of \( \text{Comp} \) to a different vertex.

So we can free up \( f \), and it can in turn free up some other edge, and etc, until we perhaps freed up one of the edges incident to the deficient vertex of \( \text{Comp} \) (note that this vertex can also change if we swapped \( u \rightarrow v \) into \( T_k \) instead). This leads to a process reminiscent of either finding augmenting paths in flow networks, or searching for sequences of exchanges in matroid intersection. This step is formally called Closure Computation.

2 Closure Computation

The goal of closure computation is to extend \( \text{Comp} \) by figuring out which edge to add to \( \text{Comp} \). The goal is to build \( F \), the set of edges that can be released. If edge in \( F \) connects to vertex outside of \( \text{Comp} \), we’re done. Otherwise, we maintain the candidate roots \( R \), along with a set \( C \) which is the ENDPOINTS of the edges in \( F \). Note that \( C \) may be a superset of \( R \).

The algorithm initializes \( F \) to all unused edges pointing into the deficient vertex of \( \text{Comp} \), and grows \( C \) and \( F \). Specifically, it grows \( C \) by looking for some \( e = u \rightarrow v \in F \), and some tree \( T_i \), and computes:

1. The entire fundamental cycle in \( T_i \) between \( u \) and \( v \).
2. Add all vertices on this cycle to \( C \).
3. Add all edges on this cycle to \( F \).
4. Add all heads of those edges to \( R \), the potential new ‘deficient vertices’, and add all unused edges incident to them to \( F \) as well.

Note that we are done as soon as we add some edge with \( u \notin \text{COMP} \): such an edge can be added to \( T_k \) without breaking the ‘no cycle’ requirement.

When \( i = k \), this is equivalent to changing the deficient vertex of \( \text{COMP} \) to somewhere else along the fundamental cycle involving \( uv \). The more interesting case is where \( i \neq k \), that is, we make an adjustment on some earlier tree. Note that in this case, if the fundamental cycle involving \( uv \) in some \( T_i \) goes outside of \( \text{COMP} \), we are immediately done: we just swap an edge on that cycle that exists \( \text{COMP} \) over from \( T_i \) to \( T_k \). The total in-degree are unchanged, and that new edge does not create a cycle (in the undirected sense) in \( T_k \) because it’s on the boundary of \( \text{COMP} \).

To bound the running time of this procedure, the key observation is that ANY of the vertices visited must be inside \( \text{COMP} \), or we are immediately done. So we can charge the cost of all these to the total size (measured in sum of degrees) of \( \text{COMP} \): we can maintain using dynamic tree data structures the identifies of the vertices, and query along a fundamental cycle in \( T_i \) whether there is something that’s in a different component. Formally, this tree maintenance can be done using the following black-box:

**Lemma 1.** There is a data structure that takes a dynamically changing forest \( F \), with vertices labeled 0 and 1, and performs the following operations in worst-case \( O(\log n) \) time:

1. Add/remove edge.
2. Relabel some vertex.
3. On the path from \( u \) to \( v \), find some vertex labeled 1, or report that none exist.

We actually need two of these: one to use 1s to represent the things outside of \( \text{COMP} \) (to see whether we went outside of it), and another to use the 1s to represent the vertices in \( \text{COMP} \), but are not in \( C \) yet.

### 3 \( C \) Also Gives Small Cut

It remains to argue that if we are not able to ‘free up’ some edge leaving \( \text{COMP} \), we can also recover a cut of size \( k - 1 \). For this, we will argue very directly that \( C \) represents such a cut.

First, observe that \( C \) is always connected in each \( T_i \): it’s a union of vertices on fundamental cycles. This means there are at least

\[
k \cdot (|C| - 1)
\]

edges, and thus that much contribution to the total in-degree of vertices in \( C \).

Secondly, observe that there can be no unused edges pointing into \( C \): if so, such an edge would be in \( F \), and \( C \) will have to grow (by the other endpoint of that edge). This
plus the fact that the initial deficient vertex of COMP is incident to \( C \) means the total in-degree of \( C \) is at most

\[
k \cdot |C| - 1 - k \cdot (|C| - 1) \leq k - 1,
\]

which means it gives us a small cut.

There is a slight caveat: we only added edges pointing somewhere into \( R \) to \( F \). Here we can actually show that when \( C \) has stabilized, we must have \( R = C \) as well, that is, we can’t have a vertex that has 0 in-degree across all the teres. This is once again by a similar degree-based summation argument: if there is an in-degree 0 vertex, the average in-degree of all other vertices in \( C \) needs to be \( k \). But since the max in-degree is at most \( k \), this means everyone, including the current deficient vertex of COMP, also need to have in-degree \( k \), a contradiction.

4 Applications

Now for the elephant in the room: how is any of this useful?

A big reason is that tree packing also work for vertex subsets: that is, we want to certify that a subset of vertices (often referred to as terminals) are \( c \)-edge connected with each other.

Tree packing algorithms, including the one presented here by Gabow, can be adapted to those settings. This type of methods led to almost-linear time algorithms for \( c \)-limited Steiner connectivity: finding the smallest cut that separates two terminal vertices \cite{CH03}, and was used later in the algorithms for computing small \( s-t \) connectivities \cite{BHKP08}, as well as the \( c \)-limited Gomory-Hu tree \cite{HKPB07}.

This kind of terminal questions, up to recursion, is very closely related to multi-pair max-flow type questions. Open problems here include:

**Open Problem.** Give a sub-quadratic time algorithm for computing an 1.1-approximate minimum Steiner cut.

**Open Problem.** Give a sub-quadratic time algorithm for 1.1-approximations to \( n \) source/sink pairs.

**Open Problem.** Give a nearly-linear time algorithm for finding 5 disjoint directed trees rooted at \( r \), with all edges pointing away from \( r \).

References


