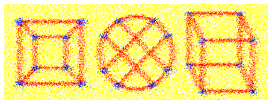
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 23, no. 5, pp. 781–813 (2019)
DOI: 10.7155/jgaa.00512

Faster algorithms for shortest path and network flow based on graph decomposition

Manas Jyoti Kashyop[1] *Tsunehiko Nagayama*[2]
Kunihiko Sadakane[2]

[1]Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India
[2]Department of Mathematical Informatics, Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan

Abstract

We propose faster algorithms for the maximum flow problem and shortest path problems based on graph decomposition. Our algorithms first construct indices (data structures) from a given graph, then use them for solving the problems. Time complexities of our algorithms depend on the size of the maximum triconnected component in the graph, say r. Max flow indexing problem is a basic network flow problem, which consists of two phases. In a preprocessing phase we construct an index and in a query phase we process the query using the index. We can solve all pairs maximum flow problem and minimum cut problem using the indices. Our algorithms run faster than known algorithms if r is small. The maximum flow problem can be solved in $\mathcal{O}(nr)$ time, which is faster than the best known $\mathcal{O}(nm)$ algorithm [29] if $r = o(m)$, where n and m are the numbers of vertices and edges in the given network, respectively. Distance oracle problem is a basic problem in shortest path, consisting of two phases. In preprocessing phase we construct index and in query phase we use the index to find shortest path between two vertices. We use these indices to solve single source shortest path and all pair shortest path problems. If the given graph is undirected and all the weights are non-negative integers, then our algorithm finds shortest path between two vertices in $\mathcal{O}(m)$ time. If the given graph is directed or the weights are non-negative real numbers then our algorithm finds shortest path between two vertices in $\mathcal{O}(m + n \log r)$ time. If the edge weights are real numbers (i.e some of the weights are negative) then our algorithm finds shortest path between two vertices in $\mathcal{O}(m + nr)$ time.

Submitted:	Reviewed:	Revised:	Accepted:
March 2018	February 2019	March 2019	April 2019
	Final:	Published:	
	Septemebr 2019	October 2019	
Article type:		Communicated by:	
Regular Paper		M.S. Rahman, W.-K. Sung, R. Uehara	

Preliminary version of this work appeared in 12^{th} International Conference and Workshops on Algorithms and Computation (WALCOM), pages 80-92, 2018. The work is supported by JSPS KAKENHI 16K12393 and JST CREST JPMJCR1402.

E-mail addresses: manasjk@cse.iitm.ac.in (Manas Jyoti Kashyop) tsunehiko_nagayama@me2.mist.i.u-tokyo.ac.jp (Tsunehiko Nagayama) sada@mist.i.u-tokyo.ac.jp (Kunihiko Sadakane)

1 Introduction

In this paper, we study shortest path problems and network flow problems. We propose faster algorithms for shortest path problems and maximum flow problem and their variants based on graph decomposition.

Network flows is one of the very well studied problems. This problem is of both theoretical and practical importance[1]. A *network* is a directed or undirected graph $G(V, E)$ with non-negative real capacities (c_e) associated with every edge $e \in E$. Let $|V| = n$ and $|E| = m$ throughout this paper. The *terminals* of network G are elements of $Q \subseteq V$. A *flow* f in G is an assignment $(f : E \to R^+)$ of non-negative real values (f_e) to each edge $e \in E$ such that $f_e \leq c_e$ and net flow out of each nonterminal vertex is zero. Net flow out of a vertex is the sum of flows on the edges going out of the vertex minus the sum of flows on the edges coming into the vertex. We consider the following problems in network flow:

Max Flow Indexing Problem(MFIP): Given a network, we preprocess it to construct an index (data structure). This phase is called preprocessing phase. Then given two vertices s and t, we compute the value of the maximum $s - t$ flow using the index. This phase is called query phase. We measure preprocessing time, size of the index, and query time.

In this paper, we propose construction of new data structures for solving the MFIP problem. We also consider the following problems using our algorithm for MFIP:

Maximum $s - t$ flow problem: Given a network and two vertices s and t, compute the maximum $s - t$ flow.

All Pairs Max Flow Problem (APMFP): Given a network, compute the values of the maximum flow between every pair of vertices.

Minimum Cut Problem (MCP): Given a network, compute the value of the minimum cut of the graph.

The MFIP problem consists of two phases: a preprocessing phase for constructing an index from a given graph and a query phase for computing the desired value using the index given two vertices. A graph is *static* if the vertices and edges or properties associated with vertices and edges do not change with time. If the graph is static and we need to compute the maximum flow values for many pairs of vertices, by using an index (data structure) constructed in the preprocessing phase, the queries can be done faster than computing the value without preprocessing. The extreme case is that in the preprocessing phase we compute maximum flow values for every pair of vertices and store them in a two-dimensional array. Then a query is trivially done in constant time. However this approach is not efficient because the index uses $\mathcal{O}(n^2)$ space even if the input size is linear in n and a naive algorithm for constructing the index solves the maximum flow problem $\mathcal{O}(n^2)$ times. Another extreme case is to use

the input graph as the index. Then the preprocessing time is constant but the query time is equal to that for solving a maximum flow problem. Therefore, there is a trade-off between preprocessing time, query time, and index size.

The shortest path problem is very well studied across multiple disciplines in literature [27]. Shortest path problem has been studied under different settings. The graph contains either *directed* or *undirected* edges. The weights over the edges can be *negative* or *non-negative*. The values of the weights can be real or integer numbers.

In this paper, we work with the following setting: Given graph $G(V, E)$, where $|V| = n$ and $|E| = m$, is either directed or undirected and static. The weights on the edges are: 1) non-negative integers, 2) non-negative real numbers and 3) real numbers (i.e some of the weights are negative). We consider the following shortest path problems:

Distance Oracle: Given a graph G, a distance oracle consist of

1. A preprocessing algorithm that construct an index (data structure).
2. A query algorithm that uses the index and computes shortest path between two given vertices.

In this paper, we propose construction of new data structures for distance oracle problem. Our algorithms for distance oracle problem returns the exact distance. Such an oracle is called *exact distance oracle*. Using our algorithms for distance oracle problem, we have also considered the following problems:

SSSP (Single Source Shortest Path): Given a graph $G(V, E)$ and a source $s \in V$, compute the distance $\delta(s, v), \forall v \in V$ where $\delta(s, v)$ is the path between s and v of minimum weight.

APSP (All Pairs Shortest Paths): Given a graph $G(V, E)$, compute the distance $\delta(u, v), \forall u, v \in V$ where $\delta(u, v)$ is the path between u and v of minimum weight.

Thorup and Zwick proposed the term distance oracle [32]. Distance oracle operates in two phases: a preprocessing phase and a query phase. In the preprocessing phase, index is constructed from the given graph. In the query phase, shortest path between two given vertices is computed using the index constructed in preprocessing phase. A distance oracle provides a trade-off between preprocessing time, size of index, and query time.

For a problem consisting of a preprocessing phase and a query phase, an algorithm is called a $\langle p(n), q(n)\rangle$ time algorithm if the preprocessing time is $p(n)$ and the query time is $q(n)$.

1.1 Related work

The maximum flow problem is well studied [13, 11, 17, 26, 29]. Among them, the fastest algorithm runs in $\mathcal{O}(nm)$ time [29]. There are also algorithms for special cases of graphs, for example the $\mathcal{O}(n \log\log n)$ time algorithm for undirected

planar graphs [24], the $\mathcal{O}(n \log n)$ time algorithm for directed planar graphs [6], and the linear time algorithm for constant tree-width graphs [21].

For MFIP and APMFP on undirected graphs, the Gomory-Hu tree [18] can be used as an index. However it is known [5] that there is no such structure for directed graphs. For constant tree-width graphs, APMFP is solved in $\mathcal{O}(n^2 + \gamma^3 \log \gamma)$ time on planar graphs, or $\mathcal{O}(n^2 + \gamma^4 \log \gamma)$ time if $m = \mathcal{O}(n)$ [3], where γ is the number of hammocks obtained by the hammock decomposition [15].

For the minimum cut problem, there are $\mathcal{O}(nm + n^2 \log n)$ time algorithm for undirected graphs [28] and $\mathcal{O}(nm \log(n^2/m))$ time algorithm for general graphs [22]. Tables 1 and 2 summarize complexities of existing algorithms and our algorithms.

Table 1: Complexities of max-flow problem, APMF, and MCP where n, m, γ, r denote the number of nodes, the number of edges, the number of hammocks, and the maximum size of triconnected components, respectively.

Problem	Reference	Graph class	Time complexity
maximum flow	[26, 29]	general	$\mathcal{O}(nm)$
	[24]	undirected planar	$\mathcal{O}(n \log \log n)$
	[6]	directed planar	$\mathcal{O}(n \log n)$
	[21]	constant tree-width	$\mathcal{O}(n)$
	Theorem 7	**general**	$\mathcal{O}(m + nr)$
APMFP	[3]	constant tree-width	$\mathcal{O}(n^2)$
	[3]	planar	$\mathcal{O}(n^2 + \gamma^3 \log \gamma)$
	[3]	$m = \mathcal{O}(n)$	$\mathcal{O}(n^2 + \gamma^4 \log \gamma)$
	Theorem 9	**general**	$\mathcal{O}\left(m + nr^3 + n^2\right)$
MCP	[28]	undirected	$\mathcal{O}(nm + n^2 \log n)$
	[22]	general	$\mathcal{O}(nm \log(n^2/m))$
	Theorem10	**general**	$\mathcal{O}\left(nr^3 + n^2\right)$

Table 2: Complexities of MFIP. If $T_1(k, n) = \lambda(k, n)$, $T_2(k, n) = 1$. If $T_1(k, n) = 1$, $T_2(k, n) = \alpha(n)$. The functions $\lambda(k, n)$ and $\alpha(n)$ are the inverse Ackermann functions defined in Section 2.5

Reference	Graph class	Complexity
[3]	constant tree-width	$\langle \mathcal{O}(nT_1(k, n)), \mathcal{O}(T_2(k, n)) \rangle$
Theorem 8	**general**	$\langle \mathcal{O}(m + nT_1(k, n) + nr^3), \mathcal{O}(T_2(k, n)) \rangle$

The shortest path problem is very well studied. If the weights are non-negative integers and the graph is undirected then SSSP problem is solved in $\mathcal{O}(m)$ time [31]. If the weights are non-negative real numbers then SSSP is solved in $\mathcal{O}(m + n \log n)$ time [16]. In the presence of negative weights SSSP is solved in $\mathcal{O}(mn)$ time [4],[14]. APSP problem is solved in $\mathcal{O}(mn + n^2 \log n)$ time [25] even in the presence of negative weights. Table 3 and Table 4 summarize complexities of existing algorithms and our algorithms.

Table 3: Complexities of s-t shortest path problem, SSSP and APSP where $n, m, r, \alpha(n)$ denote the number of vertices, the number of edges, the maximum size of triconnected components, and inverse Ackermann function respectively.

Problem	Reference	Graph class	Time complexity
s-t shortest path	**Theorem 3**	general	$\mathcal{O}(m + nr)$
	Theorem 3	undirected and non-negative integer weights	$\mathcal{O}(m)$
	Theorem 3	non-negative real weights	$\mathcal{O}(m + n \log r)$
SSSP	[31]	undirected and non-negative integer weights	$\mathcal{O}(m)$
	[16]	non-negative real weights	$\mathcal{O}(m + n \log n)$
	[4],[14]	general	$\mathcal{O}(mn)$
	Theorem 5	**general**	$\mathcal{O}(m + nr^2 + n\alpha(n))$
APSP	[25]	general	$\mathcal{O}(mn + n^2 \log n)$
	Theorem 6	**general**	$\mathcal{O}(m + nr^3 + n^2\alpha(n))$

1.2 Our contribution

We propose faster algorithms for the above problems based on graph decomposition. Namely, we use BC-trees [23] and SPQR-trees [10] for decomposition. A BC-tree represents the biconnected components of a graph and an SPQR-tree represents the triconnected components of a biconnected graph. The performance of our algorithms depends on a parameter of graphs: the size of the maximum triconnected components, denoted by r. If a given graph is decomposed into small triconnected components, our algorithms run faster than existing algorithms.

For MCP, our algorithm is faster than [28] if $r = \mathcal{O}(n^{1/3})$, and faster than [22] if $r = \mathcal{O}(m^{1/3})$. For the maximum flow problem, our algorithm is faster than [29] if $r = o(m)$. For MFIP, the algorithm of Arikati et al. [3] works efficiently for constant tree-width graphs. However the time complexities are doubly exponential to the tree-width, and finding the tree decomposition is NP-hard. On the other hand, the time complexity of our algorithm is polynomial in r.

For computing shortest path between two vertices, if the given graph is undirected and all the weights are non-negative integers, performance of our algorithm is same as linear time algorithm due to Thorup [31]. If the graph is directed or the weights are non-negative real numbers then our algorithm is faster than the algorithm due to Fredman and Tarjan [16] if $r = o(n)$. In the presence of negative weights, our algorithm is faster than [4] and [14] if $r = o(m)$.

For SSSP, in the presence of negative weights our algorithm is faster than [4] and [14] if $r = o(\sqrt{m})$. If the weights are non-negative real numbers than our algorithm is faster than [16] if $r = o(\sqrt{\log n})$.

For APSP, our algorithm is faster than [25] if $r = o(m^{1/3})$.

Table 4: Complexities of Exact Distance Oracle problem, where $n, m, r, \alpha(n)$ denote the number of vertices, the number of edges, the maximum size of triconnected components, and inverse Ackermann function respectively. Theorem 4 explains these results.

Graph class	Time Complexity	Space Complexity
undirected and non-negative integer weights	$\langle \mathcal{O}(m), \mathcal{O}(m) \rangle$	$\mathcal{O}(n)$
non-negative real weights	$\langle \mathcal{O}(m + n \log r), \mathcal{O}(m + n \log r) \rangle$	$\mathcal{O}(n)$
undirected and non-negative integer weights	$\langle \mathcal{O}(m + nr^2), \mathcal{O}(r) \rangle$	$\mathcal{O}(m)$
non-negative real weights	$\langle \mathcal{O}(m + nr^2), \mathcal{O}(r \log r) \rangle$	$\mathcal{O}(m)$
general	$\langle \mathcal{O}(m + nr), \mathcal{O}(m + nr) \rangle$	$\mathcal{O}(n)$
general	$\langle \mathcal{O}(m + nr^2), \mathcal{O}(r^2) \rangle$	$\mathcal{O}(m)$
general	$\langle \mathcal{O}(m + nr^3), \mathcal{O}(\alpha(n)) \rangle$	$\mathcal{O}(mr)$

2 Preliminaries

2.1 BC-trees

Let $G = (V, E)$ be a connected graph. A vertex $v \in V$ is called a *cut vertex* of G if removing v from G makes the graph disconnected. A maximal connected subgraph of G that does not have any cut vertex is called a *block* of G. BC-trees [23] are trees representing the biconnected component decomposition of a connected graph, defined as follows. A tree $T = (B \cup C, F)$ is called a BC-tree of G if it satisfies the following.

- C is the set of cut vertices of G and B is the set of blocks of G.
- Any $c \in C$ and any $b \in B$ are adjacent in T i.e. $(b, c) \in F \iff$ the block corresponding to b contains the cut vertex c.

For a given graph G with $|V| = n$ and $|E| = m$, BC-tree T can be computed in $\mathcal{O}(m+n)$ time [30]. Figure 1 shows an example. Cut vertices are the ones with labeled $2, 6, 7, 8, 9$, and 10. In the BC-tree, blocks are shown by squares.

2.2 SPQR trees

SPQR tree data structure is used to maintain the triconnected components of a graph. Battista et al. introduced SPQR tree data structures in [9] for planar graphs. In [10], Battista et al. extended the SPQR tree data structures for general graphs. Given a graph G, its SPQR tree decomposition T can be computed in linear time [20]. Battista et al. introduced dynamic SPQR trees in [10]. A complete description about SPQR trees can be found in [19].

For a given biconnected graph G, a *separation pair* is a pair of vertices $\{u, v\}$ whose removal disconnects G. A pair of vertices $\{u, v\}$ is called a *split pair* if $\{u, v\}$ is a separation pair or there is an edge between u and v. The split pair

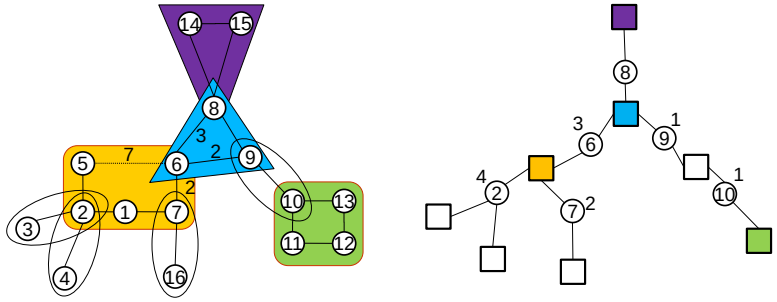

Figure 1: An input graph G (left) and its BC-tree (right). Numbers along edges in G show the weights of the edges. Edge weights are omitted if they are one. In the BC-tree, circle nodes and square nodes show cut vertices and blocks, respectively. Cut vertices have the same labels as those in the input graph. Numbers along cut vertices show the distance to the node to its parent cut vertex.

$\{u, v\}$ is said to be *maximal* with respect to edge (s, t) if for any other split pair $\{u', v'\}$, vertices u, v, s, and t are in the same split class graph.

Let $\{u, v\}$ be a split pair of G. The edges in G can be partitioned into sets E_1, E_2, ..., E_k such that two edges belong to same set E_i if they lie in a path which does not involved any vertex from the set $\{u, v\}$ as an intermediate vertex. Every such E_i, $1 \leq i \leq k$ is called a split class of the split pair $\{u, v\}$ and the graph G_i induced by the split class E_i is called split class graph.

Let $\{u, v\}$ be a split pair of graph G. The *pertinent graph* of a split pair $\{u, v\}$ with respect to an edge $e = (s, t)$ is the union of split class graphs of $\{u, v\}$ except the one containing the edge e. Informally, pertinent graph of split pair $\{u, v\}$ with respect to (s, t) is the portion of the original graph that can be reached from s and t only via u or v.

SPQR tree T for a biconnected graph G is a tree with nodes labeled as S, P, Q and R. Every node μ in SPQR tree is associated with a graph G_μ called *skeleton* of the node. Skeletons are also referred to as triconnected components of G. SPQR tree T satisfy the following properties:

- For every node μ in T, vertices in G_μ, denoted by $V(G_\mu)$, is a subset of V.
- For every edge (μ_1, μ_2) in SPQR tree, $V(G_{\mu_1}) \cap V(G_{\mu_2})$ is a split pair $\{u, v\}$ in G and there is a virtual edge (u, v) in each of G_{μ_1} and G_{μ_2}.
- For every node μ in the SPQR tree, every edge in G_μ is either an edge in E or a virtual edge corresponding to an edge in the SPQR tree.
- If μ is an R-node, then G_μ is a triconnected graph.

- If μ is an S-node, then G_μ is a polygon (a cycle).
- If μ is a P-node, then G_μ is a triconnected multigraph consisting of bundle of multiple edges.
- If μ is a Q-node, then G_μ is a biconnected multigraph consisting of two multiple edges. T has a Q-node associated with every edge of G.

SPQR tree is constructed by decomposing the original graph into triconnected components. The decomposition starts at any arbitrary edge $e = (s, t)$ of the graph which is called as *reference edge* of the decomposition. The Q-node corresponding to this edge is the root of the tree, say μ_r. The trivial case is when the graph has only one edge. In that case μ_r is the only node in the SPQR tree. In all other cases, μ_r has children $\mu_1, \mu_2, \ldots, \mu_k$ ($k \geq 1$). For every children μ_i, $1 \leq i \leq k$, the split pair $V(\mu_r) \cap V(\mu_i)$ is considered as the reference edge. If we regard SPQR trees as unordered trees, they are uniquely determined from the graph.

Figure 2 shows an example of SPQR tree. The tree consists of one R-node (red in color), one P-node (purple in color), three S-nodes (sepia in color) and twelve Q-nodes. The S-nodes are labeled as α, β, and γ. Root of the SPQR tree is the Q-node corresponding to the edge $(7, 5)$. Reference edge for the R-node is $(7, 5)$, for the P-node is $(1, 2)$, for the node α is $(1, 2)$, for the node β is $(4, 5)$, and for the node γ is $(7, 8)$. Battista et al. [10] proved the following :

Lemma 1 *[10] Suppose m be the number of edges and n be the number of vertices in G. The SPQR tree T of G has m Q-nodes, $\mathcal{O}(n)$ S, P, R-nodes. Also the total number of vertices of the skeletons stored at the nodes in T is $\mathcal{O}(n)$.*

Gutwenger et al. [20] presented a linear time implementation of SPQR trees.

Theorem 1 *[20] Given a biconnected graph G with m edges and n vertices, SPQR tree decomposition T can be computed in time $\mathcal{O}(m + n)$.*

2.3 Mimicking Networks

In this section, we review mimicking networks [21].

Let $N = (G = (V, E), c)$ be a network and let $Q = \{q_1, \ldots, q_k\} \subseteq V$. If a function $f : E \to \mathbf{R}_{\geq 0}$ satisfies $f(e) \leq c(e)$ for all $e \in E$, and $i_f(v) = 0$ for all $v \in V \setminus Q$, where $i_f(v) = \sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e)$, $\delta^+(v)$ denotes the set of edges going out of v and $\delta^-(v)$ denotes the set of edges entering v and f is called a Q-flow. For a Q-flow f, $(i_f(q_1), \ldots, i_f(q_k))$ is called the *external flow* with respect to f. If we consider all feasible Q-flows, the set of all external flows define a subset of $\mathbf{R}^Q$. We call it the external flow pattern of N with respect to Q. It is proved [21] that any external flow pattern is expressed by a set of $2^k + 1$ linear inequalities. External flow patterns can be also expressed by mimicking networks. Let $N' = (G' = (V', E'), c')$ be a network satisfying $Q \subseteq V'$. If the external flow pattern of N' with respect to Q coincides with that

of N with respect to Q, N' is called a mimicking network of N with terminal set Q. Hagerup et al. [21] proved the following.

Lemma 2 *[21] For any network and its vertex subset Q, there exists a mimicking network $N' = (G' = (V', E'), c')$ with terminal set $Q \subseteq V'$ such that $|V'| \leq 2^{2^{|Q|}}$.*

Therefore, if the number of terminals is constant, the size of the mimicking network is also constant. Furthermore, for undirected graphs with four terminals, there exists a mimicking network with five nodes [7]. We denote a mimicking network of N with terminal set Q satisfying the condition in Lemma 2 as $M(N, Q)$.

2.4 Range Minimum Queries

Range minimum query: Given an array $A[1, n]$ of length n and a range $[s, t] \subset [1, n]$, compute the position of the minimum value in the sub-array $A[s, t]$.

Given the array A, after linear time processing, a range minimum query can be solved in constant time. The size of the data structure is $2n + o(n)$ bits [12]. Note that the algorithm does not use the input array A at query time; it works just using the $2n + o(n)$ bit data structure.

2.5 Tree Product Queries

We use algorithms for the tree product query problem, defined as follows.

Tree product query: Given a semi-group $(S, \circ)$, a tree $T = (V, E)$, and a function $f : V \to S$, compute $f(u) \circ f(w_1) \circ f(w_2) \circ \cdots \circ f(v)$ for given $u, v \in V$, where $(u, w_1, \ldots, v)$ denotes the $u - v$ path.

If the preprocessing time is $p(n)$ and the time for a query is $q(n)$, we denote the time complexity by $\langle p(n), q(n) \rangle$. The following is known.

Theorem 2 *[2, 8] There exists algorithms for tree product queries with time complexity $\langle \mathrm{O}(kn\lambda(k, n)), \mathrm{O}(k) \rangle$ and $\langle \mathrm{O}(kn), \mathrm{O}(\alpha(n)) \rangle$ for any $k > 0$ where $\lambda(k, n)$ and $\alpha(n)$ are the inverse Ackermann functions. The index sizes are $\mathrm{O}(kn\lambda(k, n))$ and $\mathrm{O}(kn)$, respectively.*

Inverse Ackermann Hierarchy: The inverse Ackermann hierarchy is a class of functions $\alpha_c(n)$, $c = 1, 2, 3, \ldots$ defined as follows:

$\alpha_1(n) = \lceil \frac{n}{2} \rceil$
For $c \geq 2$,
$\alpha_c(1) = 0$
$\alpha_c(n) = 1 + \alpha_c(\alpha_{c-1}(n))$, $n \geq 2$

Inverse Ackermann Function: The inverse Ackermann Function is defined as:

$\alpha(n) = \min\{c : \alpha_c(n) \leq 3\}$
$\lambda(k, n) = \min\{c : \alpha_c(n) \leq 3 + \frac{k}{n}\}$

3 Basic Idea

Here we explain the basic idea of our algorithms. Consider the BC-tree $T = (B \cup C, F)$ of a connected graph $G = (V, E)$, which represents the biconnected component decomposition.

Consider to compute the shortest path from vertices v to w in G. Let t_v and t_w be nodes of T containing v and w, respectively. Note that these nodes are either cut vertices or blocks. First assume that $t_v \neq t_w$ and both t_v and t_w are cut vertices. Consider the unique path P from t_v to t_w in T. We assume that the graph has no cycles with negative weights. Then, the shortest path lies on P and its length is obtained as follows: for each node in P which is a block, it has two cut vertices on P and we amount the shortest path length between them inside the block. If t_v is a block, let c_v be the cut vertex in t_v which is on P. We compute the shortest path length from v to c_v inside the block and add to the shortest path length from c_v to w. If t_w is a block, we do similarly.

To compute the max flow value from v to w in G, we do similarly to the shortest path case, except we take the minimum value instead of summation.

To accelerate this computation, we precompute some values and store them in a data structure. We choose an arbitrary leaf node r of T and make T a rooted tree with root node r. For each block t, let t_0 be the cut vertex of t which is the closest to r among all cut vertices of t, and let $t_1, t_2, \ldots, t_k$ be other cut vertices of t. We call t_0 as the parent of t_i's. For the shortest path case, we compute the shortest path length from each of t_i $(1 \leq i \leq k)$ to t_0 inside t and store it in t_i. We also compute the shortest path length from t_0 to t_i and store it in t_i if the graph is directed.

In Figure 1, the distance from vertex 7 to 6 is 2, and it is stored in node 7 of the BC-tree. The distance from vertex 2 to 6 is 4 and it is stored in node 2. The node 10 of the BC-tree stores weight 1, which is the distance from vertex 10 to 9. In nodes 6 and 9 of the BC-tree, we store the distance from those vertices to 8.

Consider to compute the distance from vertex 2 to 10. The shortest path can be divided into three paths: 2 to 6, 6 to 9, and 9 to 10. The first and the third values are stored in nodes 2 and 10 of the BC-tree. To obtain the second value, we compute the distance from 6 to 9 in the block. In general, the shortest path from v to w is divided into five paths: v to t_0, t_0 to a_v, a_v to a_w, a_w to u_0, and u_0 to w, where t_0 and u_0 are the cut vertices of the blocks containing v and w, respectively, that are on the shortest path, a_v (a_w) is the cut vertex of the lowest common ancestor node a between v and w in the BC-tree that is on the path between t_0 (u_0) and a. We compute the first, the third, and the

fifth values using some data structure for the blocks. To compute the second and the fourth values, we use the tree product query. We set $\circ$ to be addition, and use two tree product queries. To compute the max flow value, we set $\circ$ to be the minimum operator.

In summary, shortest path lengths and max flow values are computed by using at most two tree product queries and computation of shortest path lengths and max flow values inside at most three blocks. Thus the problem can be reduced to the case the graph is biconnected.

4 Shortest Path Algorithms

4.1 Preprocessing

In this section we describe preprocessing algorithms for distance oracle problem using SPQR tree. Let G be a weighted graph with m edges and n vertices. If G is directed, we assume that whenever there is an edge from u to v, there is also an edge from v to u (possibly of weight ∞). First we assume that G is biconnected and describe our preprocessing algorithms. Then we describe the extension to general graphs. Given a biconnected graph G, we obtain the SPQR tree decomposition T of G in $\mathcal{O}(m+n)$ time. We construct data structures for the nodes of T in the preprocessing stage.

4.1.1 Constructing D_0 structure

We define the data structure D_0 as follows.

Definition 1 *Let μ be a node of SPQR tree T and $\{u,v\}$ be the reference edge of μ. The data structure D_0 for node μ stores an edge between u and v whose weight is the length of the shortest path between u and v in the pertinent graph of the pair $\{u,v\}$ with respect to $\{u,v\}$, that is, the skeleton of node μ after removing the reference edge $\{u,v\}$.*

Therefore, D_0 data structure of a node μ with reference edge $\{u,v\}$ stores the shortest path between u and v which is computed using the portion of the graph G corresponding to the skeleton of μ. If the graph is directed, we store two weights of the edge.

When we compute the shortest path from s to t, The D_0 data structure of node μ is used if both s and t are outside of the pertinent graph of the pair $\{u,v\}$ with respect to $\{u,v\}$. Because the pertinent graph is connected to the rest of the graph at only u and v, if the shortest path from s to t goes through the pertinent graph, the length of the shortest path between u and v in the pertinent graph does not depend on the vertices s and t, and therefore the length can be precomputed and stored as the D_0 data structure.

If μ is a Q-node then its skeleton is two parallel edges, one of them is the reference edge and the other is an edge $\{u,v\}$ in the original graph. Then D_0 data structure of node μ stores the edge $\{u,v\}$ and its associated distance is the weight of that edge.

If μ is an S-node then its skeleton is a polygon. The skeleton consists of reference edge $\{u, v\}$ and a path between u and v. Here the reference edge corresponds to the portion of the graph outside the skeleton. After removing the reference edge, remaining skeleton is a path between u and v. Suppose this path consists of edges $\{u, v_0\}$ of weight w_0, $\{v_0, v_1\}$ of weight w_1, ..., and edge $\{v_{d-1}, v\}$ of weight w_d. Then D_0 data structure of node μ stores an edge between u and v with weight $\sum_{i=0}^{d} w_i$. If the graph G is directed, D_0 data structure for node μ stores two edges $u \to v$ and $v \to u$. The weight of the edge $u \to v$ is the shortest path from u to v computed as described above in the skeleton of node μ. Similarly the weight of the edge $v \to u$ is the shortest path from v to u in the skeleton of node μ.

If μ is a P-node then its skeleton consists of two vertices u and v and k ($k \geq 3$) multiple edges between them. One of those k edges is a reference edge. The D_0 structure of node μ is an edge between u and v whose weight is the minimum of the weights of the $k-1$ edges after removing the reference edge in the skeleton of μ. If the graph G is directed then D_0 structure for node μ stores two edges $u \to v$ and $v \to u$ whose weights are computed analogously.

If μ is an R-node, its skeleton is a triconnected graph. Let $\{u, v\}$ be the reference edge of μ. In the skeleton of μ we remove the reference edge $\{u, v\}$ and compute the shortest path between u and v (Let w be the length of the shortest path). The D_0 structure for node μ stores an edge between u and v whose weight is w. If the graph G is directed then after removing the reference edge in the skeleton of μ we compute u to v shortest path (Let w_1 be the length of the shortest path) and v to u shortest path (Let w_2 be the length of the shortest path). The D_0 structure for node μ stores an edge $u \to v$ with weight w_1 and an edge $v \to u$ with weight w_2.

Figure 2 shows an SPQR tree. In the SPQR tree, there are one R-node (red in color), one P-node (purple in color), and three S-nodes (sepia in color). The S-nodes are labeled α, β, and γ. Numbers beside nodes of the SPQR tree show the D_0 data structure. The number associate with S-node α is 12 which is obtained by summation of edge weights 5 on edge $(2, 3)$ and 7 on edge $(3, 1)$. Similarly we obtain 30 for node β and 15 for node γ. The number associated with the P-node is 3 which is minimum between 3 and 12. The number associated with the R-node is 6 which is the value of shortest path between 7 and 5 ($7-4-8-5$) in the skeleton after removing edge $(7, 5)$.

Lemma 3 *Given SPQR tree decomposition T of a biconnected graph G with n nodes and m edges, D_0 data structures are stored in $\mathcal{O}(n)$ space. The construction time for D_0 data structures are as follows:*

1. *If the weights are non-negative integers and G is undirected then total time to compute D_0 data structures is $\mathcal{O}(m)$.*

2. *If G is directed or the weights are non-negative real numbers then total time to compute D_0 data structures is $\mathcal{O}(m + n \log r)$, where r is the maximum size of the triconnected component in T.*

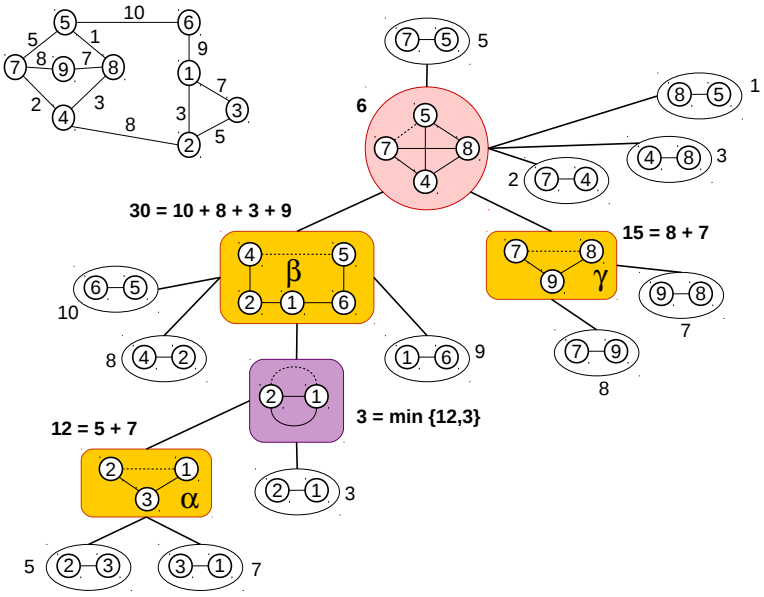

Figure 2: An input graph G (upper left) and its SPQR tree and the D_0 data structure. In the graph, numbers beside edges show edge weights. In the SPQR tree, dotted edges are reference edges.

3. *If the weights are real numbers (i.e some of the weights are negative) then total time to compute D_0 data structure is $\mathcal{O}(m + nr)$, where r is the maximum size of the triconnected component in T.*

Proof: Let μ be a node in the SPQR tree T with reference edge $\{u, v\}$.

If μ is an S-node then time required to construct D_0 for μ is proportional to the length of the path between u and v. If μ is a P-node then time required to construct D_0 for μ is proportional to the number of edges between u and v. By property of SPQR tree, two S-nodes cannot be adjacent in T. Similarly two P-nodes cannot be adjacent in T. An S-node adjacent to a P-node shares exactly one edge. But this edge is a reference edge for one of the nodes and hence considered only once in the computation of D_0 structure. Therefore, total time in the computation of D_0 structure for all the S-nodes and P-nodes is $\mathcal{O}(m)$.

If μ is an R-node, then we divide the analysis into following three cases:

1. G is undirected and all the weights are non-negative integers. Then we use linear time algorithm by Thorup [31] to compute shortest path between u and v. Let r_i be the number of edges and n_i be the number of vertices in μ. Therefore, time required to compute D_0 for one R-node μ is $\mathcal{O}(r_i)$. For all the R-nodes total time required is $\mathcal{O}(\sum_i r_i) \approx \mathcal{O}(m)$. Therefore, total time to compute D_0 data structures for all the nodes is $\mathcal{O}(m) + \mathcal{O}(m) = \mathcal{O}(m)$.

2. G is directed or the weights are non-negative real, then we use algorithm by Fredman and Tarjan [16] to compute shortest path between u and v. Therefore, time required to compute D_0 for one R-node is $\mathcal{O}(r_i + n_i \log n_i) \leq \mathcal{O}(r_i + n_i \log r)$, where r is the maximum size of the triconnected component. Total R-nodes in T is $O(n)$. Therefore, total time to compute D_0 structure for all the R-nodes is $\mathcal{O}(\sum_i (r_i + n_i \log r)) \approx \mathcal{O}(m + n \log r)$. Therefore, total time to compute D_0 structure for all the nodes is $\mathcal{O}(m) + \mathcal{O}(m + n \log r) = \mathcal{O}(m + n \log r)$.

3. G has negative weights. We use algorithm by Bellman [4] and Ford [14] to compute shortest path between u and v. Let r_i be the number of edges and n_i be the number of vertices in μ. Therefore, time required to compute D_0 for one R-node μ is $\mathcal{O}(n_i r_i) \leq \mathcal{O}(n_i r)$. For all the R-nodes total time required is $\mathcal{O}(\sum_i n_i r) = \mathcal{O}(nr)$. Therefore, total time to compute D_0 data structures for all the nodes is $\mathcal{O}(m) + \mathcal{O}(nr) = \mathcal{O}(m + nr)$.

For every S-node, P-node, and R-node of T we are storing an edge as D_0 structure. Since the total number of S-, P-, and R-nodes in T is $\mathcal{O}(n)$, therefore, the total space required by D_0 data structure is $\mathcal{O}(n)$. □

4.1.2 Constructing D_1 structure

We define D_1 data structure as follows.

Definition 2 *Let ν be a node of an SPQR tree T and μ be the parent of ν. Let $\{s, t\}$ and $\{u, v\}$ be the reference edge of ν and μ, respectively. The D_1*

data structure for ν is a network with at most four vertices $\{s, t, u, v\}$ storing the lengths of shortest paths between the vertices in the union of the pertinent graphs of children of μ except that of ν.

When we compute the shortest path from s to t, the D_1 data structure of ν is used if s is in the pertinent graph of ν and t is in the rest of the pertinent graph of μ. In this case, the shortest path from s to t goes through one of s, t and one of u, v (we break tie arbitrarily). The shortest path may go through the pertinent graphs of children of μ other than ν, but its length does not depend on s or t. Therefore we can precompute shortest path lengths between the four vertices $\{s, t, u, v\}$ and store them as the D_1 data structure.

To compute D_1 data structure for each node of the SPQR tree T, we use the following lemma.

Lemma 4 *Let ν be a node of an SPQR tree T and μ be the parent of ν. Let $\{s, t\}$ and $\{u, v\}$ be the reference edge of ν and μ, respectively, and G_μ be the skeleton of μ. We assign the weight w_e of an edge $e = (u_i, v_i)$ of G_μ so that w_e is equal to the weight of the D_0 data structure for the child node of μ corresponding to e. Then the D_1 data structure for ν is obtained by computing shortest paths among s, t, u, v in $G_\mu \backslash \{\{s, t\} \cup \{u, v\}\}$.*

Proof: Let $\bar{G}$ be the union of the pertinent graphs of children of μ except that of ν. Consider to compute the shortest path from s to u in $\bar{G}$. If it passes the pertinent graph G_i of a child of μ, its length inside G_i is equal to that of the reference edge for the child, and it is stored in the D_0 data structure for the edge. Therefore, the shortest path lengths are the same in $\bar{G}$ and $G_\mu \backslash \{\{s, t\} \cup \{u, v\}\}$. □

We compute D_1 data structure which stores in each node of the SPQR tree a constant size graph with at most four vertices (maximum graph is a K_4, i.e., complete graph with four vertices). Let μ be a node in SPQR tree T with reference edge $\{u, v\}$ and let $v_1, v_2, \ldots, v_{k-1}$ be its children.

If μ is an S-node, its skeleton is a polygon. Let $q_1 = u$, $q_2, \ldots, q_k = v$ be the nodes of the skeleton. The weights of the edges $\{q_1, q_2\}$, $\{q_2, q_3\}$, $\ldots$, $\{q_{k-1}, q_k\}$ are stored in the D_0 data structures of the nodes corresponding to the edges. Let w_i be the weight of the edge $\{q_i, q_{i+1}\}$. For the node of the SPQR tree corresponding to the edge $\{q_i, q_{i+1}\}$, we store the following graph with at most four vertices $\{q_i, q_{i+1}, u, v\}$. The graph has at most two edges $\{u, q_i\}$ and $\{q_{i+1}, v\}$. The weight of the edge $\{u, q_i\}$ is $\sum_{j=1}^{i-1} w_j$ and the weight of the edge $\{q_{i+1}, v\}$ is $\sum_{j=i+1}^{k-1} w_j$. The weights of the edges $\{u, q_2\}$, $\ldots$, $\{u, q_{k-1}\}$ and $\{q_1, v\}$, $\{q_2, v\}$, $\ldots$, $\{q_{k-1}, v\}$ can be computed in $\mathcal{O}(k)$ time. Therefore, time required for computing D_1 data structure for all the children of μ is $\mathcal{O}(k)$. In case of directed graph we store at most four edges $\{u, q_i\}$, $\{q_i, u\}$, $\{q_{i+1}, v\}$ and $\{v, q_{i+1}\}$. The weights of all these edges are defined similarly to the undirected case. It is an easy observation that in case of directed graphs also, time required for computing D_1 data structure for all the children of μ is $\mathcal{O}(k)$.

If μ is a P-node, its skeleton has $k (k \geq 3)$ edges between u and v. Let e_1, e_2, $\ldots$, e_k corresponds to those edges and e_k be the reference edge. Let w_i be

the weight of the edge e_i and edge e_i corresponds to child v_i of μ. For each v_i, in D_1 structure we store a graph consisting of a single edge between u and v. The weight of this edge is $\min\{w_1, w_2, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{k-1}\}$. By using range minimum data structure, D_1 data structures for all the children of μ can be computed in $\mathcal{O}(k)$ time. In case of directed graphs, we store two edges (u, v) and (v, u) whose weights are defined analogously. It is an easy observation that in case of directed graphs also, time required for computing D_1 data structure for all the children of μ is $\mathcal{O}(k)$.

If μ is an R-node, its skeleton is a triconnected graph. Let v_i is a child of μ with reference edge $\{s, t\}$. In the skeleton of μ we remove edge $\{u, v\}$ and $\{s, t\}$. In this reduced skeleton graph we compute shortest path between every pair of vertices in the set $\{u, v, s, t\}$ and add an edge between every pair whose weight is the value of the shortest path computed between the two vertices. This results in a K_4 graph and we store it as D_1 structure for node v_i.

If G is directed, D_1 data structures are defined analogously.

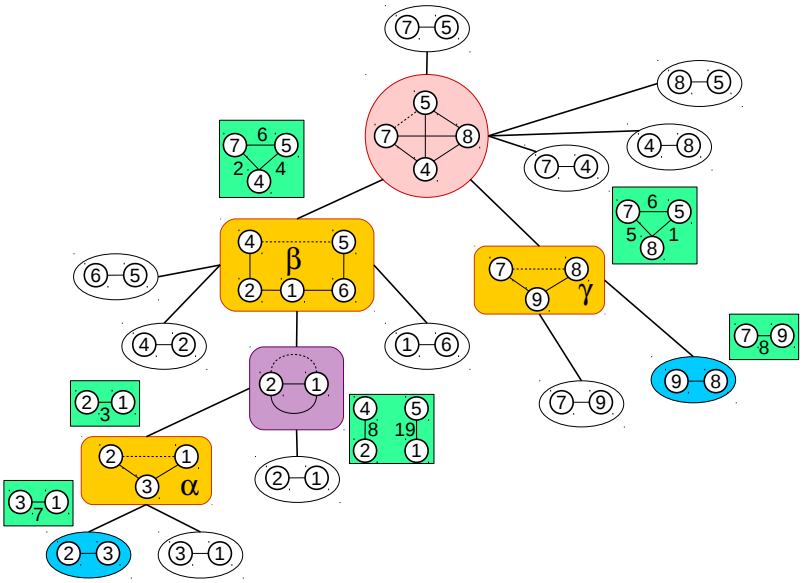

Figure 3: The D_1 data structure for the graph.

Figure 3 shows the D_1 data structure (green in color) for the S-nodes (α, β and γ), P-node (purple in color) and Q-nodes $(2, 3)$ and $(9, 8)$ (blue in color). Each node has a constant size graph with at most four vertices.

Lemma 5 *Given SPQR tree decomposition T of a biconnected graph G with n nodes and m edges, D_1 data structures are stored in $\mathcal{O}(m)$ space. The con-*

struction time for D_1 data structures is $\mathcal{O}(m + nr^2)$. Here G may be directed or undirected and weights may be real or integers and may be negative.

Proof: As shown in Lemma 3, time required to compute D_1 data structure for $k-1$ children of an S-node or P-node is $\mathcal{O}(k)$. Therefore, total time to compute D_1 data structures for children of all the S-nodes and P-nodes is $\mathcal{O}(m)$.

For an R-node, let n_i and r_i be the number of vertices and edges in the skeleton and $\{u, v\}$ be the reference edge. We use Bellman [4] and Ford [14] algorithm to compute shortest path. Therefore, time required to compute D_1 data structure for one child of an R-node is $\mathcal{O}(n_i r_i)$. Therefore, total time required to compute D_1 structure for all the children of the R-node is $\mathcal{O}(n_i {r_i}^2) \leq \mathcal{O}(n_i r^2)$. Since total R-nodes in T is of $\mathcal{O}(n)$, total time required to compute D_1 data structures for the children of all the R-nodes is $\mathcal{O}(nr^2)$. Therefore, total time to compute D_1 data structures for all the nodes is $\mathcal{O}(m) + \mathcal{O}(nr^2) = \mathcal{O}(m + nr^2)$.

We store a constant size graph for all the nodes of T. Total S-nodes,P-nodes and R-nodes in T is $\mathcal{O}(n)$ and there are m Q-nodes. Therefore, total space requirement is $\mathcal{O}(m)$. □

4.1.3 Constructing D_2 structure

The D_2 data structure is constructed for the R-nodes. Let μ be an R-node with reference edge $\{u, v\}$. Let v_1 be a child of μ with reference edge $\{s_1, t_1\}$ and v_2 be a child of μ with reference edge $\{s_2, t_2\}$. The D_2 data structure for the pair v_1 and v_2 is a K_4 (complete graph with four vertices) with vertices $\{s_1, t_1, s_2, t_2\}$. Weight of every edge is the shortest path between the two endpoints of the edge. We store such a structure for every pair of children of μ. We use Bellman [4] and Ford [14] algorithm to compute shortest path. If G is directed then D_2 data structures are defined analogously.

When we compute the shortest path from s to t, the D_2 data structure of μ for the pair v_1 and v_2 is used if s is in the pertinent graph of v_1 and t is in the pertinent graph of v_2.

Lemma 6 *Given SPQR tree decomposition T of a biconnected graph G with n nodes and m edges, D_2 data structures are stored in $\mathcal{O}(mr)$ space and constructed in $\mathcal{O}(nr^3)$ time, where r is the maximum size of the triconnected component. Here G may be directed or undirected and edge weights are real or integers and may be negative.*

Proof: Let μ be an R-node with n_i vertices and r_i edges. As explained in section 4.1.3, time required to compute D_2 data structure for one pair of children of μ using Bellman and Ford algorithm is $\mathcal{O}(n_i r_i) \leq \mathcal{O}(n_i r)$. Therefore, total time to compute D_2 structure for all the pairs of children of the R-node is $\mathcal{O}(n_i r^3)$. Therefore, total time required to compute D_2 structure for all the R-nodes is $\mathcal{O}(nr^3)$.

Since for every child of an R-node we need to store at most $r-1$ constant size graphs, total space required by D_2 data structures is $\mathcal{O}(mr)$. □

For general graph, first we compute BC-tree in $\mathcal{O}(m+n)$ time. Every block in the BC-tree is a biconnected component of the given graph. Two adjacent blocks share exactly one vertex i.e. the cut vertex and all the blocks are edge disjoint. So we compute SPQR tree for every block and then compute data structures D_0, D_1 and D_2 for every block. Overall time and space requirement is same as in the case of biconnected graphs.

4.2 Computing $s-t$ shortest path in $\mathcal{O}(m+nr)$ time and faster

In this section we present an algorithm to compute $s-t$ shortest path using the D_0 data structure. First we will consider G to be biconnected.

Given a graph G, we first compute the SPQR tree. Then for every node of the SPQR tree we compute D_0 data structure. For each of the given nodes s and t, let μ_s be an arbitrary Q-node containing s and let μ_t be an arbitrary Q-node containing t. Let p be the lowest common ancestor of nodes μ_s and μ_t. Let $v_0 = \mu_s$, v_1, v_2, ..., $v_d = p$ be the nodes in the SPQR tree along the path from μ_s to p. For each node v_i in v_1,.....,v_{d-1} we compute a constant size graph with at most four vertices. Let $\{u, v\}$ be the reference edge for node v_{i+1} and $\{x, y\}$ be the reference edge for node v_i. We remove edge $\{u, v\}$ and $\{x, y\}$ from the skeleton of v_{i+1} and compute shortest path between all the pairs in the set $\{u, v, x, y\}$. As explained in section 4.1.2 , this graph is the same as D_1 for node v_i. Therefore, resultant graph is a K_4 where weight of every edge is the shortest path computed between the two vertices. We draw the attention of reader to the fact that we do not compute it for all the children of v_{i+1} and compute only for the child having Q-node for s in its subtree. Let r_i be the number of edges and n_i be the number of vertices in v_{i+1}. Therefore, if G is undirected and all the weights are non-negative integers, then time required to construct the constant size graph for node v_i is $\mathcal{O}(r_i)$ [using algorithm [31]]. If G is directed or the weights are non-negative real numbers then time required to construct the constant size graph for v_i is $\mathcal{O}(r_i + n_i \log n_i)$ [using algorithm [16]]. If G has negative weights then time required to construct the constant size graph for v_i is $\mathcal{O}(n_i r_i)$ [using algorithm [4],[14]]. Similarly for nodes between μ_t and p we compute the constant size graph. We also compute the constant size graph for nodes between p and the root of the SPQR tree.

Finally we merge all these constant size graphs. Two adjacent nodes in SPQR tree has exactly two vertices in common. Let $G(c_1)$ be a constant size graph with vertices u, u', x, y and $G(c_2)$ be an adjacent constant size graph with vertices x, y, v, v'. In the merging process of $G(c_1)$ and $G(c_2)$, we create a new graph $G(c)$ with vertices $\{u, u', v, v'\}$ where weight of every edge is equal to shortest path length between the two endpoints of the edge computed using $G(c_1) \cup G(c_2)$. The graph $G(c_1) \cup G(c_2)$ contains all the vertices and edges in $G(c_1)$ and $G(c_2)$ with the exception that for a common edge between $G(c_1)$ and $G(c_2)$, $G(c_1) \cup G(c_2)$ will contain the edge with minimum weight. Therefore, if $G(c_1)$ and $G(c_2)$ are of constant size, then construction of $G(c)$ takes constant time. Therefore, total time required by the merging operation is proportional

to the length of the paths from μ_s to p, μ_t to p and p to the root of the SPQR tree. Finally we will have a constant size graph with vertices s, s', t and t' where s' and t' are the other endpoints in μ_s and μ_t respectively. Since this final graph is of constant size, we will compute shortest path length between s and t in constant time.

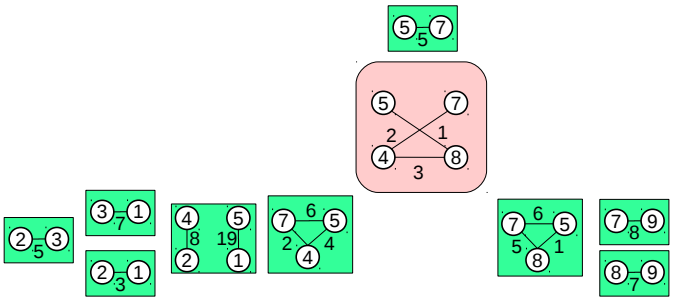

Figure 4: Computing the shortest path between 3 and 9.

Suppose we want to compute shortest path between 3 and 9 in the example graph G. In Figure 3, Q-node $(2,3)$ (blue in color) is chosen as μ_3 and Q-node $(9,8)$ (blue in color) is chosen as μ_9. R-node (red in color) is the lowest common ancestor p of μ_3 and μ_9. All the constant size graphs (green in color) are computed along the path from μ_3 to p and μ_9 to p. Figure 4 shows how to merge the constant size graphs to compute the shortest path between 3 and 9.

For general graph, first we compute BC-tree in $\mathcal{O}(m+n)$ time. We get SPQR tree decomposition and then compute D_0 data structure for every biconnected component. If s and t belongs to the same biconnected component then we are done. Otherwise different biconnected components are separated by cut vertices. So from the biconnected component containing s to the biconnected component containing t we compute shortest path along the cut vertices.

Theorem 3 *For a graph G with n vertices and m edges whose maximum triconnected component size is r, shortest path between s and t is computed in*

1. *$\mathcal{O}(m)$ time if G is undirected and all the weights are non-negative integers.*

2. *$\mathcal{O}(m + n \log r)$ time if G is directed or the weights are non-negative real numbers.*

3. *$\mathcal{O}(m + nr)$ time if G has negative weights.*

Proof: We first consider G to be biconnected. We get the SPQR tree decomposition in $\mathcal{O}(m + n)$ time.

1. G is undirected. We construct the D_0 data structure in $\mathcal{O}(m)$ time [Lemma 3]. As explained in section 4.2, time required to construct the constant size graph for node v_i is $\mathcal{O}(r_i)$ where r_i is the number of edges in the skeleton of the parent node of v_i in the path. Since all these skeletons are edge disjoint, total time required to compute the constant size graphs for all the nodes along the three paths i.e. μ_s to p, μ_t to p and p to the root of SPQR tree is $\sum_i \mathcal{O}(r_i) = \mathcal{O}(m)$. Merging of all these graphs takes time proportional to length of the paths which is again $\mathcal{O}(m)$. Therefore, overall time taken to compute s to t shortest path in G is $\mathcal{O}(m)$.

2. G is directed or weights are non-negative real numbers. We construct D_0 data structure in $\mathcal{O}(m + n \log r)$ time [Lemma 3]. As explained in section 4.2, time required to construct the constant size graph for one node is $\mathcal{O}(r_i + n_i \log r_i)$ where r_i is the number of edges and n_i is the number of vertices in the skeleton of the parent node of v_i in the path. Since all these skeletons are edge disjoint and total number of S-nodes,P-nodes and R-nodes are $\mathcal{O}(n)$, total time required for construction for all the nodes along the paths is $\sum_i \mathcal{O}(r_i + n_i \log n_i) \leq \mathcal{O}(r_i + n_i \log r) \approx \mathcal{O}(m + n \log r)$. Merging of all these graphs takes time proportional to length of the paths which is $\mathcal{O}(m)$. Therefore, overall time taken to compute s to t shortest path in G is $\mathcal{O}(m + n \log r)$.

3. G has negative weights. We construct D_0 data structure in $\mathcal{O}(m + nr)$ time [Lemma 3]. As explained in section 4.2, time required to construct the constant size graph for one node is $\mathcal{O}(n_i r_i)$ where r_i is the number of edges and n_i is the number of vertices in the skeleton of the parent node of v_i in the path. Since all these skeletons are edge disjoint and total number of S-nodes,P-nodes and R-nodes are $\mathcal{O}(n)$, total time required for construction for all the nodes along the paths is $\sum_i \mathcal{O}(n_i r_i) \leq \mathcal{O}(n_i r) \approx \mathcal{O}(nr)$. Merging of all these graphs takes time proportional to length of the paths which is $\mathcal{O}(m)$. Therefore, overall time taken to compute s to t shortest path in G is $\mathcal{O}(m + nr)$.

If G is not biconnected, then we compute the BC-tree decomposition in time $\mathcal{O}(m+n)$. Since two adjacent blocks in the BC-tree have exactly one cut vertex in common and they are edge disjoint, the remaining computation takes same time as in the case of biconnected graph. □

4.3 Algorithms for Distance Oracle Problem

Suppose we want to compute the shortest path between two vertices s and t in a graph G. Let x be a cut vertex in G and $G \setminus \{x\}$ results in two components G^1 and G^2. Suppose s belongs to G^1 and t belongs to G^2. So we will compute shortest path from s to x in G^1 and shortest path from x to t in G^2 and combine the solution to find the shortest path between s and t. The order in which solution is computed in the subgraph G^1 and G^2 does not have any impact on the final solution. Similar arguments holds true if we assume s and t are separated by a split pair $\{x, y\}$. Therefore, merging operation mentioned section 4.2 for the shortest path problem is an associative operation. Therefore, we can use Tree-Product-Query data structures. Given the SPQR tree decomposition, we can construct the Tree-Product-Query data structures in $\mathcal{O}(m)$ time and perform the merging operation in time $\mathcal{O}(\alpha(n))$ where $\alpha(n)$ is inverse Ackermann function.

4.3.1 Algorithm using D_0 data structure

We have already explained this algorithm in section 4.2. For the given graph G with m edges and n vertices, preprocessing time is $\mathcal{O}(m)$ and query time is $\mathcal{O}(m)$ if G is undirected and all the weights are non-negative integers. If G is directed or weights are non-negative real numbers then preprocessing time is $\mathcal{O}(m + n \log r)$ and query time is $\mathcal{O}(m + n \log r)$. In the presence of negative weights, preprocessing time is $\mathcal{O}(m + nr)$ and query time is $\mathcal{O}(m + nr)$. Size of index (data structure D_0) is $O(n)$.

4.3.2 Algorithm using D_1 data structure

As explained in section 4.2, the constant size graphs that we construct using D_0 data structure are exactly the D_1 structures. Therefore, if we use more space then we can reduce the query time. So total time required for preprocessing stage i.e. construction of D_0 and D_1 data structures is $\mathcal{O}(m + nr^2)$. Let $\{u, v\}$ be the reference edge of p. Let $\{x, x'\}$ be the reference edge for the child of p in the path from μ_s to p and $\{y, y'\}$ be the reference edge for the child of p in the path from μ_t to p. In the query stage we need to construct a graph with four vertices $\{x, x', y, y'\}$ using skeleton of p. This is stored in the D_2 data structure and therefore we have to construct it at query time. Time for this construction is $\mathcal{O}(r)$ if G is undirected and all the weights are non-negative integers and $\mathcal{O}(r \log r)$ if G is directed or the weights are non-negative real numbers and $\mathcal{O}(r^2)$ if the weights are negative. For merging operations we will use tree product query data structures and hence total time required for all the merging is $\mathcal{O}(\alpha(n))$.

4.3.3 Algorithm using D_2 data structure

If we use D_2 data structure then in the preprocessing stage we compute D_0,D_1 and D_2 data structures. Total preprocessing time is $\mathcal{O}(m + nr^3)$. In the

query stage we use tree product query data structures and hence query time is $\mathcal{O}(\alpha(n))$.

Theorem 4 *For a graph G with n vertices and m edges,*

1. *If G is undirected and all the weights are non-negative integers then after $\mathcal{O}(m)$ preprocessing time, using index of size $\mathcal{O}(n)$, the shortest path between two vertices is computed in $\mathcal{O}(m)$ time. If G is directed or weights are non-negative real numbers then after $\mathcal{O}(m+n\log r)$ preprocessing time, using index of size $\mathcal{O}(n)$, the shortest path between two vertices is computed in $\mathcal{O}(m+n\log r)$ time. If the weights are negative then after $\mathcal{O}(m+nr)$ preprocessing time, using index of size $\mathcal{O}(n)$, the shortest path between two vertices is computed in $\mathcal{O}(m+nr)$ time.*

2. *If G is undirected and all the weights are non-negative integers then after $\mathcal{O}(m+nr^2)$ preprocessing time, using index of size $\mathcal{O}(m)$, the shortest path between two vertices is computed in $\mathcal{O}(r)$ time. If G is directed or weights are non-negative real numbers then after $\mathcal{O}(m+nr^2)$ preprocessing time, using index of size $\mathcal{O}(m)$, the shortest path between two vertices is computed in $\mathcal{O}(r\log r)$ time. If the weights are negative then after $\mathcal{O}(m+nr^2)$ preprocessing time, using index of size $\mathcal{O}(m)$, the shortest path between two vertices is computed in $\mathcal{O}(r^2)$ time.*

3. *For G (directed or undirected) with weights integers or real (negative weights allowed), after $\mathcal{O}(m+nr^3)$ preprocessing time, using index of size $\mathcal{O}(mr)$, the shortest path between two vertices is computed in $\mathcal{O}(\alpha(n))$ time.*

where r is the maximum size of the triconnected component in the SPQR tree decomposition of the given graph.

4.4 Algorithm for SSSP and APSP

Using data structures D_0, D_1, D_2 and tree product query data structures, we can compute shortest path from a given vertex to all other vertices and between all the pairs of vertices. Recall that r is the maximum size of the triconnected component in the SPQR tree decomposition of G.

Theorem 5 *Given G (directed or undirected and weights may be negative) and a source vertex, SSSP problem is solved in $\mathcal{O}(m+nr^2+n\alpha(n))$ time using D_0,D_1 and tree product query data structures which uses $\mathcal{O}(m)$ space*

Theorem 6 *Given G (directed or undirected and weights may be negative), APSP problem is solved in $\mathcal{O}(m+nr^3+n^2\alpha(n))$ time using D_0,D_1,D_2 and tree product query data structures.*

5 Network Algorithms

5.1 Preprocessing

In this section, we show preprocessing algorithms for solving max-flow problem using SPQR trees. Let G be a network with m edges and n vertices. If G is directed, we assume that whenever there is an edge from u to v, there is also an edge from v to u (possibly of capacity 0). First we assume G to be biconnected and describe our preprocessing algorithms. Then we describe the extension to general graphs. Given a biconnected graph G, we obtain the SPQR tree decomposition T of G in $\mathcal{O}(m+n)$ time. We construct data structures for nodes of T in the preprocessing stage.

In this section, let D'_0, D'_1, D'_2 denote the data structures for computing max-flow, which correspond to D_0, D_1, D_2 data structures for computing shortest paths.

5.1.1 Constructing D'_0 data structure

First we give a data structure D'_0 which stores in each node μ of the SPQR tree T for a graph $G = (V, E)$, the edge capacity of the mimicking network corresponding to the node μ.

If μ is an S-node, its skeleton is a polygon, consisting of the reference edge $\{u, v\}$ and a path connecting u and v. Here the reference edge can be considered as the network outside of the skeleton. If we see the skeleton from outside, it is the path between u and v. Then we can regard the path as an edge between u and v. Its capacity is the minimum among edges on the path if the graph is undirected. If the graph is directed, we create two directed edges (u, v) and (v, u). Their edge capacities are defined analogously.

If μ is a P-node, its skeleton is a graph with two vertices u, v and k multiple edges between them. Among k edges, one is the reference edge. Therefore we store an edge between u and v whose capacity is the summation of those of the edges except the reference edge.

If μ is an R-node, its skeleton is a triconnected graph. Let $\{u, v\}$ be the reference edge. If the graph is undirected, we compute the minimum cut value c between u and v in the skeleton without the edge $\{u, v\}$, and we store an edge with capacity c. If the graph is directed, we compute both $u - v$ and $v - u$ minimum cut values and store two edges whose capacities are those values.

We analyze the time complexity of the above algorithm. For S- and P-node, it takes time proportional to the number of edges in the SPQR tree, which is $\mathcal{O}(m)$. For each R-node, we compute max-flow constant times. Let n_i and r_i be the numbers of nodes and edges in the skeleton of a node μ_i of the SPQR tree. Then it takes $\mathcal{O}(n_i r_i)$ time for computing max-flow [29]. Therefore the total time for computing the D'_0 data structure is

$$\sum_{\mu_i} O(n_i r_i) = O(\sum_{\mu_i} n_i r) = O(nr)$$

where $r = \max r_i$ is the maximum size of skeletons.

Lemma 7 *Given an SPQR tree of a biconnected graph which has n nodes and the maximum size of whose triconnected components is r, the D'_0 data structure is stored in $\mathcal{O}(m)$ space and constructed in $\mathcal{O}(nr)$ time.*

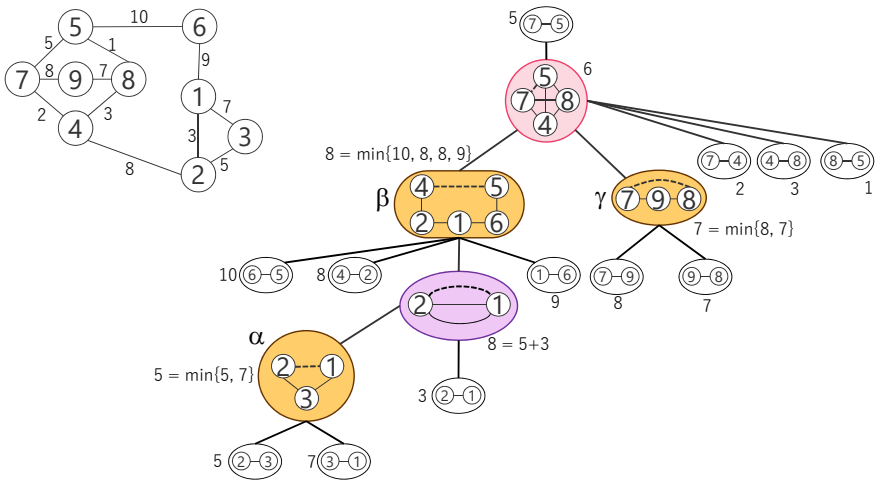

Figure 5: An input graph G (upper left) and its SPQR tree and the D'_0 data structure. In the graph, numbers beside edges show edge capacities. In the SPQR tree, dotted edges are reference edges.

Figure 5 shows an SPQR tree. In the SPQR tree, there are one R-node (red in color), one P-code (purple in color), and three S-nodes (sepia in color). The S-nodes are labeled α, β, and γ. Numbers beside nodes of the SPQR tree show the D'_0 data structure. Each node of the SPQR tre has one number, which is the capacity of the edge made by merging the edges in the node. In node α, the value is 5 because it is the minimum of $\{5, 7\}$, which are capacities of edges 2–3 and 3–1. In the P-node, the value is 8 because there are two parallel edges between node 2 and node 1 and their edge capacities are 5 and 3. In the R-node, the value is 6, which is obtained by computing the minimum cut between node 5 and node 7 in the skeleton of the R-node. The edge capacities of the edges in the skeleton are stored in the children. It is directly obtained that the minimum cut between node 5 and node 7 is 11 because we can consider that there are two parallel edges between them and their edge capacities are 5 (for the edge 5–7) and 6 (for the merged graph stored in the R-node).

5.1.2 Constructing D'_1 data structure

Next we compute D'_1 data structure which stores in each node of the SPQR tree, a mimicking network with four terminals, that is, of constant size. Let μ be a node of the SPQR tree, $v_1, v_2, \ldots, v_{k-1}$ be its children, and $\{u, v\}$ be the reference edge of μ.

If μ is an S-node, let $q_1 = u, q_2, \ldots, q_k = v$ be the nodes of the skeleton. The capacities of edges $\{q_1, q_2\}, \{q_2, q_3\}, \ldots, \{q_{k-1}, q_k\}$ are stored in the D'_0 data structure. For node v_i, we store the following graph with at most four terminals $\{q_i, q_{i+1}, u, v\}$. The graph has at most two edges: $\{u, q_i\}, \{q_{i+1}, v\}$. The edge capacities are the minimum of those of $\{q_1, q_2\}, \ldots, \{q_{i-1}, q_i\}$, the minimum of those of $\{q_i, q_{i+1}\}, \ldots, \{q_{k-1}, q_k\}$, respectively. That is, the graph is obtained by deleting the edge $\{q_{i-1}, q_i\}$ merging other edges into two. The edge capacities are computed in $\mathcal{O}(k)$ time as follows. If we know the minimum edge capacity of $\{q_1, q_2\}, \ldots, \{q_{j-1}, q_j\}$, the minimum edge capacity after adding another edge $\{q_j, q_{j+1}\}$ is computed in constant time. By repeating this, we obtain all the edge capacities.

If μ is a P-node, let $e_1, e_2, \ldots, e_k$ be the edges of the skeleton, and e_k be the reference edge. Assume that the edge e_i corresponds to the children v_i. Then for each v_i, we store a graph with two terminals $\{u, v\}$. The edge capacity is $\sum_{j=1}^{k-1} c(e_j) - c(e_i)$ where $c(e_j)$ is the capacity of e_j for undirected graphs. For directed graphs it is computed analogously. We can compute those graphs for all children of μ in $\mathcal{O}(k)$ time.

If μ is an R-node, for each child v_i of μ, we compute a mimicking network $M(G_\mu \backslash \{\{u, v\}, \{s, t\}\}, \{u, v, s, t\})$ where s, t are end points of the reference edge of v_i. If the skeleton of μ has n_i nodes and r_i edges, the mimicking network is computed in $\mathcal{O}(n_i r_i)$ time. Therefore for each R-node, it takes $\mathcal{O}(n_i r_i^2)$ time. The total time for all R-nodes is $\mathcal{O}(nr^2)$.

Lemma 8 *Given an SPQR tree of a biconnected graph which has n nodes and the maximum size of whose triconnected components is r, the D'_1 data structure is stored in $\mathcal{O}(m)$ space and constructed in $\mathcal{O}(nr^2)$ time.*

Figure 6 shows the D'_1 data structure for the example graph. Each node has a mimicking network with at most four terminals. The mimicking network for the P-node is obtained by merging edges 1–6 and 6–5 into one. The mimicking network for node β is obtained from the skeleton in the R-node by eleminating node 8.

5.1.3 Constructing D'_2 data structure

The D'_2 data structure is to store for each pair $\{s_1, t_1\}, \{s_2, t_2\}$ of edges of each R-node μ whose reference edge is $\{u, v\}$, the mimicking network with at most six terminals $M(G_\mu \backslash \{\{u, v\}, \{s_1, t_1\}, \{s_2, t_2\}\}, \{u, v, s_1, t_1, s_2, t_2\})$. For the node μ with n_i nodes and r_i edges, it takes $\mathcal{O}(n_i r_i^3)$ time. Then the total time is $\mathcal{O}(nr^3)$. The space is $\mathcal{O}(mr)$ because for each node we store at most $r - 1$ mimicking networks of constant size.

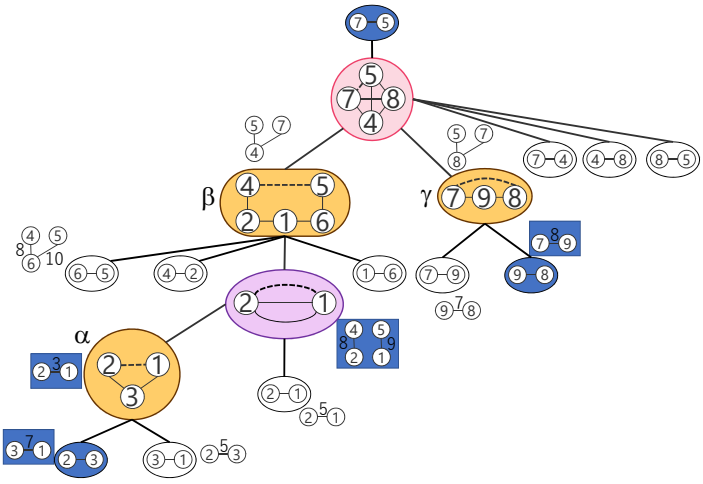

Figure 6: The D'_1 data structure for the graph.

Lemma 9 *Given an SPQR tree of a biconnected graph which has n nodes and the maximum size of whose triconnected components is r, the D'_2 data structure is stored in $\mathcal{O}(mr)$ space and constructed in $\mathcal{O}(nr^3)$ time.*

5.2 Computing $s-t$ Max Flow in $\mathcal{O}(m+nr)$ time

In this section, we show an algorithm for computing $s-t$ max flow in $\mathcal{O}(m+nr)$ time using the D'_0 data structure. First we consider an input graph is biconnected. We compute the SPQR tree in $\mathcal{O}(n+m)$ time, then construct the D'_0 data structure in $\mathcal{O}(nr)$ time.

From each of the given nodes s, t, we choose an arbitrary Q-node containing the node. Let μ_s, μ_t be the nodes, and p be their lowest common ancestor. Let $v_0 = \mu_s, v_1, \ldots, v_d = p$ be the nodes in the SPQR tree on the path from μ_s to p. For each node v_i in $v_1, \ldots, v_{d-1}$, we compute the mimicking network with at most four terminals by merging the mimicking networks for siblings of v_i. This is actually the same as $M(G_{v_{i+1}} \backslash \{\{u,v\},\{x,y\}\}, \{u,v,x,y\})$, which is in the D'_1 data structure, where $\{u,v\}$ and $\{x,y\}$ are the reference edges of v_{i+1} and v_i, respectively. Note that we do not compute those networks for all children of an R-node; only for the child having the Q-node for s in its subtree. Similarly for nodes between μ_t and p, we compute mimicking networks. We also compute for nodes between p and the root of the SPQR tree, the mimicking networks with

at most four terminals.

Finally we merge all the mimicking networks computed above. Because two mimicking networks adjacent in the tree have two common vertices, we can merge them. Let n_1, r_1 and n_2, r_2 be the number of nodes and edges in the two skeletons, respectively. The time complexity to merge the mimicking networks is $\mathcal{O}((n_1 + n_2)(r_1 + r_2)) = O((n_1 + n_2)r)$. Then the total time complexity is $\mathcal{O}(nr)$. Now we have a mimicking network with four terminals s, s', t, t' where s' and t' are the other end points of the edges in the Q-node containing s and t. By adding the edges $\{s, s'\}$ and $\{t, t'\}$ to the mimicking network and computing the $s - t$ minimum cut, we obtain the answer. This is done in constant time because the mimicking network is of constant size.

Once the value of the $s - t$ max flow is obtained, we can compute the flow itself. If the external flow of a mimicking network is fixed, we can obtain the flow in a skeleton by computing max flows constant times. And once the flow value of an edge of a skeleton is fixed, we can recursively compute the flow for the skeleton. The time complexity is the same as computing the max flow value.

Next we consider a general graph. First we compute the BC-tree in $\mathcal{O}(n+m)$ time. If s and t belong to the same biconnected component, we are done. Otherwise, for all blocks in the BC-tree on the path from the one containing s to the one containing t, we compute minimum cut values, and obtain the result. The time complexity is $\mathcal{O}(m + nr)$.

Theorem 7 *For a graph with n nodes and m edges whose maximum triconnected component is of size r, an $s - t$ max flow is computed in $\mathcal{O}(m + nr)$ time.*

5.3 Algorithms for MFIP

In this section we give algorithms for the MFIP. The results are summarized as follows.

Theorem 8 *For a directed network with n vertices and m edges,*

(i) *after $\mathcal{O}\left(m + n\lambda(k, n) + nr^3\right)$-time preprocess, using an index of size $\mathcal{O}\left(m + n\lambda(k, n) + mr\right)$, the value of the maximum flow is computed in constant time, or*

(ii) *after $\mathcal{O}\left(m + nr^3\right)$-time preprocess, using an index of size $\mathcal{O}\left(m + mr\right)$, the value of the maximum flow is computed in $\mathcal{O}(\alpha(n))$ time,*

(iii) *after $\mathcal{O}\left(m + nr^2\right)$-time preprocess, using an index of size $\mathcal{O}\left(m\right)$, the value of the maximum flow is computed in $\mathcal{O}(\alpha(n) + r^2)$ time,*

where r is the maximum size of triconnected components in the underlying undirected graph.

The proof is in the following subsections.

5.3.1 Algorithms for fast queries

To solve MFIP in a biconnected graph, we use the D'_1 and the D'_2 data structures and the tree product query data structure. In each node of the SPQR tree, a mimicking network is stored as D'_1. Because merging of mimicking networks is associative, we can use the data structure for tree product queries for those mimicking networks. The preprocess and query times are either $\mathcal{O}(n\lambda(k,n))$ and $\mathcal{O}(1)$, or $\mathcal{O}(n)$ and $\mathcal{O}(\lambda(k,n))$ for any $k \geq 0$.

Assume that vertices s, t are given as a query. For each of s, t, we choose an arbitrary Q-node containing the node. Let μ_s, μ_t be the nodes, and p be their lowest common ancestor. Let q_s and q_t be children of p on the path between μ_s and p and on the path between μ_t and p, respectively. Then the mimicking network M_s between μ_s and q_s is computed by using the tree product query data structure. Similarly the mimicking network M_t between μ_t and q_t is computed. We also compute the mimicking network M_p for nodes between p and the root of the SPQR tree using the tree product query data structure. Then we merge M_s, M_t, M_p, and the mimicking network $M(G_p \backslash \{\{u,v\},\{s_1,t_1\},\{s_2,t_2\}\},\{u,v,s_1,t_1,s_2,t_2\})$ where $\{u,v\}$ are the common vertices between M_p and $M(G_p,\{u,v,s_1,t_1,s_2,t_2\})$, $\{s_1,t_1\})$ are the common vertices between M_s and $M(G_p,\{u,v,s_1,t_1,s_2,t_2\})$, and $\{s_2,t_2\}$ are the common vertices between M_t and $M(G_p,\{u,v,s_1,t_1,s_2,t_2\})$. Because $M(G_p,\{v,u,s_1,t_1,s_2,t_2\})$ is stored in the D'_2 data structure and it is of constant size (six terminals), we can merge them in constant time, and compute the max flow value in constant time.

If the graph is not biconnected, in the preprocessing stage we construct the BC-tree and for each biconnected component we construct the SPQR tree and the tree product query data structure. Then for the BC-tree, we preprocess it for tree product queries.

5.3.2 An algorithm with small index

Here we give an algorithm using $\mathcal{O}(m)$ space based on the D'_1 data structure and the tree product query data structure. The algorithm is different from that in the previous subsection that we do not use the D'_2 data structure. Therefore for a query we have to solve max-flow problems in two nodes (p and the root). Because the number of edges in a skeleton is at most r, the max-flow can be solved in $\mathcal{O}(r^2)$ time.

In Figure 6, mimicking networks in blue boxes and ovals are used to compute the min-cut between node 3 and node 9.

Figure 7 shows how to merge mimicking networks to compute the min-cut between node 3 and node 9 using the D'_1 data structure.

5.4 Algorithms for Other Problems

The APMFP can be solved by computing the values of maximum flows for every pair of vertices using the D'_2 data structure and the tree product query data structure.

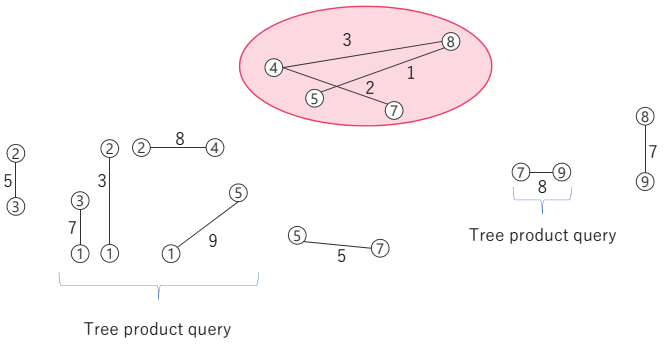

Figure 7: Computing the min-cut between node 3 and node 9.

Theorem 9 *The values of maximum flows of all pairs of vertices is computed in* $\mathcal{O}\left(nr^3 + n^2\right)$ *time.*

The minimum cut problem is also solved trivially by finding the maximum of all maximum flow values.

Theorem 10 *The value of the minimum cut is computed in* $\mathcal{O}\left(nr^3 + n^2\right)$ *time.*

6 Conclusion

We have proposed faster algorithms for network problems, especially the shortest path, the maximum flow and the minimum cut problems, based on a graph decomposition. Different from an existing work [3] based on the tree decomposition whose time complexity is doubly exponential to the tree-width, time complexities of our algorithms depend polynomially on a parameter r, the size of the maximum triconnected component. More importantly, triconnected component decomposition can be done in linear time, whereas finding a tree decomposition with minimum tree-width is NP-hard if the tree-width is not constant. Though $r = m$ in the worst case, our algorithms are faster than existing ones for small r cases. For the $s - t$ maximum flow problem, our algorithm runs in $\mathcal{O}(nr)$ time, which is faster than the fastest algorithm [29] if $r = o(m)$.

For $s-t$ shortest path problem, our algorithm runs in $\mathcal{O}(m)$ if the graph is undirected and weights are non-negative integers. If the weights are non-negative real numbers then our algorithm solves $s-t$ shortest path problem in $\mathcal{O}(m+n\log r)$ time which is faster than [16] if $r=o(n)$. If the weights are real numbers (i.e some of the weights are negative) then our algorithm solves $s-t$ shortest path problem in $\mathcal{O}(m+nr)$ time which is faster than Bellman and Ford algorithm [4],[14] if $r=o(m)$.

Our approach based on triconnected component decomposition can be easily applied for other network problems such as the network reliability problem. Our future work is to extend the scope of our approach and to show practical performance on real networks.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows - theory, algorithms and applications*. Prentice Hall, 1993.

[2] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical report, 1987.

[3] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis. All-pairs min-cut in sparse networks. *Journal of Algorithms*, 29(1):82 – 110, 1998. doi:10.1006/jagm.1998.0961.

[4] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[5] A. A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. Comput.*, 24(3):505–510, 1995. doi:10.1137/S0097539792236730.

[6] G. Borradaile and P. Klein. An o(n log n) algorithm for maximum st-flow in a directed planar graph. *J. ACM*, 56(2):9:1–9:30, Apr. 2009. doi:10.1145/1502793.1502798.

[7] S. Chaudhuri, K. V. Subrahmanyam, F. Wagner, and C. D. Zaroliagis. Computing mimicking networks. *Algorithmica*, 26(1):31–49, 2000. doi:10.1007/s004539910003.

[8] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987. doi:10.1007/BF01840366.

[9] G. Di Battista and R. Tamassia. Incremental planarity testing. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 436–441, 1989. doi:10.1109/SFCS.1989.63515.

[10] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996. doi:10.1007/BF01961541.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.

[12] J. Fischer and V. Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

[13] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8:399–404, 1956.

[14] L. R. Ford Jr. Network flow theory. *RAND paper*, 1956.

[15] G. Frederickson. Using cellular graph embeddings in solving all pairs shortest paths problems. *Journal of Algorithms*, 19(1):45 – 85, 1995. doi:10.1006/jagm.1995.1027.

[16] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

[17] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988. doi:10.1145/48014.61051.

[18] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9:551–570, 1961. doi:10.1137/0109047.

[19] C. Gutwenger. Application of SPQR-trees in the planarization approach for drawing graphs. *Ph.D thesis, Dortmund University of Technology*, 2010.

[20] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR-trees. In *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20-23, 2000, Proceedings*, pages 77–90, 2000. doi:10.1007/3-540-44541-2_8.

[21] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizations of k-terminal flow networks and computing network flows in partial k-trees. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California.*, pages 641–649, 1995.

[22] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994. doi:10.1006/jagm.1994.1043.

[23] J. E. Hopcroft and R. E. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, 1973. doi:10.1145/362248.362272.

[24] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 313–322, 2011. doi:10.1145/1993636.1993679.

[25] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977. doi:10.1145/321992.321993.

[26] V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 17(3):447–474, 1994. doi:10.1006/jagm.1994.1044.

[27] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. M. Basalamah. A survey of shortest-path algorithms. *CoRR*, abs/1705.02044, 2017.

[28] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. doi:10.1137/0405004.

[29] J. B. Orlin. Max flows in $O(nm)$ time, or better. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 765–774, 2013. doi:10.1145/2488608.2488705.

[30] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. doi:10.1137/0201010.

[31] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999. doi:10.1145/316542.316548.

[32] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.