On Applying Separator Decompositions to Path Problems and Network Flow[*]

M.J.Jansen

Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands[†]

Abstract

Separator decompositions have proven to be useful for efficient parallel shortest-path computation. In this paper the applicability of separator decompositions to maximum flow computation is explored. It is shown that efficient parallel shortest-path computation can be incorporated in the shortest augmenting path maximum flow algorithm. A class of graphs is described for which the resulting algorithm takes $O(n^{2+\mu}\log n)$ time and $O(n^3)$ work, where $0 < \mu < \frac{1}{3}$ is a class-dependent constant. For graphs with bounded treewidth an $\mathcal{NC}$-algorithm is known for the maximum flow problem. In this paper we show that width-$O(1)$ tree decompositions and separator decompositions with separators, leaf vertex sets, and boundaries of $O(1)$ size are equivalent notions under $\mathcal{NC}$ computation. $\mathcal{NC}$-algorithms are given for converting one type of graph decomposition into the other. Furthermore, the $\mathcal{NC}$-maxflow algorithm is restated in the separator decomposition framework.

1 Introduction

Network flow is a field with a rich history. It dates all the way back to the 19th-century, when physicist Kirchhoff studied the topic in the context of electrical circuits. In network flow a central role is taken by path problems, both in theoretic and algorithmic respect. Path algorithms are an essential building block in many flow algorithms, and provide us with a constructive understanding of network flow.

One of the classical problems within the field of network flow is the maximum flow problem. For sequential computing devices there is a long history of improvements on the worst-case time of the best-known maximum flow algorithm, starting with the pseudopolynomial algorithm of Ford and Fulkerson [12]. Nowadays fast polynomial algorithms are known.

[*] This research was partially supported by ESPRIT Long Term Research Project 20244 (project ALCOM IT: *Algorithms and Complexity in Information Technology*).

[†] All correspondence should be sent to M.J. Jansen, 52 Pellamwood, Grand Island, NY 14072, USA. Email: `mjj@affsys.com`

See Ahuja et al. [1] for a good survey.

On the parallel front some fundamental difficulties surround the maximum flow problem. Goldschlager et al. [14] showed that the problem is logspace complete for **P**, which can be seen as evidence that the problem is not likely to be solved by an $\mathcal{NC}$-algorithm.

Although in the general case efficient parallelization is not likely, good results are known for special cases.

In case the underlying graph is planar, Johnson [18] showed the existence of an $O(\log^3 n)$-time algorithm using $O(n^4)$ processors, and an $O(\log^2 n)$-time algorithm using $O(n^6)$ processors, both for the CREW PRAM. For outerplanar graphs a parallel algorithm is known using $O(\log n)$ time and n processors [8]. If capacities are polynomial in the number of edges m, the problem can be reduced in logspace to the maximum bipartite matching problem, which is in $\mathcal{RNC}$ [19]. Furthermore, Hagerup et al. [15] showed for graphs with treewidth $\leq k$, for constant k, the problem is in $\mathcal{NC}$.

Here, we will restrict ourselves to families of networks for which a recursive decomposition exists using small separators. A separator of a graph $G = (V, E)$ is a subset of vertices $S \subset V$ such that the subgraph induced by $V \backslash S$ is not connected. Informally, for a function $f : \mathbb{N} \to \mathbb{R}$, an $f(n)$-separator decomposition of G is a recursive tree-shaped decomposition of G into subgraphs using separators, where subgraphs of size k have separators of size $O(f(k))$ [9]. We revisit this definition in section 2. Cohen showed that for graphs with small enough separator decompositions, one can compute distances in a graph in $O(\log n)$ parallel time by augmenting the graph in a preprocessing phase with a set of edges E^+. Section 3 contains an exposition of this approach.

In trying to apply Cohen's approach to an augmenting-path based maximum flow algorithm one encounters an extra difficulty, namely that of a slowly changing underlying graph. It seems the only solution is to repeat the preprocessing phase after every augmentation. At the end of section 3 the applicability of Cohen's algorithm to the shortest augmenting-path maxflow algorithm is explored. For a specific class of graphs a parallel algorithm is given that has the same work bound and a slightly better running time than the sequential shortest augmenting-path maxflow algorithm.

We already mentioned the existence of a $\mathcal{NC}$ maxflow algorithm for graphs with treewidth $\leq k$, for constant k. The treewidth framework is very similar to the separator decompostion framework. Also in treewidth use is made of a tree shaped graph decomposition, namely the tree decomposition. The relationship between separator decompositions and tree decompositions is investigated in section 4. Here we will see that a tree decomposition of $O(1)$ width can be converted into a separator decomposition such that separators are of $O(1)$ size. Conversely, a separator decomposition of depth $O(\log n)$ and separators of $O(1)$ size, can be transformed into a tree decomposition of width $O(\log n)$.

The above mentioned $\mathcal{NC}$ maxflow algorithm will be treated in section 5. The exposition in this section diverges from the original article in the respect that the algorithm is presented with the aid of separator decompositions instead of tree decompositions. We will look at the more general case of networks with arbitrary number of sources and sinks. It is shown that a k-terminal flow in a network, i.e. flow in a network with k vertices that are source or sink, can be characterised by a set of at most 2^k equations in k variables, or a mimicking network with at most $2^{(2^k)}$ vertices. Separator decompositions facilitate the recursive computation of these characterizations. A class of graphs is given for which this implies that the maximum flow problem can be solved in $\mathcal{NC}$ time.

1.1 Acknowledgements

Previously, this report appeared as a master's thesis at Utrecht University. From this place I would like to direct some words of gratitude to the people that were involved in my graduation.

My thanks go out to Edwin and Helga, with whom I shared an office at the University. Thanks to you working on this thesis was accompanied by a healthy daily dose of fun. I would like to thank Jan van Leeuwen for being an assistant supervising professor during my graduation, and for his help and advice concerning my future in the US. I would like to thank Hans Bodlaender for being an assistant supervising professor, for the discussions we had, and for his generosity as it comes to lending books, and giving away papers. Especially, I would like to thank my supervising professor Marinus Veldhorst. Thank you for sharing your wisdom, and helping me form my ideas. Our weekly meetings were of incredible personal value. It is a wonderful thing to know that I can always walk into your office for advice.

2 Preliminaries

Unless explicitly specified otherwise, $G = (V, E)$ denotes a directed graph with $n = |V|$ nodes and $m = |E|$ edges. We assume that G does not contain isolated vertices, and that for all $v, w \in V$ only one of (v, w) and (w, v) can be in E.

Definition 2.1 *A network is a 4-tuple (G, c, s, t) where $c : E \to \mathbb{R}_{\geq 0}$ assigns capacities to the edges of G, and $s, t \in V$ are designated vertices called source and sink, respectively. A flow f in network (G, c, s, t) is a function $f : E \to \mathbb{R}_{\geq 0}$ for which:*

1. *for all edges $e \in E$, $0 \leq f(e) \leq c(e)$, and*
2. *for all $i \in V \backslash \{s, t\}$, $\sum_{(i,j)\in E} f(i,j) - \sum_{(j,i)\in E} f(j,i) = 0$.*

In other words, a flow must respect capacities (clause 1), and must be balanced in all vertices except the source and the sink (clause 2). The imbalance of a flow f in a node i is denoted by $b_f(i)$. The second clause of the above definition can therefore be reformulated to demanding that $b_f(i) = 0$ for all $i \in V \backslash \{s, t\}$. The value of a flow f is defined as the amount of imbalance $b_f(s)$ at the source. Given a network N, the maximum flow problem is the problem of finding a flow in N that has maximum value.

A cut in a network (G, c, s, t) is defined by specifying a subset $S \subseteq V$ of vertices. Given such a subset S, the corresponding cut is the set of edges that have their tails in S and their heads in $V \backslash S$. Each cut C has a certain capacity, which is defined as $\sum_{(v,w)\in C} c(v,w)$. In the following, for subsets of vertices $S \subseteq V$, if not ambiguous we will speak about the cut S instead of the cut with defining subset S. For subsets $X, Y \subseteq V$, a cut defined by subset $S \subseteq V$ is an X, Y-separating cut if $X \subseteq S$ and $Y \subseteq V \backslash S$. The famous Max-Flow Min-Cut theorem states [12]:

Theorem 2.1 *In a network (G, c, s, t) the value of any maximum flow is equal to the value of any s, t-separating cut of minimum capacity.*

Very useful in flow problems are residual networks. We will now give a definition for later use.

Definition 2.2 *For a flow f in network $N = (G, c, s, t)$ define the residual network $N(f) = (G(f), r, s, t)$, where*

1. $G(f) = (V, E(f))$,
2. $E(f) = \bigcup_{(v,w)\in E} \{(v,w),(w,v)\}$, *and*
3. $r(v,w) = \begin{cases} c(v,w) - f(v,w) & \textit{if } (v,w) \in E \\ f(w,v) & \textit{otherwise} \end{cases}$

In this paper we restrict ourselves to networks for which the underlying graph has a small separator decomposition. We now specify the separator decomposition concept.

Definition 2.3 *Given a graph $G = (V, E)$ and subsets of vertices $X, Y, Z \subset V$, we say that Y separates V into X and Z, if X, Y, Z form a partition of V, and X and Z are not connected in $G(V \backslash Y)$.*

In the following, we assume that internal nodes of rooted binary trees have exactly two children.

Definition 2.4 *A separator decomposition of a graph $G = (V, E)$ is a triple $(T_G, \mathcal{S}, \mathcal{V})$, where T_G is a rooted binary tree, and $\mathcal{S}, \mathcal{V}$ are functions $T_G \rightarrow \wp(V)$ for which the following three conditions hold:*

1. *for the root r of T_G, $\mathcal{V}(r) = V$*
2. *for leaves x of T_G, $\mathcal{S}(x) = \emptyset$*
3. *for each internal node x with children x_1 and x_2, $\mathcal{S}(x)$ separates the induced subgraph $G(\mathcal{V}(x))$ into V_1 and V_2, where for $i = 1, 2$,*
   $\mathcal{V}(x_i) = V_i \cup \{v \in \mathcal{S}(x) | \exists w \in V_i$ such that v and w incident in $G(\mathcal{V}(x))\}$.

The depth of a tree is the maximum number of edges on an acyclic path from the root to a leaf. Throughout this paper the depth of separator tree T_G is denoted with d_G. For a node x we denote with $T_G[x]$ the subtree of T_G induced by x and all its descendants. The maximum length of an acyclic path from node x to a leaf in $T_G[x]$ is denoted by height(x). For a function $f : \mathbb{N} \to \mathbb{N}$, an $f(k)$-separator decomposition is a separator decomposition with for all $x \in T_G$, $|\mathcal{S}(x)| = O(f(|\mathcal{V}(x)|)$. Separator decompositions are independent of the direction of edges in the underlying graph. It is not hard to prove the following.

Fact 2.1 *$\mathcal{S}$ is a separator decomposition of a directed graph G if and only if $\mathcal{S}$ is a separator decomposition of the undirected skeleton of G.*

For $x \in T_G$, let $G(x)$ denote the induced subgraph $G(\mathcal{V}(x))$, and define the boundary $\mathcal{B}(x)$ as:

$$\mathcal{B}(x) = \begin{cases} \emptyset & \text{if } x \text{ is the root of } T_G \\ (\mathcal{B}(y) \cup \mathcal{S}(y)) \cap \mathcal{V}(x) & \text{otherwise, and where } y \text{ is the parent of } x \text{ in } T_G. \end{cases}$$

$G(x)$ is 'connected' to the remainder of G by its boundary $\mathcal{B}(x)$. We can prove the following proposition.

Proposition 2.1 *For all $x \in T_G$,*

1. *$\mathcal{B}(x)$ separates G into $\mathcal{V}(x)\backslash\mathcal{B}(x)$ and $V\backslash\mathcal{V}(x)$.*
2. *$\mathcal{B}(x) = \mathcal{V}(x) \cap \bigcup_{y \in W_x} \mathcal{S}(y)$, where W_x is the set of all ancestors of x.*

In 1. we see that boundaries indeed have the connection-type character we just mentioned. Part 2. of the proposition is the iterative counterpart of our initial recursive boundary definition.

In this report, we will use as a model of parallel computation the CRCW PRAM. In case of a simultaneous write of the same memory location we assume that the minimum value is written in that location. For the assessment of parallel algorithms we will use the concepts of time and work. The time of a parallel algorithm A on P processors is the number of parallel steps in which at most P operations can be performed simultaneously. In case the number of processors is not specified, the time of A is the time of A on an arbitrarily large number of processors. The work of a parallel or sequential algorithm A is the total number of operations performed, and is independent of the number of processors. If p processors write the same memory location simultaneously this will give a contribution of p to the work of A. The work of A may be strictly smaller than the processor-time product because idle time is not included in it. For an explanation of these concepts we refer to JáJá [17]. Time and work are identical for sequential algorithms that have no idle time.

An algorithm of time $T(n)$ and work $W(n)$ with n the length of the input, can be realized on $P(n)$ processors and the realization runs in time $O(T(n) + W(n)/P(n))$, provided the

so-called Brent schedule [6, 17] can be applied.
In all CRCW PRAM algorithms in this paper the number of processors accessing the same memory location in one computation step is bounded by a polynomial in n, even if the total number of processors is much larger. This makes that in this paper any CRCW PRAM algorithm running in time $T(n)$ can be simulated on a EREW PRAM with the same number of processors in time $O(T(n) \cdot \log n)$.

3 Augmenting Path Maxflow Computation

The current state of the art allows for a distinction between two classes of maximum flow algorithms. We have preflow-push algorithms on one hand, and augmenting-path algorithms on the other. In this section our focus will solely be on an algorithm from the latter category, namely the shortest augmenting path algorithm.

Cohen showed for graphs with small-sized separator decompositions that distances can be computed in polylogarithmic time, and with polynomial work [9]. Given a graph G, the idea is to extend G with a set of edges E^+, without changing distances, and enabling the Bellman-Ford algorithm to take shortcuts during the path search. For the class of graphs considered, the work bound of Cohen's algorithm improves on the work bound of the sequential Bellman-Ford algorithm.

Distance computation is closely related to shortest path computation. In this section the applicability of Cohen's algorithm to the shortest augmenting path maxflow algorithm will be investigated. We will show that with a few minor changes Cohen's algorithm can be incorporated into this maxflow algorithm. For the class of graphs considered, the resulting parallel algorithm improves slightly on the time bound of the sequential shortest augmenting path maxflow algorithm. Furthermore, for a substantial subclass of graphs the algorithm and its sequential counterpart perform an equal amount of work.

The rest of this section is organised as follows. Section 3.1 reviews the Bellman-Ford algorithm and a straightforward parallelization for the CRCW PRAM. In section 3.2 this algorithm will be modified to function more efficiently under the addition of the specific set of edges E^+. An algorithm for the computation of E^+ is given in section 3.3. In section 3.4 these results will be applied to the shortest augmenting path maxflow algorithm. Section 3.5 contains an analysis of the complexity of the algorithms of this section for a specific class of graphs. The algorithms of section 3.2 and 3.3 turn out to be in $\mathcal{NC}$ for graphs with small-sized separator decompositions. Finally, in section 3.6 we end with a short digresssion on flow augmentation.

3.1 Parallel Bellman-Ford

Let $G = (V, E)$ have weights $w : E \to \mathbb{R}$ on the edges. $Reach_G(v)$ is the set of vertices $w \in V$ such that there is a path in G from v to w. The length of a path p in G is defined as the number of edges in p, and the weight of p is defined as $w(p) = \sum_{e \in p} w(e)$. For vertices $v, w \in V$ the distance $\delta_G(v, w)$ between v and w is the minimum weight of a path from v

to w. If $w \notin Reach_G(v)$, then $\delta_G(v,w) = \infty$. Given a specified source vertex s, the single source shortest path problem is the problem of calculating $\delta_G(s,v)$ for all vertices $v \in V$.

In case no negative-weight cycles exist, the single source shortest path problem can be solved by the Bellman-Ford algorithm [10]. The algorithm consists of $n-1$ phases. In each phase, for each edge $e = (v,w) \in E$ the distance from s to w is updated as the minimum of the current distance from s to w, and the sum of $w(e)$ and the current distance of s to v. This update operation is referred to as 'relaxing edge e'. Let us specify the algorithm in pseudocode. For each vertex $v \in V$, the current distance from s to v is maintained in a label $d[v]$.

```
Procedure Relax($e = (v,w)$)
1. if $d[w] > d[v] + w(e)$ then
2.     $d[w] := d[v] + w(e)$
```

```
Procedure Bellman-Ford($G = (V,E)$)
1. $d[s] := 0$
2. for all $v \in V \backslash \{s\}$ do
3.     $d[v] := \infty$
4. repeat $n-1$ times:
5.     for each edge $e \in E$ do
6.         Relax($e$)
7. for each edge $e = (v,w) \in E$ do
8.     if $d[w] > d[v] + w(e)$ then
9.         error: minimum-weight cycle detected
```

At termination of the algorithm, either it will be detected that G has a negative-weight cycle, or $d[v] = \delta_G(s,v)$ for all $v \in V$. The algorithm runs in $O(nm)$ time.

Parallelizing this algorithm for the CRCW PRAM can be done by turning the for-loops in lines 2, 5 and 7 into parallel loops. In line 6 only the minimum update value is written. On the CRCW PRAM the algorithm runs in $O(n)$ time and performs $O(nm)$ work.

3.2 Adapted Bellman-Ford

The CRCW PRAM algorithm of the previous section performs $n-1$ relaxation phases, which implies a factor n in its asymptotic running time. Actually, the necessary number of relaxation phases equals the minimum length L of any minimum-weight path from s to v, maximized over all $v \in Reach_G(s)$. In order to reduce the running time of the algorithm, we augment G with a set E^+ of edges with weights that do not change distances between vertices, and such that L is small for the extended graph. After this preprocessing of G we can apply the parallel Bellman-Ford algorithm to the extended graph in a specific way, and we need less phases.

In the following text we assume that G does not contain negative-weight cycles. Furthermore, we assume we have a separator decomposition of G and a constant $l \geq 0$ such that for each leaf x in the separator decomposition tree and all vertices v and w in $G(x)$ with $w \in \mathit{Reach}_{G(x)}(v)$, there exists a path from v to w in $G(x)$ with weight $\delta_{G(x)}(v, w)$ that is of length at most l. E^+ will be chosen such that for all vertices v in G and $w \in \mathit{Reach}_G(v)$ there is a path from v to w in the extended graph with weight $\delta_G(v, w)$ and of length $O(l + d_G)$. Hence, the parallel Bellman-Ford algorithm can be implemented on the extended graph to run in $O(l + d_G)$ relaxation phases.

Definition 3.1 *Given a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of graph G, for each node $x \in T_G$ define the set of edges*

$$\begin{aligned} E_x \;=\; & \{(v, w) | v, w \in \mathcal{B}(x), v \neq w, \text{ and } w \in \mathit{Reach}_{G(x)}(v)\} \cup \\ & \{(v, w) | v, w \in \mathcal{S}(x), v \neq w, \text{ and } w \in \mathit{Reach}_{G(x)}(v)\} \end{aligned}$$

where each edge $e = (v, w) \in E_x$ has weight $w(e) = \delta_{G(x)}(v, w)$. In addition, let $E^+(x) = \bigcup_{x' \in T_G[x]} E_{x'}$.

In the above definition, we assume that if multiple edges would occur in $E^+(x)$, only the one with minimum weight is included. For the root r of T_G we denote $E(r)^+$ by E^+. Distances in G do not change by the addition of edges (v, w) to E with weight $\delta_G(v, w)$. In the following the operator $\uplus$ denotes multiset union.

Fact 3.1 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of graph G. For each $x \in T_G$ define the multigraph $G^+(x) = (\mathcal{V}(x), E(\mathcal{V}(x)) \uplus E^+(x))$, then for all $v, w \in \mathcal{V}(x)$ it holds that $\delta_{G(x)}(v, w) = \delta_{G^+(x)}(v, w)$.*

As we already mentioned the addition of edges from E^+ introduces paths of small length and specific structure. The following two recursive definitions are a first step towards describing this structure. Given a node x in T_G, two types of paths are defined: x-entry paths and x-exit paths. Roughly speaking, an x-entry path is a minimum-weight path in $G^+(x)$ consisting of three, possibly empty, subpaths. The first subpath lies completely in one of the G^+-graphs associated with the children of x. The second subpath consists of a single edge between two vertices in the separator $\mathcal{S}(x)$. The third subpath consists of a single edge from a vertex in $\mathcal{S}(x)$ to a vertex in the boundary $\mathcal{B}(x)$. Denote the empty path starting at a vertex v with λ_v, and let $+\!\!+$ denote the operator for concatenating paths. We have the following definition.

Definition 3.2 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of G, and x a node in T_G. A path p from vertex v to vertex w in $G^+(x)$ is an x-entry path if the following four conditions are satisfied:*

1. *$w \in \mathcal{B}(x)$*
2. *$w(p) = \delta_{G(x)}(v, w)$*
3. *if x is a leaf, then p is a sequence of at most l edges in $G(x)$*

4. if x is an internal node with children x_1 and x_2, then $p = p_1 {+\!\!+} p_2 {+\!\!+} p_3$, where

(a) $p_1 = \lambda_v$, or p_1 is an x_i-entry path for some $i \in \{1,2\}$,

(b) $p_2 = \lambda_{v_1}$ with $v_1 \in \mathcal{V}(x)$, or $p_2 = (v_1, v_2) \in E_x$ with $v_1, v_2 \in \mathcal{S}(x)$, and

(c) $p_3 = \lambda_{w_1}$ with $w_1 \in \mathcal{V}(x)$, or for some $j \in \{1,2\}$, $p_3 = (w_1, w_2) \in E_{x_j}$ with $w_1, w_2 \in \mathcal{B}(x_j)$.

Definition 3.3 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of G, and x a node in T_G. A path p from vertex v to vertex w in $G^+(x)$ is an x-exit path if the following four conditions are satisfied:*

1. *$v \in \mathcal{B}(x)$*
2. *$w(p) = \delta_{G(x)}(v, w)$*
3. *if x is a leaf, then p is a sequence of at most l edges in $G(x)$*
4. *if x is an internal node with children x_1 and x_2, then $p = p_1 {+\!\!+} p_2 {+\!\!+} p_3$, where*

   (a) $p_1 = \lambda_v$, or for some $i \in \{1,2\}$, $p_1 = (v_1, v_2) \in E_{x_i}$ with $v_1, v_2 \in \mathcal{B}(x_i)$,

   (b) $p_2 = \lambda_{w_1}$ with $w_1 \in \mathcal{V}(x)$, or $p_2 = (w_1, w_2) \in E_x$ with $w_1, w_2 \in \mathcal{S}(x)$, and

   (c) $p_3 = \lambda_w$, or p_3 is an x_j-exit path for some $j \in \{1,2\}$.

We have the following important fact about the lengths of entry paths and exit paths.

Lemma 3.1 *For any graph G and separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G, it holds that each x-entry path and each x-exit path has length at most $l + 2\mathit{height}(x)$, for all $x \in T_G$.*

Proof: We use induction to prove the lemma for x-entry paths. For a leaf x, an x-entry path trivially has length at most $l{+}2\text{height}(x)$. Suppose x is not a leaf. Let x_1 and x_2 be the children of x. The induction hypothesis states that each x_i-entry path has length at most $l{+}2\text{height}(x_i)$, for $i = 1, 2$. From definition 3.2 we see that an x-entry path consists of either at most $l + 2\text{height}(x_1) + 2$, or at most $l + 2\text{height}(x_2) + 2$ edges. Therefore, an x-entry path will certainly be of length at most $l + 2\max\{\text{height}(x_1), \text{height}(x_2)\} + 2 = l + 2\text{height}(x)$. The proof of the lemma for exit paths is similar. ■

In the following two lemmas we will see that entry paths and exit paths occur in the extended graph G^+ in extensive amounts. The proofs of these lemmas are obtained by applying induction in the separator decomposition tree.

Lemma 3.2 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of graph G, and let x be a node in T_G. For any two vertices $v, w \in \mathcal{V}(x)$ such that $w \in \mathit{Reach}_{G(x)} \cap \mathcal{B}(x)$ there exists an x-entry path from v to w in $G^+(x)$.*

Proof: Consider arbitrary vertices $v, w \in \mathcal{V}(x)$ with $w \in \mathit{Reach}_{G(x)} \cap \mathcal{B}(x)$. If x is a leaf, an x-entry path from v to w exists in $G(x)$ by definition of l.

Suppose x is not a leaf. Let x_1 and x_2 be the children of x. Since w is reachable from v, there exists a path q from v to w in $G(x)$ of weight $\delta_{G(x)}(v, w)$. We consider two cases.

Case 1. $q \cap \mathcal{S}(x) = \emptyset$. Now q is a path in $G(x_i)$ for some $i \in \{1, 2\}$. Observe that $w(q) = \delta_{G(x_i)}(v, w)$, and $w \in \mathcal{B}(x_i)$. The induction hypothesis tells us that an x_i-entry path p from v to w exists. Since $w(p) = w(q) = \delta_{G(x)}(v, w)$, we conclude that p is an x-entry path from v to w.

Case 2. $q \cap \mathcal{S}(x) \neq \emptyset$. Let v_1 and v_2 be the first and the last vertex of q in $\mathcal{S}(x)$, respectively. Note that possibly $v_1 = v_2$. We can look at q as being a concatenation $q_1 +\!\!+ q_2 +\!\!+ q_3$, where q_1 is a path from v to v_1, q_2 is a path from v_1 to v_2, and q_3 is a path from v_2 to w. The paths q_1, q_2, and q_3 are possibly empty. We now construct an x-entry path p from v to w.
If $q_1 = \lambda_v$, then let $p_1 = \lambda_v$. Otherwise, for some $i \in \{1, 2\}$, q_1 is a path in $G(x_i)$. Observe that $w(q_1) = \delta_{G(x_i)}(v, v_1)$. Since $v_1 \in \mathcal{B}(x_i)$, the induction hypothesis tells us that an x_i-entry path from v to v_1 exists. Let p_1 be this path, then p_1 has weight $w(p_1) = \delta_{G(x_i)}(v, v_1) = w(q_1)$.
If $q_2 = \lambda_{v_1}$, let $p_2 = \lambda_{v_1}$. Otherwise, let p_2 be the path from v_1 to v_2 consisting of the single edge $(v_1, v_2) \in E_x$. Observe that in $w(p_2) = w(q_2)$.
If $q_3 = \lambda_{v_2}$, let $p_3 = \lambda_{v_2}$. Otherwise, for some $j \in \{1, 2\}$ it must be that $v_2, w \in \mathcal{B}(x_j)$, and hence $(v_2, w) \in E_{x_j}$. Let p_3 be the path from v_2 to w consisting of the single edge $(v_2, w) \in E_{x_j}$. Observe that $w(p_3) = w(q_3)$.
Let p be the path from v to w defined by the concatenation $p_1 +\!\!+ p_2 +\!\!+ p_3$, then $w(p) = w(p_1) + w(p_2) + w(p_3) = w(q_1) + w(q_2) + w(q_3) = w(q) = \delta_{G(x)}(v, w)$. Hence p is an x-entry path from v to w. ■

In the same way the following lemma for exit paths can be proven.

Lemma 3.3 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of graph G, and let x be a node in T_G. For any two vertices $v, w \in \mathcal{V}(x)$ such that $v \in \mathcal{B}(x)$ and $w \in \mathit{Reach}_{G(x)}(v)$ there exists an x-exit path from v to w in $G^+(x)$.*

As we will see more precisely in the proof of theorem 3.1, for almost all vertices $v \in \mathit{Reach}_G(s)$ a minimum-weight path from s to v exists that is the concatenation of an entry path, an edge from E^+, and an exit path. If the edges of such a path are relaxed in path order, we know that this results in a distance label $d[v] = \delta_G(s, v)$. Therefore the following two procedures are introduced. Given a node $x \in T_G$, the procedure EntryRelax(x) performs relaxations in such a way that for each $x' \in T_G[x]$ and each x'-entry path p, the edges of p have been relaxed in path order. The notation $\|$ used in these procedures indicates that the recursive calls are to be executed in parallel.

```
Procedure EntryRelax($x \in T_G$)
1.  if leaf($x$) then
2.      Perform $l$ phases of parallel relaxation on the edges of $G(x)$.
```

```
3. else #Let $x_1$ and $x_2$ be the children of $x$.
       EntryRelax($x_1$)|| EntryRelax($x_2$)
4.     Perform one phase of parallel relaxation on all
       edges $(v, w) \in E_x \cap E^+$ with $v, w \in \mathcal{S}(x)$.
5.     for $i = 1, 2$ do
6.         Perform one phase of parallel relaxation on all
           edges $(v, w) \in E_{x_i} \cap E^+$ with $v, w \in \mathcal{B}(x_i)$.
```

Similar to EntryRelax the procedure ExitRelax traces down exit paths:

```
Procedure ExitRelax($x \in T_G$)
1. if leaf($x$) then
2.     Perform $l$ phases of parallel relaxation on the edges of $G(x)$.
3. else #Let $x_1$ and $x_2$ be the children of $x$.
       for $i = 1, 2$ do
4.         Perform one phase of parallel relaxation on all
           edges $(v, w) \in E_{x_i} \cap E^+$ with $v, w \in \mathcal{B}(x_i)$.
5.     Perform one phase of parallel relaxation on all
       edges $(v, w) \in E_x \cap E^+$ with $v, w \in \mathcal{S}(x)$.
6.     ExitRelax($x_1$)|| ExitRelax($x_2$)
```

Using these procedures we get the following CRCW PRAM algorithm for computing distances in G.

```
Procedure Bellman-Ford$^+$($G$, $(T_G, \mathcal{S}, \mathcal{V})$)
# Let $r$ be the root of $T_G$.
1. $d[s] := 0$
2. for all $v \in V \backslash \{s\}$ do
3.     $d[v] := \infty$
4. if leaf($r$) then
5.     Perform $l$ phases of parallel relaxation on the edges of $G(r)$.
6. else #Let the children of $r$ be $r_1$ and $r_2$.
       EntryRelax($r_1$)|| EntryRelax($r_2$)
7.     Perform one phase of parallel relaxation on all
       edges $(v, w) \in E_r \cap E^+$ with $v, w \in \mathcal{S}(r)$.
8.     ExitRelax($r_1$)|| ExitRelax($r_2$)
9. *Check for negative-weight cycles.*
```

Let us now prove the correctness of this algorithm.

Theorem 3.1 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of graph G. Let G have weight $w : E \to \mathbb{R}$ on the edges such that no negative-weight cycle exists. At termination of Bellman-Ford$^+$ it holds that $d[v] = \delta_{G(x)}(s, v)$, for all $v \in V$.*

Proof: In the proof we will frequently use the following claim on the Bellman-Ford algorithm given by Cormen et al. [10]:

Claim 1 [10, Lemma 25.5]. After initialization, $d[v] \geq \delta_G(s,v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps on the edges of G. Moreover, once $d[v]$ achieves its lower bound $\delta_G(s,v)$, it never changes.

Now assume no negative-weight cycle exists. Then $\delta_G(s,s) = 0$, and $d[s]$ is initialized to $\delta_G(s,s)$. For a vertex $v \notin \mathit{Reach}_G(s)$, $d[v]$ is initialized with $\delta_G(s,v) = \infty$. By Claim 1 we know that $d[s]$ and $d[v]$ keep their correct values during the algorithm.

For an arbitrary vertex $v \in \mathit{Reach}_G(s)$, a minimum-weight path p from s to v exists in G. We distinguish between two cases.

If $p \cap \bigcup_{x \in T_G} \mathcal{S}(x) = \emptyset$, then p is a path in $G(x)$ for some leaf $x \in T_G$. Therefore, a path p' from s to v exists in $G(x)$ such that $w(p') = w(p)$ and the length of p' is at most l. Unwinding the recursion of the calls in line 6, we see that we perform l relaxation phases on the edges of p'. Since p' is a minimum-weight path, at the end of these phases we have $d[v] = \delta_G(s,v)$. Claim 1 tells us this is maintained thereafter.

Suppose $p \cap \bigcup_{x \in T_G} \mathcal{S}(x) \neq \emptyset$. Let x be the highest node in T_G for which $p \cap \mathcal{S}(x) \neq \emptyset$. Let x have children x_1 and x_2, and let v_1 and v_2 be the first and the last vertex on p in $\mathcal{S}(x)$, respectively. Note that possibly $v_1 = v_2$. Consider p as a concatenation $p = p_1 +\!\!+ p_2 +\!\!+ p_3$, where p_1 is a path from s to v_1, p_2 is a path from v_1 to v_2, and p_3 is a path from v_2 to w. Possibly the paths p_1, p_2, and p_3 are empty. If $p_1 = \lambda_v$ let $p'_1 = \lambda_v$. Otherwise, for some $i \in \{1,2\}$ it holds that p_1 lies completely in $G(x_i)$, and that $v_1 \in \mathcal{B}(x_i)$. By lemma 3.2 an x_i-entry path p'_1 from s to v_1 must exist. In both cases we have that $w(p'_1) = w(p_1)$.
If $p_2 = \lambda_{v_1}$ let $p'_2 = \lambda_{v_1}$. Otherwise, we have that $v_1 \neq v_2$, so let p'_2 be the path consisting of the single edge $(v_1, v_2) \in E_x$. Trivially, we have that $w(p'_2) = w(p_2)$.
If $p_3 = \lambda_{v_2}$ let $p'_3 = \lambda_{v_2}$. Otherwise, for some $j \in \{1,2\}$ the path p_3 lies completely in $G(x_j)$ and $v_2 \in \mathcal{B}(x_j)$. By lemma 3.3 an x_j-exit path p'_3 from v_2 to v must exist. We have that $w(p'_3) = w(p_3)$.
From the above we conclude that the path $p' = p'_1 +\!\!+ p'_2 +\!\!+ p'_3$ is a minimum-weight path in G^+ from s to v. Therefore, if the edges of p' are relaxed in path order during the execution of Bellman-Ford$^+$, we know that this results in distance label $d[v] = \delta_G(s,v)$.

In case x is the root of T_G, we have that in line 6 the edges of p'_1 are relaxed in path order. In line 7 the single-edge path p'_2 is relaxed, and in line 8 the edges of p'_3 are relaxed in path order.

Suppose x is not the root of T_G. Observe that the parallel calls in line 6 will at some point cause the call EntryRelax(x) to be made. In this call, the recursive parallel calls EntryRelax(x_1) and EntryRelax(x_2) make sure that the edges of p'_1 are relaxed in path order. Furthermore, the execution of line 6 of the incarnation EntryRelax(x) relaxes the single-edge path p'_2. Similarly the parallel calls in line 8 of algorithm Bellman-Ford$^+$ and the structure of ExitRelax(x) make sure that the edges of p'_3 are relaxed in path order. We conclude that the edges of p' are relaxed in path order during the execution of the Bellman-Ford$^+$ algorithm. ■

Let us analyze the complexity of the algorithm. Unwinding the recursion of procedures EntryRelax, and ExitRelax, we find that the algorithm Bellman-Ford$^+$ relaxes each edge in E^+ at most once, and each edge in $E \setminus E^+$ at most $2l$ times. Therefore, the amount of work performed is $O(|E^+| + l|E|)$. Since edges are relaxed in parallel we find a running time of $O(l + d_G)$.

Now consider the number of concurrent reads and writes the algorithm performs on a distance label. Observe that at most $O(n)$ incarnations of the procedures EntryRelax and ExitRelax are active. Each such an incarnation relaxes $O(n^2)$ edges at the same time since the graph associated with the node currently processed trivially contains $O(n)$ vertices. We conclude that the algorithm performs $O(n^3)$ concurrent memory operations on a distance label. Therefore, the Bellman-Ford$^+$ algorithm can be simulated on a EREW PRAM with only an extra factor of $O(\log n^3) = O(\log n)$ in its running-time.

3.3 Computing E^+

Computation of the set E^+ can be done in a bottom-up fashion. Given a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of graph G, the sets E_x are computed first for all leaves $x \in T_G$. Next, this information is used to compute the sets E_x for all internal nodes x that are just above leaf-level. Processing on like this, the computations ends with computing the set of edges associated with the root of T_G.

For each leaf x the Floyd-Warshall algorithm [10, chapter 26] can be used on $G(x)$ to determine the weights on the edges in E_x. The computation at internal nodes takes some more effort. Let us now go into the details of this.

Suppose for an internal node x with children x_1 and x_2 we want to compute E_x, given the sets E_{x_1} and E_{x_2}. For each edge $e = (v, w) \in E_x$ we have that $v, w \in \mathcal{S}(x)$, or $v, w \in \mathcal{B}(x)$.

For the purpose of handling the case that $v, w \in \mathcal{S}(x)$ we define the graph $I^x = (\mathcal{S}(x), \mathcal{S}(x) \times \mathcal{S}(x))$. Let $e = (v, w)$ be an edge in I^x. In case $e \notin E$ take in the following $w(e) = \infty$. The weight on edge e in I^x is defined as:

$$w_{I^x}(e) \quad = \quad \begin{cases} \min\{\delta_{G(x_1)}(v,w), \delta_{G(x_2)}(v,w)\} & \text{if } v, w \in \mathcal{B}(x_1) \cap \mathcal{B}(x_2) \\ \delta_{G(x_1)}(v,w) & \text{if } v, w \in \mathcal{B}(x_1) - \mathcal{B}(x_2) \\ \delta_{G(x_2)}(v,w) & \text{if } v, w \in \mathcal{B}(x_2) - \mathcal{B}(x_1) \\ w(e) & \text{otherwise} \end{cases}$$

Note that $\delta_{G(x_1)}(v, w)$ or $\delta_{G(x_2)}(v, w)$ is known in the first three cases since E_{x_1} and E_{x_2} already have been computed. The following lemma tells us that in order to determine the weights on edges $e \in E_x$ with $e \in \mathcal{S}(x) \times \mathcal{S}(x)$, it suffices to compute distances in I^x.

Lemma 3.4 *Let x be an internal node in T_G. Then $\delta_{I^x}(v, w) = \delta_{G(x)}(v, w)$ for arbitrary vertices v and w in $\mathcal{S}(x)$.*

Proof: First consider the case that $w \notin Reach_{G(x)}(v)$. In other words $\delta_{G(x)}(v, w) = \infty$. Suppose that $\delta_{I^x}(v, w) \neq \infty$. In this case there must be a path p from v to w in I^x, for which all the edges have finite weight. Consider an arbitrary edge $e = (v_1, v_2)$ on p. Since $w_{I^x}(e)$ is finite, we have that (v_1, v_2) is an edge in $G(x)$ of finite weight, or $\delta_{G(x_1)}(v_1, v_2)$ is finite, or $\delta_{G(x_2)}(v_1, v_2)$ is finite. Therefore, there exists a path in $G(x)$ from v_1 to v_2. Since e was an arbitrary edge on p, a path from v to w exists in $G(x)$. This is a contradiction. Hence, $\delta_{I^x}(v, w) = \infty$.

Now consider the case that $w \in \mathit{Reach}_{G(x)}(v)$. Let p be a minimum-weight path from v to w in $G(x)$. In case p is empty, we have $\delta_{G(x)}(v, w) = 0$, and trivially, $\delta_{I^x}(v, w) \leq \delta_{G(x)}(v, w)$. In case p is not an empty path, consider any edge $e = (v_1, v_2)$ on p for which $v_1, v_2 \in \mathcal{S}(x)$. Since p is a minimum-weight path we have that $w(e) = \delta_{G(x)}(v_1, v_2)$. Observe that $w_{I^x}(e) = w(e)$. Now consider any subpath $(u_1, \ldots, u_2)$ of p of length at least 2, for which $u_1, u_2 \in \mathcal{S}(x)$, and the vertices in between are not in $\mathcal{S}(x)$. Observe that $u_1, u_2 \in \mathcal{B}(x_i)$ for some $i \in \{1, 2\}$, and that $\delta_{G(x_i)}(u_1, u_2) = \delta_{G(x)}(u_1, u_2)$. Hence, $w_{I^x}((u_1, u_2)) = \delta_{G(x)}(u_1, u_2)$. Since p is the concatenation of edges (v_1, v_2) and subpaths $(u_1, \ldots, u_2)$ as above, we conclude that there exists a path from v to w in I^x of weight $w(p)$, and thus $\delta_{I^x}(v, w) \leq \delta_{G(x)}(v, w)$.

Let us now show that $\delta_{I^x}(v, w) \geq \delta_{G(x)}(v, w)$ for vertices $v, w \in \mathcal{S}(x)$. Suppose that $\delta_{I^x}(v, w) < \delta_{G(x)}(v, w)$, for some vertices $v, w \in \mathcal{S}(x)$, and is finite. Let p be a minimum-weight path from v to w in I^x, such that p has minimum length. If p has length 0, trivially $\delta_{I^x}(v, w) = 0 = \delta_{G(x)}(v, w)$. If p has length 1, $w_{I^x}(p)$ equals $\delta_{G(x_i)}(v, w)$ for some $i \in \{1, 2\}$ or equals the weight of this edge in G. In all cases we have a contradiction. So suppose p has length at least 2. By choice of p each edge $e = (u_1, u_2) \in p$ satisfies $w_{I^x}(e) \geq \delta_{G(x)}(u_1, u_2)$. By definition of distance we get that $\delta_{G(x)}(v, w) \leq \sum\{\delta_{G(x)}(u_1, u_2) : (u_1, u_2) \in p\} \leq \delta_{I^x}(v, w)$. This is a contradiction. ■

For the purpose of handling the case that $v, w \in \mathcal{B}(x)$ we define the auxiliary graph $H^x = (V', E')$, where $V' = \mathcal{B}(x) \cup \mathcal{S}(x)$, and $E' = \mathcal{B}(x) \times \mathcal{S}(x) \cup \mathcal{S}(x) \times \mathcal{B}(x) \cup \mathcal{S}(x) \times \mathcal{S}(x)$. Each edge $e = (v, w) \in E'$ has weight $w_{H^x}(e)$, where

$$w_{H^x}(e) \;=\; \begin{cases} \delta_{I^x}(v, w) & \text{if } e \in \mathcal{S}(x) \times \mathcal{S}(x) \\ \delta_{G(x_1)}(v, w) & \text{if } e \notin \mathcal{S}(x) \times \mathcal{S}(x), \text{ and } v, w \in \mathcal{B}(x_1) \\ \delta_{G(x_2)}(v, w) & \text{if } e \notin \mathcal{S}(x) \times \mathcal{S}(x), \text{ and } v, w \in \mathcal{B}(x_2) \\ \infty & \text{otherwise} \end{cases}$$

Observe that w_{H^x} is correctly defined, since for any internal node x with children x_1 and x_2 it holds that $\mathcal{B}(x_1) \cap \mathcal{B}(x_2) \subseteq S(x)$. In the following, we define $\delta_{G(x_i)}(v, w) = \infty$ if v or w is not in $G(x_i)$.

Lemma 3.5 *Let x be an internal node with children x_1 and x_2. Given arbitrary vertices v and w in $\mathcal{B}(x)$, the following two conditions hold:*

1. $\min\{\delta_{H^x}(v, w), \delta_{G(x_1)}(v, w), \delta_{G(x_2)}(v, w)\} = \delta_{G(x)}(v, w)$
2. *if w is reachable from v in H^x, then there exists a minimum-weight path from v to w in H^x of length at most 3.*

Proof: Let x be an internal node with children x_1 and x_2. Consider arbitrary vertices v and w in $\mathcal{B}(x)$. Let us start with the first condition. In case $w \notin \mathit{Reach}_{G(x)}(v)$, we have

$\delta_{G(x)}(v,w) = \delta_{G(x_1)}(v,w) = \delta_{G(x_2)}(v,w) = \infty$. Then $\delta_{H^x}(v,w)$ cannot be finite, because otherwise there would be a path from v to w in $G(x)$. Hence, in this case, condition 1 is satisfied.

Suppose $w \in \mathit{Reach}_{G(x)}(v)$. Let p be a minimum-weight path from v to w in $G(x)$. We consider the following two cases.

If $p \cap \mathcal{S}(x) = \emptyset$, then for precisely one $i \in \{1,2\}$, p lies completely in $G(x_i)$, and $v, w \in \mathcal{B}(x_i)$. In this case $w_{H^x}((v,w)) = \delta_{G(x_i)}(v,w)$. Therefore $\delta_{H^x}(v,w) \leq \delta_{G(x_i)}(v,w) = \delta_{G(x)}(v,w)$.

Suppose that $p \cap \mathcal{S}(x) \neq \emptyset$. Let v_1 and v_2 be the first and the last vertex on p in $\mathcal{S}(x)$, respectively. The path p is the concatenation of paths p_1, p_2, and p_3, where p_1 is a path from v to v_1, p_2 is a path from v_1 to v_2, and p_3 is a path from v_2 to w. Now we construct a path in H^x with the same weight as p.

If $p_1 \neq \lambda_v$, then $v \notin \mathcal{S}(x)$. We have that p_1 lies entirely in $G(x_i)$ and $v, v_1 \in \mathcal{B}(x_i)$ for exactly one $i \in \{1,2\}$. Therefore, $w_{H^x}((v,v_1)) = \delta_{G(x_i)}(v,v_1) = w(p_1)$. If $p_2 \neq \lambda_{v_1}$ then $v_1 \neq v_2$. Since $v_1, v_2 \in \mathcal{S}(x)$, we have $w_{H^x}((v_1,v_2)) = \delta_{I^x}(v_1,v_2) = \delta_{G(x)}(v_1,v_2) = w(p_2)$. If $p_3 \neq \lambda_{v_2}$, then $w \notin \mathcal{S}(x)$. For exactly one $j \in \{1,2\}$ it holds that p_3 lies in $G(x_j)$, and $v_2, w \in \mathcal{B}(x_j)$. Therefore, $w_{H^x}((v_2,w)) = \delta_{G(x_j)}(v_2,w) = w(p_3)$. We conclude that there exists a path in H^x from v to w with the same weight as p. Hence $\delta_{H^x}(v,w) \leq \delta_{G(x)}(v,w)$.

The proof that $\delta_{H^x}(v,w) \geq \delta_{G(x)}(v,w)$ is similar to the proof of the corresponding result on $\delta_{I^x}(v,w)$ in the previous lemma. It is left to the reader as an exercise.

In order to establish the second condition, suppose $w \in \mathit{Reach}_{H^x}(v)$. Let p be a minimum-weight path in H^x from v to w that consists of more than three edges. Let v_1 and v_2 be the second and the one but last vertex in p, respectively, then $v_1, v_2 \in \mathcal{S}(x)$. Therefore, the edge $e = (v_1,v_2)$ in H^x has weight $\delta_{I^x}(v_1,v_2) = \delta_{G(x)}(v_1,v_2)$. Condition 1 implies that $\delta_{H^x}(v_1,v_2) \geq \delta_{G(x)}(v_1,v_2)$, therefore replacing in p the subpath from v_1 to v_2 with edge e results in a minimum-weight path from v to w in H^x with length at most 3. ■

Lemma 3.4 and lemma 3.5 lead to the following algorithm for computing the weights on edges from E_x for each internal node $x \in T_G$.

```
Procedure $H$-distance($v$)
1. Perform parallel relaxation on the edges $\{v\} \times \mathcal{S}(x)$ in $H^x$.
2. Perform parallel relaxation on the edges $\mathcal{S}(x) \times \mathcal{S}(x)$ in $H^x$.
3. Perform parallel relaxation on the edges $\mathcal{S}(x) \times \mathcal{B}(x)$ in $H^x$.
```

```
Procedure ComputeWeights($x$)
# Let $x_1$ and $x_2$ be the children of $x$.
1. Construct the graph $I^x$.
2. Perform all-pairs shortest-paths computation on $I^x$,
   for every edge $e = (v,w) \in \mathcal{S}(x) \times \mathcal{S}(x)$ output $\delta_{I^x}(v,w)$.
3. Construct the graph $H^x$.
4. For every vertex $v \in \mathcal{B}(x)$ call $H$-distance($v$).
5. For every edge $e = (v,w) \in \mathcal{B}(x) \times \mathcal{B}(x) - \mathcal{S}(x) \times \mathcal{S}(x)$
```

output $\min\{\delta_{H^x}(v,w), \delta_{G(x_1)}(v,w), \delta_{G(x_2)}(v,w)\}$.

Let us analyze the complexity of the procedure ComputeWeights for the EREW PRAM, in case graphs are represented with adjacency matrices. Line 1 can be done with $O(|\mathcal{S}(x)|^2)$ work, in $O(1)$ time. Using an algorithm of Han et al. [16], line 2 can be performed with $O(|\mathcal{S}(x)|^3)$ work, in $O(\log^2 n)$ time. Line 3 takes $O(|\mathcal{B}(x)||\mathcal{S}(x)| + |\mathcal{S}(x)|^2)$ work, in $O(1)$ time. Line 4 can be performed with $O(|\mathcal{B}(x)|^2|\mathcal{S}(x)| + |\mathcal{B}(x)||\mathcal{S}(x)|^2)$ work in $O(\log(|\mathcal{B}(x)| + |\mathcal{S}(x)|))$ time. Line 5 can be done with $O(|\mathcal{B}(x)|^2)$ work in $O(1)$ time. We conclude that the algorithm Computeweights requires $O(|\mathcal{S}(x)|^3 + |\mathcal{B}(x)|^2|\mathcal{S}(x)|)$ work, and runs in $O(\log^2 n)$ time.

3.4 Extension to Flow Augmentation

The algorithm Bellman-Ford$^+$ can be adapted to function as a pathfinding routine in the shortest augmenting path maxflow algorithm. Suppose we want to compute a maximum flow in a network (G, c, s, t). The shortest augmenting path maxflow algorithm proceeds as follows [10].

```
Procedure Shortest-Augmenting-Path $(G, c, s, t)$
1.  Initialize $f$ as zero-flow.
2.  Construct residual network $G(f)$.
3.  while there exists a path from $s$ to $t$ in $G(f)$ do
4.      Find a minimum-length path $p$ from $s$ to $t$ in $G(f)$.
5.      Compute the minimum residual capacity $r_{\min}$ of edges in $p$.
6.      For each edge $(v,w) \in p$ do
7.          if $(v,w) \in E$ then
8.              $f(v,w) := f(v,w) + r_{\min}$
9.          else $f(w,v) := f(w,v) - r_{\min}$
10.     Reconstruct residual network $G(f)$.
```

Let us give a brief overview of how the Bellman-Ford$^+$ algorithm is incorporated in the shortest augmenting path maxflow algorithm. Assume we have a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G. Assign to each edge in $G(f)$ the weight $w(e) = 1$. Since $(T_G, \mathcal{S}, \mathcal{V})$ is also a separator decomposition of the residual graph $G(f)$, we augment $G(f)$ with a set of edges $E(f)^+$, as described in the previous section. Denote the resulting graph by $G(f)^+$. Observe that for each edge $e = (v,w)$ in $G(f)^+$ there exists a path from v to w in $G(f)$ of weight $w(e)$. In this section we will show how with a little extra administration we can associate with each edge in $E(f)^+$ a path in $G(f)$. The algorithm ComputeWeights is modified for this purpose. Once each edge in $E(f)^+$ has been assigned a path in $G(f)$, we attach to each edge in $E(f)^+$ a capacity equal to the minimum capacity found among edges on its associated path. As a next step, the Bellman-Ford$^+$ algorithm is extended to keep information in order to traverse paths efficiently. Given a shortest path p from s to t in $G(f)^+$, the data structure is used to compute the minimum residual capacity $r_{\min}(p)$

of p. Once $r_{\min}(p)$ has been calculated, p is to be traversed, and along each encountered edge the flow f should be augmented with $r_{\min}(p)$ flow. In order to do this the procedure Augment recursively augments flow along edges in $G(f)^+$. At the base of the recursion flow is augmented along associated paths in $G(f)$. During the course of algorithm Shortest-Augmenting-Path the residual graph changes, and hence distances change. Therefore, a new set of edges $E(f)^+$ is computed after each augmentation.

3.4.1 Associated Paths

Let us show how for each edge $e = (v, w)$ in $E(f)^+$, we can recursively associate with e a path from v to w in $G(f)$ of weight $w(e)$. First consider the case $e = (v, w) \in E(f)_x$, where x is an internal node with children x_1 and x_2. If $v, w \in \mathcal{S}(x)$, then there exists a path from v to w in I^x of weight $w(e)$. In order to associate unambiguously with e a single path in I^x, we keep all-pairs shortest-path information between vertices in I^x in a predecessor matrix Π_{I^x}. (See Cormen et al. [10] for details). Π_{I^x} can be computed in line 2 of procedure ComputeWeights using the Floyd-Warshall algorithm. Note that this does not incur any increase in asymptotic time and work bounds of the algorithm ComputeWeights. Now consider an arbitrary finite-weight edge $e' = (v', w')$ in I^x. We know that $(v', w') \in E(f)$ with weight $w_{I^x}(e')$, or $(v', w') \in E(f)_{x_1}$ with weight $w_{I^x}(e')$, or $(v', w') \in E(f)_{x_2}$ with weight $w_{I^x}(e')$. Therefore, for each edge e' in I^x we have a label sub(e') pointing to one of these three possible edges.

Let us now consider the case that $v, w \in \mathcal{B}(x)$. We know a path from v to w exists in H^x and has weight $w(e)$. Therefore, for each $v' \in \mathcal{B}(x)$ we keep a predecessor vector $\pi^{v'}_{H^x}$ containing single-source shortest-path information in H^x with respect to source v'. The procedure H-distance can be modified to compute these vectors, without change in the asymptotic behavior of its running time. Now consider an arbitrary finite-weight edge $e' = (v', w')$ from H^x. If $v', w' \in \mathcal{S}(x)$, then $(v', w') \in E(f)_x$ with weight $w_{H^x}(e')$. If v' and w' not both in $\mathcal{S}(x)$, then $(v', w') \in E(f)_{x_1}$ with weight $w_{H^x}(e')$, or $(v', w') \in E(f)_{x_2}$ with weight $w_{H^x}(e')$. We have for each edge e' in H^x a label sub(e') pointing to one of these three possible edges.

Consider the case that $e \in E(f)_x$ for a leaf x in T_G. Then a path from v to w exists in $G(f)(x)$ with weight $w(e)$. Therefore, with leaf x we have a predecessor matrix $\Pi_{G(f)(x)}$, containing all-pairs shortest-path information of the graph $G(f)(x)$. Recall from the previous section, that the Floyd-Warshall was called to compute the weights on edges in $E(f)_x$. We can adapt this call to compute $\Pi_{G(f)(x)}$ without any increase in asymptotic costs.

For clarity let us explicitly give the definition of the sub-labels, and the modified pseudocode of algorithm ComputeWeights. Note that sub-labels are only defined for finite-weight edges. No problem occurs here since in the algorithms that follow only finite-weight edges are processed. In the following an edge $(v, w) \in E(f)_x$ is denoted by $(v, w)_x$. For each internal node $x \in T_G$, and each finite-weight edge $e = (v, w)$ in I^x define:

$$\text{sub}(e) = \begin{cases} \text{sub}'(e) & \text{if } v, w \in \mathcal{B}(x_1) \cap \mathcal{B}(x_2) \\ (v, w)_{x_1} & \text{if } v, w \in \mathcal{B}(x_1) - \mathcal{B}(x_2) \\ (v, w)_{x_2} & \text{if } v, w \in \mathcal{B}(x_2) - \mathcal{B}(x_1) \\ (v, w) & \text{otherwise} \end{cases}$$

where

$$\text{sub}'(e) = \begin{cases} (v, w)_{x_1} & \text{if } w_{I^x}(e) = \delta_{G(f)(x_1)}(v, w) \\ (v, w)_{x_2} & \text{otherwise} \end{cases}$$

For finite-weight edge $e = (v, w) \in H^x$ define:

$$\text{sub}(e) = \begin{cases} (v, w)_x & \text{if } e \in \mathcal{S}(x) \times \mathcal{S}(x) \\ (v, w)_{x_1} & \text{if } e \notin \mathcal{S}(x) \times \mathcal{S}(x), \text{ and } v, w \in \mathcal{B}(x_1) \\ (v, w)_{x_2} & \text{if } e \notin \mathcal{S}(x) \times \mathcal{S}(x), \text{ and } v, w \in \mathcal{B}(x_2) \end{cases}$$

These definitions form a data structure that attaches to each edge $e = (v, w)$ in $E(f)^+$ a path from v to w in $G(f)$ of weight $w(e)$. In later sections we will use this datastructure to compute efficiently minimum capacities of paths, and to implement flow augmentation.

Let us now specify the procedure ComputeWeights to compute weights, sub-labels, the predecessor matrix Π_{I^x}, and predecessor vectors $\pi^{v'}_{H^x}$ at an internal node x. Assume the relaxation in the procedure H-distance is extended to compute predecessor vectors.

```
Procedure ComputeWeights($x$)
# Let $x_1$ and $x_2$ be the children of $x$.
1. Construct the graph $I^x$, including sub-labels.
2. Perform an all-pairs shortest-paths computation on $I^x$
   to obtain distances $\delta_{I^x}$ and the predecessor matrix $\Pi_{I^x}$.
   For every edge $e = (v, w) \in \mathcal{S}(x) \times \mathcal{S}(x)$ output $\delta_{I^x}(v, w)$.
3. Construct the graph $H^x$, including sub-labels.
4. For every vertex $v \in \mathcal{B}(x)$ call $H$-distance($v$) to obtain
   distances in $H^x$ and a predecessor vector $\pi^v_{H^x}$ with respect to source $v$.
5. For every edge $e = (v, w) \in \mathcal{B}(x) \times \mathcal{B}(x) - \mathcal{S}(x) \times \mathcal{S}(x)$
   output $\min\{\delta_{H^x}(v, w), \delta_{G(f)(x_1)}(v, w), \delta_{G(f)(x_2)}(v, w)\}$.
```

3.4.2 Capacitating edges from $E(f)^+$

The capacity of an edge from $E(f)^+$ is set equal to the minimum capacity of edges on its associated path. For a leaf x and edge $e = (v, w) \in E(f)_x$ this capacity can be computed by traversing the path from v to w backwards using predecessor matrix $\Pi_{G(f)(x)}$. For an internal node x with children x_1 and x_2 we have the following procedure. Assume that the capacities of edges in $E(f)_{x_1}$ and $E(f)_{x_2}$ have already been computed.

```
Procedure SetCap($x$)
1. for all edges $e = (v, w) \in E_x \cap (\mathcal{S}(x) \times \mathcal{S}(x))$ pardo
2.     if $w(e)$ is finite then
3.         Traverse the path $p$ from $v$ to $w$ in $I^x$ using $\Pi_{I^x}$
           to calculate $r(p) = \min\{c(\text{sub}(e'))|e' \in p\}$.
4.         $c(e) := r(p)$.
5. for all edges $e = (v, w) \in E_x \cap (\mathcal{B}(x) \times \mathcal{B}(x) - \mathcal{S}(x) \times \mathcal{S}(x))$ pardo
6.     if $w(e)$ is finite then
7.         Traverse the path $p$ from $v$ to $w$ in $H^x$ using $\pi^v_{H^x}$
           to calculate $r(p) = \min\{c(\text{sub}(e'))|e' \in p\}$.
8.         $c(e) := r(p)$
```

Because the capacities of edges in $E(f)_{x_1}$ and $E(f)_{x_2}$ have already been computed, each value $c(\text{sub}(e'))$ can be determined in $O(1)$ time. Each path in line 3 has length $O(|\mathcal{S}(x)|)$, hence lines 1 to 4 run in $O(|\mathcal{S}(x)|)$ time and $O(|\mathcal{S}(x)|^3)$ work. By lemma 3.5 each path p in line 7 has length at most 3. Hence lines 5 to 8 run in $O(1)$ time and $O(|\mathcal{B}(x)|^2)$ work. Overall, algorithm SetCap runs in $O(|\mathcal{S}(x)|)$ time, and performs $O(|\mathcal{S}(x)|^3 + |\mathcal{B}(x)|^2)$ work on a CREW PRAM.
The overall process of capacitating edges from $E(f)^+$ comes down to a straightforward bottom-up computation of capacities using the procedure SetCap. In section 3.5 the overall complexity of calculating capacities will be evaluated for a specific class of graphs.

3.4.3 Shortest Augmenting Path Maximum Flow Computation

In order to use the Bellman-Ford$^+$ algorithm as a pathfinding routine, its output must not only consist of distance labels, but also generate minimum-weight paths. Therefore, for each vertex $v \in V$ we maintain a label edge(v), which points to the last edge on a minimum-weight path from the source to v. Following a number of edge-labels, one traverses a minimum-weight path backward. The Bellman-Ford$^+$ algorithm will correctly compute these labels, if we modify its relaxation procedure into:

```
Procedure Relax($e = (v, w)$)
1. if $d[w] > d[v] + w(e)$ then
2.     $d[w] := d[v] + w(e)$
3.     edge($w$) := $e$
```

Suppose we have applied Bellman-Ford$^+$ to $G(f)^+$. If $d[t] = \infty$, then f is a maximum flow. Suppose $d[t] \neq \infty$. We can use the edge-labels to traverse backwards over a shortest path p from s to t in $G(f)^+$, and compute the minimum-capacity $r_{\min}(p)$ of edges on p. From the way we have set capacities on edges in $E(f)^+$, we conclude there exists a path p' from s to t in $G(f)$ of capacity $r_{\min}(p)$. The following procedure Augment recursively augments flow along an $E(f)^+$-edge, actually resulting in an augmentation along the associated path of this edge. We augment p with $r_{\min}(p)$ flow by calling Augment(e,$r_{\min}(p)$) for each edge $e \in p$. For each edge in $G(f)^+$ a label type(e) is kept in order to distinguish between different kinds of edges from $G(f)^+$. For edges $e \in E(f)$, type(e) = normal. For edges $e \in E(f)_x$, type(e) = added x.

```
Procedure Augment($e = (v, w)$, val)
1. if type($e$) = normal then
2.     if $(v, w) \in E$ then
3.         $f(v, w) := f(v, w) + \text{val}$
4.     else $f(w, v) := f(w, v) - \text{val}$
5. else# Let $x \in T_G$ be such that type($e$) = added $x$
6.     if $v, w \in \mathcal{S}(x)$ then
7.         Use $\Pi_{I^x}$ to traverse the path from $v$ to $w$ in $I^x$.
           For each encountered edge $e'$ call Augment(sub($e'$), val).
8.     else Use $\pi^v_{H^x}$ to traverse the path from $v$ to $w$ in $H^x$.
           For each encountered edge $e'$ call Augment(sub($e'$), val).
```

Observe that in general distances change in the residual network due to augmentation. In other words after each augmentation we have to recalulate the set $E(f)^+$, together with the administration that keeps track of associated paths. Let us specify the maxflow algorithm in pseudocode.

```
Procedure Shortest-Augmenting-Path$^+(G, c, s, t)$
1. Initialize $f$ to zero-flow.
2. repeat
3.     Construct residual network $(G(f), r, s, t)$.
4.     Compute $E(f)^+$ for $G(f)$, including the associated-paths
       administration.
5.     Compute capacities for $E(f)^+$-edges.
6.     perform Bellman-Ford$^+$ in $G(f)^+$ w.r.t. source $s$.
7.     if $d[t] \neq \infty$ then
8.         Traverse the shortest path $p$ from $s$ to $t$ to find
           the minimum-capacity $r_{\min}$ of edges on $p$.
9.         Traverse the shortest path $p$ again, and call for each
           encountered edge $e$ the procedure Augment($e$, $r_{\min}$).
10. until $d[t] = \infty$
```

3.5 Graphs with n^μ-separator decompositions

Let us analyze the algorithms of this section in case there exist constants $0 < \mu < 1$ and $\frac{1}{2} < \alpha < 1$ such that the separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G satisfies the following five conditions:

1. For all $x \in T_G$, $|\mathcal{S}(x)| = O(|\mathcal{V}(x)|^\mu)$.
2. For all $x, x' \in T_G$ such that x' is a child of x, $|\mathcal{V}(x')| \leq \alpha|\mathcal{V}(x)|$.

3. For leaves $x \in T_G$, $|\mathcal{V}(x)| = O(1)$.
4. If $|\mathcal{V}(x)| = \emptyset$, then x is a leaf.
5. For any internal node x with children x_1 and x_2, $\mathcal{S}(x) \subseteq \mathcal{B}(x_1) \cup \mathcal{B}(x_2)$.

Under these restrictions the depth d_G of the tree T_G is at most $\log_{1/\alpha} n$, and $l = O(1)$. For $0 \leq i \leq \lfloor \log_{1/(1-\alpha)} n \rfloor$, denote by $T_i \subset T_G$ the set of all nodes $x \in T_G$ such that $(1-\alpha)^{i+1} n < |\mathcal{V}(x)| \leq (1-\alpha)^i n$. We have that each T_i is a forest consisting of subtrees of height at most $\lceil \log_\alpha (1-\alpha) \rceil$. Consider a node x with children x_1 and x_2, then $x \in T_k$ for some $0 \leq k \leq \lfloor \log_{1/(1-\alpha)} n \rfloor$. Conditions 2 and 5 imply that $|\mathcal{V}(x_1)| \leq \alpha |\mathcal{V}(x)|$, $|\mathcal{V}(x_2)| \leq \alpha |\mathcal{V}(x)|$, and $|\mathcal{V}(x_1)| + |\mathcal{V}(x_2)| \geq |\mathcal{V}(x)|$. Therefore for $i = 1, 2$, $|\mathcal{V}(x_i)| \geq (1-\alpha)|\mathcal{V}(x)| > (1-\alpha)^{k+2} n$. We conclude that x_1 and x_2 are in T_k or T_{k+1}. Denote $s_k = \sum_{x \in T_k} |\mathcal{S}(x)|$. We have the following proposition.

Proposition 3.1 $s_k = O(n^\mu (1-\alpha)^{k(\mu-1)})$.

Proof: Denote by $P_k \subset T_k$ the set of roots of all trees in T_k. Let $n_k = \sum_{x \in P_k} |\mathcal{V}(x)|$. Each tree in T_k has constant height, therefore $\sum_{x \in T_k} |\mathcal{V}(x)| = O(n_k)$. Since for each node $x \in T_k$, $|\mathcal{V}(x)| > (1-\alpha)^{k+1} n$, there are $O(\frac{n_k}{(1-\alpha)^{k+1} n}) = O(\frac{n_k}{(1-\alpha)^k n})$ nodes in T_k. Hence,

$$s_k = O(\frac{n_k}{(1-\alpha)^k n} (1-\alpha)^{k\mu} n^\mu)$$

Observe that $n_0 = n$, and $n_{k+1} \leq n_k + s_k$. Therefore we have that

$$n_{k+1} \leq n_k (1 + O(((1-\alpha)^k n)^{\mu-1}))$$

Hence,

$$n_{k+1} \leq n \prod_{i=0}^{k} (1 + O((\frac{1}{n} \cdot (\frac{1}{1-\alpha})^i)^{1-\mu})) \leq n \prod_{i=0}^{k} (1 + O((\frac{1}{n} \cdot (\frac{1}{1-\alpha})^{k-i})^{1-\mu}))$$

With $k \leq \log_{(1-\alpha)^{-1}} n$ we have $(1-\alpha)^{-k} \leq n$. Hence

$$n_{k+1} \leq n \prod_{i=0}^{k} (1 + O((1-\alpha)^{i(1-\mu)})) \leq n \prod_{i \geq 0} (1 + O((1-\alpha)^{i(1-\mu)}))$$

Taking logarithms we get

$$\log n_{k+1} \leq \log n + \sum_{i \geq 0} O((1-\alpha)^{i(1-\mu)}) \leq \log n + O(\sum_{i \geq 0} (1-\alpha)^{i(1-\mu)}) \leq \log n + O(1)$$

Thus $n_{k+1} = O(n)$. We conclude that $s_k = O(n^\mu (1-\alpha)^{k(\mu-1)})$. ■

Proposition 3.1 can be used to bound $\sum_{x \in T_G} |\mathcal{S}(x)|^\omega$ for constant $\omega \geq 1$.

Lemma 3.6 *For $\omega \geq 1$ we have that*

$$\sum_{x \in T_G} |\mathcal{S}(x)|^\omega = \begin{cases} O(n^{\omega\mu}) & \text{if } \omega\mu > 1 \\ O(n \log n) & \text{if } \omega\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

Proof: T_k contains at least $\frac{s_k}{(1-\alpha)^{k\mu} n^\mu}$ nodes because $|\mathcal{S}(x)| \leq (1-\alpha)^{k\mu} n^\mu$ for each $x \in T_k$. Hence for $\omega \geq 1$

$$\sum_{x \in T_k} |\mathcal{S}(x)|^\omega \leq \frac{s_k}{(1-\alpha)^{k\mu} n^\mu} ((1-\alpha)^{k\mu} n^\mu)^\omega.$$

Using proposition 3.1, we get

$$\sum_{x \in T_k} |\mathcal{S}(x)|^\omega \leq O(n^{\omega\mu} (1-\alpha)^{k(\omega\mu - 1)})$$

We use this to get a bound on $\sum_{x \in T_G} |\mathcal{S}(x)|^\omega$:

$$\sum_{x \in T_G} |\mathcal{S}(x)|^\omega = \sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} \sum_{x \in T_k} |\mathcal{S}(x)|^\omega = n^{\omega\mu} \cdot O\left(\sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} (1-\alpha)^{k(\omega\mu-1)} \right).$$

From this the lemma follows easily for the cases $\omega\mu = 1$ and $\omega\mu > 1$. In case $\omega\mu < 1$ we have

$$\sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} (1-\alpha)^{k(\omega\mu-1)} = \frac{n^{1-\omega\mu}(1-\alpha)^{\omega\mu-1} - 1}{(1-\alpha)^{\omega\mu-1} - 1}$$

and the lemma follows. ■

Lemma 3.7 *For $\omega \geq 1$, and $0 \leq k \leq \lfloor \log_{1/(1-\alpha)} n \rfloor$, we have that*

$$\sum_{x \in T_k} |\mathcal{B}(x)|^\omega = \begin{cases} O(n^{\omega\mu}) & \text{if } \omega\mu > 1 \\ O(kn) & \text{if } \omega\mu = 1 \\ O(n^{\omega\mu}(1-\alpha)^{k(\omega\mu-1)}) & \text{otherwise} \end{cases}$$

Proof: For a node $x \in T_G$ the set $\mathcal{B}(x)$ consists of separator vertices from ancestors of x. Furthermore, each occurence of a vertex v in $\mathcal{S}(x)$ contributes to the boundaries of at most two descendants at each level. For $0 \leq i \leq \lfloor \log_{1/(1-\alpha)} n \rfloor$, we have that $\sum_{x \in T_i} |\mathcal{V}(x)| = O(n_i) = O(n)$. Each node in T_i contains at least $(1-\alpha)^{i+1} n$ vertices, so there are $O((1-\alpha)^{-i-1}) = O((1-\alpha)^{-i})$ nodes in T_i. Hence,

$$\sum_{x \in T_k} |\mathcal{B}(x)|^\omega =$$

$$\sum_{i\leq k} O((1-\alpha)^{-i}(1-\alpha)^{\omega\mu i}n^{\omega\mu}) =$$
$$O\left(n^{\omega\mu}\sum_{i\leq k}(1-\alpha)^{i(\omega\mu-1)}\right) =$$
$$\begin{cases} O(n^{\omega\mu}) & \text{if } \omega\mu > 1 \\ O(kn) & \text{if } \omega\mu = 1 \\ O(n^{\omega\mu}(1-\alpha)^{k(\omega\mu-1)}) & \text{otherwise} \end{cases}$$

This completes the proof. ■

We can use the previous two lemmas to bound the amount of work needed to compute the set E^+. Since for each leaf x, $|\mathcal{V}(x)| = O(1)$, processing the leaves takes $O(n)$ work. Recall that the algorithm ComputeWeight performs $O(|\mathcal{S}(x)|^3 + |\mathcal{B}(x)|^2|\mathcal{S}(x)|)$ work at each internal node x. We have that

$$\sum_{x\in T_G} O(|\mathcal{B}(x)|^2|\mathcal{S}(x)|) \;=\; \sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n\rfloor} O\left(((1-\alpha)^k n)^\mu \sum_{x\in T_k}|\mathcal{B}(x)|^2\right)$$

From lemma 3.7 we get that for $0 \leq k \leq \lfloor \log_{1/(1-\alpha)} n\rfloor$,

$$O\left(((1-\alpha)^k n)^\mu \sum_{x\in T_k}|\mathcal{B}(x)|^2\right) \;=\; \begin{cases} O(n^{3\mu}(1-\alpha)^{k\mu}) & \text{if } 2\mu > 1 \\ O(kn^{1.5}(1-\alpha)^{0.5k}) & \text{if } 2\mu = 1 \\ O(n^{3\mu}(1-\alpha)^{k(3\mu-1)}) & \text{otherwise} \end{cases}$$

Using lemma 3.6 to bound $\sum_{x\in T_G} O(|\mathcal{S}(x)|^3|)$, we conclude that the work involved with computing E^+ is:

$$\begin{cases} O(n^{3\mu}) & \text{if } 3\mu > 1 \\ O(n\log n) & \text{if } 3\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

Since $d_G = O(\log n)$, this can be done in $O(\log^3 n)$ time.

Let us now consider the Bellman-Ford$^+$ algorithm. The running time of this algorithm is $O((l+d_G)\log n) = O(\log^2 n)$ on a EREW PRAM. The algorithm performs $O(l|E|+|E^+|) = O(|E| + |E^+|)$ work. We can evaluate this further as follows.

The set E^+ was protected from containing multiple edges by including only the edge of minimum-weight in case a multiple occurrence would arise. As a consequence of this

$$|E^+| \;\leq\; \sum_{x\in T_G} |\mathcal{S}(x)|^2 + |F|,$$

where F is the set

$$\bigcup_{x\in T_G} \mathcal{B}(x)\times\mathcal{B}(x) - \bigcup_{x\in T_G} \mathcal{S}(x)\times\mathcal{S}(x)$$

Let us bound the size of F. Observe that if an edge $(v, w) \in F$, then there exists a node x with $v, w \in \mathcal{B}(x)$. Furthermore, there exist distinct nodes $x', x'' \in T_G$ with $v \in \mathcal{S}(x')$ and $w \in \mathcal{S}(x'')$, such that x' and x'' are either equal to x or an ancestor of x. Without loss of generality we assume that x' is an ancestor of x''. Observe we can also assume that v is not contained in any separator from a node that is on the path from x' to x''. Hence,

$$\begin{aligned} |F| &= O\left(\sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} s_k \left(\sum_{j=k}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} ((1-\alpha)^j n)^\mu\right)\right) \\ &= O\left(n^{2\mu} \sum_{k=0}^{\lfloor \log_{1/(1-\alpha)} n \rfloor} (1-\alpha)^{k(2\mu-1)}\right) \\ &= \begin{cases} O(n^{2\mu}) & \text{if } 2\mu > 1 \\ O(n \log n) & \text{if } 2\mu = 1 \\ O(n) & \text{otherwise} \end{cases} \end{aligned}$$

Using lemma 3.6 we conclude that

$$|E^+| = \begin{cases} O(n^{2\mu}) & \text{if } 2\mu > 1 \\ O(n \log n) & \text{if } 2\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

Let us now bound the size of E. Observe that each edge in E is in $\mathcal{S}(x) \times \mathcal{S}(x)$ for some x, or is in $\mathcal{V}(x') \times \mathcal{V}(x')$ for some leaf x'. We conclude that

$$|E| = \begin{cases} O(n^{2\mu}) & \text{if } 2\mu > 1 \\ O(n \log n) & \text{if } 2\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

From the above we conclude that the work performed by the Bellman-Ford$^+$ algorithm is

$$\begin{cases} O(n^{2\mu}) & \text{if } 2\mu > 1 \\ O(n \log n) & \text{if } 2\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

In the overal computation of distances, the work performed in computing the set E^+ is the main bottleneck. However, the $O(nm)$ work bound of the sequential Bellman-Ford algorithm evaluates for the considered class of graphs to:

$$\begin{cases} O(n^{2\mu+1}) & \text{if } 2\mu > 1 \\ O(n^2 \log n) & \text{if } 2\mu = 1 \\ O(n^2) & \text{otherwise} \end{cases}$$

We conclude that the work performed by the parallel algorithm is in all cases less than the work performed by its sequential counterpart.

We will now analyze the maximum flow algorithm of section 3.4. Recall our observation that the altered algorithm ComputeWeights has the same asymptotic behavior as the original. The previous analysis therefore also applies to the computation of $E(f)^+$. Since separator sizes are $O(n^\mu)$, computing the capacities for edges from $E(f)^+$ takes $O(n^\mu \log n)$ time. Furthermore, the work involved with this is $\sum_{x\in T_G} |\mathcal{S}(x)|^3 + |\mathcal{B}(x)|^2$. This falls within the work bound of computing $E(f)^+$. In line 8 a shortest path from s to t in $G(f)^+$ is traversed. The length of this path is $O(\log n)$, so this takes $O(\log n)$ sequential time. The augmentation performed in line 9 can be done in $O(n^\mu \log n)$ time with $\sum_{x\in T_G} O(1+|\mathcal{S}(x)|)$ work. We find that one iteration of the maxflow algorithm takes $O(n^\mu \log n)$ time and work

$$\begin{cases} O(n^{3\mu}) & \text{if } 3\mu > 1 \\ O(n\log n) & \text{if } 3\mu = 1 \\ O(n) & \text{otherwise} \end{cases}$$

The number of iterations of the shortest augmenting path maxflow algorithm is $O(nm)$[10]. The previous analyses together with our observations about the size of E imply a running time of

$$\begin{cases} O(n^{3\mu+1}\log n) & \text{if } 2\mu > 1 \\ O(n^2\sqrt{n}\log^2 n) & \text{if } 2\mu = 1 \\ O(n^{\mu+2}\log n) & \text{otherwise} \end{cases}$$

and work bound

$$\begin{cases} O(n^3) & \text{if } 0 < \mu < \frac{1}{3} \\ O(n^3\log n) & \text{if } \mu = \frac{1}{3} \\ O(n^{3\mu+2}) & \text{if } \frac{1}{3} < \mu < \frac{1}{2} \\ O(n^3\sqrt{n}\log n) & \text{if } \mu = \frac{1}{2} \\ O(n^{5\mu+1}) & \text{otherwise} \end{cases}$$

The sequential shortest augmenting path maxflow algorithm has time bound $O(nm^2)$, which is in this case equal to

$$\begin{cases} O(n^{4\mu+1}) & \text{if } 2\mu > 1 \\ O(n^3\log^2 n) & \text{if } 2\mu = 1 \\ O(n^3) & \text{otherwise} \end{cases}$$

In all cases the parallel version improves on this computation time. Observe that for $0 < \mu < \frac{1}{3}$ the work bound of the parallel algorithm is even equal to the work bound of the sequential version. For larger μ the parallel algorithm performs more work.

3.6 Digression

During the shortest augmenting path maxflow algorithm the distance $\delta_{G(f)}(s, v)$ does not decrease for vertices $v \in V$. As a result of this, one can prove that the total number of augmentations is bounded by $O(nm)$. For specific graph classes separator decompositions can assist in lowering this bound analytically. For the class of graphs considered in section 3.5 it was shown that m was bounded by $O(n^{2\mu} + n)$, if $2\mu \neq 1$, and $O(n \log n)$ otherwise.

In this section we introduce for paths in a graph the concept of order. We will prove that the minimum order of an acyclic s, t-path in the residual network does not decrease if flow is augmented over acyclic s, t-paths of minimum order. The next step might be to apply this in a separator decomposition embedded maxflow algorithm. However, it is left as an open problem whether any such an approach is at all beneficial.

Definition 3.4 *Given graph $G = (V, E)$ and subset $X \subseteq V$, the order $\xi_G^X(p)$ of a path p in G is the number of edges from p in the cut defined by X.*

If in above definition the graph G is directed, only edges from p with their origin in X and head not in X will contribute to the order of p. Therefore, the order denotes the number of times p exits from X. If the graph G is undirected the order also counts the number times the path p enters X. In this case the order indicates the number of times the path p crosses between X and $V \backslash X$. Let us now present the promised lemma concerning order of paths and flow augmentation.

Lemma 3.8 *Let f be a flow in network $N = (G = (V, E), c, s, t)$, and let a subset $X \subseteq V$ be given. If there does not exist an acyclic s, t-path p in $G(f)$ with non-zero residual capacity and of order $\xi_{G(f)}^X(p) \leq i$, for some $i \geq 0$, then there does not exist an acyclic s, t-path in $G(f_{aug})$ with non-zero residual capacity and of order $\xi_{G(f_{aug})}^X(p) \leq i$, where f_{aug} is a flow in N resulting from augmenting f with a positive amount of flow over some acyclic s, t-path in $G(f)$ of order $i + 1$.*

Proof: Let i be such that there exists an acyclic s, t-path p in $G(f)$ with non-zero residual capacity of order $i + 1$, but not of order less than $i + 1$. Let f_{aug} be the flow in N resulting from augmenting f with some positive amount of flow over p. In order to derive a contradiction assume there exists an acyclic s, t-path p' in $G(f_{aug})$ with non-zero capacity of order $\xi_{G(f_{aug})}^X \leq i$. Observe there must be at least one edge (v, w) in p' such that:

1. the edge (w, v) is in p, and
2. the residual capacity of edge (v, w) in $G(f)$ is zero.

For $k \geq 1$, let $(v_1, w_1), \ldots, (v_k, w_k)$ be the complete sequence of these kind of edges in the same order as they appear on p'. Define $w_0 = s$, and $v_{k+1} = t$. For $1 \leq i \leq k + 1$, let p'_i be the subpath of p' between vertices w_{i-1} and v_i. Furthermore, let $(w'_1, v'_1), \ldots, (w'_k, v'_k)$ be the complete sequence of mirror-edges of the above sequence as they appear on p. Define $v'_0 = s$, and $w'_{k+1} = t$. For $1 \leq i \leq k + 1$, let p_i be the subpath of p between vertices v'_{i-1} and w'_i. Since all subpaths p'_i are mutually edge-disjoint we have that

$$\sum_{j=1}^{k+1} \xi_{G(f)}^X(p'_j) \leq \xi_{G(f)}^X(p') \leq i$$

Similarly it holds that

$$\sum_{j=1}^{k+1} \xi^X_{G(f)}(p_j) \leq \xi^X_{G(f)}(p) = i + 1$$

Hence,

$$\sum_{j=1}^{k+1} \xi^X_{G(f)}(p'_j) + \xi^X_{G(f)}(p_j) \leq 2i + 1 \tag{1}$$

Observe that each subpath p_i either ends in t, or at a vertex where a subpath p'_j begins. Conversely, each subpath p'_i either ends in t, or at a vertex where a subpath p_j begins. This suggest one can walk from s to t by alternatingly walking subpaths of p and p'. As we will see this is indeed the case.

Consider the case one starts in s and first traverses the subpath p_1. This brings us further down both paths p and p'. Consecutively, one can traverse a subpath p'_{j_1} for some $1 \leq j_1 \leq k+1$. Trivially this brings us further down path p', but since p is acyclic this also brings us further down p. In case we are not in t we can traverse a subpath p_{j_2}, for some $1 \leq j_2 \leq k+1$. This brings us further down p, but also further down p' since p' is acyclic. Reasoning on like this we conclude there exists an s,t-path q_1 that is the concatenation of alternatingly subpaths from p and p' starting with p_1.

Similarly we can argue there exists an s,t-path q_2 that is the concatenation of alternatingly subpaths from p and p' starting with p'_1. Since in the above the traversal of subpaths always brings us further down both p' and p it can be concluded that each subpath from p and p' can be traversed in only one of q_1 and q_2. Therefore, $\xi^X_{G(x)}(q_1) + \xi^X_{G(x)}(q_2)$ is less or equal to the left-hand side of (1). Therefore the order of one of q_1 and q_2 must be less than $i+1$. Discarding cycles on this path, we have shown the existence of an acyclic s,t-path in $G(f)$ of order less than $i+1$. This is a contradiction. ■

In case we have a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of a graph G, a thing that might come to mind is to apply the lemma to the graph $G(f)$, and choosing the set X equal to $\mathcal{S}(r)$, where r is the root of T_G. Augmenting over acyclic paths the maximum order of an s,t-path is bounded by the size of X. An idea could therefore be to consecutively cancel the existence of paths of minimum order. Important in this is to find a good upper bound for the number of times one augments over the same order. In case the algorithm works with integeral values, this is at most equal to value of the cut defined by $\mathcal{S}(r)$. However, this does not seem to be helpful. Also an important problem one encounters with the above idea, is the embedding of the elimination procedure in the recursive separator decomposition framework. One thing that absolutely needs to be avoided, is that the eventual algorithm iterates a substantial number of times at an internal node, with each iteration involving a recursive computation. Algorithms with this kind of structure suffer from an asymptotic explosion in their running time. However, the above mentioned idea seems to lead in this kind of direction.

4 The Connection to treewidth

As we saw in previous sections, the tree shaped structure of a separator decomposition opens the way to "Divide and Conquer"-strategies in the design of graph algorithms. Working on a specific graph problem, it is upon the algorithm designer to fit the problem into the separator decomposition framework. In this activity, flexibility is provided in terms of the properties of separator decompositions. Using these properties as parameters, one can control the generality of graphs for which one is devising an algorithm.

Closely related to the separator decomposition framework is the treewidth approach. In treewidth, we also have a tree containing information about separators, namely the tree decomposition. Each tree decomposition has a width, which is roughly the size of the largest separator in the tree. The treewidth of a graph G is defined as the minimum width over all possible tree decompositions of G. Just as with separator decompositions, the treewidth approach opens up possibilities for applying "Divide and Conquer"-strategies in graph algorithms.

The above suggests a close relationship between tree decompositions and separator decompositions. In this section, we give a mathematical exposition of this relation. It is shown that an arbitrary width-k tree decomposition can be converted into a separator decomposition with separators, leaf vertex sets, and boundaries of size at most $3k + 3$ in $O(\log^2 n)$ time with $O(n \log n)$ work.

In order to transform a separator decomposition into a tree decomposition, it is not always sufficient to do the inverse of the above mentioned conversion. However, a slight modification of this inverse results in a correct converse transformation algorithm. Given a separator decomposition of depth $O(\log n)$ with attribute sets of constant size, the algorithm returns a width-$O(\log n)$ tree decomposition of G. The computation takes $O(\log n)$ time and $O(n \log^2 n)$ work.

4.1 Treewidth

The concept of treewidth plays an important role in the design of graph algorithms. Many classes of graphs are known that have uniform constant bounded treewidth. For such a class there exists a constant that bounds the treewidth of all graphs in the class. Many NP-complete problems become polynomially solvable when restricted to graphs classes that have uniform constant bounded treewidth. The following notions were introduced by Robertson and Seymour [20].

Definition 4.1 *Let $G = (V, E)$ be a graph. A tree decomposition for G is a pair $(T_G, \mathcal{X}_G)$, where $T_G = (V_T, E_T)$ is a tree, and $\mathcal{X}_G$ is a function $V_T \to \wp(V)$ satisfying the following conditions:*

1. *For every $(v, w) \in E$ there is an $x \in V_T$ such that $v, w \in \mathcal{X}_G(x)$, and*
2. *For $x, y, z \in V_T$, if y is on the path from x to z in T_G then $\mathcal{X}_G(x) \cap \mathcal{X}_G(z) \subseteq \mathcal{X}_G(y)$.*

The width of a tree decomposition $(T_G, \mathcal{X}_G)$ *is* $\max\{|\mathcal{X}_G(x)| - 1 | x \in V_T\}$*, and the treewidth of* G *is the minimum width over all its tree decompositions.*

The graph G in the definition above can either be directed or undirected. Just as separator decompositions, tree decompositions are independent of the direction of edges in the underlying graph. As analogy to fact 2.1 we have:

Fact 4.1 $\mathcal{T}$ *is a tree decomposition of directed graph* G *if and only if* $\mathcal{T}$ *is a tree decomposition of the undirected skeleton of* G*.*

As already was made explicit, many classes of graphs are known which have uniform constant bounded treewidth. To mention a few, we have trees, forests, Halin graphs, series-parallel graphs, and outerplanar graphs. For an overview see Bodlaender [4].

For the problem of computing a tree decomposition of a graph $\mathcal{NC}$-algorithms are known. Bodlaender and Hagerup [5] gave an $O(\log^2 n)$ time algorithm using $O(n)$ operations on the EREW PRAM for the case the graph is undirected. Given a graph of treewidth k, their algorithm constructs a width-k tree decomposition.

4.2 From Tree Decomposition to Separator Decomposition

Converting a tree decomposition into a separator decomposition is a three phase process. Separator decompositions consist of rooted binary trees, therefore in the first phase the tree decomposition is made into a rooted binary tree decomposition. Here an algorithm of Bodlaender is used, which takes a width-k tree decomposition and converts it into a rooted binary tree decomposition of logarithmic depth and of width $3k+2$ [3]. For constant $k \geq 0$, the algorithm runs in $O(\log n)$ time, and uses $O(n)$ processors. Recall our assumption that in a binary tree internal nodes have exactly two children. Bodlaender's algorithm may result in a binary tree where internal nodes have only one child. Therefore, after this algorithm has been applied, for all internal nodes with only one child, a child x is added with $\mathcal{X}_G(x) = \emptyset$. Let it be obvious that this does not affect the two tree decomposition properties.

As will become clear later, the $\mathcal{X}_G$-sets of a tree decomposition may contain superfluous vertices, prohibiting the tree decomposition from being directly converted into a separator decomposition. This will be handled in the second phase. By removing vertices from $\mathcal{X}_G$-sets, the tree decomposition is turned into what is called a clean tree decomposition. Given an $O(\log n)$ depth rooted binary tree decomposition of constant width the algorithm runs in $O(\log^2 n)$ time, performing $O(n \log n)$ work.

After the tree decomposition has been made binary and clean, the actual conversion into a separator decomposition takes place. The conversion is a straightforward method, which takes $O(\log n)$ time and $O(n \log n)$ work for tree decompositions of constant width. The overall process of converting a tree decomposition of constant width into a separator decomposition takes $O(\log^2 n)$ time and $O(n \log n)$ work. Given a width-k tree decomposition the resulting separator decomposition has separators, boundaries, and leaf vertex sets of size at most $3k + 3$, and is of logarithmic depth.

4.3 Clean Tree Decompositions

As was already mentioned, the $\mathcal{X}_G$-sets of a tree decomposition sometimes contain superfluous vertices, which makes direct transformation into a separator decomposition difficult. Therefore, tree decompositions will be skimmed down in a preprocessing phase. A tree decomposition that has been processed this way is called clean. The following definition specifies the desired minimality requirement exactly.

Definition 4.2 *Given a rooted tree decomposition $(T_G, \mathcal{X}_G)$ of graph G and node $x \in T_G$, a vertex $v \in \mathcal{X}_G(x)$ is called clean with respect to x if for all children y of x it holds that, if v is not incident to any vertex in $\bigcup_{y' \in T_G[y]} \mathcal{X}_G(y') \backslash \mathcal{X}_G(x)$, then $v \notin \bigcup_{y' \in T_G[y]} \mathcal{X}_G(y')$.*

In the following a tree decomposition $(T_G, \mathcal{X}_G)$ will be called clean, when for all $x \in T_G$, all vertices in $\mathcal{X}_G(x)$ are clean with respect to x. Note that for leaves $x \in T_G$, all vertices in $\mathcal{X}_G(x)$ are clean. The following lemma and its corollary indicate that individual vertices can be cleaned by removing their occurrences in the subtree that make them dirty.

Lemma 4.1 *If we have a rooted tree decomposition $(T_G, \mathcal{X}_G)$ of graph G, internal node $x \in T_G$, child y of x, and vertex $v \in \mathcal{X}_G(x)$ that is not incident to any vertex in $\bigcup_{y' \in T_G[y]} \mathcal{X}_G(y') \backslash \mathcal{X}_G(x)$, then after removing v from all $\mathcal{X}_G(y')$ for $y' \in T_G[y]$ that contain it, the resulting $(T_G, \mathcal{X}_G)$ is still a tree decomposition of G.*

Proof: Let $(v, w) \in E$ be given, and let $z \in T_G$ be a node in the original tree with $v, w \in \mathcal{X}_G(z)$. Now suppose that v was removed from this set, because for an ancestor x of z we had that $v \in \mathcal{X}_G(x)$, and v was not incident to any vertex in $\bigcup_{y' \in T_G[y]} \mathcal{X}_G(y') \backslash \mathcal{X}_G(x)$, for child y of x. Since $z \in T_G[y]$, we have that $w \in \mathcal{X}_G(x)$. Apparently, for each node z in the tree with $v, w \in \mathcal{X}_G(z)$, if v is removed from $\mathcal{X}_G(z)$ during cleaning, there is an ancestor z' of z with $v, w \in \mathcal{X}_G(z')$. Since vertices are never removed from the $\mathcal{X}_G$-set of the root, there must be a node x in T_G for which $v, w \in \mathcal{X}_G(x)$.

Consider nodes x', y', z' in T_G with y' on the path from x' to z'. Originally, the condition $\mathcal{X}_G(x') \cap \mathcal{X}_G(z') \subseteq \mathcal{X}_G(y')$ was satisfied. Consider the case that y' is an ancestor of both x' and z'. It is easy to see that the condition still holds after deleting a vertex v, since if it is deleted from $\mathcal{X}_G(y')$, it is also deleted from $\mathcal{X}_G(x')$ and $\mathcal{X}_G(z')$, if contained at all. Now consider the case that x' is an ancestor of y', and y' is an ancestor of z'. If a vertex is removed from $\mathcal{X}_G(y')$, it is also removed from $\mathcal{X}_G(z')$, if contained at all, so again the condition holds afterwards. The case z' is an ancestor of y', and y' is ancestor of x' is symmetric. ■

Corollary 4.1 *Given a rooted tree decomposition $(T_G, \mathcal{X}_G)$ of graph G, node $x \in T_G$, and a dirty vertex $v \in \mathcal{X}_G(x)$ with respect to x, there exists a rooted tree decomposition $(T_G, \mathcal{X}'_G)$ of graph G with:*

1. for all $y \in T_G$, $\mathcal{X}'_G(y) \subseteq \mathcal{X}_G(y)$.

2. $v \in \mathcal{X}'_G(x)$, and v is clean with respect to x.

Proof: Simply repeat the procedure of lemma 4.1 for all children y of x. ■

Knowing how we can clean individual vertices, the question is how this can be used in cleaning up a whole tree. One might worry that cleaning one vertex could make another dirty. The following lemma indicates that this is not the case, if the cleaning is done in a top-down approach, cleaning vertices at nodes one-by-one.

Proposition 4.1 *Given rooted tree decomposition $(T_G, \mathcal{X}_G)$ of graph G, node $x \in T_G$, and node $z \in T_G[x]$, if we clean a dirty vertex in $\mathcal{X}_G(z)$, then all clean vertices in $\mathcal{X}_G(x)$ stay clean.*

Proof: Let $v \in \mathcal{X}_G(z)$ be a dirty vertex about to be cleaned, and let w be a clean vertex in $\mathcal{X}_G(x)$. We have for all children y of x that $w \in \bigcup_{y' \in T_G[y]} \mathcal{X}_G(y')$ only if w is incident to a vertex $u \in \bigcup_{y' \in T_G[y]} \mathcal{X}_G(y') \backslash \mathcal{X}_G(x)$. Observe that w can only become dirty if $v = u$. First note that, if $z = x$ then $u \neq v$, so w cannot get dirty.

Secondly, in case z is a descendant of x, cleaning v will not remove v from $\mathcal{X}_G(z)$. So after cleaning still $u \in \bigcup_{y' \in T_G[y]} \mathcal{X}_G(y') \backslash \mathcal{X}_G(x)$, and thus w is still clean. ■

For clarity reasons we summarize the above mentioned ideas in a straightforward pseudocode cleaning algorithm. See table 1. In order to clean a tree decomposition $(T_G, \mathcal{X}_G)$ simply call the procedure CleanTree with the root of T_G as argument. Let us analyse the performance of the algorithm in case a rooted binary tree decomposition of depth $d_G = O(\log n)$ and constant width is given.

To start with the procedure Incident, since the width of the tree is constant, the check in line 1 takes $O(1)$ time. Let the height of node c be h, then the procedure takes $O(h)$ time and $O(2^h)$ work. Similarly, we can conclude that procedure Remove takes $O(h)$ time and $O(2^h)$ work, if called for a node with height h. Moving up in the procedure hierarchy, we find that also CleanNode takes $O(h)$ time and $O(2^h)$ work, if called for nodes of height h. The procedure CleanTree called for the root of T_G therefore has running time:

$$\sum_{i=0}^{d_G} O(i) = O(d_G^2) = O(\log^2 n)$$

The number of nodes with height h is $O(2^{d_G - h})$, The amount of work performed by CleanTree is:

$$\sum_{i=0}^{d_G} O(2^i)O(2^{d_G - i}) = \sum_{i=0}^{d_G} O(2^{d_G}) = \sum_{i=0}^{d_G} O(n) = O(n \log n)$$

Summarizing these result for future reference:

```
Procedure CleanTree ($x \in T_G$)
1. CleanNode ($x$)
2. for all children $c$ of $x$ pardo
3.     CleanTree($c$)

Procedure CleanNode ($x \in T_G$)
1. if internal($x$) then
2.     for all $v \in \mathcal{X}_G(x)$ do
3.         for all children $c$ of $x$ do
4.             if not Incident($v$, $x$, $c$) then
5.                 Remove ($v$, $c$)

Procedure Incident ($v \in V$; $x, c \in T_G$)
1. if $v$ incident to vertex in $\mathcal{X}_G(c) \backslash \mathcal{X}_G(x)$ then
2.     return TRUE
3. else if leaf($c$) then
4.         return FALSE
5.     else for all children $c'$ of $c$ pardo
6.         $r_{c'}$ := Incident($v$, $x$, $c'$)
7.     return $\bigvee_{c'} r_{c'}$

Procedure Remove($v \in V$, $x \in T_G$)
1. Remove $v$ from $\mathcal{X}_G(x)$.
2. if internal($x$) then
3.     for all children $c$ of $x$ pardo
4.         Remove($v$, $c$)
```

Table 1: The Cleaning Algorithm

Lemma 4.2 *Any rooted binary constant-width tree decomposition of depth $O(\log n)$ can be cleaned in $O(\log^2 n)$ time with $O(n \log n)$ work.*

4.4 The Conversion

In order to tranform a tree decomposition $(T_G, \mathcal{X}_G)$ into a separator decomposition we exploit the fact that $\mathcal{X}_G$-sets are separators. The following lemma shows how the conversion can be done.

Lemma 4.3 *Given a graph G with a rooted binary clean tree decomposition $(T_G, \mathcal{X}_G)$ of width $k \geq 0$, let $\mathcal{S} : T_G \to V$ and $\mathcal{V} : T_G \to V$ be defined as*

$$\begin{aligned} \mathcal{S}(x) &= \begin{cases} \mathcal{X}_G(x) & \text{if } x \text{ internal,} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{V}(x) &= \bigcup_{y \in T_G[x]} \mathcal{X}_G(y) \end{aligned}$$

then $(T_G, \mathcal{S}, \mathcal{V})$ is a separator decomposition of G. Furthermore, for all nodes $x \in T_G$ it holds that $|\mathcal{S}(x)|, |\mathcal{B}(x)| \leq k+1$, and if x is a leaf then $|\mathcal{V}(x)| \leq k+1$.

Proof: Let r be the root of T_G. The first condition of the separator decomposition definition states that $\mathcal{V}(r) = V$. Since G has no isolated vertices, and for all $(v, w) \in E$, there exists a node $x \in T_G$ with $v, w \in \mathcal{X}_G(x)$, we have that $\mathcal{V}(r) = \bigcup_{y \in T_G[r]} \mathcal{X}_G(y) = \bigcup_{y \in T_G} \mathcal{X}_G(y) = V$. The second condition is satisfied trivially, since it is explicitly stated that $\mathcal{S}(x) = \emptyset$ for leaves $x \in T_G$.

The third condition needs some more work. Let x be a node in T_G with children x_1 and x_2. Define for $i \in \{1, 2\}$, $V_i = \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \backslash \mathcal{X}_G(x)$.

<u>Claim 1</u>: $\mathcal{S}(x)$ separates the induced subgraph $G(\mathcal{V}(x))$ into V_1 and V_2.

Proof: Let us first show that $\{V_1, V_2, \mathcal{S}(x)\}$ is a partition of $\mathcal{V}(x)$. Trivially, $V_1 \cup V_2 \cup \mathcal{S}(x) = \mathcal{V}(x)$. It is also easy to see that $\mathcal{S}(x) \cap V_1 = \emptyset$, and $\mathcal{S}(x) \cap V_2 = \emptyset$. Furthermore, since for all $y \in T_G[x_1], y' \in T_G[x_2]$, $\mathcal{X}_G(y) \cap \mathcal{X}_G(y') \subseteq \mathcal{X}_G(x)$, we have that

$$\begin{aligned} V_1 \cap V_2 &= \bigcup_{y \in T_G[x_1]} \mathcal{X}_G(y) \backslash \mathcal{X}_G(x) \cap \bigcup_{y \in T_G[x_2]} \mathcal{X}_G(y) \backslash \mathcal{X}_G(x) \\ &= \left(\bigcup_{y \in T_G[x_1]} \mathcal{X}_G(y) \cap \bigcup_{y \in T_G[x_2]} \mathcal{X}_G(y) \right) \backslash \mathcal{X}_G(x) = \emptyset \end{aligned}$$

In order to complete the proof of claim 1, let us now show that V_1 and V_2 are separated by $\mathcal{S}(x)$. For the purpose of contradiction, assume there exist incident vertices v and w in G, with $v \in V_1$, and $w \in V_2$. Condition 2 of definition 4.1 states that there must be a $y \in T_G$

for which $v, w \in \mathcal{X}_G(y)$. Observe that the only candidates for y are the ancestors of x. In case x is the root, we have a contradiction. Otherwise let y be an ancestor of x for which $v, w \in \mathcal{X}_G(y)$. Since $v \in V_1$, there must be a descendant y' of x with $v \in \mathcal{X}_G(y')$. We thus have a contradiction, since $\{v\} \subseteq \mathcal{X}_G(y) \cap \mathcal{X}_G(y') \subseteq \mathcal{X}_G(x)$.

Claim 2: For $i = 1, 2$ it holds that:

$$\mathcal{V}(x_i) = V_i \cup \{v \in \mathcal{S}(x) | \exists w \in V_i \text{ such that } v \text{ and } w \text{ are incident in } G(\mathcal{V}(x))\}$$

Proof: Let $i \in \{1, 2\}$ be given, then

$$\begin{aligned} \mathcal{V}(x_i) &= \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \\ &= \left(\bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \backslash \mathcal{X}_G(x) \right) \cup \left(\bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x) \right) \\ &= V_i \cup \left(\bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x) \right) \qquad (2) \end{aligned}$$

In the following, denote with W the set:

$$\{v \in \mathcal{X}_G(x) | \exists w \in V_i \text{ such that } v \text{ and } w \text{ are incident in } G(\mathcal{V}(x))\}$$

Since $(T_G, \mathcal{X}_G)$ is clean, each vertex $v \in \mathcal{X}_G(x)$ with $v \in \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y)$ is incident to some vertices in $\bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \backslash \mathcal{X}_G(x) = V_i$. Therefore, $\bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x) \subseteq W$. We will now prove that $W \subseteq \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x)$.

Let a vertex $v \in \mathcal{X}_G(x)$ be given that is incident to a vertex $w \in V_i$. Since $(T_G, \mathcal{X}_G)$ is a tree decomposition there must be a node z in T_G with $v, w \in \mathcal{X}_G(z)$. Since $w \notin \mathcal{X}_G(x)$ we have that $z \neq x$. Suppose that z is an ancestor of x. Since $w \in \mathcal{X}_G(z')$ for an $z' \in T_G[x_i]$, we would have that $w \in \mathcal{X}_G(x)$. Therefore, z must be a descendant of x. Since V_1 and V_2 are disjoint, and $w \in V_i$ we find that $z \in T_G[x_i]$, and thus that $v \in \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x)$. It can be concluded that $W = \bigcup_{y \in T_G[x_i]} \mathcal{X}_G(y) \cap \mathcal{X}_G(x)$. Substituting this in (2) proves the claim.

We conclude from claims 1 and 2 that $(T_G, \mathcal{S}, \mathcal{V})$ also satisfies the third condition of the separator decomposition definition, and it is thus established that $(T_G, \mathcal{S}, \mathcal{V})$ is a separator decomposition of G. Furthermore, we have for leaves $x \in T_G$ that $|\mathcal{V}(x)| \leq k + 1$, and for all $x \in T_G$ that $|\mathcal{S}(x)| \leq k + 1$ and $|\mathcal{B}(x)| \leq k + 1$. The last inequality follows from the fact that for $x \in T_G$,

$$\mathcal{B}(x) = \mathcal{V}(x) \cap \bigcup_{y \in W_x} \mathcal{S}(y) = \bigcup_{y \in T_G[x]} \mathcal{X}_G(y) \cap \bigcup_{y \in W_x} \mathcal{X}_G(y) \subseteq \mathcal{X}_G(x),$$

where W_x is the set of all ancestors of x. ■

Observe that in the proof of claim 2 we really need to know that the tree decomposition is clean. The necessity for cleanness is simply a consequence of the fact that in separator decompositions we only take separator vertices down the tree if they are adjacent to the separated parts.

Lemma 4.3 contains the logical sense in which tree decompositions can be transformed into separator decompositions. In practice, the need for actually performing the conversion does not seem very substantial since a tree decomposition can easily be used as a cleancut representation of a separator decomposition.

Let us now consider the case where we have a tree decomposition of depth $d_G = O(\log n)$ and constant width. In actually performing the transformation indicated by lemma 4.3, the main bottleneck is the computation of the $\mathcal{V}$-sets. Assuming that these sets are represented as linked lists, we can make a copy, and concatenate these copies of two lists of length l_1 and l_2, in $O(1)$ time with $O(l_1+l_2)$ work. Calculating all $\mathcal{V}$-sets can be done in a straightforward bottom-up copying and uniting of $\mathcal{V}$-sets. A slight problem here is that a concatenated list might contain multiple occurrences of the same vertex. This can be handled as follows.

First for each node x a copy $\mathcal{X}'_G(x)$ is made of $\mathcal{X}_G(x)$. This takes $O(1)$ time, and $O(n)$ work. The $\mathcal{X}'_G$-sets will be used to compute the $\mathcal{V}$-sets. For each internal node $x \in T_G$, vertices from $\mathcal{X}'_G(x)$ are removed from $\mathcal{X}'_G(x')$ for all descendants x' of x. On the CREW PRAM this can be done as follows. For each node x', remove a vertex from $\mathcal{X}'_G(x')$ if it is contained in a $\mathcal{X}'_G$-set of an ancestor of x'. Processing all nodes x' in parallel, this takes $O(\log n)$ time and $O(n \log n)$ work. The $\mathcal{X}'_G$-sets obtained this way can be used just as before to compute the $\mathcal{V}$-sets. Consider an arbitrary internal node x with children x_1 and x_2. From the second part of definition 4.1 it can be concluded that $\mathcal{V}(x_1) \cap \mathcal{V}(x_2) = \emptyset$. The concatenation of the lists representing these sets will not contain multiple occurrences of the same vertex.

Let us analyze the overall complexity of calculating the $\mathcal{V}$-sets. Assume lists are concatenated in parallel for all nodes at height h. Processing nodes at height h takes $O(2^h)O(2^{d_G-h}) = O(2^{d_G}) = O(n)$ work, since there are $O(2^{d_G-h})$ nodes at height h, and the $\mathcal{V}$-sets at these nodes contain $O(2^h)$ vertices. We conclude that the whole conversion can be done in $O(\log n)$ time with $O(n \log n)$ work. Summarizing the results of this section:

Theorem 4.1 *Given an arbitrary width-k tree decomposition $(T_G, \mathcal{X}_G)$ of a graph G for constant $k \geq 0$, a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G can be computed in $O(\log^2 n)$ time with $O(n \log n)$ work. Furthermore, for all nodes $x \in T_G$ $|\mathcal{S}(x)|, |\mathcal{B}(x)| \leq 3k+3$, and for leaves $x \in T_G$, $|\mathcal{V}(x)| \leq 3k+3$.*

4.5 From Separator Decomposition to Tree Decomposition

In order to convert a separator decomposition into a tree decomposition a natural first attempt is to try the inverse of the transformation mentioned in the previous section. Given a graph $G = (V, E)$, binary tree T_G, and function $\mathcal{X}_G : T_G \to \wp(V)$, we had a transformation τ on pairs $(T_G, \mathcal{X}_G)$ defined as:

$$\tau((T_G, \mathcal{X}_G)) = (T_G, \mathcal{S}, \mathcal{V}), \text{where}$$

$$\begin{aligned} \mathcal{S}(x) &= \begin{cases} \mathcal{X}_G(x) & \text{if } x \text{ internal,} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{V}(x) &= \bigcup_{y \in T_G[x]} \mathcal{X}_G(y) \end{aligned}$$

Let $\mathcal{S}'$ and $\mathcal{V}'$ both be functions of type $T_G \to \wp(V)$. Assume that for all leaves x it holds that $\mathcal{S}'(x) = \emptyset$. A simple check shows that υ defined for tuples $(T_G, \mathcal{S}', \mathcal{V}')$ as

$$\upsilon((T_G, \mathcal{S}', \mathcal{V}')) = (T_G, \mathcal{X}_G), \text{where}$$

$$\mathcal{X}_G(x) = \begin{cases} \mathcal{S}'(x) & \text{if } x \text{ internal,} \\ \mathcal{V}'(x) & \text{otherwise} \end{cases}$$

is the inverse of the transformation τ. Let a pair $(T_G, \mathcal{X}_G)$ of appropriate type be given, then

$$\upsilon(\tau((T_G, \mathcal{X}_G))) = \upsilon((T_G, \mathcal{S}, \mathcal{V})), \text{ where}$$

$$\begin{aligned} \mathcal{S}(x) &= \begin{cases} \mathcal{X}_G(x) & \text{if } x \text{ internal,} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{V}(x) &= \bigcup_{y \in T_G[x]} \mathcal{X}_G(y) \end{aligned}$$

Working this out further, we get that

$$\upsilon((T_G, \mathcal{S}, \mathcal{V})) = (T_G, \mathcal{X}'_G), \text{ where}$$

$$\mathcal{X}'_G(x) = \begin{cases} \mathcal{S}(x) & \text{if } x \text{ internal,} \\ \mathcal{V}(x) & \text{otherwise} \end{cases}$$

From this it can be deduced that for all x in T_G it holds that $\mathcal{X}_G(x) = \mathcal{X}'_G(x)$. Therefore, $\upsilon(\tau((T_G, \mathcal{X}_G))) = (T_G, \mathcal{X}_G)$.

Conversely, let $(T_G, \mathcal{S}', \mathcal{V}')$ be an arbitrary tuple of appropriate type, then

$$\tau(\upsilon((T_G, \mathcal{S}', \mathcal{V}'))) = \tau((T_G, \mathcal{X}_G)), \text{ where}$$

$$\mathcal{X}_G(x) = \begin{cases} \mathcal{S}'(x) & \text{if } x \text{ internal,} \\ \mathcal{V}'(x) & \text{otherwise} \end{cases}$$

Working this out further we get that

$$\tau((T_G, \mathcal{X}_G)) = (T_G, \mathcal{S}, \mathcal{V}), \text{ where}$$

$$\begin{aligned}\mathcal{S}(x) &= \begin{cases} \mathcal{X}_G(x) & \text{if } x \text{ internal,} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{V}(x) &= \bigcup_{y \in T_G[x]} \mathcal{X}_G(y)\end{aligned}$$

Hence, $\mathcal{S}'(x) = \mathcal{S}(x)$, for all x in T_G. Furthermore,

$$\begin{aligned}\mathcal{V}(x) &= \bigcup_{y \in T_G[x]} \mathcal{X}_G(y) \\ &= \bigcup_{\text{internal } y \in T_G[x]} \mathcal{S}'(y) \cup \bigcup_{\text{leaves } y \in T_G[x]} \mathcal{V}(y') \\ &= \mathcal{V}'(x)\end{aligned}$$

The last equality follows almost directly from the separator decomposition definition. The above calculations establish that, for all separator decompositions $\mathcal{S}$, and for all tree decompositions $\mathcal{T}$:

1. $\tau(\upsilon(\mathcal{S})) = \mathcal{S}$,
2. $\upsilon(\tau(\mathcal{T})) = \mathcal{T}$

Thus it can be concluded that τ and υ are each other's inverse. Unfortunately it cannot be guaranteed that υ applied to an arbitrary separator decomposition always results in a tree decomposition, as the following example will show.

Figure 4.5 depicts a fragment of a separator decomposition tree. Separator sets and vertex sets are indicated with dotted lines around the contained vertices. Transformation υ translates separator sets directly into $\mathcal{X}_G$-sets. We get that $\alpha, \beta, \gamma \in \mathcal{X}_G(x)$, $\beta \in \mathcal{X}_G(y)$, and $\gamma \in \mathcal{X}_G(z)$. The node y lies on a path from x to z, and $\gamma \in \mathcal{X}_G(z) \cap \mathcal{X}_G(x)$. However it is not the case that $\gamma \in \mathcal{X}_G(y)$. In this example the second condition of definition 4.1 is therefore not being obeyed.

Looking closer at the previous example one could suspect that adding boundaries to the $\mathcal{X}_G$-sets would result in the general satisfaction of condition 2 of the tree decomposition definition. Lemma 4.5 shows that this is indeed the case. In the proof of lemma 4.5 the following lemma is used.

Lemma 4.4 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of a graph G, and let z be a node in T_G. If a vertex $v \in \mathcal{S}(z) \cup \mathcal{B}(z)$, then for all descendants y of z, if $v \notin \mathcal{B}(y)$ then $v \notin \mathcal{V}(y)$.*

Proof: Let y be a descendant of z. In the following, for a node $x \in T_G$ denote with W_x the set containing x and all ancestors of x. Proposition 2.1(2) states that $\mathcal{B}(y) = \mathcal{V}(y) \cap \bigcup_{y' \in W_y} \mathcal{S}(y)$. Hence, $\mathcal{B}(y) \supseteq \mathcal{V}(y) \cap \bigcup_{y' \in W_z} \mathcal{S}(y) \supseteq \mathcal{V}(y) \cap [\mathcal{S}(z) \cup \mathcal{B}(z)]$. Therefore, if a vertex $v \notin \mathcal{B}(y)$, we know that $v \notin \mathcal{V}(y)$, or $v \notin \mathcal{S}(z) \cup \mathcal{B}(z)$. The assumption

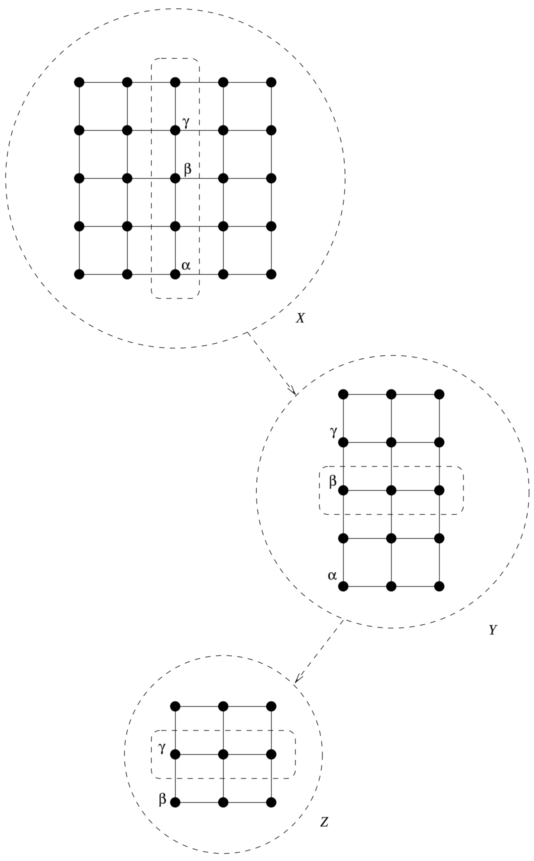

Figure 1: Fragment of a separator decomposition.

$v \in \mathcal{S}(z) \cup \mathcal{B}(z)$ implies that $v \notin \mathcal{V}(y)$. ■

In addition to the above lemma, note that in a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$, for a node x with children x_1 and x_2, if for $i = 1, 2$, $V_i = \mathcal{V}(x_i) \backslash \mathcal{S}(x)$ then

1. $\{\mathcal{S}(x), V_1, V_2\}$ is a partition of $\mathcal{V}(x)$
2. V_1 and V_2 are separated by $\mathcal{S}(x)$
3. $\mathcal{V}(x_i) = V_i \cup \{v \in \mathcal{S}(x) | \exists w \in V_i$ such that v and w incident in $G(\mathcal{V}(x))\}$

These facts follow directly from the separator decomposition definition.

Lemma 4.5 *Let $(T_G, \mathcal{S}, \mathcal{V})$ be a separator decomposition of graph G, then $(T_G, \mathcal{X}_G)$ with*

$$\mathcal{X}_G(x) = \begin{cases} \mathcal{S}(x) \cup \mathcal{B}(x) & \text{if } x \text{ internal,} \\ \mathcal{V}(x) & \text{otherwise} \end{cases}$$

is a tree decomposition of G.

Proof: Let us first show that for every $(v, w) \in E$ there is an x in T_G such that $v, w \in \mathcal{X}_G(x)$. Assume the converse, i.e. for an edge $(v, w) \in E$, for all $x \in T_G$, $v \notin \mathcal{X}_G(x)$, or $w \notin \mathcal{X}_G(x)$.

Claim: For internal nodes x with children x_1 and x_2, and $v, w \in \mathcal{V}(x)$, it holds that $v, w \in \mathcal{V}(x_i)$ for an $i \in \{1, 2\}$.
Proof: Since v, w are not both in $\mathcal{X}_G(x) = \mathcal{S}(x) \cup \mathcal{B}(x)$, they are not both in $\mathcal{S}(x)$. There are two cases to consider.
Case $v, w \notin \mathcal{S}(x)$
Without loss of generality assume that $v \in \mathcal{V}(x_1)$, and $w \in \mathcal{V}(x_2)$. In this case $v \in \mathcal{V}(x_1) \backslash \mathcal{S}(x)$, and $w \in \mathcal{V}(x_2) \backslash \mathcal{S}(x)$, which is a contradiction, since $\mathcal{V}(x_1) \backslash \mathcal{S}(x)$, and $\mathcal{V}(x_2) \backslash \mathcal{S}(x)$ are separated by $\mathcal{S}(x)$.
Case one of v, w is in $\mathcal{S}(x)$
Without loss of generality assume that $v \in \mathcal{S}(x)$. We have for an $i \in \{1, 2\}$, $w \in V_i = \mathcal{V}(x_i) \backslash \mathcal{S}(x)$. Since

$$\mathcal{V}(x_i) = V_i \cup \{v \in \mathcal{S}(x) | \exists w \in V_i : v \text{ and } w \text{ incident in } G(\mathcal{V}(x))\}$$

we conclude that $v \in \mathcal{V}(x_i)$. This proves the claim.

For the root r of T_G, $\mathcal{V}(r) = V$. Together with the claim this implies that $v, w \in \mathcal{X}_G(x)$ for a leaf $x \in T_G$, which contradicts our original assumption.

To complete the proof, we show that for x, y, z in T_G with y on the path from x to z, $\mathcal{X}_G(x) \cap \mathcal{X}_G(z) \subseteq \mathcal{X}_G(y)$. There are two cases to be considered.

In case x and z are both descendants of y, a vertex $v \notin \mathcal{S}(y)$ is not in both $\mathcal{V}(y_1)$ and $\mathcal{V}(y_2)$, for children y_1 and y_2 of y. Therefore, v is not in both $\mathcal{V}(x)$ and $\mathcal{V}(z)$. Hence,

$\mathcal{V}(x) \cap \mathcal{V}(z) \subseteq \mathcal{S}(y)$, and thus $\mathcal{X}_G(x) \cap \mathcal{X}_G(z) \subseteq \mathcal{V}(x) \cap \mathcal{V}(z) \subseteq \mathcal{S}(y) \subseteq \mathcal{S}(y) \cup \mathcal{B}(y) = \mathcal{X}_G(y)$.

In case y is a descendant of one of x and z, say z, we have that $\mathcal{X}_G(x) \cap \mathcal{X}_G(z) \subseteq \mathcal{V}(x) \cap \mathcal{X}_G(z) = \mathcal{V}(x) \cap [\mathcal{S}(z) \cup \mathcal{B}(z)] \subseteq \mathcal{S}(y) \cup \mathcal{B}(y) = \mathcal{X}_G(y)$. In order to prove this last subset relation, let $v \in \mathcal{V}(x)$, and $v \in \mathcal{S}(z) \cup \mathcal{B}(z)$. Assume that $v \notin \mathcal{S}(y) \cup \mathcal{B}(y)$. Therefore, $v \notin \mathcal{B}(y)$. Lemma 4.4 states that $v \notin \mathcal{V}(y)$, which implies that $v \notin \mathcal{V}(x)$. This is a contradiction. ■

Of course, adding the boundaries leads to an increase in the width of the resulting tree decomposition. For example, suppose all separators, and leaf vertex sets of a separator decomposition are of size at most k, for constant k. In case the separator decomposition is of depth $O(\log n)$, proposition 2.1(2) implies that boundaries are of size $O(\log n)$. Therefore, the resulting tree decomposition would have $O(\log n)$ width. Comparing this with the performance of the inverse transformation v, the result would be a pair $(T_G, \mathcal{X}_G)$ of width $k+1$. Unfortunately, it cannot be guaranteed that $(T_G, \mathcal{X}_G)$ obtained this way is always a tree decomposition.

Let us consider the case that the depth d_G of T_G is $O(\log n)$, $\mathcal{S}(x)$ is of constant size for all nodes $x \in T_G$, and $\mathcal{V}(x)$ is of constant size for all leaves $x \in T_G$. The main bottleneck in performing the above conversion is obviously the calculation of the boundaries. Let us sketch a two-phase CREW PRAM algorithm to perform this computation.

In the first phase, for each node $x \in T_G$, for each vertex $v \in \mathcal{V}(x)$, it is checked whether $v \in \mathcal{S}(x')$ for an ancestor x' of x. If the height of node x is h, then there are $O(2^h)$ vertices to consider, and there are $O(d_G - h)$ ancestors. Since there are $O(2^{d_G - h})$ nodes at height h, we find that we have to perform $O(2^{d_G-h})O(2^h)O(d_G - h) = O(2^{d_G}(d_G - h)) = O(n \log n)$ work to handle all nodes at height h. Taking care of nodes at all heights in the tree therefore takes $O(n \log^2 n)$ work.

Having established for each node x, for each $v \in \mathcal{V}(x)$ whether it is in the boundary or not, the second phases entered, which consists of constructing the boundary sets. Assuming these sets are represented by linked lists, constructing an individual boundary set $\mathcal{B}(x)$, takes work linear in the number of vertices in $\mathcal{B}(x)$, which is $O(\log n)$. Therefore, the second phase takes $O(n \log n)$ work.

Let us consider the running time of the algorithm. If in the first phase the check is performed both for all nodes x, as for all vertices $v \in \mathcal{V}(x)$ in parallel, phase one will take $O(\log n)$ time, since there are at most $O(\log n)$ ancestors to consider, and each ancestor can be handled in constant time. Furthermore, in the second phase the boundary sets can be constructed for each node in parallel, therefore the second phase also takes $O(\log n)$ time. To summarize we mention the following theorem.

Theorem 4.2 *Given a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of graph G of $O(\log n)$ depth for which $|\mathcal{S}(x)| \leq k$ for all $x \in T_G$, $|\mathcal{V}(x)| \leq k$ for all leaves $x \in T_G$, and constant $k \geq 0$, a tree decomposition of width $O(\log n)$ can be computed in $O(\log n)$ time with $O(n \log^2 n)$ work.*

5 An $\mathcal{NC}$-flow algorithm

In section 3 an efficient parallel algorithm was given for the shortest path problem in case a suitable separator decomposition of the underlying graph was available. The algorithm can be used in the shortest augmenting path maxflow algorithm, introducing some parallelism into maxflow computation. However, the amount of parallelism in this algorithm is not satisfactory. The difficulty is that the augmentations are still done one after another. Since the total number of augmentations with the shortest augmenting path maxflow algorithm is $O(nm)$, we already have a factor of $O(nm)$ in the running time of the algorithm.

In this section we will see a more successful approach. It uses separator decompositions to solve the maxflow problem in a "Divide and Conquer" manner. It is shown that flow in a k-terminal network, that is a network for which the number of sources plus the number of sinks is k, can be characterized by a set of at most 2^k equations, or equivalently by a mimicking network of at most $2^{(2^k)}$ vertices. More precisely, the possible patterns of imbalance at the terminals, which will be referred to as feasible external flow patterns, are characterized. The $\mathcal{NC}$-maxflow algorithm makes use of these characterizations in the following way.

Suppose we want to compute the maximum flow in network (G, c, s, t). Given a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G, we associate with each node x in T_G a network $N(x)$. In addition to that, we associate with each internal node x a network $N_s(x)$. In the algorithm the external flow characterizations of these networks will be computed in a bottom-up fashion. It is shown that for an internal node x with children x_1 and x_2 the characterizations of the child networks $N(x_1)$ and $N(x_2)$, together with the characterization of network $N_s(x)$ can be 'melt' into a characterization of the associated network $N(x)$. If the number of terminals in the merged networks $N(x_1), N(x_2)$ and $N_s(x)$ is $O(1)$ this can be done is $O(1)$ time. Since the network $N(r)$ associated with the root r of T_G is equal to the original network (G, c, s, t) the final result of this procedure is a characterization of external flow in the network (G, c, s, t). Once this characterization is computed, a corresponding flow can be efficiently calculated for each external flow pattern. For the special case that the separator decomposition of the underlying graph has separators, boundaries, and leaf vertex sets of constant size, this results in an $\mathcal{NC}$-maxflow algorithm. From theorems 4.1 and 4.2 we know that these conditions are satisfied if and only if the graph G has bounded treewidth.

5.1 Characterizing external flow

In the flow algorithm that follows later, networks can have more than one source and sink. We therefore generalize our network definition as follows.

Definition 5.1 *A k-terminal network is a triple (G, c, Q), where $G = (V, E)$ is a digraph with capacities on the edges specified by $c : E \to \mathbb{R}_{\geq 0}$, and Q is an ordered set $\{q_1, \ldots, q_k\} \subseteq V$ of distinguished vertices called terminals. A flow in a k-terminal network (G, c, Q) is a function $f : E \to \mathbb{R}_{\geq 0}$ for which:*

1. *$0 \leq f(e) \leq c(e)$ for all edges $e \in E$*
2. *$b_f(i) = \sum_{(i,j)\in E} f(i,j) - \sum_{(j,i)\in E} f(j,i) = 0$ for all $i \in V\backslash Q$.*

In other words, a flow in a k-terminal network is a flow which is balanced everywhere, except possibly at the terminals. For a flow f in a k-terminal network we call the imbalances at the terminals the external flow pattern of f. External flow patterns can be considered to be abstractions of flow that hide away the internal structure. In a network external flow patterns are either realisable or not:

Definition 5.2 *A realizable external flow in a k-terminal network N is a k-tuple $(x_1,\dots,x_k)$ of real numbers associated with the terminals $q_1,\dots,q_k$ for which there exists a flow f in N such that $b_f(q_i) = x_i$ for $i = 1,\dots,k$.*

Considering the terminals to be the places of inflow and outflow in the network, one would expect a flow to have its total input and output balanced. The following lemma shows that this is indeed the case. We use induction to the size of the network to prove the lemma.

Lemma 5.1 *If $(x_1,\dots,x_k)$ is a realizable external flow in some k-terminal network for some $k \geq 0$, then $\sum_{i=1}^{k} x_i = 0$.*

Proof: For networks containing no vertices the conditions holds trivially. Let $(x_1,\dots,x_k)$ be the external flow tuple of flow f in a k-terminal network $(G = (V,E), c, Q)$. Assume the graph G contains $n+1$ vertices. Consider the network $N' = (G' = (V',E'), c', Q')$, where $V' = V\backslash Q$, $E' = E(V\backslash Q)$, c' equals c restricted to E', and $Q' = \{v \in V\backslash Q | v \text{ connected to terminal in } G\}$. Observe that f is a flow in this network. Let $(x'_1,\dots,x'_k)$ be the external flow tuple of f in N' corresponding to terminals $q'_1,\dots,q'_{k'}$. We have that

$$\sum_{i=1}^{k} x_i = \sum_{i=1}^{k}\left(\sum_{(q_i,v)\in E} f(q_i,v) - \sum_{(v,q_i)\in E} f(v,q_i)\right) =$$

$$\sum_{i=1}^{k}\left(\sum_{(q_i,v)\in E, v\notin Q} f(q_i,v) - \sum_{(v,q_i)\in E, v\notin Q} f(v,q_i)\right) =$$

$$\sum_{i=1}^{k'}\left(\sum_{(w,q'_i)\in E, w\in Q} f(w,q'_i) - \sum_{(q'_i,w)\in E, w\in Q} f(q'_i,w)\right) =$$

$$\sum_{i=1}^{k'}\left(\sum_{(w,q'_i)\in E} f(w,q'_i) - \sum_{(q'_i,w)\in E} f(q'_i,w)\right) -$$

$$\sum_{i=1}^{k'}\left(\sum_{(w,q'_i)\in E, w\notin Q} f(w,q'_i) - \sum_{(q'_i,w)\in E, w\notin Q} f(q'_i,w)\right) =$$
$$\sum_{i=1}^{k'}\left(\sum_{(q'_i,w)\in E, w\notin Q} f(q'_i,w) - \sum_{(w,q'_i)\in E, w\notin Q} f(w,q'_i)\right) =$$
$$\sum_{i=1}^{k'}\left(\sum_{(q'_i,w)\in E'} f(q'_i,w) - \sum_{(w,q'_i)\in E'} f(w,q'_i)\right) = \sum_{i=1}^{k'} x'_i = 0$$

This proves the lemma. ■

We will refer to the condition of lemma 5.1 as the network balance constraint. Recall that in the conventional situation of having one source s and one sink t we have that the amount of flow that flows from s to t is at most equal to the value of the minimum s,t-separating cut. Considering an arbitrary subset $S \subseteq Q$ of terminals, lemma 5.1 implies that $\sum_{q_i\in S} x_i = -\sum_{q_i\notin S} x_i$, which is a hint towards looking at $\sum_{q_i\in S} x_i$ as the amount of flow that goes from S to $Q\backslash S$. In the light of the previous remark one would expect this amount of flow to be bounded by the value of any minimum $S, Q\backslash S$-separating cut. It is possible to prove this.

Lemma 5.2 *For a realizable external flow $(x_1,\ldots,x_k)$ in a k-terminal network (G,c,Q) we have for all subsets $S \subseteq Q$ that $\sum_{q_i\in S} x_i \leq b_S$, where b_S is the value of a minimum $S, Q\backslash S$-separating cut.*

Proof: Let f be a flow in a k-terminal network (G,c,Q) with external flow pattern $(x_1,\ldots,x_k)$. Consider an arbitrary subset $S \subseteq Q$. Let X be an $S, Q\backslash S$-separating cut. We have that

$$\sum_{q_i\in S}^{k} x_i =$$
$$\sum_{v\in X}\left(\sum_{(v,w)\in E} f(v,w) - \sum_{(w,v)\in E} f(w,v)\right) =$$
$$\sum_{\substack{(v,w)\in E\\ v\in X, w\notin X}} f(v,w) - \sum_{\substack{(w,v)\in E\\ w\notin X, v\in X}} f(w,v) \leq$$
$$\sum_{\substack{(v,w)\in E\\ v\in X, w\notin X}} f(v,w) \leq \sum_{\substack{(v,w)\in E\\ v\in X, w\notin X}} c(v,w) \leq b_S$$

this proves the lemma. ■

Lemma 5.1 and 5.2 show us two important properties of realizable external flows. It would be interesting to see whether, if these properties hold for a k-tuple $\overline{x}$, this would imply that $\overline{x}$ is a realizable external flow. If this is the case we have found a complete characterization of realizable external flows. The following is a reformulation of a theorem proposed by Hagerup *et al.* [15]. They derived this result by a slight modification and extension of a result due to Gale [13].

Theorem 5.1 *In a k-terminal network (G, c, Q), a k-tuple $(x_1, \ldots, x_k)$ is a realizable external flow if and only if the following two relations are satisfied:*

1. $\sum_{i=1}^{k} x_i = 0$
2. *For all subsets $S \subseteq Q$, $\sum_{q_i \in S} x_i \leq b_S$, where b_S is the value of any minimum $S, Q \backslash S$-separating cut.*

Proof: Let (G, c, Q) be an arbitrary k-terminal network, and let $(x_1, \ldots, x_k)$ be such that conditions 1 and 2 hold. Let us show that $(x_1, \ldots, x_k)$ is a realizable external flow in (G, c, Q). The converse follows from lemma 5.1 and lemma 5.2.

We augment the network with two new vertices s and t. Furthermore, for each i such that $x_i > 0$ an edge (s, q_i) with capacity x_i is added, and for each i such that $x_i < 0$ an edge (q_i, t) with capacity $-x_i$ is added. Let us show that in this network the cut defined by $\{s\}$ is a minimum s, t-separating cut. In this case we deduce with the max-flow min-cut theorem that there exists a flow f such that all edges (s, q_i) are saturated. Condition 1 implies that in this case also all edges (q_i, t) are saturated. Restricting f to the original network then gives us a flow with external flow pattern $(x_1, \ldots, x_k)$.

Consider an arbitrary subset $X \subset V$. The cut defined by the set $\{s\} \cup X$ consists of edges from s to $Q \backslash X$, edges from X to $V \backslash X$, and edges from $Q \cap X$ to t. The capacity of this cut is

$$\sum_{q_i \in Q \backslash X, x_i > 0} x_i + \sum_{\substack{(v,w) \in E \\ v \in X, w \in V \backslash X}} c(v, w) - \sum_{q_i \in Q \cap X, x_i < 0} x_i =$$

$$\sum_{x_i > 0} x_i - \sum_{q_i \in Q \cap X, x_i > 0} x_i + \sum_{\substack{(v,w) \in E \\ v \in X, w \in V \backslash X}} c(v, w) - \sum_{q_i \in Q \cap X, x_i < 0} x_i =$$

$$\sum_{x_i > 0} x_i + \sum_{\substack{(v,w) \in E \\ v \in X, w \in V \backslash X}} c(v, w) - \sum_{q_i \in Q \cap X} x_i$$

The first term in the last line is the capacity of the cut defined by $\{s\}$. The sum of the last two terms is non-negative, since the tuple satisfies the second condition of the theorem, which implies that $\sum_{q_i \in Q \cap X} x_i \leq b_{Q \cap X}$. Therefore, the value of the cut defined by $\{s\}$ is less than or equal to the value of the cut defined by $\{s\} \cup X$. We conclude that the cut defined by $\{s\}$ has value less than or equal to the value of any s,t-separating cut. ■

The theorem establishes that feasible external flow in a k-terminal network can be characterized by a set of 2^k inequalities. In the following, this set of inequalities will be referred to as the external flow inequalities. The right-hand side of each inequality can be found as follows.

In order to find b_S for each subset $S \subseteq Q$, new vertices s and t are added to the original graph G. For all $v \in S$ and $w \in Q\backslash S$ we add to G edges (s,v) and (w,t), respectively, of infinite capacity. In the resulting network G' we find the value r of the maximum s,t-flow. The following lemma shows that this is the value b_S we are looking for.

Lemma 5.3 *In the above described procedure r equals b_S, the value of any minimum $S, Q\backslash S$-separating cut.*

Proof: The max-flow min-cut theorem implies that the value of any minimum s',t'-separating cut X in G' is r. We will establish that for any such minimum s',t'-separating cut X, the set $X\backslash\{s'\}$ is a minimum $S,Q\backslash S$-separating cut in G with value equal to r.

Let a mininum s',t'-separating cut X in G' be given. The capacities of edges in G are finite. Therefore cuts in G' which contain no added edges all have smaller value than cuts that do contain added edges. Since X is a minimum s',t'-separating cut we find that $X \cap Q = S$. Therefore, we have that the set $X\backslash\{s'\}$ is a $S,Q\backslash S$-separating cut.

Suppose there exist an $S,Q\backslash S$-separating cut X' in G, which has smaller value than X. In this case $X' \cup \{s'\}$ would be an s',t'-separating cut in G' with smaller value than X, which contradicts the minimality of X.

Observe that cut X and cut $X\backslash\{s'\}$ contain the same edges, therefore the value of $X\backslash\{s'\}$ is r. ■

It can be concluded that the procedure described above is correct. Note that for any k-terminal network one can therefore find a characterization of its realizable external flow in time polynomial in the size of the network, and exponential in the number of terminals.

5.2 Merging external flow characterizations

In the following unifying two vertices v and w of a graph G will refer to the procedure of making v and w into one vertex u. In this procedure the edges incident to v and w become incident to the unified vertex u, multiple edges are merged into one, and loops are eliminated. Note that the unification of incident vertices v and w is equivalent to contracting all edges between v and w. Given two vertex disjoint k-terminal networks $N_1 = (G_1 = (V_1, E_1), c_1, Q_1)$ and $N_2 = (G_2 = (V_2, E_2), c_2, Q_2)$, the union of N_1 and N_2 is the network (G', c', Q') defined as:

1. $G' = (V_1 \cup V_2, E_1 \cup E_2)$
2. $c'(e) = \begin{cases} c_1(e) & \text{if } e \in E_1 \\ c_2(e) & \text{otherwise} \end{cases}$

3. $Q' = Q_1 \cup Q_2$

Having defined unification and union as above, consider the following problem.

Suppose for vertex disjoint k-terminal networks $N_1 = (G_1, c_1, Q_1)$ and $N_2 = (G_2, c_2, Q_2)$ realizable external flow is characterized by sets of inequalities $\mathcal{C}_1$ and $\mathcal{C}_2$. We assume that these sets are represented in a way that makes them suitable objects for the computations performed in subsequent text. Let $I \subseteq Q_1 \times Q_2$ be a given matching of terminals. Since in the following we do not want the difficulty of having to deal with multiple edges, assume that there are no edges between terminals that occur in the matching. If $N' = (G', c', Q')$ is the network resulting from uniting networks N_1 and N_2, and unifying all matched pairs of terminals in I, how do we derive the external flow inequalities of N' from the sets $\mathcal{C}_1$ and $\mathcal{C}_2$? For clarity, unification of two terminals results in one unified vertex which is a terminal.

In order to describe the external flow in unified terminals we introduce for each unified terminal q a new variable x_q. Since $\mathcal{C}_1$ and $\mathcal{C}_2$ characterize the external flow in the subnetworks N_1 and N_2 of N' we get a characterization $\mathcal{C}'$ of the external flow in N' by adding to $\mathcal{C}_1 \cup \mathcal{C}_2$ the equalities that correspond to the process of unifying terminals as follows.

Initially, $\mathcal{C}'$ is set equal to $\mathcal{C}_1 \cup \mathcal{C}_2$. If terminals q_1 and q_2 have external flow (x_{q_1}, x_{q_2}), and get unified into terminal q, the external flow of q is $x_{q_1} + x_{q_2}$. Therefore, for all unified terminals q_1 and q_2 the equation $x_q = x_{q_1} + x_{q_2}$ is added to $\mathcal{C}'$. Note that a terminal q in N' can be made into a non-terminal by substituting 0 for the variable x_q in $\mathcal{C}'$. The resulting set of equations and inequalities $\mathcal{C}'$ characterizes the external flow of network N'.

There is a slight problem with the set $\mathcal{C}'$, namely it is not in the form of theorem 5.1. However, we can obtain an equivalent set of inequalities $\mathcal{C}''$ in desired format by considering the constraints one-by-one. For each subset $S \subseteq Q'$ we simply have to compute b_S, the capacity of any minimum $S, Q'\backslash S$-separating cut, and add the inequality $\sum\limits_{q_i \in S} x_i \le b_S$ to $\mathcal{C}''$. The value b_S is computed by maximizing $\sum\limits_{q_i \in S} x_i$ with respect to constraints $\mathcal{C}'$ and the network balancing constraint with an algorithm for linear programming in fixed dimensions discovered by Chazelle and Matoušek [7]. In our case this will take $O(1)$ time. Assume we find a value r this way. We have that $r = b_S$, since if $r < b_S$ theorem 5.1 tells that there are external flow tuples for which $\sum\limits_{q_i \in S} x_i > r$, which contradicts the maximality of r. Conversely, if $r > b_S$ then we would have found a realizable external flow tuple for which $\sum\limits_{q_i \in S} x_i > b_S$, which contradicts theorem 5.1.

Once for all subset of terminals $S \subseteq Q'$ the value b_S has been computed, the set $\mathcal{C}''$ characterizes feasible external flow in network N'. Just as long as the total number of terminals $|Q_1| + |Q_2|$ is constant, this characterization can be computed in $O(1)$ time.

5.3 The $\mathcal{NC}$ maxflow algorithm

The merging of flow inequalities as described in previous section can be used to implement a "Divide and Conquer" maxflow algorithm. However, the way networks are merged here is slightly different. Instead of merging two networks, we now merge three networks in one step. In this merging, possibly three terminals are unified into one. In the following we will see that this does not really impose any extra difficulty.

Suppose for a network (G, c, s, t) a separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G is available. We associate a network $N(x) = (G'(x), c', Q(x))$ to all nodes x in T_G, where $Q(x) = (\mathcal{V}(x) \cap \{s, t\}) \cup \mathcal{B}(x)$, $G'(x) = (\mathcal{V}(x), E(\mathcal{V}(x)) \backslash \{(v, w) | v, w \in \mathcal{B}(x)\})$, and c' is equal to c restricted to edges of $G'(x)$. Intuitively, one can look at this as follows. The vertices in the boundary $\mathcal{B}(x)$ are taken as terminals since network $N(x)$ is merged with other networks by identification of boundary vertices. Therefore $G'(x)$ does not contain edges between boundary vertices, since we do not want the complication of having to eliminate multiple edges. Observe that the network $N(r)$ associated with the root r of T_G is equal to the original network (G, c, s, t).

In addition to the network $N(x)$, for all internal nodes we have a network $N_s(x) = (G''(x), c, \mathcal{S}(x))$, where $G''(x) = (\mathcal{S}(x), E(\mathcal{S}(x)) \backslash \{(v, w) | v, w \in \mathcal{B}(x)\})$. In the following we describe how for internal nodes x the external flow inequalities of $N(x)$ can be computed from the external flow inequalities of the networks associated with its children and the connecting network $N_s(x)$.

First observe that for internal node x with children x_1 and x_2 we have that $G'(x) = G'(x_1) \cup G'(x_2) \cup G''(x)$. Also observe that $\mathcal{V}(x_1) \cap \mathcal{V}(x_2) \subseteq \mathcal{S}(x)$. Therefore the graph $G'(x)$ can be constructed by first taking the disjoint union of graphs $G'(x_1)$, $G'(x_2)$ and $G''(x)$, and then unifying:

1. for all vertices in $\mathcal{V}(x_i) \cap \mathcal{S}(x)$ which are not in $\mathcal{V}(x_{3-i})$, the occurrences in disjoint subgraphs $G'(x_i)$ and $G''(x)$, for $i = 1, 2$.

2. for all vertices in $\mathcal{V}(x_1) \cap \mathcal{V}(x_2) \cap \mathcal{S}(x)$, the occurrences in disjoint subgraphs $G'(x_1)$, $G'(x_2)$, and $G''(x)$.

Note that $G'(x_1)$, $G'(x_2)$, and $G''(x)$ are edge disjoint. Therefore, $G'(x)$ as obtained above does not contain multiple edges. Furthermore, observe that the vertices that fall under 1 appear in $N(x_i)$ and $N_s(x)$ as terminals. Similarly, vertices that fall under case 2, appear in $N(x_1)$, $N(x_2)$, and $N_s(x)$ as terminals. We are therefore assured that we only unify terminals. Furthermore, since $\mathcal{B}(x) \subseteq \mathcal{B}(x_1) \cup \mathcal{B}(x_2) \cup \mathcal{S}(x)$ we are assured that all terminals in $N(x)$ are terminals in $N(x_1)$, $N(x_2)$ and $N_s(x)$, which is necessary for the merging of external flow inequalities of the networks $N(x_1)$, $N(x_2)$ and $N_s(x)$ to result in a complete characterization of the external flow in $N(x)$. Possibly two terminals get merged into a non-terminal. In this case the same substitution trick is applied as in previous section.

Let $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ be the external flow inequalities of networks $N(x_1)$, $N(x_2)$, and $N_s(x)$, respectively. Assuming these external flow inequalities all use different variables, we make the following notational convention. For a vertex $q \in V$ that occurs as terminal in $\mathcal{C}_i$ for $i \in \{1, 2, 3\}$, denote with x_q^i the variable refering to q in $\mathcal{C}_i$. The external flow of network $N(x)$ can be characterized by the extending the set of inequalities $\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ as follows.

For each vertex q that falls in one of the categories mentioned above, a new variable x_q is introduced. In case vertex q falls under 1, we add the equality $x_q = x_q^i + x_q^3$ to $\mathcal{C}'$. In case it falls under category 2, the equation $x_q = x_q^1 + x_q^2 + x_q^3$ is added to $\mathcal{C}'$. At the completion of these additions the set $\mathcal{C}'$ will characterize the external flow of network $N(x)$. In order to get $\mathcal{C}'$ in the form of theorem 5.1 the same linear programming procedure is followed as in the previous section.

Having seen how for an internal node x we can compute the external flow inequalities of $N(x)$ by merging the inequalities of the networks associated with its children and network $N_s(x)$, an algorithm for computing the external flow inequalities for (G, c, s, t) is obvious. As a first step, we calculate the external flow inequalities for the networks associated with leaf nodes, for all leaf nodes in parallel as in section 5.1. In consecutive steps we work up the tree, merging flow inequalities in parallel for all nodes for which the networks associated with its children are already characterized. The characterization of networks $N_s(x)$ will be computed at the time of this merging.

We will now analyze the algorithm for the case that we have a constant k such that for all $x \in T_G$, $|\mathcal{S}(x)|, |\mathcal{B}(x)| \leq k$, if x is a leaf then $|\mathcal{V}(x)| \leq k$, and T_G has depth $O(\log n)$. If G has bounded treewidth we can compute such a separator decomposition in $O(\log^2 n)$-time with $O(n \log n)$ work, as we saw in previous section.

Since the leaf vertex sets have size at most k, the characterization of the networks associated with the leaves can be computed in $O(1)$ time. After this has been done $O(\log n)$ merging phases follow. Each parallel merging at a node consists of characterizing the network induced by the separator, and performing the linear programming in fixed dimensions. The first can be done in $O(1)$ time, since separators have size at most k. The second can be done in $O(d^{O(d)} n)$ time, in case of d variables, and n constraints [7]. In our case the number of variables and constraints are bounded by a constant since boundaries are bounded by a constant. Therefore the application of the linear programming algorithm takes $O(1)$ time. We conclude that the algorithm runs in $O(\log n)$ time. At each node $O(1)$ work is performed, therefore the algorithm performs a total of $O(n)$ work.

Once the external flow inequalities of $(G, c, \{s, t\})$ are calculated, for any realizable external flow pattern (v_s, v_t) we can calculate a corresponding flow as follows. If $\mathcal{C}'$ is the extended set of inequalities characterizing $(G, c, \{s, t\})$, we substitute the values v_s and v_t in $\mathcal{C}'$ for variables x_s and x_t. With linear programming we then find values for the remaining variables that fit the inequalities. The values found this way make up feasible external flow patterns in the networks associated with the children of the root, and the N_s-network of the root. For the networks associated with the children we can now recursively apply this procedure. For node x in T_G, we compute a flow in network $N_s(x)$ as follows. The same technique is used for calculation of flow in networks associated with leaves.

First augment $N_s(x)$ with vertices s' and t'. Secondly, for each terminal q with $x_q > 0$, add an edge (s', q) to $N_s(x)$, and set its capacity to x_q. For each terminal q with $x_q < 0$, we add an edge (q, t') to $N_s(x)$, and set its capacity to $-x_q$. Once this is done, a maximum s', t'-flow f is computed in the augmented network. The flow f restricted to network $N_s(x)$ has the required external flow pattern.

It is obvious that the two flows determined recursively together with the flow determined in the N_s-network make up a flow in the original network corresponding to the requested external flow pattern. Unwinding recursion we find that the networks associated with the leaves, and the N_s-networks piece together a flow in original network. Summarizing the results of this section:

Theorem 5.2 *Given network $N = (G, c, s, t)$ and separator decomposition $(T_G, \mathcal{S}, \mathcal{V})$ of G for which separators, boundaries, and leaf vertex sets have size $O(1)$, and T_G has $O(\log n)$ depth, computing a maximum flow in N can be done in $O(\log n)$ time with $O(n)$ work.*

5.4 Mimicking Networks

Characterizing external flow with sets of inequalities, and the implied use of linear programming algorithms, is a practice that one might want to refrain from. Fortunately, there exists an alternative way to go about. In this section we will see that for each k-terminal network N we can construct a network N', in which precicely the same set of external flow tuples is realizable as in N. The network N' is referred to as being a mimicking network of N. It has size dependent on only the number of terminals in N, namely at most $2^{(2^k)}$ vertices. In the following text we will see that we can use mimicking networks in the flow algorithm of the previous section for the characterization of external flow. Although this is less time-efficient than the use of external flow inequalities, the resulting algorithm is still in $\mathcal{NC}$ if separators, boundaries, and leaf vertex sets are of constant size. Let us get more detailed.

Definition 5.3 *Given k-terminal network (G, c, Q) with for each subset $S \subseteq Q$ a certain minimum $S, Q \backslash S$-separating cut with defining subset C_S, two vertices v and w are equivalent, denoted $v \equiv w$, if and only if for all $S \subseteq Q$ it holds that $v, w \in C_S$ or $v, w \notin C_S$.*

It is easy to see that $\equiv$ is an equivalence relation. For each vertex $v \in V$ let E_v be the equivalence class that contains it.

Definition 5.4 *Given k-terminal network $N = (G, c, Q)$ with equivalence relation $\equiv$ on its vertices, the mimicking network of N is the network $(G' = (V', E'), c', Q')$ for which:*

1. *$V' = \{E_v | v \in V\}$,*
2. *for $U, W \in V'$, $(U, W) \in E'$ iff there exist $u \in U$ and $w \in W$ with $(u, w) \in E$,*
3. *each edge $e = (U, W) \in E'$ has capacity $c'(e) = \sum_{\substack{u \in U, w \in W \\ (u,w) \in E}} c(u, w)$, and*
4. *$Q' = \{E_q | q \in Q\}$.*

In order to construct a mimicking network in the above situation, first for each subset $S \subseteq Q$ the set C_S has to be calculated. Secondly, the equivalence classes of $\equiv$ have to be determined. An approach could be to determine for each vertex $v \in V$ a binary vector $\vec{r}_v$

of length 2^k, that contains the information about the containment of v in all sets C_S. The mimicking network then gets a vertex for each different vector we encounter. Initializing capacities in the mimicking network with zero, we would then have to consider all edges $e \in E$ and add $c(e)$ to the appropriate edge capacity in the mimicking network. Observe that computing mimicking networks in this fashion takes time polynomial in the number of vertices in G, and exponential in the number of terminals k.

Knowing how we can construct a mimicking network, we are still left with the important issue whether this network actually has the mimicking character its name reflects. The following theorem establishes that our construction indeed leads to a network that mimicks the original network as far as external flow is concerned.

Theorem 5.3 *Let $N' = (G', c', Q')$ be a mimicking network of k-terminal network $N = (G, c, Q = \{q_1, \ldots, q_k\})$, then any tuple of reals $(x_1, \ldots, x_k)$ associated with terminals $q_1, \ldots, q_k$ is a realizable external flow pattern in N if and only if $(x_1, \ldots, x_k)$ associated with terminals $E_{q_1}, \ldots, E_{q_k}$ is a realizable external flow pattern in N'.*

Proof: From theorem 5.1 we see that a tuple $(x_1, \ldots, x_k)$ associated with terminals $q_1, \ldots, q_k$ is realizable in N if and only if the following two conditions hold:

1. $\sum_{i=1}^{k} x_i = 0$
2. For all subsets $S \subseteq Q$, $\sum_{q_i \in S} x_i \leq b_S$, where b_S is the value of any minimum $S, Q \backslash S$-separating cut.

Also from theorem 5.1 we see that a tuple $(x_1, \ldots, x_k)$ associated with terminals $E_{q_1}, \ldots, E_{q_k}$ is realisable in N' if and only if the following two conditions hold:

1. $\sum_{i=1}^{k} x_i = 0$
2. For all subsets $S' \subseteq Q'$, $\sum_{E_{q_i} \in S'} x_i \leq b_{S'}$, where $b_{S'}$ is the value of any minimum $S', Q' \backslash S'$-separating cut.

Since an equivalence class can not contain two or more terminals, the second condition of the above is equivalent to stating that for all subsets $S \subseteq Q$, $\sum_{q_i \in S} x_i \leq b_{S'}$, where $S' = \{E_q | q \in S\}$, and $b_{S'}$ is the value of any minimum $S', Q' \backslash S'$-separating cut. Let us prove the theorem by showing that for all subsets $S \subseteq Q$ it holds that $b_S = b_{S'}$. Consider an arbitrary subset $S \subseteq Q$. Let X be the defining subset of an $S', Q' \backslash S'$-separating cut. Observe that the cut defined by $X' = \bigcup_{U \in X} U$ is an $S, Q \backslash S$-separating cut, and that the capacity of X and X' is the same. Therefore, it must be that $b_S \leq b_{S'}$. Conversely, let

C_S be the minimum $S', Q' \backslash S'$-separating cut used in the construction of the mimicking network. Consider the cut in N' defined by subset $C'_S = \{E_v | v \in C_S\}$. We have that

$$C'_S \cap Q' = \{E_v | v \in C_S\} \cap \{E_q | q \in Q\} =$$
$$\{E_q | q \in C_S \cap Q\} = \{E_q | q \in S\} = S'.$$

Since each equivalence class in C'_S is completely contained in C_S we have that the capacity of C'_S is at most equal to the capacity of C_S. Therefore, it holds that $b'_S \leq b_S$. ■

We now show that a mimicking network has size independent of the size of the original network.

Theorem 5.4 *A mimicking network of a k-terminal network (G, c, Q) has at most $2^{(2^k)}$ vertices.*

Proof: The number of cuts C_S used in definition 5.3 is at most 2^k. For a vertex $v \in V$ the equivalence class E_v can be specified by a binary vector of length 2^k, containing information in which of the 2^k cuts v is contained. We conclude that there are at most $2^{(2^k)}$ equivalence classes. ■

Let us now briefly address to the issue of using mimicking networks in the flow algorithm of previous section. Recall we had here, for each node x an associated network $N(x)$, and for each internal node x a network $N_s(x)$. For internal nodes x with children x_1 and x_2 we argued that the network $N(x)$ could be formed by first taking the disjoint union of the networks $N(x_1)$, $N(x_2)$, and $N_s(x)$, and then unifying certain vertices in this disjoint union. Let us assume the mimicking network of the networks $N(x_1)$, $N(x_2)$, and $N_s(x)$ are already calculated. In order to calculate the mimicking network of $N(x)$, we first take the disjoint union of these three mimicking networks. Theorem 5.3 gives us a one-to-one correspondence between the terminals in the networks $N(x_1)$, $N(x_2)$, and $N_s(x)$, and the terminals in their mimicking networks. Therefore, for each unification of vertices in the original networks, we simply unify the corresponding vertices in the mimicking network. The network obtained this way features the same external flow behavior as the network $N(x)$. However, it might not be in the form of definition 5.4 anymore. Therefore, as a final step we apply the mimicking network construction procedure on this network, obtaining the mimicking network of $N(x)$.

Let us consider the resources involved in case we have a constant k such that for all $x \in T_G$, $|\mathcal{S}(x)|, |\mathcal{B}(x)| \leq k$, if x is a leaf then $|\mathcal{V}(x)| \leq k$, and T_G has depth $O(\log n)$. Observe that in this case the mimicking networks of the networks associated with the leaves of T_G, and the N_s-networks can be computed in $O(1)$ time. Furthermore, since boundaries are bounded by k we have that the number of terminals of the associated networks is at most k. Therefore, we find that mimicking network construction applied in the merging of networks takes time polynomial in the sum of the sizes of the merged networks. From theorem 5.4 we see that the mimicking networks we produce have at most

$2^{(2^k)} = O(1)$ vertices. As a consequence, for each node computing the mimicking network of the associated network takes $O(1)$ time. We conclude that computing the mimicking network of the original network can be done in $O(\log n)$ time with $O(n)$ work.

Computing an actual flow in network N can be done by processing down the tree. At an internal node x with children x_1 and x_2, the procedure is to calculate a flow in the network that was formed by merging $N(x_1)$, $N(x_2)$, and $N_s(x)$, of desired external flow pattern. From this flow external flow patterns in the constituent networks $N_s(x)$, $N(x_1)$ and $N(x_2)$ can be determined recursively. We conclude that in above situation computing an actual flow in network N can be done in $O(\log n)$ time with $O(n)$ work.

References

[1] R.K. Ahuha, T.L. Magnanti, and J.B. Orlin. Some recent advances in network flows. *SIAM Review,* 33(2):175–219, 1991.

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

[3] H.L. Bodlaender. NC-algorithms for graphs with small treewidth. In *Proceedings of the 14th International Workshop on Graph-Theoretic Concepts in Computer Science.* Springer Lecture Notes in Computer Science, 344:1–10, 1988.

[4] H.L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. Technical Report UU-CS-1996-02, Department of Computer Science, Utrecht University.

[5] H.L. Bodlaender and T. Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. Preprint, 1996.

[6] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM,* 21:201–208, 1974.

[7] B. Chazelle and J. Matoešek. On linear-time deterministic algorithms for optimization problems in fixed dimensions. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms,* 281–290, 1993.

[8] M. Chrobak and K. Diks. Network flows in outerplanar graphs, 1987.

[9] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms,* 21:331–357, 1996.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, 1990.

[11] J. Edmonds and R.M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM,* 19:248–264, 1972.

[12] L.R. Ford, Jr., and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[13] D. Gale. A theorem on flows in networks. *Pacific Journal of Mathematics,* 7:1073–1082, 1957.

[14] L.M. Goldschlager, R.A. Shaw, and J. Staples. The maximum flow problem is log space complete for P. *Theoretical Computer Science,* 21:105–111, 1982. North-Holland Publishing Company.

[15] T. Hagerup, N. Nishimura, J. Katajainen, and P. Ragde. Characterisations of k-terminal flow networks and computing network flows in partial k-trees. *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms,* 641–649, 1995.

[16] Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all-pairs shortest-paths in directed graphs. In *Proceedings 4th Annual ACM Symposium on Parallel Algorithms and Architectures* pp.353–362 Assoc. Comput. Mach., New York 1992.

[17] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley Publ. Co., 1992.

[18] D.B. Johnson. Parallel algorithms for minimum cuts and maximum flow in planar networks. *Journal of the ACM,* 34(4):950–967, 1987.

[19] R. M. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random NC. *Combinatorica,* 6:35–48, 1986.

[20] N. Robertson and P. D. Seymour. Graph Minor. II. Algorithmic aspects of treewidth. *Journal of Algorithms,* 7:309–322, 1986.