Implementing Gabow's Algorithm on Edge Connectivity

Omar A. Garcia, Pablo Blanco, Aurora Hiveley, and Lucy Martinez

Spring 2025

1 Introduction

Let $G = (V, E)$ be a directed graph with $|E| = m$ edges and $|V| = n$ vertices. The edge connectivity of a graph is the minimum number of edges whose deletion from a graph G disconnects G i.e. the size of the minimum edge cut. Moreover, denote $\lambda(G)$ to be the the size of the smallest edge cut of G. There are several algorithms that compute λ, all of which have different time bounds. For instance, the Ford-Fulkerson method computes the edge connectivity in $O(mn^2)$ [4]. Some other algorithms include Even and Tarjan [3], Schnorr [9], and Mansour and Schieber [8] with $O(m^{3/2}n), O(\lambda mn)$ and $O(\min\{mn, \lambda^2 n^2\})$ times respectively. In this paper, we focus on implementing Gabow's algorithm of a digraph which runs in $O(\lambda m \log n)$ time [5, 6]. Gabow's algorithm computes the edge connectivity of a digraph by computing a minimum s-cut which is constructed via a collection of k edge-disjoint spanning trees. This method was inspired by the matroid intersection problem (finding a largest common independent set in two matroids) and the idea of packing spanning trees.

This paper is structured as follows. We first provide all the necessary background in Section 2. We then give an overview of the algorithm in Section 3. We conclude this paper with all the Maple procedures used to implement Gabow's algorithm in Section 4.

2 Background

We standardize our notation here. $G = (V, E)$ is a directed graph, or *digraph*, on $n = |V|$ vertices with $m = |E|$ (directed) edges. The graph with the same underlying vertices and all edges' directions reversed is denoted G^R. A *spanning tree* of G is an acyclic spanning subgraph of the undirected graph G. Adding any other edge e from G to a spanning tree T introduces a cycle, which is the *fundamental cycle* of e in T, denoted $C(e, T)$. We say the digraph G is *strongly connected* if there is a path between any two distinct vertices. Removing a directed edge from G may remove this property, or *disconnect* G. The *edge connectivity* of a graph G is thus the minimum number of edges required to be deleted to disconnect G, and is denoted $\lambda(G)$. Then $\lambda(G)$ is bounded above by the number of outgoing edges from any vertex s, and also by the number of incoming edges to s. Generalizing, define an *s-cut* as a set of edges that point from S to $V - S$ for any $S \subsetneq V$ such that $s \in S$. Then $\lambda(G)$ is exactly the minimal cardinality of all s-cuts of G and G^R.

Gabow's algorithm computes the edge connectivity efficiently by applying the matroid characterization of s-cuts i.e. the edges of a directed graph can be partitioned into k s-arborescences if and only if they can be partitioned into k spanning trees and every vertex except s has in-degree k. In practice, Gabow's algorithm works with spanning trees, so we need not focus too heavily on s-arborescences. The result above is referred to as the Matroid Characterization of Minimum-Cuts since spanning trees with the above property are the result of intersecting two matroids.

3 Overview of the Algorithm

In this section, we describe Gabow's algorithm and we follow the algorithm description from [5, 6, 7]. We break this section into two subsections: one for the theory and the other one for the implementation in Maple.

3.1 Algorithm Theory

The algorithm breaks down into *iterations*, which in turn break down into *rounds*. For convenience, we use the depth-first search (DFS) algorithm in every iteration, as in [7]. Recall that the DFS algorithm is a recursive algorithm for searching a graph. For our implementation, we use DFS to search for a tree in a directed graph. The algorithm starts at the root vertex of a tree and traverses as far as it can down a given path, then backtracks until it finds an unexplored path, and then explores it, and so on. The algorithm does this until the entire graph has been explored. We implement DFS using stacks.

An iteration takes an input $T = (T_1, \ldots, T_{k-1}, T_k)$ of $k-1$ spanning trees and a spanning forest T_k, all edge-disjoint from each other, and attempts to enlarge T_k by adding edges to turn it into a spanning tree. The trees inside the forest T_k as referred as f-trees. The process of attempting to add a single edge to T_k is a *round*. When it is impossible to construct a T_k, the algorithm returns $T_1, \ldots, T_{k-1}$ indicating edge-connectivity $k-1$.

In order for Gabow's algorithm to properly function, there are a few additional conditions placed on the construction of the trees $T_1, \ldots, T_{k-1}$ and the forest T_k. In particular, we are looking for a *k-intersection*, which is defined as follows:

Definition 3.1. For any fixed vertex s, a *k-intersection* is a collection $T = (T_1, \ldots, T_k)$ where:

1. $T_1, \ldots, T_{k-1}$ are spanning trees, and T_k is a spanning forest
2. T_i and T_j are edge-disjoint for $i \neq j$
3. s has in-degree 0, and any other vertex v has in-degree at most k.

The k-intersection T is *complete* if T_k is a spanning tree. We note that this definition differs from that in [7], where they consider each T_i to be a spanning forest. In practice, our definition is sufficient. At the k-th iteration of the algorithm, T_k is a forest of rooted trees (f-trees). Each of these rooted trees is rooted at a vertex z.

Furthermore, in each round we will attempt to add an edge to T_k by constructing an augmenting path. This construction also has some specific conditions which will guarantee the algorithm's success. The definition of an augmenting path, and its necessary conditions, are outlined below.

Definition 3.2. A *partial augmenting path* P from a vertex z is a sequence of distinct edges $e_1, e_2, \ldots, e_\ell$ such that

1. $e_1 \in E^-(z) \setminus T$.
2. For each $i < \ell$ either
   (a) $e_{i+1} \in C(e_i, T_j)$ where $e_{i+1} \in E(T_j)$ and $e_i \notin E(T_j)$.
   (b) $e_i, e_{i+1} \in E^-(v)$ for some vertex v where $e_i \in E(T)$ and $e_{i+1} \notin E(T)$.
3. Executing all swaps of P (i.e., the pairs e_i, e_{i+1} of (a)) gives a new collection of forests.

An *augmenting path* P from a vertex z is a partial augmenting path from z where the last edge of P, namely e_ℓ, is joining for z. We say that an edge $e = (x, y)$ is *joining* if x and y are in different f-trees of the forest T_k. In the next section, we describe the steps to construct an augmenting path at each iteration.

3.2 Algorithm Steps

Each iteration is a sequence of at most $\lceil \log(n) \rceil$ where n is the number of vertices. We denote $C(e, T)$ to be the *fundamental cycle* of e in T, that is, the cycle that is formed whenever e is added to T. To execute the steps for finding an augmenting path, we begin with a *double-ended queue*. A double-ended queue, or DEQueue for short, is a generalized version of a queue where elements can be added to or removed from both the front and back. We denote these two operations by `pop_front()` and `push_back(g)` where g is an edge. The augmenting path algorithm proceeds as follows. In the first step of execution, let Q be a DEQueue, set $i = 1$ and let l to a null variable. For each i, we will consider tree T_i.

1. Label step: Execute this step for all edges $g \in E^-(z) \setminus T$. Returns an edge g if it is a joining edge. Otherwise, use `push_back(g)` which adds the edge to the end of Q.

2. If no joining edge g is found in $E^-(z) \setminus T$ then there are 3 steps to execute. First, let A be a queue.

   (a) Next edge step: If Q is empty, then the algorithm returns FAIL since the search was unsuccessful and the execution is terminated. Otherwise, use `pop_front()`, i.e. delete the first edge e from Q. If the edge e is some edge such that $e \in T_i$ then $i \coloneqq (i \mod k)+1$.

   (b) Fundamental cycle step: For this step, we will need the notion of a subtree of T_i that consist of the vertex z and every vertex on a labeled edge of T_i. Denote this subtree by L_i. The fundamental cycle step considers the following. If both endpoints of e are in L_i then continue to Step 1. Otherwise, set A to be the path of unlabelled edges of $C(e, T_i)$ which are ordered from L_i to an endpoint of e.

   (c) Label A step: For the edges $a = (a_1, a_2) \in A$ we do the following in order:
   - Set $l = e$, use `pop_front()` for a and execute Step 1 for the edge a.
   - For each $g \in E^-(a_2) \setminus T$, execute Step 1.

We conclude with a summary of Gabow's algorithm.

1. Once and for all, fix a vertex $s \in G$.

2. In every iteration, initialize T_k by using DFS on $G \setminus T$, starting from s.

3. At the start of a round, consider an f-tree in T_k. Denote this tree F_z with root z. Then attempt to build an augmenting path P from z to the root of some other f-tree. This path uses edges of $G \setminus T_k$ and is allowed to traverse through $T_1, T_2 \ldots, T_{k-1}$. The path P is constructed as follows:

   (a) If the algorithm cannot build P, it stops and returns $T_1, T_2 \ldots, T_{k-1}$.

   (b) Otherwise, construct P. If P uses edges that belong to any of $T_1, \ldots, T_{k-1}$, then Gabow's algorithm swaps the edges along the path with the edges used from $T_1, \ldots, T_{k-1}$.

4 Maple Procedures

Accompanying this paper is the Maple package `proj5.txt`. In this section, we describe the main procedures need to implement Gabow's algorithm.

4.1 Objects

The vast majority of our code is written using *objects*. The benefit of this style of programming is that we are more easily able to track changes over time, and we can write procedures which apply only to specific objects. As iterations of the algorithm progress, we can easily and simultaneously update the list of trees $T_1, \ldots, T_k$, the list of roots of our f-trees, indicators for which edges are used in T or in the forest T_k, etc. Similarly, writing a procedure specific to a given object means that that procedure can only be applied to that object, which removes the opportunity for user errors when attempting to use a procedure incorrectly. These procedures are also able to access any of the parameters of the object which are encoded as local variables without them being fed as inputs to the procedure, making our implementation neater. The two key objects, as well as some of the important procedures related to each, are detailed below.

`Gabow(G)`: inputs a digraph G and then uses an initialized object `kInt(G,1)` to return the edge-connectivity of G.

`kInt(G,s)`: inputs a digraph G (as represented above) and a source vertex s. Outputs a `kInt` object, which itself contains `RootedTree` objects.

- `size()` returns the current size of the `kInt` object.
- `used()` Returns a table for the edges used by `kInt`. The table is indexed by directed edges e (if $e = u \to v$, represent as $[u, v]$), the table maps to the index of the `RootedTree` object that used e. If e is not used, the table value for e is 0.
- `extend()` adds a new `RootedTree` object to `kInt`. Uses `generateT1`(?) to initialize the new object. Returns FAIL when tree from `generateT1` does not span the graph.
- `grow()` only use after executing `extend()` first. Uses Gabow's algorithm to augment the k-th `RootedTree` object. Returns FAIL if there is no such augmentation.

`RootedTree(n,k,D,R,s)` inputs a positive integer `n`, a table `D` which specifies the in-degree of all the `n` vertices, a table `R` which stores the roots of the f-trees, and a source vertex `s`, and initializes the `k`-th rooted tree on an `n`-vertex graph. Moreover, `RootedTree` is an object which contains the following procedures: `join(e), par(v), dep(v), paredge(v),` and `isTree()` where

- `join(e)` adds an edge `e` to `F`,
- `par(v)` gets the parent of `v` in the rooted tree,
- `dep(v)` gets the depth of `v` in the rooted tree,
- `paredge(v)` returns the edge between `v` and its parent
- `isTree()` returns true iff the RootedTree is a spanning tree of its graph.

Procedures inside `RootedTree` will update the tables `D` and `R`.

4.2 Standard Procedures

In this section, we detail procedures that do not exist within objects, but which are still integral to the operation of our code. Recall that for a vertex v_i, we let $E^-(v_i)$ denote the set of vertices with an edge pointing into v_i, and $E^+(v_i)$ the set of vertices with an edge coming from v_i. The structure of a digraph G is given as follows $G = (E^-(v_1), E^-(v_2), \ldots, E^-(v_n))$.

`DFS(G,s,type)` inputs a directed graph `G`, a source vertex `s`, and an optional parameter `type` which by default will be set to `dir`, otherwise, it can be set to `undir` for undirected edges. This procedure uses the depth-first search (DFS) algorithm to traverse the graph `G` and returns an ordering on the vertices.

`generateT(n,E,k,s)` inputs a $G \setminus T$ with n vertices and edge set E in iteration k, as well as a root vertex s. This procedure uses DFS to build an f-tree rooted at s. The tree is maximal where no non-root vertex's in-degree exceeds $k-1$ (where k is the iteration), and the root's in-degree is 0. Once the tree is built, the procedure returns *FAIL* if the tree is not a valid f-tree, otherwise it returns `[vT,T]` where vT is the vertex set and T is the edge set of the f-tree. Notice that this procedure will be used to determine which edges are added to T using `join(e)` in `RootedTree` detailed in the previous section.

Disclaimer: the following procedure is implemented inside a `kInt` object. It is used by `grow()`. Because of this, `FindAugPath` is not for the end-user. Furthermore, it will use information that is stored inside `kInt`.

`FindAugPath(z)` Given a vertex z inside of G, finds an augmenting path P from z. It will output a table `L` (that labels the edges in P) and the last edge of the path e; e.g. the last three edges of P in path order are `L[L[e]]`,`L[e]`, `e`. When repeated application of `L` to `e` returns `NULL`, P has no more edges.

Various procedures were modified from class, including: `LC(p)`, which returns true with probability p where p is a rational such that $0 \leq p \leq 1$. `LC(p)` is used in `RG(n,p)`, which creates a random graph on n vertices where any possible edge appear with probability p. We also used `IsCo(G)` which checks if the graph G is connected. These procedures were used for the the generation of test graphs (see below).

`RSCG(n,k)` constructs a "random" directed graph with $\lambda = k$. One way to guarantee that $\lambda = k$ when constructing a graph on n vertices is to start with a complete graph K_n and remove $n-k$ edges pointing out from a fixed vertex. But this method only spits out graphs with lots of edges. So, in our alternative approach, `RSCG` partitions $n = (k+1) + (k+1) + \cdots + (k+1) + K$, where $K \geq k+1$ and there are N terms in this sum, then uses `RG` to make a random graph H on N vertices. G is then initialized, by making N "chunks" of vertices (of size $k+1, k+1, \ldots, k+1, K$) which correspond to vertices of H, and are complete graphs. Then if two vertices in H are connected by an edge, the corresponding chunks in G are given k edges from one to the other. When done precisely, this guarantees an edge-connectivity of k. See the documentation for a more detailed proof. Note also that `RSCG` returns an error if $k+1 \geq n/2$.

5 Conclusion

In practice, Gabow's algorithm is very complex to implement and to understand. As this project progressed, we collectively realized that we would need to spend much more time than we originally anticipated dissecting the literature and understanding the nuances of this algorithm. This meant that coding progressed much slower than we expected, and there are aspects of the algorithm which remain incomplete or buggy in our implementation.

For example, for larger example graphs produces with `RSCG(n,k)`, the procedure `generateT` produces unexpected errors that have yet to be debugged. Even more fundamentally, the implementation of the augmenting path finder is incomplete as it requires a (currently empty) helper procedure which traces a fundamental cycle. Additionally, once the augmenting path P has been constructed, the algorithm must shift edges which overlap with the existing trees $T_1, \ldots, T_{k-1}$, and this step is also nonexistent in the current version of our code.

In conclusion, Gabow's algorithm proved to be a much gnarlier beast than we had initially anticipated. While some aspects of our implementation remain incomplete or buggy, we have still managed to implement the lion's share of the steps outlined in [7] while also developing our own understanding of the algorithm.

References

[1] J. Edmonds. *Edge-disjoint branchings.* Combinatorial Algorithms, 91-–96, 1972.

[2] J. Edmonds. *Submodular functions, matroids, and certain polyhedra.* Combinatorial Structures and their Applications, 69-–81, 1970.

[3] S. Even and R. E. Tarjan. *Network flow and testing graph connectivity.* SIAM Journal on Computing **4** (1975), no. 4, 507–518.

[4] D. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, USA, 2010.

[5] H. N. Gabow. *A matroid approach to finding edge connectivity and packing arborescences.* Journal of Computer and System Sciences **50** (1995), no. 2, 259–273.

[6] H. N. Gabow. *A matroid approach to finding edge connectivity and packing arborescences.* Technical Report 539-91, Department of Computer Sciences, University of Colorado, Boulder, 1991. `https://spl.cde.state.co.us/artemis/ucbserials/ucb51110internet/1991/ucb51110539internet.pdf`.

[7] L. Georgiadis, D. Kefallinos, L. Laura, and N. Parotsidis. *An experimental study of algorithms for computing the edge connectivity of a directed graph.* In the 2021 Proceedings of the Symposium on Algorithm Engineering and Experiments, 85–97. URL: `https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.7`.

[8] Y. Mansour and B. Schieber. *Finding the edge connectivity of directed graphs.* Journal of Algorithms **10** (1989), no. 1, 76–85.

[9] C. P. Schnorr. *Bottlenecks and edge connectivity in unsymmetrical networks.* SIAM Journal on Computing **8** (1979), no. 2, 265–274.