Math. Program. 84: 595–640 (1999) / DOI 10.1007/s10107990075b © Springer-Verlag 1999

András A. Benczúr

Parallel and fast sequential algorithms for undirected edge connectivity augmentation*

Received November 1995 / Revised version received July 1998

Abstract. In the *edge connectivity augmentation problem* one wants to find an edge set of minimum total capacity that increases the edge connectivity of a given undirected graph by τ. It is a known non-trivial property of the edge connectivity augmentation problem that there is a sequence of edge sets $E_1, E_2, \ldots$, such that $\bigcup_{i \leq \tau} E_i$ optmially increases the connectivity by τ, for any integer τ. The main result of the paper is that this sequence of edge sets can be divided into $O(n)$ groups such that within one group, all E_i are basically the same. Using this result, we improve on the running time of edge connectivity augmentation, as well as we give the first parallel (RNC) augmentation algorithm. We also present new efficient subroutines for finding the so-called *extreme sets* and the *cactus representation of min-cuts* required in our algorithms. Augmenting the connectivity of *hypergraphs* with ordinary edges is known to be structurally harder than that of ordinary graphs. In a weaker version when one exceptional hyperedge is allowed in the augmenting edge set, we derive similar results as for ordinary graphs.

Key words. graph augmentation – edge connectivity – hypergraphs – randomized algorithms – parallel algorithms – combinatorial optimization

1. Introduction

In the paper we obtain improved algorithms for the undirected edge connectivity augmentation problem by a new insight to the structure of edge sequences giving successive optima. Let $G = (V, E)$ be an undirected graph with integral edge capacities. A *cut* is a partition of the vertex set into two subsets; the value of a cut is the total capacity of edges between the two sets. Cuts with minimum value c are *min-cuts*; c is the (edge) *connectivity* of G. Given a *connectivity increment* τ (or a *target connectivity* $k = c + \tau$), the *edge connectivity augmentation problem* is to find an edge set of minimum total capacity whose addition to G increases its connectivity by τ.

Our most efficient Monte Carlo algorithm has a running time[1] of $\tilde{O}(n^2 \log(U/c))$ where n is the number of vertices and U is the highest edge capacity in the graph. In comparison the deterministic algorithm of Gabow [18] runs in $\tilde{O}(n^2 m)$ time while a very recent algorithm [33] improves this to $\tilde{O}(nm)$. The best algorithm for graphs

A.A. Benczúr: Computer and Automation Institute, Hungarian Academy of Sciences and Department of Operations Research, Eötvös University, Budapest, e-mail: benczur@cs.elte.hu
Supported from EC grant ALTEC-KIT, OTKA grants T-16503, T-16524, T-17580, T-30132 and FKFP grant 0206/1997.

Mathematics Subject Classification (1991): ■ ■ ■

* Preliminary version: Augmenting undirected connectivity in RNC and in randomized $\tilde{O}(n^3)$ time, *STOC–94.*

[1] We use the $\tilde{O}$ (soft-O) notation for asymptotics up to a polylogarithmic factor.

with unit edge capacities with runtime $\tilde{O}(k\tau m)$ is due to Gabow [16]. Based on this work, very recently the Monte Carlo runtime of edge augmentation was improved to $\tilde{O}(n^2)$ [6].

Our new results stem from the following observation. On one hand, it is known that edge connectivity augmentation can be done in strongly polynomial time, independently of the edge capacities or the increment value [12]. On the other hand, for small increments τ, algorithms increasing connectivity by one in τ phases turn out to be more efficient [16]. In particular, these algorithms use min-cut structures that can take advantage of the improved running times of some new min-cut algorithms [15]. However, these algorithms have their running time, seemingly inherently, dependent on τ and hence are not polynomial. In the paper, we improve on these algorithms so that the running time becomes independent of the increment value.

We show that the consecutive augmentation phases can be grouped into $O(n)$ segments, such that within a segment, the augmentation steps are basically identical. This result is of independent theoretical interest, as well as it already yields an $\tilde{O}(nm \min\{n, \tau\})$-time augmentation algorithm. We show that the situation is somewhat analogous to the Ford–Fulkerson augmenting path-algorithm for maxflows: in the original algorithm one finds paths that increase the flow value one by one, hence that algorithm is not polynomial for capacitated graphs. The key to obtain a polynomial algorithm is to show that there is a polynomial sequence of augmenting paths such that saturating flow on them is sufficient for computing the maximum flow. Note that the output of such an algorithm for some flow value contains all other outputs for smaller flow values. Previous known polynomial time augmentation algorithms, on the other hand, do not have this property: a τ_1-edge connected augmentation produced by the algorithm is not necessarily the subgraph of that for some $\tau_2 > \tau_1$. We give an algorithm that can produce an element of a fixed sequence of successive optima, in polynomial time independently of the increment.

By our grouping technique, we can give various very efficient augmentation algorithms. This paper describes the first parallel (RNC) algorithm for this problem. We give randomized sequential algorithms with running time

$$\tilde{O}(n^2 \min\{n, \log \tau/c, \log nU/c\}) ,$$

where c is the original connectivity, τ is the connectivity increment, and U is the maximum edge capacity. The deterministic version runs in $\tilde{O}(nm \min\{n, \tau\})$ time. For an arbitrary value of τ, we can also find an element of the increasing sequence of optima $(\bigcup_{i \le \tau} E_i)$, with running time not depending on τ.

Recently, finding efficient edge connectivity augmentation algorithms became a well-studied area [38, 7, 12, 32, 15, 18]. The algorithms use various min-cut structures and are generally based on maxflow computations. Our algorithms require efficient subroutines for finding two cut data structures: the extreme sets introduced by Watanabe and Nakamura [38] and the cactus representation of Dinitz et al. [10]. We improve on the efficiency of finding the extreme system: we present the first RNC algorithm by a new analysis of Karger's [24] algorithm, as well as a very efficient sequential algorithm based on a representation of near-minimum cuts [3]. By recent breakthroughs in the design of algorithms for finding edge connectivity and min-cuts [23, 33, 15, 26], it turns

out that several min-cut structures are much easier to build than to find even a single source-sink min-cut. Our algorithms take advantage of these results and, in particular, avoid maxflow computations.

The rest of this paper is organized into two independent parts. In the first part, we discuss the theory of edge connectivity augmentation algorithms. We sketch those ideas and analyses of earlier algorithms that we build on. Then two new algorithms are described in Sections 3 and 4. The first algorithm is our strongly polynomial time successive algorithm; the second one runs in RNC. The correctness of these algorithms are proved in Section 5. Section 6 deals with the hypergraph connectivity augmentation problem.

Part II describes sequential, randomized and parallel subroutines required in our algorithms. Section 9 contains cactus algorithms; Section 8 extreme sets algorithms. The running times of the best known augmentation algorithms are given in the concluding section.

1.1. Basic notation

We assume that the input graph have n vertices, and m edges; parallel edges are not allowed and thus $m = O(n^2)$. In the terminology of [22], an algorithm is *polynomial*, if its running time is polynomial in the bit length (the logarithm) of the input numbers and in n. And an algorithm is *strongly polynomial*, if the number of arithmetic operations and the space usage is polynomial in n. The arithmetic operations are addition, comparison, multiplication and division. In the paper, we use the term strongly polynomial in a stronger sense: we require that the arithmetic operations be performed on numbers not larger than a polynomial in n plus the maximum of the input numbers.

For $X \subseteq V$, let $\overline{X} = V - X$. A cut with bipartition to C and $\overline{C}$ is denoted $(C|\overline{C})$. The *value* $d_G(C)$ of a cut $(C|\overline{C})$ is the total weight of the edges connecting C and $\overline{C}$. The *connectivity* c_G of G is the minimum value of any of its cuts. The *k-demand* of a set C is $\text{dem}_G(C, k) = \max\{0, k - d(U)\}$. We omit subscripts G if it causes no ambiguity.

A *subpartition* $\mathcal{P}$ of V is a set of disjoint subsets of V. Two sets C and D are called *intersecting* if neither of $C \cap D$,$C \cap \overline{D}$ and $\overline{C} \cap D$ is empty and *crossing* if in addition $\overline{C} \cap \overline{D} \neq \emptyset$. A set system is *laminar* if it contains no intersecting pair. Sets of a laminar system $\mathcal{F}$ can be viewed as nodes of a tree where X and $Y \in \mathcal{F}$ are joined by an edge if $X \subset Y$ but there is no $Z \in \mathcal{F}$ with $X \subset Z \subset Y$.

Part I: Extreme sets based augmentation algorithms

There are two approaches to solving the augmentation problem. One approach is first described by Cai and Sun [7]: they add augmenting edges to an extra artificial node and then use edge splitting [30] to remove that node. Frank [12], based on this approach, presented the first strongly polynomial time algorithm which solves the (integral) capacitated case in $O(n^5)$ time. Gabow [18] improved Frank's running time to $O(n^2 m \log(n^2/m))$, which was very recently further improved to $O(nm \log n)$ by Nagamochi and Ibaraki [33]. In these algorithms, the computational bottleneck is edge splitting from the extra node. So

far, all known edge splitting algorithms are either based on flow computations [12,18], or their running time depends on the total capacity of the augmenting edges [3].

Our new algorithms are based on a historically earlier approach, when the connectivity is repeatedly increased by one unit until the target is reached. The first edge connectivity augmentation algorithm by Watanabe and Nakamura [38] has this approach. A more efficient version with running time $O(\tau^2 nm)$ was given by Naor et al. [32]; Gabow [16] improved this time to $\tilde{O}(k\tau m)$. All these running times are valid for graphs with unit edge capacities only.

Algorithms of this latter approach [38,32,16] have an additional property that they can find a *successive*, increasing sequence of intermediate optima, for increasing values of target connectivity k. We will pay special attention to algorithms with this successive property. For example Naor et al. [32] notice that such an algorithm can always be used to solve the modified “no target” problem where instead of specifying a connectivity target, the total capacity of edges to be added is bounded and the connectivity should be increased by as much as possible. Unfortunately an algorithm that increases the connectivity one by one is inherently not polynomial. We give an alternate definition of a successive algorithm that is sufficient for example to solve the no target problem but allows polynomial time algorithms:

Definition 1 (Successive augmentation). *An edge augmentation algorithm is* successive *if its output for target k is the subgraph of its output for target $k' > k$.*

We remark that Cheng and Jordán [8] proved that successive algorithms exists for a general class of augmentation problems. The polynomiality of such successive algorithms is not addressed in their paper.

Augmenting by one: the cactus-increase

A main subroutine of our algorithms is the algorithm of Naor et al. [32] to increase edge connectivity by one, for connected graphs. It finds a minimum cardinality edge set E' covering all min-cuts by the cactus representation of all min-cuts [10]. For the purposes of the paper, we call this algorithm the *cactus-increase* subroutine.

A *cactus* is a graph which contains no cut edges and no two cycles with a common edge. In other words, a cactus is built up from a single vertex by recursively joining cycles (of length possibly two!) to existing vertices. The *cactus representation* (Fig. 1) of an (ordinary) connected graph G is a cactus $\mathcal{K}$ such that a partition of V corresponds to the vertex set of the cactus, with an exceptional case that some cactus-vertices may contain the empty vertex set (as in Fig. 1). The min-cuts of the cactus are precisely those which arise by erasing two edges of a cycle. Then the min-cuts of G are precisely the edge sets of G connecting two components of a min-cut of the cactus.

Given the cactus representation, the connectivity of a graph can be increased optimally by one as follows [32]. Each cactus graph has an Eulerian cycle; we fix such a cycle $\mathcal{C}$. In $\mathcal{C}$, the degree two vertices of the cactus are ordered cyclically. By the definition of the representation, each min-cut of G divides $\mathcal{C}$ into two consecutive parts. Hence if we connect all pairs of *opposite* degree two vertices by edges, we add one edge to each min-cut of G.

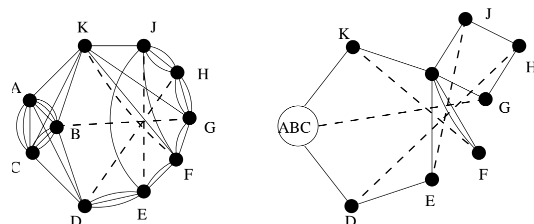

Fig. 1. A graph with connectivity 6 and the cactus representation of its min-cuts. Dashed edges form an optimum augmentation to connectivity 7 of `cactus-increase`

We sketch why the cactus-increase step uses minimal number of edges. It is easy to prove that sets S contained by degree two vertices of the cactus have $d(S) = c$. One edge has to be added to each such set S; and one edge can cover at most two of them. Let there be ℓ such vertices; then at least $\lceil \ell/2 \rceil$ edges are required to increase connectivity. This is equal to the actual number of edges used, proving optimality. It is important to remember that if ℓ is odd, we have to add *two* edges to one of the sets S.

2. A generic augmentation algorithm

Our augmentation algorithms build on the common structure of those of [38,32,16]. These algorithms on one hand use some information about min-cuts to increase connectivity, as it is done in the cactus-increase subroutine; on the other hand, global optimality is reached by using the structure of the so-called *extreme sets*, a higher-value generalization of minimal min-cuts. Next we describe the general structure of these extreme-sets based algorithms.

Extreme-sets based algorithms (see Algorithm 1) increase connectivity in *phases*, until the connectivity is increased by τ. In phase t, an intermediate augmented graph G_{t-1} is further augmented by an edge set E_t; $G_t = G_{t-1} + E_t$. We use subscripts t to refer to properties of G_t: we let $d_t(C)$ be a shorthand for $d_{G_t}(C)$, and let c_t denote the connectivity of G_t. Unlike all previous algorithms, our algorithms will not necessarily have $c_{t+1} = c_t + 1$.

In each phase, the edge set E_t is found in *two steps*. In the *connectivity-increase* step, an edge set E'_t is found that increases connectivity to c_t. The second step is a *lookahead* step. In the first step, we may as well contract all subsets of G_{t-1} not separated by cuts of value at most c_t; the end-vertices of the augmenting edges can be placed anywhere within a contracted subset. It is known [32] that an arbitrary placement of E'_t may cause that the next augmentation step cannot yield an optimum augmentation, if compared to G_{t-1} and not just to G_t. In the lookahead step, the end-vertices of augmenting edges are kept within the same contracted subset, but replaced so as to preserve optimality for further augmenting steps. We start by an example to show that in an optimal augmentation phase, one requires a lookahead step.

Example 1. Let a graph G have six highly connected components $A_1, \dots, A_6$. Let the first and last three A_i be connected by edges of two triangles; let there be a further edge

between A_1 and A_6. G has a single min-cut of value 1 separating the two triangles. To augment connectivity by one, an arbitrary edge may be added connecting the first three and last three A_i. Assume one adds another edge between A_1 and A_6. Then the resulting graph has connectivity 2, with four minimal min-cuts A_2, A_3, A_4 and A_5. Hence another two edges connecting these four sets are necessary to augment the connectivity of G to three. On the other hand, if we augment by one by an edge between A_3 and A_4, say, then another edge between A_2 and A_5 suffice for the augmentation.

□

In the above example, we considered (inclusion-wise) minimal sets with respect to a given demand (indeed, $A_1 \cup A_2 \cup A_3$, $A_4 \cup A_5 \cup A_6$, and $A_2, \dots, A_5$ are all such sets). We define such sets as extreme:

Definition 2 (Extreme sets). *A subset $U \subset V$ is d-*extreme *(as in [38]) if $d(U) = d$ and $d(U') > d$ for all $U' \subset U$. The collection of all extreme sets for all values of d is the* extreme system.

Extreme sets satisfy the following properties. Their collection forms a laminar system. The collection of d-extreme sets for a fixed d forms a subpartition. The subpartition of d-extreme sets, for all possible values of d, define *levels* of the extreme system. For $c = c_G$, the edge connectivity value of G, it is equivalent that a set S is c-extreme, or that S is minimal such that $(S|\overline{S})$ is a min-cut, or that S is contained by a degree two node of the cactus representation. Among extreme sets, c_G-extreme sets are inclusion-wise maximal (but other extreme sets can be maximal as well).

Algorithm 1: extreme-sets based algorithms

```
input: a graph G; the connectivity target k.
  t ← 0; G_0 ← G; c_0 ← c_G
Phase: until c_t = k do
     F_t ← the extreme sets of G_t
     t ← t + 1
     E'_t ← connectivity-increase(G_{t−1})
     E_t ← lookahead(F_{t−1}, E'_t, k)
     G_t ← G_{t−1} + E_t
  end do
output G_t
```

In order to prove the optimality of an extreme-sets based augmentation algorithm, we introduce a potential function based on the system of extreme sets. First we generalize the notion of the demand to laminar systems – the potential will then be the demand of the extreme system $\mathcal{F}_0$ of the input graph. Let the *k-demand of a subpartition* $\mathcal{P} = \{V_1, \dots, V_m\}$, $\mathrm{dem}_G(\mathcal{P}, k)$, be the sum of the demands of individual sets; let the *demand* $\mathrm{dem}_G(\mathcal{F}, k)$ *of a laminar family* $\mathcal{F}$ be the maximum k-demand of a subpartition consisting of elements of $\mathcal{F}$. (Note that this subpartition, as well as the value of the demand, can be found by recursion on the tree corresponding to the laminar system.)

The potential $\mathrm{dem}_G(\mathcal{F}_0, k)$ is proved to give a tight bound for the augmentation cost (for ordinary graphs) by Watanabe and Nakamura [38]. Let the *cost of an edge* be its capacity times its size (two for ordinary edges). Notice that for ordinary graphs, this is *twice* the natural cost measure. The goal of the rest of this section is on one hand to prove the theorem of Watanabe and Nakamura below; on the other hand, to give an optimality criterion for our further algorithms using the general-lookahead step.

Theorem 1 ([38]). *The minimum number of (ordinary) edges needed to augment connectivity to target value $k \geq 2$ is*

$$\lceil \tfrac{1}{2} \max\{\mathrm{dem}(\mathcal{P}, k) : \mathcal{P} \textit{ is a sub-partition}\} \rceil .$$

Theorem 2. *For all t, in phase t of an extreme-sets based augmentation algorithm, let E_t be the augmenting edge set added to G_{t-1} to obtain the next intermediate graph G_t. Let $\mathcal{F}_t$ be the system of extreme sets of G_t. Assume that for all t,*

(2.A) $\mathrm{dem}_{G_{t-1}}(\mathcal{F}_{t-1}, k) - \mathrm{dem}_{G_t}(\mathcal{F}_{t-1}, k)$ *is (at least) the cost of E_t' ($\mathcal{F}_{t-1}$ is fixed as reference!);*
(2.B) each extreme set of G_t is extreme in G_{t-1} as well (note that the requirement is sufficient to hold for the positive demand extreme sets only); and finally that
(2.C) the output graph of the final phase has connectivity k.

Then the output of the augmentation algorithm has optimum cost $\mathrm{dem}_G(\mathcal{F}, k)$. *And if (2.A) holds in all phases except for the last one, and in the last phase τ, the cost of the augmenting edges E_τ is at most $2\lceil \frac{1}{2}\mathrm{dem}_{G_{t-1}}(\mathcal{F}_{t-1}, k)\rceil$, then the output of the augmentation algorithm has optimum cost as in Theorem 1.*

Proof. We consider the case when (2.A) holds in the last phase τ as well. The other case can be proved in exactly the same way, by taking care of the fractional values in the last augmentation phase. Clearly, $\mathrm{dem}_G(\mathcal{F}_0, k)$ is a lower bound on the cost of any augmenting edge set, and in particular on $\bigcup_{t \leq \tau} E_t$. An upper bound on the cost of this edge set, by (2.A), is

$$\sum_{t \leq \tau} \mathrm{dem}_{G_{t-1}}(\mathcal{F}_{t-1}, k) - \mathrm{dem}_{G_t}(\mathcal{F}_{t-1}, k) .$$

By (2.B), $\mathcal{F}_{t-1} \supseteq \mathcal{F}_t$; thus $\mathrm{dem}_{G_t}(\mathcal{F}_{t-1}, k) \geq \mathrm{dem}_{G_t}(\mathcal{F}_t, k)$. By rewriting the above sum,

$$\mathrm{dem}_{G_0}(\mathcal{F}_0, k) - \mathrm{dem}_{G_\tau}(\mathcal{F}_{\tau-1}, k) + \sum_{\tau > t > 0} \mathrm{dem}_{G_t}(\mathcal{F}_t, k) - \mathrm{dem}_{G_t}(\mathcal{F}_{t-1}, k) \leq \mathrm{dem}_G(\mathcal{F}_0, k) - \mathrm{dem}_{G_\tau}(\mathcal{F}_{\tau-1}, k) .$$

Since the k-demand of the final graph equals 0 by (2.C), the claim is proved. □

We prove Theorem 1 in the next subsection. We describe a lookahead step and prove that an extreme sets based algorithm with cactus-increase and this lookahead step achieves optimality. The proof is based on the theorem of Naor et al. [32] that (2.B) holds for the edge set given by the cactus-increase step. For completeness, we prove this fact in Section 5.

2.1. The general-lookahead step

Next we describe our *general-lookahead* step (see Algorithm 2), a generalized form of the lookahead step of Naor et al. [32] (the original form is also given in Section 2.2). As Example 1 indicates, the lookahead step is necessary to ensure optimality. In phase t, the connectivity-increase step adds an edge set E'_t. The lookahead step takes each end-vertex v' in E'_t and replaces it by another v to obtain an edge set E_t (in particular, $|E_t| = |E'_t|$).

In Algorithm 2 we process edge-endvertices v_i of E'_t as follows. Let $W_i \in \tilde{\mathcal{F}}$ be the inclusion-wise minimal set containing v_i with $\tilde{\text{dem}}(W)$ maximum. Then W_i is recursively replaced by one of its inclusion-wise maximal extreme subsets, until the final W_i has either demand 0 or consists of a single vertex w_i. In E'_t, v_i is then replaced by w_i. We also update the values of $\tilde{\text{dem}}(W)$ by subtracting one for each immediate extreme set W_i visited.

Algorithm 2: general-lookahead

```
input: an edge set E'_t;
         a laminar family F~ with demands dem~(W) for W ∈ F~;
         (the target value k is contained implicitly in dem~;
         F~ contains all extreme sets of G_{t−1}.)
for all edge-endvertices v_i in E'_t do
  W_i ← the maximal element of F~ containing v_i
  repeat until W_i is a single vertex w_i or dem~(W_i) = 0
    select a maximal extreme subset W ⊂ W_i with dem~(W) > 0
    dem~(W_i) ← dem~(W_i) − 1
    W_i ← W
  end repeat
  replace v_i by w_i in E'_t
end do
output E'_t and all dem~(W)
```

Theorem 3. *Assume that E_t is obtained from E'_t by the general-lookahead algorithm where $\tilde{\mathcal{F}}$ is a laminar family containing all extreme sets of G_{t-1} and the input* $\tilde{\text{dem}}(W)$ *is equal to* $\text{dem}_{G_{t-1}}(U, k)$ *for all $W \in \tilde{\mathcal{F}}$. Assume that (i) no edge of E'_t has two endvertices in the same set of $\tilde{\mathcal{F}}$; furthermore that (ii) no maximal extreme set U has $d_{E'_t}(U) > \tilde{\text{dem}}(U)$. Then (2.A) of Theorem 2 is satisfied; furthermore the output* $\tilde{\text{dem}}(W)$ *is equal to* $\text{dem}_{G_{t-1}+E_t}(U.k)$ *for all $W \in \tilde{\mathcal{F}}$.*

In the proof of this theorem, we use the notion of the recursive demand of a set in a laminar family. Notice that the maximum demand subpartition of extreme sets can be found by starting with minimal extreme sets, and replacing them by their parents in the laminar family if this increases the total demand. Formally, in a laminar family $\mathcal{F}$, let the *recursive demand* $\text{rdem}(U|\mathcal{F}, k)$ be identical to $\text{dem}(U, k)$ for all minimal $U \in \mathcal{F}$.

And given $\mathrm{rdem}(U_i|\mathcal{F}, k)$ for all maximal subsets U_i of $U \in \mathcal{F}$, let

$$\mathrm{rdem}(U|\mathcal{F}, k) = \max\{\mathrm{dem}(U, k), \sum_i \mathrm{rdem}(U_i|\mathcal{F}, k)\} .$$

Clearly, the demand of a laminar family $\mathcal{F}$ as we defined before is the sum of the recursive demands of its maximal elements.

Proof. Let an input vertex v_i be replaced by a vertex w_i in the output; let $W_i \in \mathcal{F}$ be maximal with $v_i \in W_i$. Then $\tilde{\mathrm{dem}}(U)$ decreases by one for all extreme sets U $w_i \in U \subset W_i$ with positive initial value of $\mathrm{dem}(U, k) = \tilde{\mathrm{dem}}(U)$. For all $U \in \tilde{\mathcal{F}}$ the value of $\tilde{\mathrm{dem}}(U)$ remains non-negative throughout the procedure by (ii); by (i) its decrease is equal to the number of edges added to U. Thus the output $\tilde{\mathrm{dem}}(U)$ becomes the new demand value as required.

Finally (2.A) follows if we show that $\mathrm{rdem}(W_i|\tilde{\mathcal{F}}, k)$ decreases by $d_{E_t}(W_i)$ for all maximal extreme W_i. Let us define $\tilde{\mathrm{rdem}}(U)$ in the same way as $\mathrm{rdem}(U|\tilde{\mathcal{F}}, k)$ by using the values of $\tilde{\mathrm{dem}}(U)$ instead of $\mathrm{dem}(U, k)$. By the previous argument both the initial and the final values of $\tilde{\mathrm{dem}}(U)$ and $\mathrm{rdem}(U|\tilde{\mathcal{F}}, k)$ agree. Hence we prove for $\tilde{\mathrm{rdem}}(U)$; we show that its value decreases by one whenever a vertex w_i is processed. Let us consider all intermediate $U \in \tilde{\mathcal{F}}$, $w_i \in U \subseteq W_i$ in the inclusion-wise increasing order. The last such $U \in \tilde{\mathcal{F}}$ with $\tilde{\mathrm{dem}}(U) > 0$ has $\tilde{\mathrm{dem}}(U') = 0$ for all $U' \subset U$; hence $\tilde{\mathrm{rdem}}(U) = \tilde{\mathrm{dem}}(U)$ and this value decreases by one when w_i is added. For all remaining U both $\tilde{\mathrm{dem}}(U)$ as well as (by induction) $\tilde{\mathrm{rdem}}(U')$ decreases by one; thus we get that $\tilde{\mathrm{rdem}}(U)$ decreases by one as needed.

□

Finally we prove the main Theorem 1. We require the following property of extreme sets:

Lemma 1. *Any vertex set U contains an extreme set $W \subseteq U$ with $d(W) \leq d(U)$. In particular a graph with connectivity k must have k-extreme sets.*

Proof. Let W be an inclusion-wise minimal subset of U with $d(W) \leq d(U)$. Then W is extreme.

□

Proof. We prove that an extreme-sets based algorithm with cactus-increase and general-lookahead steps satisfies (2.A–C) of Theorem 2. Since by Lemma 1 a graph has connectivity k if all of its extreme sets U have $d(U) \geq k$, (2.C) holds by the definition of Algorithm 1. (2.A) holds in all except the last phase by Theorem 3. It is proved in [32] that an arbitrary edge set found by cactus-increase satisfies (2.B); we give a proof of this fact for completeness in Section 5.

The only remaining case is the last phase when Theorem 3 cannot be applied: it is possible that the increment is one and there are an odd number of maximum-demand extreme sets. Then two edges are added to some extreme set; however, its demand is only one, violating (ii) of Theorem 3. In this scenario, the exceptional case of Theorem 2 still applies.

□

2.2. *Obtaining a successive sequence of optima: balanced-lookahead*

An algorithm using the general-lookahead step is not a successive algorithm in our definition. Although it obtains an optimum solution for the given target k, the two output graphs for targets $k_1 < k_2$ are not necessarily subgraphs of one another. In contrast, the algorithm of Naor et al. [32] is a successive algorithm using the cactus-increase step together with a more specific lookahead routine. Here we describe this *balanced-lookahead* routine that we also use in our successive algorithm. It is noticed in [32] that a successive algorithm can solve the so-called No Target Problem when the number of edges we may add to the input graph is bounded, and we want to increase the connectivity by as much as possible.

By using the cactus-increase step, we may obtain a successive algorithm by requiring for all target values the lookahead step proceed in the same way. In the general-lookahead step, the choice of the augmenting edges depend on demand values and hence on the choice of the target. The balanced-lookahead step fixes this "flaw".

The *balanced-lookahead* step is a specific implementation of general-lookahead. When an extreme set W_i has to be replaced by another W, we choose some of its extreme subsets W with $d(W)$ *minimum*. Then the value of $d(W)$ is increased by one and the procedure is repeated as in general-lookahead. Note that this is an implementation of the general-lookahead subroutine. This algorithm continues selecting extreme subsets of possibly already zero-demand sets, however in that case, the choice is arbitrary in general-lookahead.

3. An efficient successive algorithm

In this section we present our strongly polynomial time successive augmentation algorithm. Recall that we call an algorithm successive if its output for target k is the subgraph of its output for target $k' > k$. We aim to use the general framework of extreme-sets based algorithms that increase connectivity in phases. Our idea is that we compute the augmenting edges of several phases ("groups") together, without recomputing any information about the intermediate augmented graphs. By this idea, the running time of our algorithm becomes independent of the connectivity increment.

We get to our successive polynomial-time algorithm in two steps: in this section, we describe a simple connectivity-increase step that increases connectivity by *two* by requiring the knowledge of the extreme sets only; then in the next subsection, we show that the phases of this new algorithm can be divided into $O(n)$ groups such that within one group, the same edges can be added by the connectivity-increase step of each phase. This "grouping" is possible since the extreme system is robust to the addition of the augmenting edges, as it is also indicated by property (2.B) of Theorem 2. The final implementation details of how to find the groups and how to perform lookahead is in the last subsection.

The main idea of our connectivity-increase step is that an arbitrary *cycle of weight* $1/2$ connecting all c-extreme sets increases connectivity by one. This step is thus dependent only on the maximal extreme sets of the intermediate graph. In comparison, the cactus-increase step requires the knowledge (the representation) of all min-cuts of

the intermediate augmented graphs at each call. However, we do not want to allow fractional weights in the augmenting edge set. In case there are no $(c+1)$-extreme sets (or all $(c+1)$-extreme sets are subsets of c-extreme ones and at least one new edge is added to each of them), we may add the same cycle with weight *one*, to increase connectivity by *two*. However, in the following example, this step cannot be applied in an arbitrarily long sequence of augmentation phases:

Example 2. In a starting phase t, let there be an odd number of c_{t-1}-extreme sets $U_1, \dots, U_{2\ell-1}$, and a single maximal extreme set U_0 with $d(U_0) = c_{t-1}+1$. We cannot add a cycle connecting all c_{t-1}-extreme sets, since that does not decrease the demand of U_0. Thus we have to increase connectivity by one by, say, cactus-increase. Let E''_t be an edge set increasing connectivity by one; then wlog $d_{E''_t}(U_1) = 2$, while $d_{E''_t}(U_i) = 1$ for $i \geq 2$. Hence in the next phase, the c_t-extreme sets are $U_0, U_2, U_3, \dots, U_{2\ell-1}$, while U_1 is (c_t+1)-extreme. In subsequent phases, U_0 and U_1 keep exchanging roles, and we may never add cycles connecting maximum-demand extreme sets.

□

Now we give the formal description of the cycle-increase step. The solution to handle the situation as in the example is to add a cycle connecting all the U_i, *including* the $(c+1)$-extreme one – whenever possible. For the sake of simplicity, we will be slightly more restrictive in specifying the cases when cycle-increase does not apply than we need to. Also notice that the augmenting edge set E' has to include an edge set E'' increasing connectivity optimally by one, in order to obtain a successive algorithm.

Definition 3 (cycle-increase). *Let G be the input to the connectivity-increase step, let the connectivity of G be c. Cycle-increase cannot be applied in the following cases (and the augmenting edge set is given by cactus-increase, say): if G has*

- *a maximal extreme set U with* $d(U) = c+2$*;*
- *more than one maximal extreme sets U with* $d(U) = c+1$*;*
- *a maximal extreme set U with* $d(U) = c+1$*, when there are an even number of c-extreme sets;*
- *a maximal extreme set U with* $d(U) = c$ *or* $c+1$*, with more than one extreme subsets* U' *that have* $d(U') = d(U)+1$*.*

Assume that we are not in any of the above cases. Let E'' *be a minimum cardinality edge set that increases G's connectivity by one (*E'' *may, for example, be the output of cactus-increase). Then the output edge set* E' *of cycle-increase is a cycle* E' *with* $E'' \subset E'$ *that connects all maximal extreme sets U with* $d(U) = c$ *or* $c+1$*. Furthermore, if such an extreme set U has an extreme subset* U' *with* $d(U') = d(U)+1$*, then at least one endvertex of* E' *has to be placed to* U'*. (This latter condition is satisfied by using the balanced-lookahead step.)*

□

Theorem 4. *An extreme-sets based augmentation algorithm with cycle-increase and balanced-lookahead steps finds an optimum cost augmentation of the input graph to connectivity k, belonging to an increasing sequence of graphs for different values of k (successive algorithm).*

Proof. By the definition of cycle-increase and balanced-lookahead, the algorithm is successive. To prove optimality, we show that the conditions of Theorem 2 hold: (2.A) follows by Theorem 3, (2.C) is trivial. We prove (2.B) separately in Section 5.

□

3.1. Groups of phases: definition and bound

We continue the analysis of our successive algorithm using the cycle-increase and balanced-lookahead steps. Our next goal is to show that the augmentation phases can be divided into $O(n)$ consecutive *groups*, such that within a group, the cycle-increase step may add the same augmenting edge set to the input graph. We define groups by also ensuring that the balanced-lookahead step behaves in a similar way within a group of phases. However, it cannot be required that the augmenting edge sets within a group remain the same after processed by balanced-lookahead – all we achieve is that an efficient $O(n^2)$ time algorithm can compute the entire augmenting edge set of the group as it were given by the balanced-lookahead step. The details of this algorithm are in the next subsection.

By definition, we know that the cycle-increase step depends on the system of maximum, maximum-1 and maximum-2 demand extreme sets. The next example describes how balanced-lookahead depends on the demand differences of the extreme system.

Example 3. Let us consider those phases of the algorithm where balanced-lookahead processes a fixed extreme set W. Let $U_1, \dots, U_\ell$ be the extreme subsets of W with $d_t(U_i) = d$ minimum. Then in ℓ subsequent such phases, W is replaced first by U_1 and this increases $d(U_1)$ to $d + 1$, then the same thing happens with U_2, etc. Finally, all $d_{t+\ell}(U_i) = d + 1$, which will be the new minimum value. The algorithm is unchanged as long as there are no other d and $(d + 1)$-extreme sets than the U_i.

□

In summary, the steps of our algorithm depend on the properties of extreme sets described by the following functions (see Fig. 2). If $U \in \mathcal{F}_t$, i.e. U is extreme in G_t, then let

- $\text{children}_t(U)$ denote the (inclusion-wise) maximal extreme subsets of U;
- $\text{maxchildren}_t(U)$ denote the maximum demand extreme subsets of U;
- $\text{max–1children}_t(U)$ ($\text{max–2children}_t(U)$) denote the collection of extreme subsets with demand maximum or maximum–1 (or maximum–2);

We use these functions with argument (V) to denote extreme subsets of the entire graph.

By Example 3, we extract the following definition. We say that the two intermediate augmented graphs G_t and G_{t+1} of two consecutive phases are *structurally equivalent* if

(i) $\mathcal{F}_t = \mathcal{F}_{t+1}$;
(ii) $\text{max–2children}_t(V) = \text{max–2children}_{t+1}(V)$; and
(iii) for all $U \in \mathcal{F}_t$, $\text{max–1children}_t(U) = \text{max–1children}_{t+1}(U)$.

If (i–iii) holds, both cycle-increase and balanced-lookahead behaves in the same way in phases t and $t + 1$. We define a *group of phases* as the longest sequence of phases where all consecutive intermediate graphs are structurally equivalent.

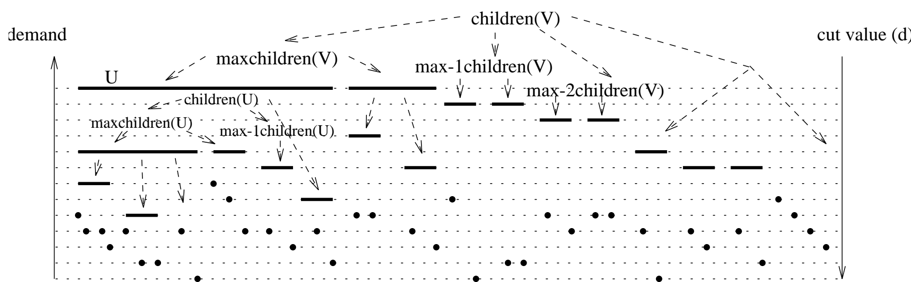

Fig. 2. The laminar system of extreme sets. Thick lines denote vertex sets, dots denote single vertices. Dashed lines illustrate pointers of the extreme system. The two vertical arrows point in the direction of increasing values (of demand and degree)

Lemma 2. *All augmentation phases can be divided into at most $O(n)$ groups.*

Proof. We bound the number of phases when (i), (ii) or (iii) is violated separately. By (2.B) of Theorem 2 (which we prove in Section 5), (i) is violated only if there is a $U \in \mathcal{F}_t - \mathcal{F}_{t+1}$. This may only happen once for each initial extreme set, $O(n)$ times altogether. The number of phases (ii) is violated can be bounded similarly. By the choice of the cycle-increase step, $U \in \text{max–2children}_t(V) - \text{max–2children}_{t+1}(V)$ only if $U \in \mathcal{F}_t - \mathcal{F}_{t+1}$. The same holds for the cactus-increase step; notice even though there may be an exceptional set $U \in \text{maxchildren}_t(V)$ whose demand is decreased by two, such a set remains in $\text{max–1children}_t(V)$. Hence (ii) may be violated at most twice for each initial extreme set: once when it first becomes the member of max–2children_t, and once when it becomes non-extreme.

It is less straightforward to give a tight analysis for (iii). As before, $U' \in \text{max–1children}_t(U) - \text{max–1children}_{t+1}(U)$ only if U' becomes non-extreme. However, U' may get added to $\text{max–1children}_t(U)$ for each $U \supset U'$. By this analysis, we only get an $O(n^2)$ bound.

We bound the number of phases where (iii) is violated by a potential function argument. For an extreme set U, let $q(U)$ be the number of distinct values $d(U_j)$ take for $U_j \in \text{children}(U) - \text{max–1children}(U)$. Let $p(U) = 1$ for all minimal extreme sets. For a general extreme U, let

$$p(U) = 1 + q(U) + \sum_{U_i \in \text{children}(U)} p(U_i) \; .$$

Finally, let p be the sum of $p(U)$, for all maximal extreme U.

Obviously, p remains the same if (i) and (iii) holds. Whenever (i) is violated, an extreme set U becomes non-extreme; p decreases then since each extreme U has its own contribution of at least one to p. Initially, $p = O(n)$. We show that p decreases if (iii) is violated. By the definition of balanced-lookahead $\text{max–1children}(U)$ may only increase; this happens if the minimum degree of an extreme subset of U increases and reaches within distance one from the extreme sets $U_j \in \text{children}(U) - \text{max–1children}(U)$ with $d(U_j)$ minimum (all U_j) are added to $\text{max–1children}(U)$ then). But in this case $q(U)$ decreases by one. The proof is complete.

□

We complete the description of groups by proving that we have to call cactus-increase (or some other routine increasing connectivity by one) only a constant times in a phase. By definition, the output E'_t of cycle-increase should contain an edge set E''_t increasing connectivity optimally by one. E''_t might have to be computed individually for each phase, hence we cannot take advantage of the simple structure of the groups. By the next lemma, it turns out that E''_t may be the same for all phases within a group. We also show below that each group may have no more than two initial phases when cycle-increase does not apply. This completes the description of a group.

Lemma 3. *Within a group of phases, cycle-increase applies for all phases except possibly the first two phases. Let phases $t, t+1, \ldots, t+\tau_0$ belong to the group and let cycle-increase apply for them. Let $E''_t \subset E'_t$ be the output of cactus-increase on G_{t-1}. Then E''_t increases the connectivity of $G_{t+k} = G_{t-1} + kE'_t$ by one, for all $k \leq \tau_0$.*

Proof. To prove the first claim, notice that after two phases in which the connectivity of the initial graph G_{t-1} increased by one, all maximal extreme sets W with $d_t(W) \leq c_t + 2$ have $d_{t+2}(W) = c_{t+2} = c_t + 2$, with at most one exceptional $d_{t+2}(W_0) = c_{t+2} + 1$. Hence if the next phase is in the same group (i.e. there are no new sets in max–2child(V)), then cycle-increase applies.

To prove the second claim, notice that by assumption on τ_0, all min-cuts of $G_{t-1}+kE'_t$ separate maximum or maximum+1 demand extreme sets. However, kE'_t adds $2k$ edges to each cut with this property. Hence the min-cuts of $G_{t-1} + kE'_t$ are all min-cuts of G_{t-1}, proving the claim.

□

3.2. Processing phases within a group

In this section, we complete the description of our successive algorithm by showing that the augmenting edge set of an entire group of phases can be computed in $O(n^2)$ time. Within the same time, we also show how to compute the maximum number of phases that remain within the same group. Together with Lemma 2 and the fact that the intermediate extreme system does not have to be recomputed ((2.B) of Theorem 2), this shows:

Theorem 5. *The augmentation algorithm with cycle-increase and balanced-lookahead requires the computation of the initial extreme system, cactus representations for $O(n)$ intermediate graphs. All remaining steps of the algorithm require an additional $O(n^3)$ time.*

□

Before giving our algorithm to process a group of phases, we give some insight into the structure of successive optima. We fix a group and assume that all intermediate graphs are structurally equivalent. A statement one would obviously like to have is that the augmenting edges are the same or have low length periods. For example, if the same cycles could be added, that corresponds to a period of length two. Surprisingly, this statement fails because of the effect of balanced-lookahead. Our next example shows that even without a structural change in the extreme system, the sequence of the final vertices selected by balanced-lookahead can have *exponentially long* periods!

Example 4. Let the vertex set of G be partitioned into ℓ maximum demand extreme sets $U_1, \dots, U_\ell$. Let $p_1, \dots, p_\ell$ be the first ℓ prime numbers; let U_i consist of p_i vertices, each with the same small demand. Then the augmenting edges connecting U_i to U_j will change with a period of length $p_i p_j$. The shortest period length of the entire augmenting edge set is thus $\prod p_i$, which is exponentially large in n.

Within a group of τ phases, our algorithm will compute the multiplicities of each possible $m = O(n^2)$ edges in the augmenting edge set. Because of the exponentially large period lengths, we have to be careful with the arithmetic model we use. Strongly polynomiality, in general (as in [22]), allows arithmetic operations with, say, numbers of n times larger size than any number in the input. We shall be more restrictive and require all arithmetic operations be performed on numbers not exceeding the maximum of the connectivity target value k (which is part of the input) and n^2.

3.2.1. Task 1. First we show how to find the largest value τ_0 such that phase $t + \tau_0$ is in the same group as phase t. We find τ_0 by an upward recursion over the extreme system: we define $\tau(U)$ as the maximum number of edges that balanced-lookahead can add to a fixed extreme set U by keeping U and its descendants structurally equivalent. In the discussion below, U will always denote the current extreme set; $U_1, \dots, U_{\ell'}$ its extreme subsets with maximum demand; and $U_{\ell'+1}, \dots, U_\ell$ the maximal ones with demand less by one. The balanced rule selects U_i in the increasing order of i.

The value of $\tau(U)$ can be initialized to be infinite for minimal extreme sets (i.e. single vertices). Given $\tau(U_i)$ for $i \le \ell$ as above, $\tau(U)$ is the minimum of the following three upper bounds:

- no more than $\tau(U_i)$ edges can be added to any U_i: let $m = \min\{\tau(U_i) : i \le \ell\}$, then $\tau(U) < m\ell + \min\{i : \tau(U_i) = m\}$;
- $d(U) < d(U_i)$ must remain valid as long as U is extreme: $d_{G_t}(U) + \tau(U) < d_{G_t}(U_1) + \lceil \tau(U)/\ell \rceil$; and finally
- for all maximal extreme subsets U' of U distinct from the U_i, $d(U') > d(U_{\ell'+1})$ must hold:

$$d_{G_t}(U') > d_{G_t} + \lceil \frac{\tau(U) - \ell'}{\ell} \rceil \, .$$

Provided $\tau(U_i)$ as above is known for $i \le \ell$, $\tau(U)$ can be computed in $O(n)$ time. Hence all τ-values can be found in $O(n^2)$ time. The value of the maximum increment with structurally equivalent intermediate graphs, τ_0, is the minimum of $\tau(U)$ for the maximum and maximum-1 demand extreme sets, and the number of phases before a new maximal extreme U' attains $d(U') = c_t + 2$. This number is $\min\{d_{G_t}(U') - c_t - 1 : U'$ is maximal extreme with $d_{G_t}(U') > c_t + 2\}$.

3.2.2. Task 2. Next we turn to our second task: we compute the set of augmenting edges added in a group of phases. We proceed by downward recursion. For each pair of extreme sets U and W, let times(U, W) be the total number of augmenting edges between U and W. Our goal is to compute times(w_i, w_j), where the w_i and w_j are single vertices. We initialize times$(U, W) = 0$ or times$(U, W) = \tau_0/2$ for pairs of maximal extreme sets, according to the cycle-increase step.

Let us consider the recursive evaluation step now. For a pair U, W, let balanced-lookahead select, in order, the extreme subsets $U_1, \ldots, U_{\ell_1}$ and $W_1, \ldots, W_{\ell_2}$, respectively. Now times(U_i, W_j) is determined as follows. Let h be the least common multiple of ℓ_1 and ℓ_2; h divides $\ell_1 \ell_2$. Then we determine the sequence $(U_{i_1}, W_{j_1}), \ldots, (U_{i_h}, W_{j_h})$, in which the edges are selected. Finally,

$$\text{times}(U_{i_t}, W_{j_t}) = \lfloor (\text{times}(U, W) - t + 1)/h \rfloor .$$

The augmenting edge set is the collection (w_i, w_j) with weight times(w_i, w_j), where the w_i are extreme sets with single vertices (i.e. minimal ones).

Finally we analyze the running time of the procedure. Since times(U) decreases as the size of U decreases, we never use arithmetic operations on values exceeding $\max\{n^2, \tau_0\}$. And for each pair of extreme sets U, W, we perform a constant number of arithmetic operations to determine times(U, W). Thus the running time is $O(n^2)$.

4. A more efficient non-successive algorithm

In this section, we drop the successivity requirement to obtain an improved sequential algorithm. We also describe the first known parallel augmentation algorithm. The algorithm will require the cactus-increase step at most once, in the last augmentation phase; in other words the "parity-like" problems in the previous algorithm are all postponed to the end. Based on this algorithm, in Section 6 we also obtain a hypergraph-augmentation algorithm using ordinary (two-vertex) augmenting edges, with at most one exceptional hyperedge in the last phase.

The basic goal of our next algorithm is to avoid cactus computations that require the construction of current intermediate graphs which are then sequentially dependent on previous cactus computations. Our new ideas will ensure all phases be dependent only on the system of extreme sets. While we are no longer restricted to successivity, notice that in an algorithm using cycle-increase, we still need cactus computations at the end of each group if its total connectivity increment is *odd*.

We give separate ideas to handle the last phases of a group, for the different possibilities why a group may end. There are two (main) reasons why a new group has to be started: either a maximal extreme set becomes of maximum demand, or a maximum demand extreme set has more than two maximum-1 demand extreme subsets.

Our first idea is that we connect *all* maximal extreme sets with demand at least *two* by a cycle – not only the maximum demand ones. It is possible to prove that this idea gives a correct connectivity-increase step (by modifying the proof in Section 5 for cycle-increase). However, even then there can be $\Omega(n)$ groups, each of which may possibly have an odd connectivity increment.

Our next aim is to add two edges to a maximal extreme set U even if it has more than two extreme subsets $U_1, \ldots, U_\ell$ with $d_t(U_i) = d_t(U) + 1$, in a phase t. We know that if U has maximum demand in phase t, then we have $c_{t+1} \leq c_t + 1$. We relax the goal to increase the connectivity by two; instead, we only require (2.B) of Theorem 2 – this still implies optimality. In the next example, however, (2.B) always fails:

Example 5. Let there be a single maximal extreme set U with extreme subsets $U_1, \dots, U_\ell$, $\ell = 3$ as above. Let $d_t(U) = c_t$. Then all extreme sets $W \in \mathcal{F}_t$ not contained by U have $d_{t+1}(W) \geq c_t + 2$; $d_{t+1}(U) = c_t + 2$; finally wlog $d_{t+1}(U_1) = d_{t+1}(U_2) = c_t + 2$. Hence U_3 is the only element of $\mathcal{F}_t$ with $d_{t+1}(U_3) = c_t + 1$. By Lemma 1, the complement of U_3 must contain a $(c_t + 1)$-extreme set; this set is in $\mathcal{F}_{t+1} - \mathcal{F}_t$.

□

The above example motivates our second idea. We want to make sure that there are $(c_t + 1)$-extreme sets in phase $t + 1$ that are extreme in phase t as well. Let us consider a cycle connecting all maximal extreme sets. Let us select two maximum demand ones and remove the edge connecting them; these sets will be $(c_t + 1)$-extreme in phase $t + 1$. We obtain our connectivity-increase step by connecting maximal extreme sets in this path in a particular order, as described next:

Definition 4. *The* path-increase *step is as follows. Let $U_1, \dots, U_\ell$ be the (inclusion-wise) maximal extreme sets of G_{t-1} (see Fig. 2) with k-demand at least two, such that*

$$d(U_\ell) = d(U_1) \leq d(U_2) \leq \dots \leq d(U_{\ell-1}) \; .$$

Then let E'_t be a collection of edges e_i for $i = 1, \dots, \ell-1$, where e_i connects U_i to U_{i+1}. Path-increase applies unless the connectivity increment is one*, when cactus-increase has to be used.*

□

Theorem 6. *An extreme-sets based augmentation algorithm with path-increase and general-lookahead steps finds an optimum cost augmentation of the input graph.*

Proof. We show that the conditions of Theorem 2 hold: (2.A) follows by Theorem 3, (2.C) is trivial (we terminate the algorithm when the intermediate graph has connectivity k). We prove (2.B) separately in Section 5.

□

We complete this subsection by showing that if a maximal extreme set is ever selected to receive a single edge by path-increase (U_1 or U_ℓ in Definition 4), then we may keep that set as long as it remains extreme. Notice that this fact is crucial in handling groups of phases. We prove a slightly stronger lemma:

Lemma 4. *Let E'_t and $E'_{t'}$ be the edge sets selected by path-increase in phases t and $t' > t$. Let U_1 and U_ℓ be the maximal extreme with $d_{E'_t}(U_1) = d_{E'_t}(U_\ell) = 1$. Let U'_1 and $U'_{\ell'}$ be these sets in phase t'. Then by appropriately breaking ties in path-increase, we may assume $U'_1 \subseteq U_1$ and $U'_{\ell'} \subseteq U_\ell$. Furthermore, if t and t' are in the same group, we have equality.*

Proof. It suffices to show the claim for $t' = t + 1$. By definition, no extreme set has demand more than U_1 and U_ℓ in G_{t-1}; furthermore all non-maximal extreme sets have strictly greater demand. Hence in G_t, U_1 and U_ℓ have maximum demand. We are done if U_1 and U_ℓ are still extreme; otherwise, they contain another extreme U'_1 and $U'_{\ell'}$ with the same (maximum) demand.

□

4.1. Groups of path-increase and lex-lookahead

In the previous section, we described path-increase as a connectivity-increase step that enables us to increase connectivity by the knowledge of the extreme system only. Now we describe a simple implementation of the general-lookahead step that makes the grouping technique (Section 3.1) particularly simple. In the discussion, we also fix the target value k and always talk about k-demands.

The *lex-lookahead* step is the following implementation of general-lookahead. Whenever an extreme set W is replaced by its extreme subset, we choose the *lexicographically first* such set with positive demand. If W has no positive demand extreme subsets, then we select the same vertex of W that was selected the previous time lex-lookahead processed W. We assume that the vertices of G are numbered by a depth-first search of the tree corresponding to the system of (initial) extreme sets $\mathcal{F}$ (note the vertices of G are leaves of this tree). We remark that this lookahead step does not yield successive algorithms.

First we prove that if processed by lex-lookahead, the augmenting edge set of each phase within a group is the same – recall the augmenting sets could even have exponentially long periods when processed by balanced-lookahead. We in fact show a stronger property: under the particular numbering of the vertex set we required, lex-lookahead selects the same vertex v in an interval of phases:

Lemma 5. *Assume that the vertices of G are numbered by a DFS ordering of the system of (initial) extreme sets $\mathcal{F}$. Let us consider an extreme-sets based algorithm that uses lex-lookahead and satisfies (2.B) of Theorem 2. Then the phases when an augmenting edge is added to a vertex v form an interval.*

Proof. Assume an edge is added to vertex v in phase t. Let W be the maximal extreme set of phase $t+1$ with $v \in W$; assume W has demand at least one and hence another edges are added to it in phase $t+1$. Assume that one of these edges is added to some $w \neq v$, $w \in W$. Then the claim follows if we show that no further edges may be added to vertex v. To show, let $U \subseteq W$ be the minimal extreme set of phase $t+1$ containing both v and w. By (2.B), U is extreme in phase t as well; let U' and U'' be its lexicographically first extreme subsets with positive demand in phases t and $t+1$, respectively. Clearly $w \in U''$; to see $v \in U'$ notice that lex-lookahead is based on a depth-first search numbering of the tree corresponding to the extreme system and is hence not affected by deletions of sets. We get $U' \neq U''$ by the maximality of U. The subset U'' may not exist; in this case all further edges inside U are added to v by the definition of lex-lookahead. Otherwise again by definition the demand of U' (as well as any of its subsets) must be 0 and no further augmenting edge may be added to U'.

□

Theorem 7. *In a sequence of consecutive steps when the positive-demand extreme sets of the intermediate graphs are the same, we may add the same augmenting edges to G. The time to find one such edge set is $O(n)$. Hence the augmentation algorithm with path-increase and lex-lookahead finds an optimum augmentation by computing the initial extreme sets, a cactus representation of an intermediate graph, and using $O(n^2)$ extra (sequential) time.*

Proof. Recall that consecutive phases when the system of extreme sets does not change are called groups. Assume we know the number of phases τ within a group. Path-increase can be assumed to select the same edge set E'_t by Lemma 4; the set E_t given by lex-lookahead will also be the same over the phases by Lemma 5. Hence we may choose τE_t as the augmenting edge set of the entire group. Lex-lookahead can be performed in $O(n)$ time by depth-first search on the system of extreme sets. Hence the augmenting edge set of the group can be found in $O(n)$ time. The number of groups is bounded by $O(n)$, the number of initial extreme sets.

We show how to compute τ, the number of phases in a group starting at phase t. Let U be an extreme set with $d_{E_t}(U)$ equal 1 or 2. Let U_1 be the lexicographically first maximal extreme subset of U; let U_0 be the maximum demand one distinct from U_1. Assuming that U_1 is extreme in phase $t' + t$, $d_{t'+t}(U) = d_{t-1}(U) + (t' + 1)d_{E_t}(U)$. By Lemma 1, if U is non-extreme, it has an extreme subset U' with $d_{t'+t}(U') \leq d_{t'+t}(U)$. Thus by construction, U becomes non-extreme in phase t' exactly if $d_{t'+t}(U_0) = d_{t'+t}(U)$; i.e. $t'(U) = (d_{t-1}(U_0) - d_{t-1}(U))/d_{E_t}(U)$. Then τ is the minimum of $t'(U)$, over all extreme U with $d_{E_t}(U) \geq 1$. This value is found by depth-first search again. Note here that this value may be fractional when $d_{E_t}(U) = 2$; this may mean exceptional cases when the two edges of phase $\tau + t$ get added to distinct vertices. The number of such exceptional phases is not more than $O(n)$, the total number of groups.

□

4.2. Parallel implementation

We implement our extreme-sets based algorithm with path-increase and lex-lookahead routines on a PRAM. In the implementation we ignore details about EREW/CRCW, number of processors and steps other than the fact that there are a polynomial number of processors and the time used is polylogarithmic. We assume that the initial extreme system $\mathcal{F}$ is given and that we can compute the cactus representation in the final phase. We give parallel algorithms for these tasks in Part II. The `cactus-increase` subroutine also requires an Euler tour of the cactus representation; an $O(n^2)$-processor parallel algorithm to find an Euler tour is in [1]. The numbering of the vertex set necessary for lex-lookahead can be obtained in parallel $O(\log n)$ time with n processors by computing, say, a pre-order numbering of the leaves of the tree corresponding to all extreme sets [37].

Similar to the previous sequential algorithm, our parallel algorithm will have the following high-level steps. First for each vertex v, we compute the intervals when augmenting edges are added to that vertex as in Lemma 5. By sorting all starting points of these intervals, we obtain all group boundaries. We compute the maximal extreme sets of phase t, for all group boundaries t; we also compute the two exceptional maximal extreme sets that receive a single edge by path-increase. Then we can compute the augmenting edge set for each group in parallel. The final augmenting edge set is the parallel sum of all these edge sets.

The above procedure will require the following data structure D1–5. As in Section 3, we let $\text{children}_t(U)$ denote the maximal extreme subsets of U, in phase t. In particular,

$\text{children}_t(V)$ are the current inclusion-wise maximal extreme sets in phase t. Without the subscript t, we refer to the initial extreme system $\mathcal{F}$.

D1 The sets in $\text{children}_t(V)$ for all t. By taking the union over all possible t we get a sub-tree of the initial system of extreme sets which we denote $\mathcal{T}$.
D2 For each node $U \in \mathcal{T}$, a mark by the first phase t when $U \in \text{children}_t(V)$. By $\mathcal{T}$ and the marks, $\text{children}_t(V)$ is computable for any input t by a parallel algorithm.
D3 Two sub-paths of $\mathcal{T}$, which consist of the sets that receive a single edge in path-increase (there are such paths by Lemma 4).
D4 For each vertex $v \in V$, the interval of phases (as in Lemma 5) when augmenting edges are added to v.
D5 For all $U \in \mathcal{T}$, $\text{first}(U)$, the first phase when U receives an augmenting edge.

Before describing the parallel construction of the above data structure, we notice that we face the technical difficulty that there are "exceptional" current maximal extreme sets that get a single augmenting edge in a phase, while the "typical" sets get two edges. Instead of keeping track of the exceptional sets, we first compute D1–5 under the assumption that each maximal extreme set receives only one edge per phase. Then we compute an initial state of the extreme system when all end-vertices are already added to such exceptional sets. Hence we handle end-vertices added separately as exceptional and as typical ones.

In summary D1–5 is computed in the following sub-steps:

1. Compute D1, D2, D4 and D5, under the assumption that each (current) maximal extreme set gets a single augmenting edge in a phase. The parallel implementation of this step is given at the end of this subsection.
2. Select the two sub-paths of $\mathcal{T}$ as in D3. Start from two maximum demand extreme sets; recursively select the current maximum demand subset of each element over the path. In order to select $\mathcal{T}$, we can compute the current demand of an extreme set U from its initial demand and $\text{first}(U)$. Then the parallel path selection is a pointer jump operation over this tree (see e.g. [29]).
3. From $\mathcal{T}$, select all extreme sets U that are maximal subject to not occurring in the path selected in step 2. For each such U, offset the phase number by the number t of the first phase when $U \in \text{children}_t(V)$; for all extreme subsets of U, use as initial demands the demand in phase t.
4. Erase from $\mathcal{T}$ all sets selected in step 2. Recompute D1, D2, D4 and D5 for all remaining sets by modifying the procedure of step 1 such that two edges are added to each maximal extreme set in a phase.

Next we show how to compute step 1. We use the assumption of Lemma 5 on the numbering of the vertex set. We will obtain D1, D2, D4 and D5 (under the assumption of step 1) in a sequence of steps.

4.2.1. Parallel tree operations. Our parallel algorithm is based on pointer jump and parallel prefix computations over trees. Next we briefly survey the necessary background as in [29,37,9].

A *parallel prefix computation* is an $O(\log n)$ time parallel procedure that, given an input sequence of constants $a_1, \dots, a_n$, computes another sequence $y_1, \dots, y_n$ where

$y_1 = a_1$ and $y_i = y_{i-1} \oplus a_i$. Here $\oplus$ is an arbitrary associative operation; addition and maximization are both examples of a prefix computation.

The *leaf fix* and *root fix* operations are parallel prefix computations over all subpaths of a tree; for leaf fix, these paths lead from the root towards the leaves while for root fix, the orientation is opposite. By the Euler tour technique of [37], these operations can be reduced to segmented prefix computations over the Euler tour of the tree. Root fix enables one to compute in parallel a wide variety of functions with constants and operators stored in trees [29].

4.2.2. An exceptional tree computation: rdem. First of all our algorithm will require the values $\mathrm{rdem}(U)$ for $U \in \mathcal{F}$ of the initial extreme system $\mathcal{F}$. Unfortunately the recursive formula of rdem involves addition and maximization. Since the latter is not distributive over the former, we may not directly use the root fix operation or the Euler tour technique as above.

We give the following algorithm to compute rdem. In the i-th parallel step we will compute $\mathrm{rdem}(U)$ for all $U \in \mathcal{F}$ that has at least 2^{i-1} and at most $2^i - 1$ distinct extreme subsets. We are done if the i-th parallel step runs in polylogarithmic time. This is trivially the case if U has no extreme subset of at least 2^{i-1} further subsets.

Next we turn to the case when some extreme sets U with at most $2^i - 1$ extreme subsets have subsets U' with at least 2^{i-1} extreme subsets. Such sets $U \in \mathcal{F}$ form subpaths of the tree corresponding to $\mathcal{F}$. We show that in this case $\mathrm{rdem}(U)$ can be computed in polylogarithmic time for all elements $U_1 \subset U_2 \subset \ldots \subset U_\ell$ of a subpath, in parallel. For $i \leq \ell$, let d_i denote the sum of $\mathrm{rdem}(U')$ for all extreme subsets $U' \neq U_{i+1}$ of U_i. Then

$$\begin{aligned}\mathrm{rdem}(U_k) = \max \{ & \, dem(U_k), d_k + \mathrm{dem}(U_{k+1}), d_k + d_{k+1} + \mathrm{dem}(U_{k+2}), \ldots , \\ & d_k + d_{k+1} + \ldots + d_{\ell-1} + \mathrm{dem}(U_\ell), d_k + d_{k+1} + \ldots + d_\ell \} ,\end{aligned}$$

where the computation consists of two parallel prefix operations (addition and then maximization) for each $k \leq \ell$.

4.2.3. For each extreme set U, we assume that U is the only set that receives augmenting end-vertices. For all extreme subsets $U' \subset U$, we compute $\mathrm{start}(U, U')$, the number of augmenting end-vertices added to U before any of them is added to U'. We do a top-down recursion by computing $\mathrm{start}(U, U_i)$ first for $U_1, U_2, \ldots , U_\ell \in \mathrm{children}(U)$:

$$\mathrm{start}(U, U_i) = \sum_{j<i} \mathrm{rdem}(U_j) \; .$$

These values can be found by a parallel prefix computation over the sequence of the U_i.

Finally the recursive step is a root fix operation. For $U' \in \mathrm{children}(U)$, let $\mathrm{parent}(U') = U$; then for a fixed U_0 and another extreme $U \supset U_0$,

$$\mathrm{start}(\mathrm{parent}(U), U_0) = \mathrm{start}(U, U_0) + \mathrm{start}(\mathrm{parent}(U), U) \; .$$

4.2.4. Under the scenario of Step 2, we compute the maximum number of edges $\text{end}(U)$ that can be added to U so that the demand of U is larger than any of its extreme subsets. Assume τ new end-vertices are added to U; then we can compute the current demand of any subset of U by the values of $\text{start}(U, U')$. Hence for $U' \subset U$, we can also compute the minimum τ (if there is such) when the demand of U' is not less than that of U. Then $\text{end}(U)$ is the minimum of all these values, over all extreme subsets of U.

4.2.5. Now we can find all extreme sets that become maximal extreme at some phase of the algorithm. Hence we can form the sub-tree $\mathcal{T}$ of the extreme system as required in D1. The pointers of the sub-tree are found by a pointer jump operation. All we need is an easy observation: an extreme set U' becomes maximal in some phase exactly if for all extreme $U \supset U'$,

$$\text{end}(U) < \text{end}(U') + \text{start}(U, U') .$$

4.2.6. We compute $\text{first}(U)$ (D5), for $U \in \mathcal{T}$, by leaf fix. The base operation for an immediate successor U' of U in $\mathcal{T}$ is

$$\text{first}(U') = \text{first}(U) + \min\{\text{end}(U), \text{start}(U, U')\} .$$

Then we easily get D2: if U' is an immediate successor of U in $\mathcal{T}$, then the first phase when U' becomes maximal is $\text{first}(U) + \text{end}(U)$.

4.2.7. For each vertex v, compute the interval of phases when an edge gets added to it (D4). As an auxiliary data, compute for each fixed $U \in \mathcal{T}$, $v \in U$ the sub-intervals when an edge is added to v and U is extreme. These intervals can be computed by the knowledge of $\text{first}(U)$ and $\text{start}(U, w)$ for all $w \in U$. Then the intervals required in D4 can be formed by taking the minimum of the sub-interval starts and the maximum of the ends.

5. Proofs of optimality

In this section, we prove (2.B) for three connectivity-increase steps: cactus-increase, cycle-increase and path-increase; hence we complete the proof of Theorems 4 and 6. The results of this section can be read independently of Sections 3 and 4; all is required is the definition of the corresponding connectivity-increase steps.

Throughout the proofs, let graph G' arise by adding a set of edges E' to G. Observe that in all three cases at most two edges are added to any extreme set W of G ($d_{E'}(W) \leq 2$). We prove the correctness (i.e. that (2.B) holds) by contradiction: we assume U is an extreme set of G' but not of G.

The next three lemmas form the base of the analyses. We note here that all we require for G is that the cut value function d is symmetric submodular, hence the results of this section apply if G is a hypergraph. For more details, see Section 6.

Lemma 6. *Let $W \subset U$ be an extreme set of G. Then $d_G(U) \leq d_G(W) + d_{E'}(W) - d_{E'}(U) - 1$.*

Proof. By definition of extreme in G', $d_{G'}(U) \leq d_{G'}(W) - 1$. Expanding $d_{G'}$ as the sum of d_G and $d_{E'}$ gives the result.

□

Lemma 7. *U contains an edge of E' and contains or is intersecting with at least two extreme sets X_1 and X_2 of G with $d_{E'}(X_i) \geq 1$ for $i = 1, 2$.*

Proof. If U is not extreme in G, by definition there is a $U' \subset U$ with $d_G(U') \leq d_G(U)$. If we apply Lemma 6 to U and U', we get that $d_{E'}(U') \geq d_{E'}(U) + 1$, implying that there must be an edge e of E' within U. If e connects the maximal extreme sets X_1 and X_2, then these sets are either intersecting with or contained by U, as required.

□

Lemma 8. *If U intersects an extreme set W of G, then $d_{E'}(W) = d_{E'}(W \cap U, U - W) = 2$; $d_{E'}(U) = 0$; and $d_G(W - U) = d_G(W) + 1$.*

Proof. By definition of extreme, $d_G(W - U) \geq d_G(W) + 1$. This implies by submodularity that $d_G(U - W) \leq d_G(U) - 1$. On the other hand, by the definition of extreme, $d_{G'}(U - W) \geq d_{G'}(U) + 1$. Combining the two latter inequalities, we get that $d_{E'}(U - W) \geq d_{E'}(U) + 2$. Since all edges not counted in $d_{E'}(U)$ but counted in $d_{E'}(U - W)$ connect $U \cap W$ and $U - W$, there are at least two such edges; furthermore $d_{E'}(W) \geq 2$. Since $d_{E'}(W) \leq 2$ by the construction of E', equality must hold everywhere, proving the claims.

□

The first theorem is also found in Naor et al. [32]; we include it here for completeness.

Theorem 8 ([32]). *(2.B) holds for the cactus-increase subroutine.*

Proof. Let U be an extreme set of G' but not of G. By Lemma 7, there is an edge $e \in E'$ connecting extreme sets W and W' of G, with $e \subset U$. By the definition of cactus-increase we may assume $d_{E'}(W) = 1$. Hence by Lemma 8, $W \subset U$. Now by Lemma 6, $d_G(U) \leq c - d_{E'}(U)$. Thus $d_{G'}(U) \leq c$, contradicting our assumption that G' has connectivity $(c + 1)$.

□

Theorem 9. *(2.B) holds for the cycle-increase subroutine.*

Proof. Let G and G' be the graphs before and after adding the augmenting edges; let G have connectivity c. The claim is true by Theorem 8 when E' is just the output of cactus-increase. Otherwise let U be a set which is extreme in G' but not in G. By Lemma 7, U contains or is intersecting with maximal extreme sets W with $d_{E'}(W) \geq 1$. By the definition of cycle-increase, all such W have $d_G(W) = c$ or $d_G(W) = c + 1$ as well as $d_{E'}(W) = 2$.

First assume that there is a W as above that is intersecting with U. Then by Lemma 8, $d_G(W - U) = d_G(W) + 1$, and by Lemma 1, $W - U$ must contain an extreme set U' with $d_G(U') \leq d_G(W) + 1$. By Lemma 8, we also get that both end-vertices of E' in W are in $W \cap U$; thus $d_{E'}(U') = 0$ since $U \subset W$. This contradicts the definition of cycle-increase

where we require all $(d_G(W)+1)$-extreme subsets U' of a maximal extreme W with $d(W) \le c+1$ have $d_{E'}(U') > 0$.

Next assume that no W as above is intersecting with U, but there is a maximal c-extreme $W \subset U$. By Lemma 6, $c \le d_G(U) \le c+1-d_{E'}(U)$, thus $d_{E'}(U) \le 1$. By the algorithm, $d_G(U) = c$ implies $d_{E'}(U) \ge 2$ and $d_G(U) = c+1$ implies $d_{E'}(U) \ge 1$. This contradicts the above inequalities.

Finally assume that U avoids all c-extreme sets, but contains the (unique) $(c+1)$-extreme W. Since E' connects c-extreme sets and W by a cycle, $d_{E'}(U) \ge 2$. By Lemma 1, $c+1 \le d_G(U)$. By Lemma 6, $d_G(U) \le c+2-d_{E'}(U)$, thus $d_{E'}(U) \le 1$, a contradiction.

□

Theorem 10. *(2.B) holds for the path-increase subroutine.*

Proof. In path-increase, $d(U_\ell) = d(U_1) \le d(U_2) \le \ldots \le d(U_{\ell-1})$ are the maximal extreme sets with $w(U_i) \ge 2$; the algorithm selects an edge set $E' = \{e_1, \ldots, e_{\ell-1}\}$ with e_i connecting U_i and U_{i+1}. Let U be an extreme set of G' but not of G.

Lemma 9. *There exists no $i < \ell$ such that both vertices of e_i belong to U and both vertices of e_{i-1} or e_{i+1} belong to $\overline{U}$.*

Proof. Recall that both e_{i+1} and e_i have a vertex in U_{i+1}. The claim is immediate if $U_{i+1} \subset U$; else U_{i+1} and U intersect and the claim follows by Lemma 8.

□

Corollary 1. *If $d_{E'}(U) = 0$, then both vertices of e_i are contained in U for all $i < \ell$ and $U \cap U_i \ne \emptyset$ for $i \le \ell$.*

Proof. By Lemma 7 both vertices of e_i are contained in U for some i. The claim follows by applying Lemma 9 inductively for increasing and decreasing values of i.

□

Lemma 10. $U_1, U_\ell \subseteq \overline{U}$.

Proof. By Lemma 8, U does not intersect U_1 and U_ℓ since by the construction of E', $d_{E'}(U_1) = d_{E'}(U_\ell) = 1$. If one of them is contained by U, we get $d_G(U) \le c+1-d_{E'}(U)-1$ by Lemma 6; this is possible only if $d_G(U) = c$ and $d_{E'}(U) = 0$. By Corollary 1 $U \cap U_i \ne \emptyset$ for $i \le \ell$ then. However by Lemma 1 there is a c-extreme subset X of $\overline{U}$; $d(X) = c$ implies $w(X) \ge 2$. We reached a contradiction, since X should be included among the U_i.

□

By Corollary 1 and Lemma 10, we get that $d_{E'}(U) \ge 1$. By Lemma 9 and by the facts that $U_1, U_\ell \subseteq \overline{U}$ we also get that $d_{E'}(U) \ge 2$ then. By Lemma 8 no extreme set U_i can intersect U. Let i be minimal such that $U_i \subset U$.

We complete the proof by using the above choice of U_i and exhibiting a maximal extreme set X with $d_G(X) < d_G(U_i)$ and $X \not\subset \overline{U}$. Such an X has $w(X) \ge 2$ and hence must be equal to some U_k for $k < \ell$. By the choice of i, $k > i$. This provides

a contradiction with the choice of E', since $d_G(U_k) \geq d_G(U_i)$ for all $\ell > k > i$. To find this set X, by Lemma 6 we get $d_G(U) \leq d_G(U_i) + 2 - d_{E'}(U) - 1 \leq d_G(U_i) - 1$. Hence there is an extreme set $W \subset U$ of G (by Lemma 1) with $d_G(W) < d_G(U_i)$. There must also be a maximal extreme set $X \supseteq W$ with $d_G(X) < d_G(U_i)$ (possibly $X = W$). We reached the desired contradiction.

□ □ □

6. Hypergraph augmentation

We show that the general augmentation scheme can be applied to the hypergraph connectivity augmentation problem. For a hypergraph G with vertex set V, we define the value of a cut $(C|\overline{C})$ as the total capacity of the hyperedges containing vertices on both sides of the cut. The connectivity value is defined as the minimum cut value. We may define the k-demand of sets, partitions and laminar families in the same way as for ordinary graphs.

There may be different interpretations of the hypergraph augmentation problem. Unlike in the ordinary graph case, augmentation is possible either by ordinary or by hyperedges. The more vertices there are in a (hyper)edge, the more efficient that edge is in augmenting the connectivity. It turns out that the hardest case for the augmentation problem is when only ordinary edges are allowed. This most restrictive problem was completely solved by [2]. For other formulations of the hypergraph augmentation problem, we refer the reader to [36, 13].

We consider a different and in several senses simpler problem. We allow hyperedges to be added as well. We measure the cost of a hyperedge by its size (number of vertices contained). However, we only allow a *single* hyperedge in the solution. For this problem, a similar bound as in Theorem 1 can be obtained for the optimum cost. As it is noticed in [8], this same bound on the augmentation cost is insufficient if no hyperedge is allowed. We note it here that an independent proof of the single-hyperedge-augmentation result is found in [5].

Theorem 11. *The minimum cost of hyperedges needed to augment hypergraph connectivity to target value k is* $\max\{\text{dem}(\mathcal{P}, k) : \mathcal{P}$ *is a sub-partition*$\}$. *There is an optimum solution containing at most one hyperedge.*

We will prove this theorem by showing that the extreme-sets based augmentation algorithm with greedy-lookahead and path-increase steps is optimal in all except for the last augmentation phases (when path-increase cannot be applied). First of all, we have to show that the definition of extreme sets makes sense for hypergraphs:

Definition 5. *A non-negative valued function d on subsets of V satisfying $d(X) = d(\overline{X})$ and $d(X)+d(Y) \geq d(X \cup Y)+d(X \cap Y)$ is called* symmetric submodular. *The hypergraph cut value function is known to be symmetric submodular. For a symmetric submodular function d, $X \subset V$ is called* extreme *if $d(X) < d(X')$ for all $X' \subset X$. The extreme sets form a laminar family.*

Now we notice that neither the lookahead nor the path-increase steps require the underlying graph be ordinary (with no hyperedges). The proof of Theorem 10 works for arbitrary submodular functions d. Hence we may prove the main result:

Proof. Let the input hypergraph be G, the target connectivity be k. We know that (2.A–B) holds if we use the path-increase and greedy-lookahead steps to optimally augment G to G' with connectivity $k-1$ by ordinary edges. Let $\mathcal{F}'$ consist of the system of k-extreme sets of G'. Let e be a hyperedge with one vertex in each set of $\mathcal{F}'$. By Lemma 1, each min-cut of G' contains k-extreme sets on both cut sides; thus (2.C) holds if we add edge e. (2.B) holds vacuously. Finally to show (2.C), notice that the cost of e is $\text{dem}_{G'}(\mathcal{F}', k)$. The proof is complete by Theorem 2.

□

Part II: Extreme sets and cactus algorithms

In the second part of the paper, we describe the necessary subroutines (extreme sets and cactus algorithms) to obtain efficient augmentation algorithms for ordinary graphs. Notice that we no longer consider the hypergraph augmentation problem. We will summarize the running times we achieve in the Conclusion (Section 10).

Until recently, there has been few effort in giving efficient cactus and extreme sets algorithms. Gabow [16] described efficient algorithms for graphs with unit edge weights based on a surprisingly efficient $\tilde{O}(cm)$ time min-cut algorithm [14]. For weighted graphs, however, $\tilde{O}(nm)$ time flow and min-cut algorithms (see e.g. [20,23]) have been used. The basically only known cactus algorithm by Karzanov and Timofeev [28] (with slight modifications and improvements in [34]) requires the computation of $n-1$ flows. The extreme sets can be found again by $n-1$ flow computations via a Gomory–Hu tree [21], as described in [32].

The flow-based algorithms seem hard to be improved. There has been a lot of (unsuccessful) effort in improving Gomory–Hu algorithms ([23,26,3] etc). The only positive result is that the $n-1$ flow computations required to find min-cuts can be pipelined to a single one. However, the Karzanov–Timofeev cactus algorithm requires these flows in a specific order that the Hao–Orlin algorithm cannot guarantee.

As for parallel algorithms, it was known for relatively long that min-cuts can be found by the randomized algorithm of Mulmuley, Vazirani and Vazirani [31]. Based on this result, Naor and Vazirani [34] described a parallel cactus algorithm. On the negative side, finding general flows (and hence Gomory–Hu trees) is known to be P-complete. Prior to our result, no parallel extreme sets algorithm has been known.

The recent breakthrough of Karger and Stein [26] shows that (even all) min-cuts can be found much quicker that the current best flow algorithms. Provided Monte Carlo algorithms are allowed, all min-cuts can be found in $\tilde{O}(n^2)$ time; this algorithm is very efficient in parallel as well. Recently, Karger [25] also gave an $\tilde{O}(m)$-time algorithm that finds a single min-cut.

Our most efficient algorithms are based on the Karger–Stein algorithm. One may say that using a faster min-cut algorithm is a trivial way to improve cactus or extreme sets algorithms. However as we describe in Section 7, the time to even list all min-cuts can be $\Omega(n^3)$; hence it is a non-trivial task stay within the $\tilde{O}(n^2)$ time in our algorithms. In our algorithms, we strongly build on the cactus representation and a recent near-mincuts representation [3] to efficiently handle the system of cuts found by the Karger–Stein algorithm.

Summary of new results

In the bulk of this section we present deterministic, randomized and parallel (RNC) algorithms for building the cactus representation and the system of extreme sets. Our algorithms are based on the new approaches to find min-cuts, and in particular on the contraction-based algorithms of [24, 26].

The most important new results are Monte Carlo algorithms for finding extreme sets in RNC and in sequential

$$\tilde{O}(n^2 \min\{n, \log(\tau/c), \log(nU/c)\})$$

time, with high probabilities. Notice that our augmentation algorithm is the first, where computing extreme sets is the computational bottleneck. Our sequential algorithms (Section 8) rely on the Karger–Stein algorithm [26] and the near-minimum cuts data structure of [3] that efficiently handles the output of the Karger–Stein algorithm. The parallel algorithm (Section 7.2) is based on the previously unknown property of the contraction algorithm of Karger [24] that it finds a collection of $O(n^3)$ sets that include all extreme sets. Our results indicate that finding extreme sets is a much simpler task than finding a Gomory–Hu tree, in a similar manner as finding min-cuts is easier than computing flows.

We present efficient cactus algorithms based on both the Hao–Orlin and the Karger–Stein algorithms. Recently Lisa Fleischer [11] showed that the Hao–Orlin algorithm can be turned to an efficient cactus algorithm with (deterministic) running time is $\tilde{O}(nm)$; an $\tilde{O}(n^2)$-time Monte Carlo cactus algorithm based on the Karger–Stein algorithm is given in [27]. Independent of these two results, we give a somewhat different cactus algorithm based on a slight modification of the Karzanov–Timofeev cactus algorithm [28]; our specific implementations match the time of [11, 27]. The advantage of our algorithms is that they are based on a general framework that may be compatible with potential new min-cut algorithms. While our presentation of the Hao–Orlin implementation in Section 9.4 is brief and we refer to [11] for more detail, we give the full details of our Karger–Stein based algorithm in Section 9.5.

7. Randomized contraction-based connectivity algorithms

In this section, we describe the Karger–Stein algorithm. We start by describing its main subroutine, the *contraction algorithm* of Karger [24]. In the discussion of the contraction algorithm, we also show the previously unknown fact that the contraction algorithm can find the extreme sets. Our parallel extreme sets algorithm (Section 7.2) is based on this fact.

7.1. The contraction algorithm

Karger [24] describes the *contraction algorithm* as follows. We repeatedly pick a random edge[2] of the existing graph and contract its end-vertices. The procedure terminates when

[2] For graphs with edge capacities we replace a capacity u edge by a set of u unweighted parallel edges; uniform random selection is then made among the multi-edges. For more details see [24].

the graph has $k \geq 2$ remaining vertices; k is a parameter given in advance. A cut *survives the contractions* if no cut edge is contracted, i.e. if the two vertices of the final graph are the contracted sides of the cut.

Theorem 12 (Karger [24]). *A fixed min-cut survives contractions to $k \geq 2$ vertices with probability $\Omega(n^{-2})$.*

□

We will need the following extension to this theorem, which we prove below. The proof is almost exactly as that of Theorem 12. In the lemma below, notice the difference that contraction to *two* vertices is not allowed.

Lemma 11. *Let G have an exceptional vertex v such that all cuts of G except $(v|V-v)$ have value at least $\tilde{c}$ ($d(v)$ may be arbitrarily low). Then a fixed cut with value $\tilde{c}$ survives contractions to $k \geq 3$ vertices with probability $\Omega(n^{-2})$.*

Proof. An intermediate contracted graph with k vertices has at least $(k-1)\tilde{c}/2$ edges, since each $w \in V-v$ corresponds to a vertex subset of the initial graph and thus $d(w) \geq \tilde{c}$. On the other hand, the fixed cut $(C|\overline{C})$ contains $\tilde{c}$ edges, provided it has survived contractions up to k vertices. Hence the probability that $(C|\overline{C})$ does not survive the contraction to the next $(k-1)$-vertex graph, given that it has survived contractions up to k vertices, is at most

$$\frac{\tilde{c}}{(k-1)\tilde{c}/2} = 2/(k-1) \; .$$

Hence the probability that $(C|\overline{C})$ survives contractions to three vertices is at least

$$\left(1-\frac{2}{(n-1)}\right)\left(1-\frac{2}{(n-2)}\right)\cdot \ldots \cdot \left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) = \Theta\left(n^{-2}\right) \; .$$

□

Finally, we show that the contraction algorithm finds not only min-cuts, but extreme sets as well. By running the contraction algorithm until the graph is contracted to two vertices, we consider all $n-2$ sets that ever get contracted to a single vertex in this procedure. By the next theorem, these sets will be our extreme sets candidates.

Lemma 12. *In the contraction algorithm, a fixed extreme set U is contracted to a single vertex before any of the edges leaving U gets contracted with probability $\Omega(n^{-2})$.*

Proof. If we contract $\overline{U}$ to a single vertex for an arbitrary extreme set U, $(U|\overline{U})$ becomes the (unique) min-cut of the contracted graph. If we apply the contraction algorithm to this graph, U survives contractions with probability $\Omega(|U|^{-2}) = \Omega(n^{-2})$, by Theorem 12. This means that if we apply the contraction algorithm to the *original* graph, U is contracted to a single vertex before any edge of the boundary of U is contracted, with probability $\Omega(n^{-2})$.

□

We also mention another result of Karger [24] relating to Theorem 12. The contraction algorithm can be used to find cuts with value within αc, i.e. within α times the minimum. Here the same calculations as in the proof of Lemma 11 give a probability of $\Theta(2^{-2\alpha})$ for such a cut to survive contractions to $\lceil 2\alpha \rceil$ vertices.

All four of the above results can be applied as follows. We repeat the contraction algorithm $\Theta(n^2 \log n)$ times (in parallel). In the case of cuts within value αc, we increase this amount to $\Theta(n^{2\alpha} \log n)$. Then the probability that a fixed min-cut or extreme set is found is $1 - \text{poly}^{-1}(n)$. Since there are $O(n)$ extreme sets and $O(n^2)$ min-cuts, this proves that all of them are found, with high probability.

In Section 7.3 we present a more efficient way to apply the contraction algorithm for finding min-cuts. Next we give a parallel implementation of the contraction algorithm to find extreme sets.

7.2. A parallel (RNC) extreme sets algorithm

We are left with the following task to solve. The independent runs of the contraction algorithm produce a system $\mathcal{S}$ of $\Theta(n^3 \log n)$ candidate extreme sets ($n - 1$ per each run). Then the non-extreme sets have to be removed from $\mathcal{S}$. A straightforward way to select extreme sets is to compare all pairs $S_1, S_2 \in \mathcal{S}$; if $S_2 \subset S_1$ and $d(S_2) \leq d(S_1)$, then S_1 can be removed from $\mathcal{S}$. Assume that initially $\mathcal{S}$ contains all extreme sets. Then by Lemma 1, for all non-extreme sets $S_1 \in \mathcal{S}$ there exists an extreme set S_2 satisfying the above pair of inequalities. Hence the remaining sets will all be extreme. This results in an algorithm that – although not in a very efficient way – finds all extreme sets. We give improved extreme sets algorithm in Section 8.

We are ready to present the first known parallel (RNC) algorithm to find extreme sets. First we get the system $\mathcal{S}$ of candidate extreme sets as in the previous section as follows. A parallel implementation of the randomized contraction algorithm is given by Karger [24]. We select $\Theta(n^2 \log n)$ random sequences to run the contraction algorithm with. Then for all the sequences and the values $k = n, n - 1, \ldots, 2$ in parallel, we run contractions to k vertices by the parallel algorithm of Karger. Thus by selecting the last newly created contracted vertex in each of the runs, we get the system $\mathcal{S}$.

Finally, we compare all candidate extreme sets pairs in parallel. We assign n processors for each pair of candidate sets, one for each graph-vertex. In parallel $O(\log n)$ time, we may decide then if one element of the pair contains the other. If for an $S \in \mathcal{S}$ we find another such $S' \in \mathcal{S}$ with $S' \subset S$, $d(S') \leq d(S)$, then we mark S as non-extreme. By removing all marked sets, we are left with the system of extreme sets. Finally, the pointers of the extreme system can be built as follows: for all extreme sets U in parallel, we select the minimal extreme set containing U. This completes our algorithm.

7.3. The Karger–Stein algorithm

The Karger–Stein algorithm [26] finds all min-cuts in time $O(n^2 \log^3 n)$. This algorithm can be modified to find all cuts with value at most αc, where c is the connectivity and α is a constant. The running time becomes $\tilde{O}(n^{2\alpha})$. These running times are surprisingly

low, if we notice the following. The number of min-cuts can be as large as $\Omega(n^2)$ ([10] and [24]). Hence the space needed to list all min-cuts may be $\Omega(n^3)$. Hence the Karger–Stein algorithm runs in "sub-linear" time, i.e. it must produce some kind of representation of the min-cuts with $\tilde{O}(n^2)$ size.

In algorithmic applications that need all min-cuts, one has to be very careful with applying the Karger–Stein algorithm. Both the cactus-increase step and (as we will see it in Section 8) our efficient extreme sets algorithms need access to all min-cuts. It turns out that the output of the Karger–Stein algorithm itself is (most likely) insufficient to provide the necessary access to all cuts. Hence algorithms that turn the output of the Karger–Stein algorithm to other representations with easier access to cuts are crucial in our applications. One such algorithm is the cactus algorithm of Section 9.5; another is the recent *near-mincuts* representation of Benczúr [3]. We crucially build on this representation in our extreme sets algorithm (Section 8.3).

Let us have a closer look at the Karger–Stein algorithm itself and the representation of min-cuts given as its output. The algorithm builds a collection of binary trees $\mathcal{T}_i$, $i = 1, \ldots, t$ as follows. For a graph G_0 with n_0 vertices on the current leaf level of some $\mathcal{T} = \mathcal{T}_i$, the two new children will contain graphs with $n_0/\sqrt{2}$ vertices. These two graphs arise by applying the contraction algorithm of the previous subsection twice for G_0. The final leaves of $\mathcal{T}$ contain graphs with two vertices, or equivalently, a two sides of a cut, each contracted to a single vertex.

We remark that in the cases of second minimum cuts or cuts with value within αc, we have to modify the algorithm. First of all we terminate contractions with more than two vertices as in the corresponding cases of Section 7.1; all cuts defined by partitions of these two or more vertices are found by the algorithm. Furthermore for a constant $\alpha \geq 1$, we contract an n_0-vertex graph to $n_0/\sqrt[2\alpha]{2}$ vertices. The running time then increases to $\tilde{O}(n^{2\alpha})$.

As Karger and Stein [26] show, a fixed min-cut survives all contractions up to at least one of the leaves with probability $\Omega(1/\log n)$. Hence by setting $t = \Theta(\log^2 n)$, the leaves of the trees will contain all min-cuts with high probability.

To achieve the very efficient running time, it is crucial that for an intermediate graph G_0, we may spend only $O(n_0^2)$ time for contraction. But in that case if $n_0 = o(n)$, we are not allowed to carry information on the *original* graph. Hence at a vertex v of G_0, all we know is the set of vertices in the parent of G_0 that are contracted to v. Vertices of a cut can be "unwrapped" by traversing the path up to the root.

So far, we have discussed how the Karger–Stein algorithm finds all min-cuts. It remains open whether it could find the extreme sets as well. Note that finding the extreme sets is the bottleneck of both of our sequential (randomized) and parallel algorithms. Hence such an application of the Karger–Stein algorithm could improve the augmentation problem.

8. Sequential algorithms to find extreme sets

We describe two extreme sets algorithms. The first algorithm with running time $\tilde{O}(n^2 \min\{\tau, n\})$ (Section 8.1) is based solely on finding min-cuts and serves as an illustration of the main ideas in an extreme sets algorithm. Our second extreme sets

algorithm (Section 8.3) uses the Karger–Stein algorithm to find cuts within a multiplicative factor of the minimum: extreme sets are repeatedly extracted from all cuts of value between $\delta^{i-1} \cdot c$ and $\delta^i \cdot c$, for $i = 1, 2$, etc. The $\tilde{O}(n^2 \log(U/c))$ running time of this algorithm is thus dependent on $\log(U/c)$, where U is the largest edge weight in the input graph. Notice that one may expect U and c be within a polynomial factor of one another, in which case the running time is just $\tilde{O}(n^2)$. Our algorithm uses the *polygon representation* of all such cuts [3] as a query structure to the Karger–Stein algorithm.

8.1. A first sequential randomized algorithm

In our first sequential algorithm, we build on the fact that the minimum weight extreme sets of a graph are all minimal min-cut sides. This fact is noticed by Naor et al. [32]. Hence the first step of our algorithm is easy: we build the cactus representation of all min-cuts by the algorithm of Section 9.5; then we use another observation of Naor et al. [32] that the minimum weight extreme sets are precisely the degree-two nodes of the cactus.

We continue with finding heavier extreme sets. Recall that a set X is called *d-extreme* if X is extreme with $d(X) = d$. Assuming that we have found all d'-extreme sets over all $d' < d$, we aim to recurse over all maximal sets of vertices M such that for all known extreme sets X, $M \subseteq X$ or $M \cap X = \emptyset$. Such sets M form a partition of G's vertex set V. In order to find the next level of heavier d-extreme sets, we consider a graph G_M with $V - M$ contracted to a single vertex. If the connectivity value of this graph is d, then we find d-extreme sets as min-cut sides as before. However, G_M may have a single min-cut $(M|V - M)$ with $d(M) < d$; then the next level of heavier extreme sets within M correspond to *second* minimum cut sides of G_M.

We give a procedure to select d-extreme sets from the second minimum cuts of a graph G_M that has a unique min-cut with a single vertex on one side. We saw in Lemma 11 and Section 7.3 that the Karger–Stein algorithm can be used to find the second minimum cuts of value $\tilde{c}$ in the above G_M. However, we may no longer build a cactus representation to find the extreme sets. Instead, we describe a direct method similar to the cactus algorithm of [26]. The Karger–Stein algorithm builds $O(\log^2 n)$ binary trees $\mathcal{T}_i$ whose $\tilde{O}(n^2)$ leaves contain the candidate sets. Our procedure is an upwards recursion over these trees.

Given that the Karger–Stein algorithm has output all second-minimum cuts of G_M, we select a subpartition of candidate extreme sets for each intermediate contracted graph G_0. Let G_1 and G_2 be the two contracted subgraphs of G_0. Consider the union of extreme candidates selected for G_1 and G_2. We may assume that none of the candidate sets C contain the contracted vertex $V - M$: $V - M$ is the unique min-cut side in G_M, implying that C is not extreme. If any two such sets C and C' intersect, we claim that both of them can be removed. Notice that $C - C' \neq V - M$, $C' - C \neq V - M$ and they cannot be equal to M either since one of them would then be V. Thus by submodularity

$$2\tilde{c} \leq d(C - C') + d(C' - C) \leq d(C) + d(C') = 2\tilde{c}\ .$$

This implies that neither C nor C' is extreme and they can both be removed from the system of candidate extreme sets. Finally we also remove a set C if another set $C' \subset C$

is contained in the candidate system, since such sets cannot be extreme. Once all these pairs are eliminated, we are left with a subpartition of the set M.

If we show that in an n_0-vertex graph G_0 the above procedure can be performed in $O(n_0^2)$ time, we get that we spend $\tilde{O}(n^2)$ time for each tree $\mathcal{T}_i$: recall from Section 7 that we may stay within the time bound of the Karger–Stein algorithm if we spend $O(n_0^2)$ time to process an n_0-vertex intermediate graph. Finally, the extreme candidates of the $\log^2 n$ roots of trees $\{\mathcal{T}_i\}$ can be merged two at a time by the exact same procedure, in $O(n^2)$ time each.

Now we give the procedure of eliminating non-extreme sets from the candidate systems $\mathcal{F}_1$ and $\mathcal{F}_2$ found in the two contracted subgraphs G_1 and G_2 of G_0. We assume that both $\mathcal{F}_1$ and $\mathcal{F}_2$ are subpartitions of $V - M$; we output another subpartition of $V - M$ as a subset of $\mathcal{F}_1 \cup \mathcal{F}_2$. We represent a subpartition $\mathcal{F} = \{X_1, \ldots, X_r\}$ by a vector $(x_1, \ldots, x_{n_0})$ where $x_i = j$ if the i-th vertex of G_0 is contained in X_j. We use a special null symbol if this vertex is not in any of the sets. First of all, we start with $\mathcal{F}_1$ and $\mathcal{F}_2$ as subpartitions of G_1 and G_2; we may uncontract these graphs to obtain the required vectors in $O(n_0)$ time. Then by using the vector representing $\mathcal{F}_2$, we may in $O(n_0)$ time decide if a given element $C \in \mathcal{F}_1$ intersects or contains some element of $\mathcal{F}_2$. For all $C \in \mathcal{F}_1$ the operation takes $O(n_0^2)$ time. The procedure is completed if we apply the last steps by exchanging the role of $\mathcal{F}_1$ and $\mathcal{F}_2$.

In summary we can find the next level of heavier extreme sets within a given subset M in randomized time $\tilde{O}(n_M^2)$, where n_M denotes the number of vertices of G_M. Observing that $\sum_M n_M \leq 2n$, we derive that the total time necessary to find the next level of extreme sets over all such sets M is $\tilde{O}(n^2)$. We proceed recursively; two upper bounds on the maximum depth of the recursion are $\tau = k - c$ if we are interested in d-extreme sets with $d \leq k$ only, and the bound n on the depth of the extreme system. This gives us the randomized running time

$$O(n^2 \log^3 n \min\{\tau, n\}) .$$

8.2. The polygon representation of near-mincuts

We aim to improve the previous algorithm by considering cuts within a *constant times* the minimum instead of (second) minimum cuts. Notice that the Karger–Stein algorithm is capable of finding all such cuts. The main achievement we will thus get is that the running time dependence on τ can be replaced by a dependence on $\log(\tau/c)$.

The obvious choice of the improved algorithm is to run the Karger–Stein algorithm and find the extreme sets by an upward recursion in its output. In the case of (second) minimum cuts, we were able to use submodularity to discard non-extreme sets. In a similar approach, two intersecting close-minimum cuts C and C' must have either $d(C - C') \leq d(C)$ or $d(C' - C) \leq d(C')$ – but not necessarily both. In order to decide which is the case, we have to know values of cuts that we do not necessarily have located in the output of the Karger–Stein algorithm. Hence each of these "uncrossing" operations may take $\Omega(m)$ time and we may have to spend $\Omega(mn_0^2) \gg n_0^2$ time on an intermediate contracted graph G_0 with n_0 vertices.

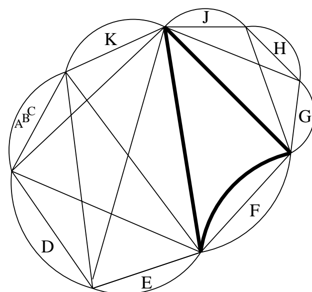

Fig. 3. The cactus representation of Fig. 1 turned to a polygon representation. The three thick lines separate the representation into three sub-polygons, each of which is corresponding to a single cycle of the cactus

The actual algorithm to select extreme sets from close-minimum cuts is more complex. The main tool we need is a representation for all cuts within a constant times the minimum, in which it takes $O(1)$ time to get the value of $d(C - C')$, for given C and C'. The *polygon representation* of Benczúr [3] has this property; we describe this representation now. We say a cut is an α*-near-mincut* if its value is less than αc. We consider α-near-mincuts for $\alpha \leq 6/5$. In the representation, the system of α-near-mincuts is divided into subsets $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$. All we need to know about these subsets is that no cut of one subset may cross a cut of another one (we mentioned this same property of the cactus representation in Section 9.1). Now each subset $\mathcal{C}_i$ possesses a *representing polygon* defined next. In Fig. 3 we give an example of a single representing polygon corresponding to the set of all min-cuts of the graph in Fig. 1.

- A *representing polygon* $\mathcal{P}_i$ is a polygon with a collection of distinguished *representing diagonals*, with all polygon-edges and diagonals drawn by straight lines in the 2D plane.
- The representing diagonals divide $\mathcal{P}_i$ into *cells*.
- The vertex set V is partitioned into *atoms*; each atom is mapped to a cell of $\mathcal{P}_i$. However, some cells may contain no atoms (typically, most of them are empty).
- Each representing diagonal defines a cut, with sides as the union of atoms contained by cells on the diagonal sides.
- The collection of cuts $\mathcal{C}_i$ equals to the collection of cuts defined by representing diagonals of the polygon.
- An array with one entry for each pair of polygon-vertices stores the fact whether the diagonal connecting the pair of vertices is present. If the diagonal is present, the value of the corresponding cut is also stored.

In [3] two key facts are shown. First, the total number of polygon-vertices in all polygons is $O(n)$ (whence it follows that there are at most $O(n^2)$ 6/5-near-mincuts). Second, the polygon representation can be built by the Karger–Stein algorithm, with exceeding its running time bound only by a factor $O(\log n)$.

8.3. An improved extreme sets algorithm

We show that all d-extreme sets with $d \leq \alpha c$ can be selected from the polygon representation of all α-near-mincuts in (deterministic) $O(n^2 \log n)$ time [3,4]. This implies that all k-extreme sets can be found by recursing in contracted graphs G_M – as in the algorithm of Section 8.1. The running time of our algorithm based on this idea can be determined as follows. In the i-th step we find all d-extreme sets with $d < \alpha^i \cdot c$ by setting $\alpha = 1 + 1/\log n < 6/5$. This step requires the computation of the polygon representation in randomized $O(n^{2+2/\log n} \log^4 n)$ time. Provided we want to find all d-extreme sets with $d < k$, we need to take $O(\log_{(1+1/\log n)}(k/c))$ such steps. This totals to

$$O\left(n^2 \log^5 n \log(k/c)\right) ,$$

which yields $\tilde{O}(n^2 \log(U/c))$ if we notice that there are no d-extreme sets with $d > nU$.

We select the extreme sets among represented near-mincuts as follows. For each representing polygon $\mathcal{P}_i$, we select a laminar family $\mathcal{F}_i$ containing all extreme sets represented by that polygon; we record the value $d(C)$ for each $C \in \mathcal{F}_i$. For the entire collection of polygons, $\mathcal{F}_0 = \bigcup_i \mathcal{F}_i$ is a cross-free family since no pair of cuts from different polygons may cross. Then it is trivial to remove non-extreme sets from $\mathcal{F}_0$ in $O(n^2)$ time.

We show how to find $\mathcal{F}_i$ in $O(n_i^2)$ time, where n_i is the number of polygon-vertices in $\mathcal{P}_i$. We maintain a laminar family $\mathcal{F}$ which is initially empty. We scan all $O(n_i^2)$ sets C with $d(C) \leq \alpha c$ represented by $\mathcal{P}_i$ one-by-one. We show that C can either be added to $\mathcal{F}$ or it can be proved that C is non-extreme, in amortized $O(\log n)$ time. This will complete the analysis.

For a fixed laminar family $\mathcal{F}$ and a set C such that all sets correspond to diagonals in $\mathcal{P}_i$ we select all diagonals of $\mathcal{F}$ crossing C; this takes time $O(|\mathcal{F}|)$. If there are no such sets, we add C to $\mathcal{F}$. Otherwise for a $C' \in \mathcal{F}$ crossing C, we check if there is a diagonal corresponding to $C - C'$ and another to $C' - C$. Hence we may decide in constant time whether $d(C) \geq d(C - C')$ or $d(C') \geq d(C' - C)$; one of the two holds by the submodularity of d. Hence in constant time, we may remove C or C' from $\mathcal{F} + C$.

The above procedure may take $|\mathcal{F}| = \Omega(n_0)$ time for each set C. We improve on this by considering the system $\mathcal{S}$ of sets incident to the same polygon-vertex A at once. Provided we manage to handle one polygon-vertex in time $O(n_0 \log n)$, we spend an amortized $O(\log n)$ time per diagonal. This completes the analysis of our extreme sets algorithm.

Consider the two laminar families $\mathcal{F}$ and $\mathcal{S}$ as above. Each time a crossing pair in $\mathcal{S} \cup \mathcal{F}$ is detected, we may remove one set from $\mathcal{S} \cup \mathcal{F}$; thus the maximum number of crossing pairs that have to be detected becomes $O(n_0)$. In $O(\log n)$ time we will either find a crossing pair or conclude that some element of $\mathcal{F}$ crosses none of $\mathcal{S}$. Remember that all diagonal sides in $\mathcal{S}$ share a polygon-vertex S; we may assume $\mathcal{S}$ is sorted around this vertex. Let a diagonal AA' belong to $\mathcal{F}$. Then we may in $O(\log n)$ time insert the diagonals AS and $A'S$ into the sorted list $\mathcal{S}$. Then the elements of $\mathcal{S}$ crossing AA' are precisely those between AS and $A'S$ in this order. This shows that extreme sets can be extracted from the polygon representation in $O(n^2 \log n)$ time.

9. Computing the cactus representation

In this section, first we describe a general cactus algorithm that may potentially serve as a base for implementations based on new min-cut algorithms as well. The general algorithm is a slight generalization of the Karzanov–Timofeev algorithm [28] (also see [34]). Then we show that this algorithm can be implemented by running either the Karger–Stein [27] or the Hao–Orlin [23] algorithms that produce all min-cuts. Notice that the $\tilde{O}(n^2)$ and $\tilde{O}(nm)$ running time of these algorithms is less than the $O(n^3)$ time to list all min-cuts. Hence we have to be careful and not access all the min-cuts when we build the cactus representation. Our cactus algorithms will match the time bound of the corresponding min-cut algorithms. While we describe our Karger–Stein based algorithm in full detail in Section 9.5, the Hao–Orlin based implementation is only sketched in Section 9.4 and the reader is advised to check for a full implementation in Fleischer [11].

9.1. Cactus representation: uniqueness

Recall that the cactus representation of all min-cuts consists of a cactus graph $\mathcal{K}$ such that a partition of V is mapped to certain cactus-vertices. Consider the partitions of G's vertex set into C and $\overline{C}$ arising by deleting a pair of edges of the same cactus-cycle and considering the vertex sets mapped to each side. The min-cuts of G are precisely the $(C|\overline{C})$ defined in this way.

The cactus representation as defined above is not uniquely determined. First, we may always choose a cycle $\mathcal{C}$ and on of its vertices k. We may then subdivide k into a pair of new vertices k' and k'' connected by a length two cycle such that k' belongs to cycle $\mathcal{C}$ while k'' is incident to all remaining edges (Fig. 4, left). All graph-vertices previously mapped to k are now mapped to k''. Second, we may replace a length three cycle on vertices u_1, u_2 and u_3 by making a new vertex k with no graph-vertices mapped to it and then creating three new cycles k, u_i $(i = 1, 2, 3)$ of length two.

We will use the definition of [34] that uniquely determines the cactus representation. Roughly speaking, we will perform both of the above two changes as long as no cut is represented more than once. We say that

- a length two cycle $\mathcal{C}$ represents the single cut defined by the sides of the cactus arising by deleting the edges of $\mathcal{C}$; while
- a cycle $\mathcal{C}$ of length at least four represents all cuts arising by deleting two *non-adjacent* edges of $\mathcal{C}$.

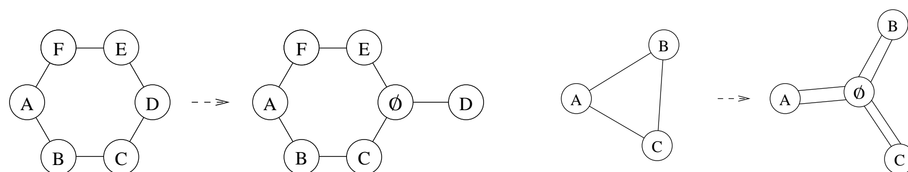

Fig. 4. Two transformations of cactus representations that show the representation is not uniquely determined

We disallow cycles of length three and require each cut be represented as in the above definition exactly once. Notice that cuts represented by distinct cycles may not cross each other. Now min-cuts represented by length two cycles are exactly those that do not cross any other min-cuts and the subsets of cuts represented by the same cycle give the finest partition of the set of min-cuts so that cuts of distinct sets do not cross each other. For the remaining details see [34] or [4].

9.2. The residual cactus

We start by giving a base of algorithms for building the cactus representation: a recursive procedure that, given a cactus representation of a graph with two vertices v_1 and v_2 contracted and the collection of all min-cuts separating v_1 from v_2, yields the cactus representation of the initial graph. Our procedure is similar to the cactus algorithm of Karzanov and Timofeev [28]; there the pair of vertices v_1 and v_2 is selected by a special rule that our procedure will not require.

In the bulk of Section 9, let G be the input graph and let $\mathcal{K}$ be its unique cactus representation as defined in Section 9.1. Let v_1 and v_2 be two vertices of G; let graph G' with cactus representation $\mathcal{K}'$ arise by contracting v_1 and v_2 to a single vertex in G.

While our aim is to build the cactus representation $\mathcal{K}$ from $\mathcal{K}'$, we start our discussion with an opposite process: given $\mathcal{K}$, we describe the collection of min-cuts separating v_1 and v_2. All these cuts are described by the residual cactus $\mathcal{K}_{v_1,v_2}$ arising from $\mathcal{K}$ as defined next.

Definition 6 (Residual cactus). *Let $\mathcal{K}$ be the cactus representation of a graph G, let v_1 and v_2 be a pair of vertices. Consider the shortest path $\mathcal{P}$ from the cactus-vertex k containing v_1 towards k' containing v_2 as well as the next shortest path $\mathcal{P}'$ in $\mathcal{K} - \mathcal{P}$ (see Fig. 5). The* residual cactus *$\mathcal{K}_{v_1,v_2}$ is a cactus graph arising by the contraction of all cactus-edges other than $\mathcal{P} \cup \mathcal{P}'$.*

In what comes next, first we prove that the residual cactus is a cactus representation of a subcollection of min-cuts that includes all those separating v_1 from v_2; then in Lemma 14 we show how $\mathcal{K}$ and the residual cactus (Fig. 5) determine $\mathcal{K}'$, the cactus representation of G' arising by contracting v_1 and v_2 (Fig. 6). The opposite procedure is used in Algorithm 3 to build $\mathcal{K}$ from $\mathcal{K}'$. The first lemma also characterizes the residual cactus as a "path" of cycles.

Lemma 13. *The residual cactus $\mathcal{K}_{v_1,v_2}$ is a cactus graph consisting of cycles $\mathcal{C}_1, \ldots, \mathcal{C}_t$ that can be numbered such that $\mathcal{C}_j$ and $\mathcal{C}_{j+1}$ are adjacent for $1 \le j < t$, and there are no other adjacent cycle pairs. A pair of vertices $k \in C_1$ and $k' \in \mathcal{C}_t$ contain v_1 and v_2, respectively (see Fig. 5). The residual cactus $\mathcal{K}_{v_1,v_2}$ represents all min-cuts separating v_1 from v_2.*

Proof. After deleting edges of path $\mathcal{P}$ (the shortest one between v_1 and v_2), the path from v_1 to v_2 without vertex repetition is uniquely defined as the remaining edges of all cycles $\mathcal{C}_i$ for $i \le t$ that contain edges of $\mathcal{P}$. If a cycle does not contain edges of path $\mathcal{P}$, then it is contracted in the residual cactus. Thus the residual cactus consists precisely

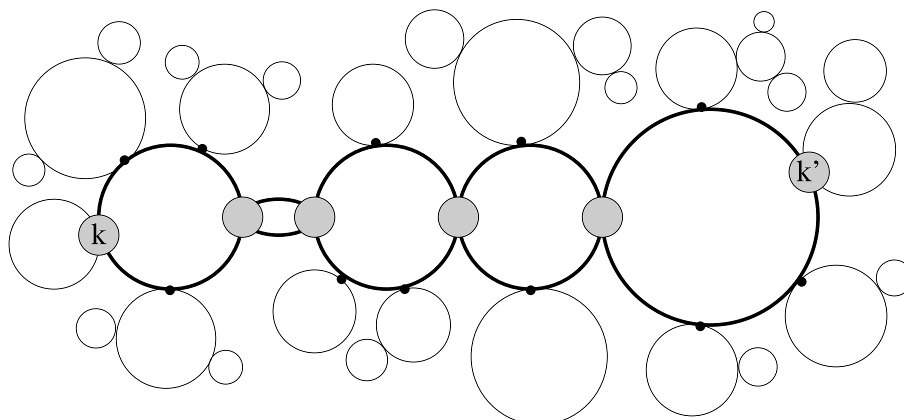

Fig. 5. The cactus $\mathcal{K}$ and the cycles (thick) of $\mathcal{P} \cup \mathcal{P}'$ that define the residual cactus $\mathcal{K}_{v_1,v_2}$. Cactus-vertices k and k' contain v_1 and v_2, respectively

of the $\mathcal{C}_i$. If we number them by the sequence visited by $\mathcal{P}$, we get a numbering as required. Finally for the last part, notice that any cut separating v_1 and v_2 also separates k and k'.

□

Now we complete our goal: we build G's cactus representation $\mathcal{K}$ from $\mathcal{K}_{v_1,v_2}$ and the cactus $\mathcal{K}'$ of G' arising by contracting two of its vertices v_1 and v_2 (Figs. 5 and 6). The algorithm below builds a cactus representation where all cuts from $\mathcal{K}'$ as well as from $\mathcal{K}_{v_1,v_2}$ are represented. Hence this cactus is $\mathcal{K}$.

We use the following notation. Let cycles $\mathcal{C}_i$ and two distinguished cactus-vertices k and k' containing vertices v_1 and v_2, respectively, be as in Lemma 13. Let $\mathcal{P}$ be the

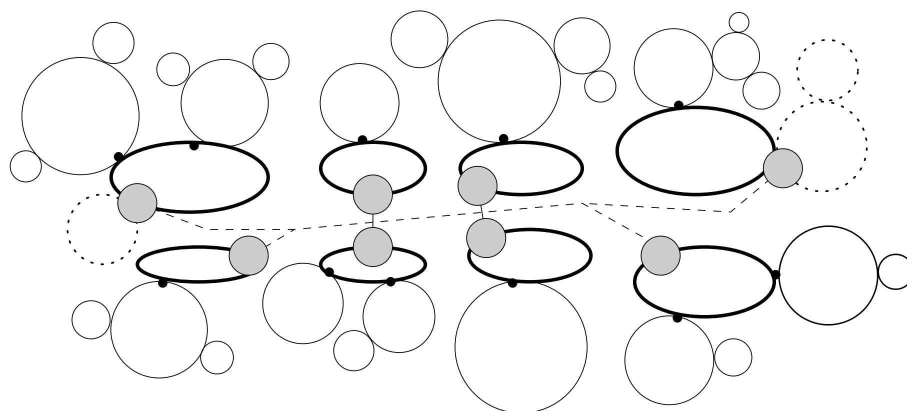

Fig. 6. The cactus $\mathcal{K}'$ arises by modifying the cycles (thick) of the residual cactus $\mathcal{K}_{v_1,v_2}$ in Fig. 5. Shaded vertices are all joined to a single cactus-vertex u containing both v_1 and v_2. The components adjacent to u are those incident to the thick cycles as well as the two dotted components

shortest path of $\mathcal{K}$ connecting k and k'; let k_i be the unique vertex of $\mathcal{P}$ that belongs to both $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$, for $i < t$. Let $k_0 = k$ and $k_t = k'$. If the two vertices k_{i-1} and k_i are non-incident, let two cycles $\mathcal{C}_i'$ and $\mathcal{C}_i''$ arise by contracting k_{i-1} and k_i in $\mathcal{C}_i$; else let a single cycle $\mathcal{C}_i'$ arise by contracting k_{i-1} and k_i again.

Lemma 14. *The cactus representation $\mathcal{K}'$ arises by (i) contracting all k_i for $0 \le i \le t$ to a single vertex u; and (ii) for each length three cycle arising in this way, contracting the two vertices different from u to single new vertices (and thus obtaining length two cycles).*

Proof. By Lemma 13, the contraction procedure removes all cuts separating v_1 and v_2. We show that no other min-cut gets removed. We contract each cycle $\mathcal{C}_i$ into (possibly two) smaller cycles; these cycles represent all cuts not separating v_1 and v_2. If any of these cycles is of length three, the only cut represented by this cycle (not arising by the removal of two incident cactus-edges) arise by the removal of the two cactus-edges adjacent to the contracted vertices. These cuts are represented by the length two cycles as described. This proves the theorem.

□

Algorithm 3. The cactus $\mathcal{K}$ arises from $\mathcal{K}'$ and $\mathcal{K}_{v_1,v_2}$ as follows.

1. From $\mathcal{K}_{v_1,v_2}$, we build the cycles $\mathcal{C}_i'$ and $\mathcal{C}_i''$. We replace each length three cycle by a length two one.
2. All cycles $\mathcal{C}_i'$ and $\mathcal{C}_i''$ in $\mathcal{K}'$ are joined at a single cactus-vertex u as in Lemma 14. We find all these cycles and vertex u in $\mathcal{K}'$.
3. We cut $\mathcal{K}'$ into the set of components adjacent to u (as in Fig. 6).
4. We arrange all components containing a $\mathcal{C}_i'$ or $\mathcal{C}_i''$ by forming $\mathcal{K}_{v_1,v_2}$ from the $\mathcal{C}_i'$ and $\mathcal{C}_i''$.
5. Finally we have to join the remaining components $\mathcal{C}$ (that do not containing any $\mathcal{C}_i'$ or $\mathcal{C}_i''$) to one of the residual cactus vertices k_i. We do this by picking a graph-vertex w corresponding to component $\mathcal{C}$ and for each $i \le t$ selecting a cut $(A_i|\overline{A_i})$ separating k_{i-1} and k_i with $v_1 \in A_i$ and $v_2 \in \overline{A_i}$. We join component $\mathcal{C}$ to the following vertex of $\mathcal{K}_{v_1,v_2}$:
   - k_h if $w \subset A_{h+1} - A_h$;
   - $k = k_0$ if $w \subset A_1$;
   - $k' = k_t$ if $w \subset \overline{A}_t$.

□

9.3. The Karzanov–Timofeev algorithm

We mention the main difference between the Karzanov–Timofeev algorithm [28] and our general cactus-building procedure. While the Karzanov–Timofeev algorithm uses a notion similar to our residual cactus, their residual cactus is assumed to have a very simple structure. Namely, they assume that each pair v_1, v_2 ever arising in their algorithm have an edge connecting them. Notice that this assumption can be made whenever the input graph is connected. Then $\mathcal{K}_{v_1,v_2}$ has the following simple structure:

Lemma 15 (Karzanov and Timofeev [28,34]). *If there is an edge connecting v_1 and v_2, then the shortest path $\mathcal{P}$ of the cactus from v_1 to v_2 contains at most one edge from each cycle. Hence the system of s min-cuts $\{(A_i|\overline{A}_i)\}$ separating v_1 and v_2 form a chain with $A_1 \subset A_2 \subset \ldots \subset A_s$; $s = O(n)$.*

□

In Lemma 14 we saw that the cycles $\mathcal{C}_i$ of the residual cactus $\mathcal{K}_{v_1,v_2}$ can be turned either to a single cycle $\mathcal{C}_i'$ or to two cycles $\mathcal{C}_i'$ and $\mathcal{C}_i''$ of $\mathcal{K}'$. Lemma 15 implies that in the scenario of the Karzanov–Timofeev algorithm the first case must hold for all $i \le t$. This fact can be used to simplify Algorithm 3.

9.4. The Hao–Orlin algorithm

The Hao–Orlin algorithm performs the Goldberg–Tarjan max-flow algorithm [20, 19] for source–sink pairs w_i and $\{w_1, \ldots, w_{i-1}\}$, in some permutation $\{w_j\}$ of the vertex set. The novelty of this algorithm is that the flow computations are pipelined so that the sequence of $n - 1$ max-flow computations terminate within the time bound of a single run of the Goldberg–Tarjan algorithm. Hence min-cuts are found in $\tilde{O}(nm)$ time, an amount less than the potential space to list all min-cuts. It is true but not entirely obvious that the Hao–Orlin algorithm finds all min-cuts; these min-cuts are described by the Picard–Queyranne representation [35].

In what follows next, we show how to build the cactus representation from the output of the Hao–Orlin algorithm, thus yielding a deterministic $\tilde{O}(nm)$-time cactus algorithm. In order to do this, we build the residual cactus $\mathcal{K}_{v_1,v_2}$ from the output of a v_1–v_2 max-flow computation. Unfortunately in the algorithm below we may not use the simplifying assumption of Karzanov and Timofeev (Lemma 15): the permutation $\{w_j\}$ of the vertex set is part of the output of the Hao–Orlin algorithm and cannot be given in advance.

A main tool for building the residual cactus $\mathcal{K}_{v_1,v_2}$ is the representation of Picard and Queyranne [35] for all cuts with minimum value separating v_1 and v_2. Notice that if there exist min-cuts separating v_1 and v_2 at all (i.e. the max-flow value equals c), then the system of cuts contained by the Picard–Queyranne representation are all included in $\mathcal{K}_{v_1,v_2}$. Else if there are no min-cuts separating v_1 and v_2, then the v_1–v_2 max-flow value is more than c and then v_1 and v_2 will always belong to the same node of the cactus representation.

The Picard–Queyranne representation is as follows. For a maximum v_1–v_2 flow in graph G, one defines a residual flow as a directed graph arising by orienting some edges of G. We orient all edges that are not saturated (i.e. the amount of flow carried is less than the weight of the edge) in the direction of the flow carried; we also orient all edges that carry a positive amount of flow in the reverse direction (so some edges can be oriented both ways). A v_1–v_2 min-cut may only have backwards edges (from the side of v_2 towards the side of v_1); hence no v_1–v_2 min-cut may divide a subgraph of positive directed connectivity (i.e. a strongly connected subgraph) in the residual graph. Linear time algorithms that find these components are known [9]; after contracting all these components to single vertices, the resulting graph is a so-called DAG with no directed

cycle. This DAG is the Picard–Queyranne representation; in this DAG, all min-cuts can be found as bipartitions with backward edges only (see [35] or [17]).

The Picard–Queyranne representation can be transformed to the residual cactus $\mathcal{K}_{v_1,v_2}$ as follows. First of all we notice that the vertex set of the DAG is equal to the partition $\{V_1, \dots, V_\ell\}$ mapped to the cactus-vertices $\mathcal{K}_{v_1,v_2}$: both of these partitions are equal to the coarsest partition in which no element is divided by any min-cut. Thus we have to turn the directed edges of the Picard–Queyranne representation to cactus-edges.

Recall that $\mathcal{K}_{v_1,v_2}$ consists of a collection $\mathcal{C}_1, \dots, \mathcal{C}_t$ of cycles with $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ sharing a vertex, for all $i < t$. We transform the DAG to $\mathcal{K}_{v_1,v_2}$ in two steps. First we sort the set system $\{V_h\}$ such that for a pair of its non-empty sets, $V_{h'}$ comes before V_h if there are cycles $\mathcal{C}_i$ and $\mathcal{C}_j$ with $i < j$, V_h mapped to $\mathcal{C}_i$ and $v_{h'}$ mapped to $k' \in \mathcal{C}_j$. In a *topological sort* of the DAG, it is easy to see that the sets V_k have the required property. A topological sort means a linear arrangement of a DAG's vertex set such that there are no forward edges; a topological sort can be performed in $O(m)$ time [9].

In a topological order of the DAG-vertices, the sets V_h that belong to a fixed $\mathcal{C}_i$ form an interval of the topological order. Consider the vertices k_{i-1} and k_i shared by $\mathcal{C}_i$ and $\mathcal{C}_{i-1}$, respectively by $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$.

– If these vertices are of distance at least two in $\mathcal{C}_i$, then the topological order of this interval is not unique: we may merge vertices over the two paths between k_{i-1} and k_i arbitrarily. This property identifies the vertices of all such cycles $\mathcal{C}_i$.
– Otherwise (if k_i and k_{i-1} are incident) the topological order of the $\mathcal{C}_i$-vertices is unique.

All left to be done is to determine the length of the cycle $\mathcal{C}_i$. Notice that all incident pairs of vertices over the path between k_{i-1} and k_i have $c/2$ edges connecting them. This characterizes all vertices of $\mathcal{C}_i$ with one exception: if one could form a length three cycle replacing two consecutive length two cycles $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$, then there are $c/2$ edges between the pairs k_{i-1}, k_i and k_i, k_{i+1} as well. We may easily distinguish this case. Hence we may form all cycles of $\mathcal{K}_{v_1,v_2}$.

9.5. Transforming the Karger–Stein output to a cactus

We transform all min-cuts in the output of the Karger–Stein algorithm to a cactus representation by using the inductive cactus algorithm of Section 9. Recall that the cactus representation can be built from another $\mathcal{K}'$ arising by contracting a pair of vertices v_1 and v_2 and the so-called residual cactus $\mathcal{K}_{v_1,v_2}$ (Definition 6).

In our algorithm, we fix a permutation $w_1, w_2, \dots, w_n$ of the vertex set. Given the permutation $\{w_i\}$, we proceed as follows: we build the cactus representation $\mathcal{K}'$ of the graph arising by contracting all vertices $\{w_1, \dots, w_i\}$; from this cactus, we build that of the graph with $\{w_1, \dots, w_{i-1}\}$ contracted. In other words, we equate

$$v_1 = \{w_1, \dots, w_{i-1}\} \quad \text{and} \quad v_2 = w_i$$

and use the procedure of Section 9. In order to make the simplifying assumption of the Karzanov–Timofeev cactus algorithm as in Lemma 15, we choose the permutation such

that there is always an edge connecting w_i to $\{w_{i+1}, w_{i+2}, \dots, w_n\}$. Such a permutation can be selected by traversing a search tree of the graph.

In summary, we have to build $\mathcal{K}$ from $\mathcal{K}'$ for $i = n-1, n-2, \dots, 2$, in an (amortized) $\tilde{O}(n)$ time, given the output of the Karger–Stein algorithm. The algorithm requires the knowledge of the residual cactus $\mathcal{K}_{v_1,v_2}$; the main difficulty is that we do not have this cactus at hand. In our algorithm (Algorithm 4) we find the residual cactus and build $\mathcal{K}$ from $\mathcal{K}'$ as in Algorithm 3 at the same time.

The high-level structure of the algorithm is to select cuts $(A|\overline{A})$ with $v_1 \in A$ and $v_2 \in \overline{A}$ and incorporate them one-by-one into the representation. Each such cut $(A|\overline{A})$ is represented by a unique cycle of $\mathcal{K}$. This cycle must be one of the cycles $\mathcal{C}_1, \dots, \mathcal{C}_t$ of the residual cactus $\mathcal{K}_{v_1,v_2}$ (see Lemma 13) where $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ share a vertex k_i. By Lemma 15 k_i and k_{i+1} are adjacent in the cactus and thus they form the vertices of a path $\mathcal{P}$ of the residual cactus. We also know that $\mathcal{K}'$ arises by contracting all edges of $\mathcal{P}$, or, in other words, all vertices k_i are contracted to a vertex $u \in \mathcal{K}'$. Also recall that if an edge of a length four cycle is contracted, the resulting length three cycle is replaced by three new length two cycles sharing a common vertex $k \in \mathcal{K}'$.

Now whenever $(A|\overline{A})$ is selected, we unfold the cycle $\mathcal{C}_i$ representing $(A|\overline{A})$ by subdividing a vertex to two new vertices k_i and k_{i+1}. Provided $\mathcal{C}_i$ has length four, this also means replacing the three length two cycles of $\mathcal{K}'$ by a new length four one. Hence in our algorithm we will maintain a cactus representation $\mathcal{K}''$ with $\mathcal{K}'' = \mathcal{K}'$ as input and $\mathcal{K}'' = \mathcal{K}$ as output and with the property that $\mathcal{K}''$ arises by contracting some consecutive segments of the vertices $k_0, \dots, k_\ell \in \mathcal{K}$ and by replacing length three cycles by three length two ones sharing a common vertex.

Given A and $\overline{A}$, we give a key notion to identify cycle $\mathcal{C}_i$. First of all, we modify the cactus representation $\mathcal{K}''$ by deleting all vertices W mapped to a vertex k whenever both $W \cap A$ and $W \cap \overline{A}$ are non-empty. Then we say that a cycle $\mathcal{C}$ of the cactus *belongs to* A (or, symmetrically, to $\overline{A}$) if all graph-vertices mapped to its vertices belong to A. Since there may be cycles whose vertices have no graph-vertex mapped to them at all, we extend this notion to such cycles as follows. If all cycles that share vertex k belong to A with one exceptional cycle $\mathcal{C}$ with no vertices mapped to it, we declare k to belong to A and if all vertices of a cycle belong to A with at most one exception that is undetermined, then the entire cycle belongs to A. Now the notion can be recursively extended: by removing exactly one edge from each cycle with no vertex mapped to it, the edges of these cycles form disjoint subtrees of the cactus. Hence if we start from the leaf level, we may in $O(n)$ time either decide for a cycle whether it belongs to A or $\overline{A}$ or declare it unknown.

Theorem 13. *Consider a cut $(A|\overline{A})$ with $v_1 \in A$ and $v_2 \in \overline{A}$ represented by a cycle $\mathcal{C}_i$ of $\mathcal{K}$. Let $\mathcal{K}''$ be an intermediate cactus representation arising in our algorithm. Then either*

1. *all cycles belong to either A or $\overline{A}$; there is a single cactus-vertex u that belongs to cycles of both types; and $\mathcal{C}_i$ is a length two cycle that can be added by subdividing u;*
2. *there is exactly one cycle $\mathcal{C}_i'$ of length at least four that belongs to neither A nor $\overline{A}$ and $\mathcal{C}_i$ arises by increasing this cycle by a new edge;*
3. *or there is exactly one cycle of length two with vertices u and k such that k has no graph-vertex mapped to it and two further cycles C_A and $C_{\overline{A}}$ sharing k where C_A*

belongs to A and $C_{\overline{A}}$ to $\overline{A}$. Then $\mathcal{C}_i$ has length four and arises by the subdivision of u into two new vertices and the deletion of k.

Proof. It is immediate by the definition of $\mathcal{K}''$ that all of its cycles except for those created by the contraction of an edge of $\mathcal{C}_i$ (recall there may be three such cycles) belong to either A or $\overline{A}$. Now the three cases as in the theorem arise by distinguishing the cases of a cycle $\mathcal{C}_i$ with length two, length more than four and finally of length exactly four.

□

Algorithm 4: building $\mathcal{K}$ from $\mathcal{K}'$.

```
1. queue←all min-cuts separating v1 and v2;
2. truncate the queue to less than n elements
   until the queue is empty
3.   pop a cut (A|Ā) from queue;
     decide for all cycles whether they belong to A or Ā;
       if all cycles belong to either A or Ā
         then select the unique common vertex u
           of cycles that belong to A and of those that belong to Ā
           subdivide u by a new length two cycle for (A|Ā)
       if a length two cycle C belongs to neither A nor Ā
         then select the vertex k ∈ C that has
           no graph-vertex mapped to it and
           is adjacent to two length two cycles C_A and C_Ā
           belonging to A and Ā, respectively.
           replace C, C_A and C_Ā by a length four cycle
       if a cycle C'_i of at least four belongs to neither A nor Ā
         then insert a new edge into C'_i for (A|Ā)
             to form the new cycle C_i
4.           remove from queue all cuts represented by
             the augmented cycle C_i
   repeat
```

Algorithm 4 performs the above procedure. In the analysis, we have to be careful since we are not allowed to explicitly read all vertices in all min-cut sides. We have to read all vertices of cuts $(A|\overline{A})$ arising in Step 3 of Algorithm 4; we have to make sure that there are at most $O(n)$ such cuts in the entire run of the algorithm then. This is the reason why we inserted Step 4: each cut $(A|\overline{A})$ forces a new cactus-edge to be added, while the number of cactus-edges is $O(n)$ altogether. However we also have to make sure step 4 is not called too often, which could be caused by a high number (possibly even $\Omega(n^2)$) repetitions of the exact same cut in the output of the Karger–Stein algorithm. The truncation in step 2 will make sure that no more than n cuts reach the main loop. We analyze the four main Steps 1, 2, 3 and 4 of Algorithm 4 separately.

9.5.1. Step 1. First we consider the implementation of the queues in Step 1. Recall that all min-cuts are contained in the leaves of the trees generated by the Karger–Stein algorithm. We show how to label all these cuts by the value of i such that w_i is separated from $\{w_1, \dots, w_{i-1}\}$. We use the following simple fact:

Lemma 16. *For a graph with n_0 vertices arising by contracting some vertices of G, there are at most $n_0 - 1$ values of i such that w_i is in a different contracted vertex than all of $w_1, \dots, w_{i-1}$.*

□

We maintain the following data structure for all intermediate contracted graphs G_0 produced by the Karger–Stein algorithm. For all values i where w_i can be separated from $\{w_1, \dots, w_{i-1}\}$ in G_0, we record the vertex u_i of G_0 that contain w_i and the subset U_i that contain $\{w_1, \dots, w_{i-1}\}$. Let G_0 have n_0 vertices; then the size of the data is $O(n_0^2)$. Notice that each leaf of the computation tree of the Karger–Stein algorithm (and thus each min-cut) has a unique value i; this value is what we require.

Now assume G_0 gets contracted. Then we update the data structure as follows. For all values of i, we find the new contracted vertices that contain u_i and U_i. We check if these sets are disjoint; if not, we remove i from the data. The procedure takes $O(n_0^2)$ time.

9.5.2. Step 2. We give a very simple rule to decide whether one cut is a repeated copy of another one in the queue. Since all theses cuts form a chain by Lemma 15, the number of vertices on the same side as v_1 give unique names of the cuts in the queue. Given the size as a label, we may immediately remove repeated cuts in time $O(n)$ plus the queue size.

We are hence done if we implement the size count in the Karger–Stein algorithm. However this is easy: we maintain the number of vertices contracted to each intermediate vertex; whenever two new vertices are contracted, we may hence in $O(1)$ time follow up the vertex count.

9.5.3. Step 3. For a cut $(A|\overline{A})$ that arises in Step 1, we have to know all vertices of A in order to perform the consequent steps of the algorithm. For each such cut, we saw that we need to spend $\Theta(n)$ time. Step 3 does not form a computational bottleneck by the following lemma:

Lemma 17. *A cactus representation has $O(n)$ edges. Hence Step 3 is performed for $O(n)$ min-cuts only.*

Proof. The first part is due to Karzanov and Timofeev [28] (see also [4]). As for the second part, notice that each cut $(A|\overline{A})$ in Step 3 generates a new edge of the cactus representation.

□

9.5.4. Step 4. Finally, we analyze the number of cut–cycle pairs that may reach Step 4. By this analysis, we will complete the description of our cactus algorithm. We give an amortized analysis for all calls to Step 4 for all the $n - 1$ choices of the pair v_1 and v_2.

Theorem 14. *Step 4 of Algorithm 4 is performed on $O(n)$ cuts for each newly created cactus-edge. The time spent on one removal step is $O(\log n)$. Hence the cactus representation can be built in randomized $O(n^2 \log^3 n)$ time.*

Proof. The first part follows by Step 2. If we show the second part, the $O(n^2 \log^3 n)$ total running time will arise as follows. Within this time bound, we run the Karger–Stein algorithm and also build the auxiliary information needed in Steps 1 and 2. Steps 1–3 also fit within the time bounds. Finally in Step 4 we take $O(\log n)$ steps for $O(n)$ cuts each for each single cactus-edge. We are done since the number of cactus-edges is $O(n)$ by Lemma 17.

Now we give a procedure to efficiently decide that a given cut $(B|\overline{B})$ is already represented by an intermediate cactus $\mathcal{K}''$. Let $(A|\overline{A})$ be a cut separating cycle $\mathcal{C}'_i$ as in Step 3 of Algorithm 4; let a new edge be added to $\mathcal{C}'_i$ by subdividing a vertex to two new ones k and k'. Consider an arbitrary pair of graph-vertices x and x' mapped to the subgraphs of $\mathcal{K}''$ containing k and k', respectively, that arise by the deletion of the edges of $\mathcal{C}_i$. Then by Lemma 15 another cut $(B|\overline{B})$ of the queue has x and x' on different sides if and only if $(B|\overline{B})$ is represented by the augmented cycle $\mathcal{C}_i$.

To complete the proof, we show that it can be decided in $O(\log n)$ time whether $(B|\overline{B})$ has x and x' on different sides. Notice that the vertex pair x and x' can be found in an $O(n)$ time pre-processing step when the new cactus-edge between k and k' is formed.

We are done if we may decide for a given min-cut side A and a vertex v in $O(\log n)$ time whether $v \in A$. In order to do this, while we are performing recursive contractions, we have to add additional pointers to access the data produced, as follows. In the Karger–Stein algorithm, a single contraction can spend $O(n_0)$ time on G_0 by using adjacency matrices as data structure. Within this time, we can build pointers from the vertices of G_0 to the contracted vertices. Thus to decide whether $v \in A$ for a vertex v and cut $(A|\overline{A})$ found by the algorithm, we have to scan the path of length $O(\log n)$ from the root to the leaf containing the cut. This takes $O(\log n)$ time, as required.

□

10. Conclusion

We presented various edge connectivity augmentation algorithms for undirected graphs with integer edge capacities. Our successive algorithm (producing an increasing sequence of augmented graphs for increasing target connectivity values) requires the computation of $O(n)$ cactus representations, the initial extreme system, and uses $O(n^3)$ extra work. This algorithm is inherently sequential; the best running times by the results of Part II are $\tilde{O}(n^2 m)$ deterministically and $\tilde{O}(n^3)$ if Monte Carlo algorithms are allowed.

Our unrestricted (non-successive) augmentation algorithm computes a single cactus representation and the initial extreme system; it uses $O(n^2)$ extra work. An implementation of this algorithm is the first RNC algorithm to solve the augmentation problem. The randomized sequential running time is improved compared to our successive algorithm to

$$\tilde{O}(n^2 \min\{n, \log \tau/c, \log nU/c\}) ,$$

where U is the highest capacity of an edge in the input graph and τ is the connectivity increment.

In our paper, several questions remain open. The bottleneck of our augmentation algorithm is to find the initial extreme system. An improvement for this task would immediately improve our non-successive augmentation algorithm. Very recently it was shown in [6] that extreme sets can be found in (randomized) $\tilde{O}(n^2)$ time, yielding a matching runtime for edge augmentation. For the hypergraph augmentation problem, it remains open whether an extreme sets based algorithm may find the optimum solution. The efficiency of neither the hypergraph augmentation nor the (ordinary) parallel augmentation problems is attacked in this paper.

Acknowledgements. András Frank contributed a main part to the paper with discussions, questions and inspiration that led to finding the grouping technique. Thanks to Eddie Cheng for discussions about the hypergraph augmentation problem. Many people helped with clarifying various parts of this paper: Lisa Fleischer in the cactus algorithm part; David Karger (my co-author in [6]) in the extreme sets algorithm part; Michel Goemans (my advisor [4]) in most parts and particularly in those related to the polygon representation.

References

1. Atallah, M., Vishkin, U. (1984): Finding Euler tours in parallel. J. Comp. System Sci. **29**(3), 330–337
2. Bang-Jensen, J., Jackson, W. (1999): Augmenting hypergraphs by edges of size two. Math. Program. **84**. This issue
3. Benczúr, A.A. (1995): A representation of cuts within 6/5 times the edge connectivity with applications. Proc. 34th Annual Symp. on Found. of Comp. Sci. 92–102
4. Benczúr, A.A. (1997): Cut structures and randomized algorithms in edge-connectivity problems. Ph.D. Thesis, M. I. T.
5. Benczúr, A.A., Frank, A. (1999): Covering symmetric supermodular functions by graphs. Math. Program. **84**. This issue
6. Benczúr, A.A., Karger, D.R. (1998): Augmenting undirected edge-connectivity in $\tilde{O}(n^2)$ time. Proc. 9th ACM-SIAM Symp. on Discrete Algorithms
7. Cai, G.-P., Sun, Y.-G. (1989): The minimum augmentation of any graph to a k-edge-connected graph. Networks **19**, 151-172
8. Cheng, E., Jordán, T. (1999): Successive connectivity augmentation algorithms. Math. Program. **84**. This issue
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L. (1990): Introduction to Algorithms. MIT Press, Cambridge, MA
10. Dinits, E.A., Karzanov, A.V., Lomonosov, M.L. (1976): On the structure of a family of minimal weighted cuts in a graph. In: Fridman, A.A., ed., Studies in Discrete Optimization, pp. 290–306 (in Russian). Nauka, Moscow
11. Fleischer, L. (1997): Building chain and cactus representations of all minimum cuts from Hao–Orlin in the same asymptotic run time. manuscript ■ More information yet? ■
12. Frank, A. (1992): Augmenting graphs to meet edge connectivity requirements. Proc. 31st Annual Symp. on Found. of Comp. Sci.(1990) and SIAM J. Discr. Math. **5**, 25–53
13. Fleiner, T., Jordán, T. (1999): Coverings and structure of crossing families. Math. Program. **84**. This issue
14. Gabow, H.N. (1991): A matroid approach to finding edge connectivity and packing arborescences. Proc. 23rd Annual ACM Symp. on Theory of Comp. 112–122
15. Gabow, H.N. (1991): Applications of a poset representation to edge connectivity and graph rigidity. Proc. 32nd Annual Symp. on Found. of Comp. Sci. 812–821
16. Gabow, H.N. (1991): Applications of a poset representation to edge connectivity and graph rigidity. Dept. of Comp. Sci., University of Colorado, Techn. Rept. CU–CS–545–91
17. Gabow, H.N. (1993): A representation for crossing set families with applications to submodular flow problems. Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 202–211
18. Gabow, H.N. (1994): Efficient Splitting Off Algorithms for Graphs, Proc. 26th Annual ACM Symp. on Theory of Comp., 696–705

19. Goldberg, A.V., Tardos, É., Tarjan, R.E. (1990): Network flow algorithms. In: Korte, Lovász, Prömel Schrijver, eds., Paths, Flows, and VLSI-Layout, pp. 101–164. Springer
20. Goldberg, A.V., Tarjan, R.E. (1988): A new approach to the maximum flow problem. J. ACM **35**, 921–940
21. Gomory, R.E. Hu, T.C. (1961): Multi-terminal network flows. SIAM J. Appl. Math. **9**, 551–560
22. Grötschel, M., Lovász, L., Schrijver, A. (1988): Geometric algorithms and combinatorial optimization. Springer
23. Hao, J. Orlin, J.B. (1992): A faster algorithm for finding the minimum cut in a graph. Proc. of 3rd ACM–SIAM Symposium on Discrete Algorithms 165–174
24. Karger, D.R. (1993): Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. Proc. of 4th ACM–SIAM Symposium on Discrete Algorithms 21–30
25. Karger, D.R. (1996): Global min-cuts in near linear time. Proc. 28th Annual ACM Symp. on Theory of Comp.
26. Karger, D.R., Stein, C. (1993): An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts. Proc. 25th Annual ACM Symp. on Theory of Comp. 757–765
27. Karger, D.R., Stein, C. (1996): A new approach to the min-cut problem. J. ACM **43**(4), 601–640
28. Karzanov, A.V., Timofeev, E.A. (1986): Efficient Algorithms for Finding all Minimal Edge Cuts of a Nonoriented Graph. Cybernetics 156–162, translated from Kibernetika **2**, 8–12
29. Leighton, F.T. (1992): Introduction to parallel algorithms and architectures: arrays, trees, hypercubes. Kaufmann
30. Mader, W. (1978): A reduction method for edge-connectivity in graphs. Annales Discrete Math. **3**, 145–164
31. Mulmuley, K., Vazirani, U.V., Vazirani, V.V. (1987): Matching is as easy as matrix inversion. Combinatorica **7**(1), 105–113
32. Naor, D., Gusfield, D., Martel, Ch. (1990): A fast algorithm for optimally increasing the edge connectivity. Proc. 31st Annual IEEE Symposium on Foundations of Comp. Sci. 698–707
33. Nagamochi H., Ibaraki, T. (1992): A linear time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. Algorithmica **7**, 583–596
34. Naor, D., Vazirani, V.V. (1991): Representing and enumerating edge connectivity cuts in RNC. Proc. Second Workshop on Algorithms and Data Structures. Lect. Notes Comput. Sci. **519**, 273–285. Springer
35. Picard, J.–C., Queyranne, M. (1980): On the structure of all minimum cuts in a network and applications. Math. Program. Study **13**, 8–16
36. Szigeti, Z. (1999): Hypergraph connectivity augmentation. Math. Program. **84**. This issue
37. Tarjan, R.E., Vishkin, U. (1984): Finding biconnected components and computing tree functions in logarithmic parallel time. Proc. 25th Annual IEEE Symp. on Foundations of Comp. Sci. 12–20
38. Watanabe, T., Nakamura, A. (1984): Edge connectivity augmentation problems. Comput. System Sci. Eng. **35** (1987), 96–144