Reconstructing Edge-Disjoint Paths Faster

Chao Xu\textsuperscript{a,*}

\textsuperscript{a}Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801

Abstract

For a simple undirected graph with $n$ vertices and $m$ edges, we consider a data structure that given a query of a pair of vertices $u, v$ and an integer $k \geq 1$, it returns $k$ edge-disjoint $uv$-paths. The data structure takes $O(n^{3.375})$ time to build, using $O(\sqrt{mn}^{1.5} \log n)$ space, and each query takes $O(\sqrt{n})$ time, which is optimal and beats the previous query time of $O(kn\alpha(n))$.

Keywords: ancestor tree, connectivity, edge-disjoint paths

1. Introduction

For a simple undirected graph $G$ with $n$ vertices and $m$ edges, we are interested in building a data structure to return $k$ edge-disjoint paths between two vertices. Conforti, Hassin and Ravi [3] demonstrated a data structure that takes $O(n \text{MF}(n, m))$ preprocessing time, uses $O(nm)$ space and queries in $O(kn\alpha(n))$ time, where $\alpha$ is the inverse Ackermann function and MF($n, m$) is the running time for computing a maximum flow in an undirected unit capacity graph with $n$ vertices and $m$ edges.

Our data structure is simple and reaches the optimal query time of $O(\sqrt{n})$ while improving the space usage to $O(\sqrt{mn}^{1.5} \log n)$. The query time is optimal as there exist graphs where every $k$ edge-disjoint $st$-paths uses $\Omega(\sqrt{n})$ edges [5].

2. Preliminaries

Throughout the paper, we fix a simple undirected graph $G = (V, E)$ with $n$ vertices and $m$ edges. Denote $\lambda(s, t)$ to be the local edge-connectivity between $s$ and $t$ in $G$, i.e. the maximum number of edge-disjoint paths between $s$ and $t$. The degree of a vertex is $\text{deg} v$. $\lambda(s, t)$ is bounded above by both $\text{deg} s$ and $\text{deg} t$.

For a rooted tree $T$ with root $r$, the lowest common ancestor of two nodes $u$ and $v$, denoted $\alpha_{uv}$, is the node farthest away from the root that is contained in both the $ru$-path and the $rv$-path. $T_w$, denotes the subtree of $T$ rooted at $\alpha_{uv}$. For any internal node $v$, we abuse the notation and say $u$ is a leaf of $v$ if $u$ is a leaf of the subtree rooted at $v$. A binary tree is full if each internal node has two children.

A rooted full binary tree $T$ with weights on the internal nodes is an ancestor tree of $U \subset V$ if the set of leaves coincides with $U$ and $\lambda(u, v)$ equals the weight of $\alpha_{uv}$ for all $u, v \in U$. An immediate consequence of the definition is $\lambda(u, v) \leq \lambda(x, y)$ for all leaves $x, y$ of $T_{uv}$. An ancestor tree can be found in $O(|U| \text{MF}(n, m))$ time [2].

3. Previous data structure

We give a quick sketch of the data structure of Conforti et al. The heart of their data structure exploits that edge-disjoint paths are effectively “composable”.

Theorem 1 (Theorem 3.1 [3]). Given $k$ edge-disjoint $uv$-paths and $k$ edge-disjoint $vw$-paths with a total of $m$ edges, a set of $k$ edge-disjoint $uv$-paths can be found in $O(m)$ time.

Remark 1. For anyone familiar with the original proof would notice it actually obtain the bound $O(m + k^2)$, where $k^2$ comes from the dummy edges that force a perfect stable matching between the paths. Fortunately, avoiding dummy edges is easy: find any stable matching and match the unmatched paths arbitrarily.

Every $k$ edge-disjoint paths contain $O(kn)$ edges, hence composing $k$ edge-disjoint paths takes $O(kn)$ time. One can construct an auxiliary graph $H$, such that for each edge $uv$ in $H$, we precompute the maximum number of edge-disjoint $uv$-paths in $G$ using any maximum flow algorithm. A query of $k$ edge-disjoint $v_i v_{i+1}$-paths can be answered by a sequence of composition of $k$ edge-disjoint $v_i v_{i+2}$-paths, $v_{i+2} v_{i+3}$-paths, $\ldots v_{i+1} v_{i+l}$-paths, where $v_i, \ldots, v_l$ is a path in $H$ and $\lambda(v_i, v_{i+1}) \geq k$ for all $i \leq l - 1$. The total query time is therefore $O(kn\lambda)$. By augment a flow equivalent tree with Chazelle’s semigroup product structure for free trees [1], it returns a graph $H$ with $O(n)$ edges and at most $O(\alpha(n))$ composition per query. The preprocessing time is $O(|H| \text{MF}(n, m)) = O(n \text{MF}(n, m))$ using $O(nm)$ space, and the query time is $O(kn\alpha(n))$.

*Corresponding author

Preprint submitted to Elsevier November 18, 2015
4. Data structure

On the high level, our data structure is the same as the previous one: we precompute some edge-disjoint paths, and compose them during query time. The difference is the edge-disjoint paths are short, at most one composition per query and the implementation is a simple binary tree.

4.1. Composition of short edge-disjoint paths

It’s easy to find examples where $k$ edge-disjoint paths contain $\Omega(kn)$ edges, even returning the edge-disjoint path itself already exceed our bound. Fortunately, there are always short edge-disjoint paths. A set of $k$ edge-disjoint paths is short if it contains at most $2\sqrt{kn}$ edges.

**Theorem 2.** There exist short $\lambda(s,t)$ edge-disjoint st-paths $P_{st}$, and they can be found in $O(\text{MF}(n,m))$ time. Moreover, the $k$ shortest paths in $P_{st}$ have a total of $O(\sqrt{kn})$ edges for all $k \leq \lambda(s,t)$.

**Proof.** Find any maximum 0-1 st-flow from $s$ to $t$. There is a $O(m)$ time procedure to decycle the flow and then decompose the flow to unit flows along st-paths. Let $P_{st}$ be the paths in the flow decomposition, then $P_{st}$ fits the requirement. Indeed, any acyclic maximum st-flow in a unit capacity simple graph saturates at most $2\sqrt{kn}$ edges [5].

The $k$ shortest paths in $P_{st}$ have total length at most

$$k \frac{2\sqrt{\lambda(s,t)n}}{\lambda(s,t)} = k \frac{2n}{\sqrt{\lambda(s,t)}} \leq 2k \frac{n}{\sqrt{k}} = 2\sqrt{kn}.$$

Short edge-disjoint paths are closed under our implementation of composition. Let $f_{uv}$ denote some $\lambda(u,v)$ short edge-disjoint uw-paths. Let $\ell = \min(k, \lambda(u,w), \lambda(w,v))$. The previous two theorems imply $\text{Compose}(f_{uw}, f_{uv}, k)$ in Figure 1 returns $\ell$ short edge-disjoint uw-paths. The algorithm runs in $O(\sqrt{kn})$ time.

**Figure 1:** Compose $f_{uw}$ and $f_{uv}$.

```
COMPOSE($f_{uw}$, $f_{uv}$, $k$):
$\ell \leftarrow \min(k, |f_{uw}|, |f_{uv}|)$
$p_{uw} \leftarrow \ell$ shortest edge-disjoint paths in $f_{uw}$
$p_{uv} \leftarrow \ell$ shortest edge-disjoint paths in $f_{uv}$
$F' \leftarrow \text{compose } p_{uw}$ and $p_{uv}$
$f \leftarrow \text{push a unit of flow on all paths of } F'$
Decycle $f$
return a path decomposition of $f$
```

4.2. Cache paths and queries

The algorithm first finds $T$, an ancestor tree of $V$, in $O(n \text{MF}(n,m))$ time [2]. If $k \leq \lambda(u,v)$, then there exist $k$ edge-disjoint uw and we-paths, where $w$ is any leaf of $T_{uw}$.

For each internal node $r$ of an ancestor tree, we can assign one single leaf $w$ of $r$ called a hub of $r$, such that for any other leaves $u$ and $v$, either we have already precomputed edge-disjoint paths for $uw$, or we can compose edge-disjoint path of $uw$ and $vw$. It turns out we can assign hubs in a way so we only need to precompute $O(n \log n)$ pairs of edge-disjoint paths.

Let $c(u)$, the heavier child, be the child of $u$ in $T$ with larger number of leaves. The heavier child is the root of the larger subtree. If both children have same number of leaves, then $c$ break ties arbitrarily.

Let the hub of $u$ be $h(u)$, and defined recursively:

$$h(u) = \begin{cases} u & \text{if } u \text{ is a leaf} \\ h(c(u)) & \text{otherwise}. \end{cases}$$

$h(u)$ is always a leaf of $u$. For every internal node $v$ and each leaf $u$ of $v$, the data structure saves maximum edge-disjoint $h(u)u$-paths.

We design a recursive function $\text{CacheFlows}$ to satisfy the above requirement. It maintains the invariant that if $v$ is the input, then it saves flow $f_{h(v)u}$ for each $u$ a leaf of $v$. For an internal node $v$ with children $v_1$ and $v_2$, $\text{CacheFlows}(v)$ begins by running both $\text{CacheFlows}(v_1)$ and $\text{CacheFlows}(v_2)$. Assume $v_2$ is the heavier child, then $h(v_2) = h(v)$, and $f_{h(v)u}$ is cached for all $u$ a leaf of $v_2$. It remains to compute $f_{h(v_1)u}$ for all $u$ a leaf of $v_1$. This can be done by composing $f_{h(v_1)u}$ with $f_{h(v_2)h(v)}$. All $f_{h(v_1)u}$ has been computed due to the last call to $\text{CacheFlows}(v_1)$. Finding $f_{h(v_1)h(v)}$ takes a single maximum flow computation. See Figure 2.

```
\text{CacheFlows}(v):
if $v$ is an internal node
$v_1, v_2$ are children of $v$, where $v_2$ is the heavier child
$\text{CacheFlows}(v_1)$
$\text{CacheFlows}(v_2)$
$f_{h(v_1)h(v)} \leftarrow \text{MAXIMUMFLOW}(h(v_1), h(v))$
for all leaf $u$ of $v_1$
$f_{h(v_1)u} \leftarrow \text{Compose}(f_{h(v_1)u}, f_{h(v_2)h(v)}, \infty)$
else
\text{do nothing}
```

Figure 2: Cache flows.

Let $F$ be the set of pairs $\{s, t\}$ such that we have cached an st-flow at the end of $\text{CacheFlow}(r)$, where $r$ is the root of the ancestor tree $T$. The size of $F$ is an upper bound on the number of times the algorithm applied $\text{Compose}$. Let $\ell(v)$ be the number of leaves of the subtree rooted at $v$. Applying a standard heavy-path decomposition argument [7], $|F|$ is bounded by

$$\sum_{v \text{ an internal node of } T} \ell(v) - \ell(c(v)) = O(n \log n).$$

In each recursive call of the algorithm, the dominating factor of the running time is the maximum flows and compositions. There are $n - 1$ maximum flow computations each taking $O(MF(n,m))$ time, and $O(|F|) = O(n \log n)$
compositions each taking $O(m)$ time. The time spent on CacheFlows is $O(n\cdot MF(n, m) + mn \log n)$.

Because we cache $O(n \log n)$ flows and each flow uses at most $O(m)$ edges, the number of edges stored is bounded by $O(mn \log n)$. A more careful analysis can produce a stronger bound. For fixed $u$ and $v$, the number of edges in the flow is $O(\sqrt{\lambda(u,v)n}) = O(\sqrt{\min\{\deg u, \deg v\}n})$. The total number of edges is

$$\sum_{\{u,v\} \in F} O(\sqrt{\min\{\deg u, \deg v\}n})$$

For every cached flow $f_{st}$, $s$ is called a non-hub for $f_{st}$ if $s$ is not the hub of $\alpha_{st}$. The main observation is that every leaf can partake as a non-hub for $O(\log n)$ cached flows. Indeed, the number of times $s$ occurs as a non-hub equals to the number of non-heavy child in the root to $s$ path, which is $O(\log n)$ [7]. We can charge the space to the vertex that acts as the non-hub. The total space used is therefore

$$\sum_{\{u,v\} \in F} O(\sqrt{\min\{\deg u, \deg v\}}) \leq O(\log n) \sum_{v \in V} \sqrt{\deg v}$$

Using the fact that $\sqrt{\cdot}$ is a concave function,

$$\sum_{v \in V} \sqrt{\deg v} \leq \sum_{v \in V} \sqrt{2m/n} = O(\sqrt{mn}).$$

Putting the above together shows the space usage is $O(\sqrt{mn}^{1.5} \log n)$.

When querying vertices $u$ and $v$ for $k$ edge-disjoint paths, the algorithm finds the hub $w = h(\alpha_{uv})$, and return the composition of $k$ shortest edge-disjoint paths of $f_{uw}$ and $f_{uw}$. The query run time is dominated by the composing procedure. Composing the paths take time proportional to the total number of edges involved, which is $O(\sqrt{kn})$.

**Theorem 3.** There is a data structure that preprocesses an undirected simple graph $G$ of $n$ vertices and $m$ edges in $O(n(MF(n, m) + m \log n))$ time, use $O(\sqrt{mn}^{1.5} \log n)$ space and answer queries for $k$ edge-disjoint st-paths in $O(\sqrt{kn})$ time.

Although there is no known non-trivial lower bound for $MF(n, m)$, every known maximum flow algorithm dominates $m \log n$ by at least a polynomial factor. It’s safe to assume the preprocessing time is $n$ maximum flows. Using the state of art max flow algorithm by Duan [4], the preprocessing time is $O(n^{\beta}\epsilon^{3.375})$.

**Remark 2.** Often one is only interested in edge-disjoint paths between a set of $n'$ terminal vertices $U \subset V$. We can find an ancestor tree for $U$ and apply the rest of the algorithm without modification. The preprocessing time becomes $O(n'(MF(n, m) + m \log n'))$ and the data structure occupies $O(\sqrt{m'n'n' \log n'})$ space, where $m'$ is the sum of degree of vertices in $U$.

If there is an upper bound $k_{max}$ on the query integer $k$, then all occurrences of $m$ can be replaced by $k_{max}$ using sparsification [6].

**Acknowledgement**

We like to thank Chandra Chekuri for bringing the problem to our attention, and Hsien-Chih Chang, Jiahui Jiang, Urvashi Khandelwal and Vivek Madan for reading the draft copy.

**References**


