Augmenting undirected connectivity in RNC and in randomized $\tilde{O}(n^3)$ time

András A. Benczúr*
Department of Mathematics
MIT
Cambridge MA 02139
benczur@math.mit.edu

Abstract

In the edge connectivity augmentation problem one wants to find an edge set of minimum total capacity so that if one adds it to the input graph, the maxflow value is at least k between each pair of nodes. In this paper we present the first RNC algorithm to solve this problem. The sequential version of our algorithm has running time $\tilde{O}(\min\{n^3, \tau n^2\})$ for undirected graphs with integer capacities, where τ is the amount by which the connectivity must be increased. The version not using randomization runs in time $O(n^4)$. The most efficient earlier algorithms are those of Gabow with running times $\tilde{O}(k\tau n^2)$ for unit capacities and $\tilde{O}(n^2 m)$ for general capacities, where k is the desired connectivity value.

Our main improvements are the following. First we give a grouping technique, which eliminates the dependence of the running time on the connectivity increment for an algorithm similar to that of Naor et al. Then we provide an algorithm where the groups of augmenting edges are sufficiently independent of each other that they can be computed in parallel. Also we extend the randomized *contraction algorithm* of Karger and Stein for finding connectivity cuts to provide all mincut structures required by the algorithm. We give an $\tilde{O}(n^2)$ randomized algorithm to compute the *cactus representation* of all mincuts, which was introduced by Dinits, Karzanov and Lomonosov; our algorithm is *sublinear* in the total size of the mincuts. The cactus representation is the main building block of the algorithm of Naor et al. The best known previous algorithm, due to Gabow, has running time $\tilde{O}(m + k^2)$ for graphs with unit capacities. Another basic structure needed is the laminar system of *extreme sets* introduced by Watanabe and Nakamura; we provide the first RNC algorithm to compute extreme sets, by modifying the contraction algorithm of Karger.

*Research was done while the author was a student at the Mathematical Institute of the Hungarian Academy of Sciences, and the Department of Computer Science, Eötvös University, Hungary. Author was partially supported by OTKA grants no. 1909, 7559 and 2118

0 Introduction

Let $G = (V, E)$ be an undirected graph with integer capacities. The edge connectivity of a graph G is the maximum value c such that for each pair of nodes s, t, the value of the maximum flow from s to t is at least c. Given a connectivity increment τ, the *edge augmentation problem* is to find an edge set of minimum total capacity whose addition to G increases its edge connectivity by τ. In this paper we present algorithms for undirected graphs with integer capacities. We describe the first RNC algorithm to solve this problem. The sequential version of this algorithm has running time $\tilde{O}(\min\{n^3, \tau n^2\})$ using randomization. (The $\tilde{O}$ notation is used to mean asymptotics up to a polylogarithmic factor.) We also present a deterministic version with running time $O(n^4)$.

There are two approaches to solving the augmentation problem. The first approach works for graphs with unit capacity edges, and repeatedly increases the connectivity by one unit until the desired increment is obtained, hence we call such an algorithm *successive*. This approach was first described by Watanabe and Nakamura [17], who also proved a min-max theorem. An efficient version with running time $O(\tau^2 nm)$ was given by Naor et al. [15] and Gabow [6] gave an $\tilde{O}(k^2 n)$-time modification, where k is the desired connectivity value. These algorithms use elementary manipulations, provided some necessary structures of minimum cuts are known. On the other hand, Frank [3], based on some ideas of Cai and Sun [1], presented a different algorithm which solves the (integral) capacitated case in $O(n^5)$ time. The algorithm uses maxflow algorithms and hence is not really efficient, but it can solve the capacitated problem in polynomial time. Very recently, Gabow [7] improved Frank's running time to $O(n^2 m \log(n^2/m))$.

As Karger and Stein describe in their paper [12], we may well assume that finding (possibly all) mincuts is a reasonably simpler task than finding maxflows. Also, the algorithm of [12] is very efficient for finding mincuts both sequentially *and in parallel*. The best known methods for finding connectivity cuts are summarized in [12]: for graphs with unit (or small) capacities the fastest algorithm is that of Gabow [4], while both for arbitrary integer capacities and for parallel processing the most efficient method is that of Karger and Stein [12] with running time (respectively processor requirement) $O(n^2 \log^3 n)$, provided we allow randomization. This algorithm implicitly finds *all* mincuts as well. Hence in this paper we aim to improve on the first (successive) type of augmentation algorithms to get efficient sequential *and parallel* algorithms, using the algorithms of Karger and Stein. Note however that to modify such "successive" algorithms to run in polynomial time or in NC seems hard: the number of phases of the algorithm is equal to the connectivity increment and these steps seem to depend on previous ones.

We improve on the earlier successive augmentation algorithms, and hence we sketch some of their ideas now. More details are found in Section 1. Observe that in order to augment the connectivity of G by one, we must add one edge to each minimum cut of G. Let S be a minimal set such that $(S|\overline{S})$ defines a mincut. Then in augmenting by 1, one may contract S; in other words the endvertices of an augmenting edge can be placed arbitrarily within S. However, two successive optimal augmentations by one may not be optimal [15], and so to preserve optimality for later augmentation one has to take care of those sets which may become minimal mincuts later. These sets are the so-called *extreme sets* described by Watanabe and Nakamura [17].

The first step we make in improving successive augmentation algorithms is that of eliminating the dependence of their running time on the increment. Note that by the result of Frank [3] we know that the capacitated problem can also be solved in polynomial time. 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. Frank's augmentation algorithm does not have a similar property: a τ_1-edge connected augmentation produced by the algorithm is not necessarily the subgraph of that for some $\tau_2 > \tau_1$.

In this paper we find a successively augmented sequence of graphs in polynomial time by grouping together successive augmentation steps into a polynomial number of large steps. Inside a step essentially the same set of edges has to be added to the graph. Our technique applied in the algorithm of Naor et al. already improves on the running time of Frank's algorithm [3] to $O(n^4)$. We have to show that the properties describing the augmenting edge set change only a polynomial number of times. We show first that consecutive augmentations can be divided into $O(n)$ phases such that the system of extreme sets does not change its structure within a phase, and second, that within a phase, the set of edges to be added can be computed knowing only the mincuts of the *initial* graph *of that phase*. Then we further improve on this so that are able to compute these edges for each phase by using only the *extreme set structure of the input graph*. It will thus be possible to compute augmenting edges *in parallel*.

In Section 2 we give a polynomial time algorithm which, for a given connectivity increment τ, builds a graph G_τ in a sequence satisfying $G_i \subset G_j$ for $i < j$. Crucially, our algorithm is strongly polynomial with respect to both the increment τ and the edge capacities. Next in Section 3 we describe a parallel algorithm using only polylog(n) steps to find an optimally augmented graph. Unlike the previous one, this algorithm will not necessarily produce an increasing sequence of outputs.

In order to achieve an RNC algorithm and to provide the $\tilde{O}(n^3)$ time bound for our sequential version, we have to efficiently compute the extreme sets. Naor et al. show how the extreme sets are naturally described by the *Gomory–Hu tree* of the graph. A Gomory–Hu tree [9] compactly represents a mincut between each pair of nodes. The best known algorithm for computing such a tree requires $n-1$ maxflow computations with total running time $\tilde{O}(n^2 m)$. Gabow [6] has shown that extreme sets can be computed more efficiently for graphs with unit capacities. Instead of flows, this algorithm uses other methods to find connectivity cuts. In Section 4 we present two different extreme-sets-algorithms without using flows.

Our parallel algorithm for finding extreme sets is based on the following fact, which we prove in Section 4.1: the contraction algorithm of Karger [11] does not only find all mincuts, but in some other sense all extreme sets as well. Note that here we are not able to use the most efficient recursive contraction algorithm of Karger and Stein [12]. Improving on this fact would not only yield a more efficient parallel algorithm, but also a better sequential one.

Our sequential extreme sets algorithm repeatedly solves the following problem: given G and a node s of low degree (in a capacitated sense), find all mincuts that are different from $(s|v(G)-s)$. By modifying the algorithm of Karger and Stein, we give an $\tilde{O}(n^2)$ algorithm for this latter task, which yields an $\tilde{O}(n^3)$ algorithm to find the extreme sets.

The trouble one faces when using the recursive contraction algorithm of Karger and Stein [12] is that dealing with a set of $O(n^2)$ cuts means dealing with a set of $O(n^3)$ data: the tight upper bound for the number of mincuts of a graph is $O(n^2)$ ([2] and [12]). Hence if we do not want to exceed the time bound of the contraction algorithm, we are not allowed to write or access all the data. Also note that the same argument suggests that it should be impossible to find all mincuts within an $\tilde{O}(n^2)$ bound. The Karger–Stein algorithm does this by finding a representation requiring $\tilde{O}(n^2)$ space that implicitly contains all mincuts (with some error probability). Let $(A|\overline{A})$ be a cut that is given in this sense. For each v, it requires $O(n)$ time to decide whether $v \in A$ or $v \in \overline{A}$.

We shall use the recursive contraction algorithm in the following way. Dinits, Karzanov and Lomonosov [2] prove a concise *cactus representation* of all mincuts using only $O(n)$

space. This representation contains all the necessary information our algorithms require for mincuts. We shall find the cactus by using at most $O(n^2)$ tests to decide whether $v \in A$ for a cut $(A|\overline{A})$ and a vertex v. We show that the Karger–Stein algorithm can be modified so that such a test takes $O(\log n)$ time. Note that in order to find the edge set covering all mincuts, our algorithm also uses the cactus representation. Naor et al. [15] use the cactus as well, while Gabow [6] uses a similar structure, the m-tree representation of [5].

Thus another contribution in this paper is an algorithm for computing the cactus representation in randomized $\tilde{O}(n^2)$ time. The original paper of Dinits et. al. does not give an algorithm for computing the cactus. An algorithm was given by Karzanov and Timofeev [13] with running time $O(nm)$ for graphs with unit capacities. It requires $n-1$ flows with value at least the connectivity. Hao and Orlin [10] show that such a task can be pipelined to one maxflow computation, which yields an $\tilde{O}(nm)$ algorithm for arbitrary integer capacities. By using other methods than flows for finding mincuts, Gabow [5] gave an $O(m+k^2\log(n/k))$-time algorithm. Using the Mulmuley–Vazirani–Vazirani [14] result for finding mincuts in *parallel*, Naor and Vazirani [16] described a parallel cactus algorithm. Karger and Stein [12] note that by using their parallel algorithm instead of that of [14], the required number of processors decreases. However the processor bound of the contraction algorithm is lower than the space needed to read all mincuts; they do not describe how data should be mainpulated in order not to exceed this processor bound.

1 The generic successive augmentation algorithm

In this section we review the basic definitions and ideas of earlier edge augmentation algorithms ([17], [15]), in a generalized form which will be a basis of our new algorithms. In the paper we always consider *undirected* graphs $G=(V,E)$ with non-negative integer *edge capacities*. For the sake of simplicity, we shall view edges with capacity α as a set of α parallel edges. Note however that the input size will be the number of vertices; the number of operations performed on capacities will be a polynomial in the number of vertices. For $X \subseteq V$, let $\overline{X} = V - X$. Let $d_G(X)$ denote the total capacity of edges between X and $\overline{X}$ (we omit G if it is clear from the context). An (edge) cut is the set of edges leaving a set $C \subset V$; it will also be denoted $(C|\overline{C})$. We shall call the minimum of $d(C)$ over $C \subset V, C \neq \emptyset, V$ the (edge) **connectivity** of G. If G is a c-connected graph, then cuts of capacity c are the *mincuts*. The submodular inequalities are used throughout the paper:

$$d(X) + d(Y) \geq d(X \cup Y) + d(X \cap Y)$$

We say that G' is an **augmentation** of G to connectivity k if G' is k-connected and arises from G by adding new edges. In the **edge augmentation algorithm** we are interested in an

Figure 1: Extreme tree: the laminar system of extreme sets.

augmentation of G to connectivity k which minimizes the total capacity of new edges.

In the next lemma we give a trivial lower bound on the capacity of new edges of an augmentation. In order to do this, we need some definitions regarding set systems. 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 connected if $X \subset Y$ but there is no $Z \in \mathcal{F}$ with $X \subset Z \subset Y$. A **subpartition** $\mathcal{P}$ of V is a set of disjoint subsets of V. $\mathcal{P}$ is a partition if $\bigcup \mathcal{P} = V$.

Definition. We define the k-th **demand** of a set U to be $\mathrm{dem}_k(U) = \max\{0, k - d(U)\}$. The demand of a *subpartition* $\mathcal{P} = \{V_1, \ldots, V_m\}$, $\mathrm{dem}_k(\mathcal{P})$, is the sum of the demands of individual sets. The demand $\mathrm{dem}_k(\mathcal{F})$ of a *laminar system* $\mathcal{F}$ over V is defined recursively, as follows. Let $U \in F$. If U is minimal in $\mathcal{F}$, let $\mathrm{dem}_k(U, \mathcal{F}) = \mathrm{dem}_k(U)$. If $\mathrm{dem}_k(U_i, \mathcal{F})$ is defined for all maximal subsets $U_1, \ldots, U_m$ of U, we define $\mathrm{dem}_k(U, \mathcal{F}) = \max\{\mathrm{dem}_k(U), \sum_{i=1}^m \mathrm{dem}_k(U_i, \mathcal{F})\}$. Then $\mathrm{dem}_k(\mathcal{F}) = \mathrm{dem}_k(V, \mathcal{F})$.

Lemma 1.1. *The minimum total capacity of edges needed to augment G to connectivity k is at least $\mathrm{dem}_k(\mathcal{F})/2$, where $\mathcal{F}$ is an arbitrary laminar system over V.*

The algorithms of both [17] and [15] agree in that the connectivity is increased successively one by one. As a simple example of [15] shows, the difficulty in designing such an algorithm is that an optimum augmentation by one, followed by another optimum augmentation by one does not yield an optimum augmentation by two. The reason why this is the case is the following. An augmentation of a c-connected graph to connectivity $c+1$ can be solved by looking at sets with degree c; within a minimal set with degree c the endvertices can be placed arbitrarily. However for larger connectivity increments these sets may have subsets with positive demand and the placement of all edges must be made so as to satisfy these demands as well.

The correct sets to consider are not only sets of minimum degree, but all sets which are minimal with respect to some fixed demand, as noted by Watanabe and Nakamura [17]. Formally, we call a subset $U \subset V$ ℓ-**extreme** if $d(U) = \ell$ and $d(U') > \ell$ for all $U' \subset U$. These sets form a laminar system: to show that two extreme sets U and W cannot intersect, we apply submodularity to get that otherwise either $d(U - W) \leq d(U)$ or $d(W - U) \leq d(W)$. We denote both the laminar system and the tree corresponding to it by $\mathcal{F}(G)$ (or $\mathcal{F}$). This tree is called the **extreme tree** of G (see Fig. 1, another example is in Fig. 3). Since an ℓ-extreme set U cannot contain an ℓ'-extreme set $W \neq U$ for $\ell' \leq \ell$, the extreme tree can be divided into

Box 1: **Generic algorithm**

`input`: graph G, target k, and the extreme tree $\mathcal{F}$ of G

`connect maximals`: choose pairs of (inclusionwise) maximal sets of $\mathcal{F}$ to be connected by new edges

`endvertex placement`: push down edges to the leaves of the extreme tree, by repeating until possible:

    let edge e be chosen to enter $U \in \mathcal{F}$,
    let U_1 be a child of U with positive k-demand,
    choose e to enter U_1.

remove non-extreme or 0 k-demand sets from $\mathcal{F}$
repeat until there exist sets with positive k-demand

levels so that level ℓ contains the ℓ-extreme sets. In [17] it is proved that this system which maximizes the demand over all laminar systems.

The generic framework of the type of successive augmentation algorithm we investigate is in Box 1. The fact that the output graph is k-connected must be verified for the particular implementations of the `connect maximals` step. The output is optimal if the number of endvertices of new edges isthe same as the extreme demand. This fact is ensured by showing that each new edge satisfies a demand of the *initial* extreme tree $\mathcal{F}$. Let an endvertex of a new edge be assigned to some $v \in V$. Then we shall require that the minimal set of $\mathcal{F}$ containing v has no child in $\mathcal{F}$ with remaining positive demand; thus the demand of $\mathcal{F}$ as a laminar system decreases by one for each new vertex. Because of the leveled structure of $\mathcal{F}$, the condition on new vertices can be guaranteed by recursively "pushing them down" in $\mathcal{F}$, as described by the `endvertex placement` step.

Theorem 1. *If the generic algorithm outputs a k-connected graph, it is an optimum augmentation.* □

A final note is that in each implementation we describe it will be true that intermediate augmented graphs contain no new extreme sets compared to $\mathcal{F}$. Hence after removing non-extreme or 0 k-demand sets from $\mathcal{F}$, we get the extreme tree of the intermediate graph. This fact proves useful in the analysis of correctness (altough optimality is proved with respect to the *initial* extreme tree as a laminar system).

1.1 Connecting maximals à la Naor et al.: the cactus representation.

In order to augment connectivity by one, a set of edges covering all mincuts is needed. The basic similarity in the `connect maximals` step of earlier algorithms is the use of some kind of structure representation of all mincuts. In Naor et al. [15] the **cactus** structure of mincuts [2] is used. A cactus is a multigraph which contains no cut edges and no two cycles with common edge. In other words a cactus is built up from a single vertex by recursively joining cycles to existing vertices. The cactus representation (Fig. 2) of a graph G is a cactus $\mathcal{K}$ such that a

Figure 2: A 6-edge-connected graph and the cactus representation of its mincuts.

partition of V corresponds to the vertex set of the cactus. The mincuts of the cactus are precisely those which arise by erasing two edges of a cycle. Then the mincuts of G are precisely the edge sets of G connecting two components of a mincut of the cactus.

Now the (in our terms) `connect maximals` step of Naor et al. is described in a simple way as follows. It is a simple fact [15] that degree-2 vertices of the cactus are precisely the extreme sets with *maximal demand*. We notice that the cactus is an Eulerian graph and consider *degree-2* vertices ordered around this cycle. Then the step is to connect pairs which are opposite in this cyclic order. Note that this is possible only if the number of degree-2 vertices is even; if this number is odd, we have to add two edges to one of these extreme sets. In the paper of Naor et al. the `endvertex placement` is handled separately in this case. In our more general setting of the `endvertex placement` step we can (and shall) add more than one edge to an extreme set at the same time.

1.2 The generic grouping idea.

The drawback of using representations of mincuts in the `connect maximals` step is that the structure of mincuts is sensitive to the addition of new edges to the graph. Whenever an augmentation step is made, the representation has to be recomputed; hence the running time of the algorithm will be inherently dependent on the connectivity increment value.

The basic idea of our new augmentation algorithm is the following: an arbitrary Hamiltonian cycle increases by at least 2 the capacity of *each* cut. In the next two sections we shall see two different implementations of this idea, where cycles are added to connect maximal extreme sets. Both implementations will basically depend only on the extreme tree of the graph. We shall call two extreme trees **structurally equivalent** if the `connect maximals` and `endvertex placement` steps behave in the same way on the underlying graphs. As opposed to mincut structure, the extreme tree turns out to be robust in the sense that the number of structural changes during the entire augmentation procedure will be $O(n)$. Hence we get a polynomial time algorithm by determining the number of steps with structurally equivalent extreme trees and compute all edges to be added at once.

2 The No Target problem: the "Balanced" algorithm

In this section we propose a polynomial time algorithm which, for a given connectivity increment k, builds a graph G_k in a sequence satisfying $G_i \subset G_j$ for $i < j$. In other words, the output can be considered as if instead of specifying the target in advance, we bound the number of edges we may add to the input graph: the algorithm must halt when edges run out, and the output graph must still be an optimum augmentation. This task is called the **no-target problem** in [15]. Next we describe our `connect maximals` step, which is the key in achieving polynomial running time even if the increment is large. The proof of correctness of the algorithm is similar to that in [15]; it is sketched in the same subsection. The proof of the polynomial running time is in Section 2.3 together with the discussion of the `endvertex placement` rules.

2.1 Balanced cycles: first step towards grouping.

Let $U_1, \ldots, U_m$ be the *maximum demand* extreme sets, in an arbitrary order (see Fig. 1). Assume that every other maximal extreme set has demand by at least 2 less than the U_i, and furthermore that each U_i remains extreme after the `endvertex placement` of 2 new edges entering it. Then the **balanced cycle** step is to connect U_i to U_{i+1} for $i = 1, \ldots, m-1$ and U_m to U_1.

When one `endvertex placement` destroys one U_i or causes a new set to have maximum demand, we cannot apply our step. To overcome such "parity-like" problems, we have to use the cactus representation (the method of [15]) as described in Section 1.1. In order to solve the no-target problem, we may and shall assume that the edge set described by the balanced rule contains those edges the rule of Naor et al. [15] add to the graph. Finally, note that in deciding which rule to use, explicit reference is made to the `endvertex placement` rule, which we describe only in the next subsection.

Now we prove that using our `connect maximals` step, the algorithm outputs an optimally augmented graph. By Theorem 1 it suffices to show that this graph has the required connectivity. The algorithm at each step increases the degree of initially extreme sets with maximum remaining demand. Hence to complete the proof it suffices to show that it is impossible that after some steps of the main loop, a new extreme set is created in the graph. This is true when we use the `connect maximals` step of Section 1.1; the proof is simple and can be found in [15]. For balanced cycles, this same proof can easily be modified; we omit both proofs. We note that in Section 3.1 we do prove that no new extreme set is created for another `connect maximals` step; that proof is more complex but contains the same ideas as what we need here.

2.2 Balanced and Bold endvertex placements.

At a general stage of the `endvertex placement` we decide to place an endvertex of an augmenting edge in an extreme set U (see Box 1). Until U has extreme subsets with positive demand, we have to select one of them to place the endvertex to it. We

Figure 3: Example when balanced placement has long periods because of the large least common multiple of degrees. Nodes of the graph on the left contain the 200-extreme sets.

present two different, natural rules to select this extreme subset. The first rule is basically that of [15].

Balanced placement: choose the first extreme subset with *maximum demand*.

Bold placement: choose the first extreme subset with *arbitrary positive* demand.

Let us compare the two placement rules. First of all it is important that only the balanced placement solves the no-target problem: if k is not given in advance, demands are undefined in the algorithm and we cannot implement the bold rule. On the other hand if there is an extreme subset with positive k-demand for some k, then those picked by the balanceed rule (with maximum demand) certainly are such ones.

As for implementing the two rules, the bold rule has the advantage of simplicity: it selects the same set (and vertex), as long as possible. The implementation is particularly easy and runs in linear time. The balanced rule, however, will cyclically go through the the set of maximum and maximum-1 demand children of a fixed extreme set. A change to this set is a *structural change* of the extreme tree with respect to the balanced rule. To perform the balanced placement between two structural changes, instead of computing augmenting edge sets successively one by one, we compute the period lengths arising by this cyclic placement. We do not describe in detail how this can be done: we traverse each extreme set once and make calculations with numbers refering to period lengths; these numbers are $O(n)$-th powers of degrees, hence have size $O(n \log n)$. The time required to compute all augmenting edges is thus $O(n^2 \log n)$.

To justify why we cannot implement the balanced rule directly without period length computations, we show the following: even without a structural change in the extreme tree, the sequence of the final vertices selected by the `endvertex placement` can have *exponentially long* periods! The final period lengths at vertices will be the least common multiple of the number of children with maximum demand for some extreme sets. When these numbers are distinct primes for different extreme sets, the period length, which is the product of primes summing up to $\Omega(n)$, can be exponentially large. On the example of Fig. 3, this period length is 210.

Finally we prove that the number of structural changes of the extreme tree with an algorithm using balanced cycles is $O(n)$, whichever of the two `endvertex placement` rules we choose. The structural changes are summarized as follows. If an extreme set becomes non-extreme, it affects both `endvertex placement` rules if the set is not on the top level; if it does have maximal demand then its removal affects the balanced cycles. It will also affect the cycles if by decreasing the maximum demands a new set attains the maximum demand. There are no more changes if we use the bold rule, since no new extreme set is created during the augmentation procedure (see previous subsection). For the balanced rule, as we noted, a further structural change is when, for an arbitrary extreme set U, a new child attaines maximum demand. Each of the above changes can be caused by an extreme set at most once, thus $O(n)$ times altogether. This proves our claim.

Theorem 2. *The augmentation algorithm with balanced cycles requires the computation of the initial extreme tree and $O(n)$ cactus representations. $O(n^3)$ extra time is needed if we apply balanced cycles; this time is $O(n^2)$ for bold cycles.* □

3 RNC implementation: the "Bold" algorithm

The Bold algorithm does not aim to build a sequence of successively augmented graphs. The philosophy of the bold `connect maximals` step is to add edges whenever it is "safe" to, without using any information not described by the extreme tree. It will turn out that dependencies of later augmentation steps on earlier ones are low enough so that the algorithm is parallelizible. Most importantly, the augmenting-by-one step of [15] (Section 1.1) is used only twice, at the end of the algorithm; in other words those "parity-like" problems in the balanced algorithm are all postponed to the last phase. The (sequential) time requirement of the bold algorithm is bounded by the computation of the extreme tree, two cactus representations, and $O(n^2)$ time to perform the bold `endvertex placement` steps.

3.1 Bold cycles and correctness.

Let $U_1, \ldots, U_m$ be the (inclusionwise) maximal extreme sets with k-demand at least two, in an arbitrary order (see Fig. 1). Provided there is a U_i with k-demand at least 3, the **bold cycle** step is to connect U_i to U_{i+1} for $i = 1, \ldots, m$ and U_m to U_1.

The first difference we notice in comparing bold to balanced cycles is that (almost) all maximal sets, which do not necessarily have maximum *demand*, are connected. But there is another, even more important difference: we do not check whether two endvertices can be added to an extreme set U_i without destroying it. Hence starting with a c-connected graph, in one step we always get a $(c+1)$-connected graph; the graph is however not necessarily $(c+2)$-connected if a c-extreme set has $(c+1)$-extreme children. This pecularity means that at demand 2, the augmenting edges may become non-optimal. This is justified in Lemma 3.1 where it turns out (and indeed it is easy to construct examples) that a new set may become extreme and the demand can increase then. The correctness of the algorithm is proved exactly as in Section 2.1 for the balanced algorithm; here we only give the step of the proof that no new set may become extreme while we use bold cycles with maximum demand at least 3.

Lemma 3.1. *Let us add one bold cycle to a graph G and apply bold* `endvertex placement` *to it to get a graph G'. Then if U is extreme in G', then so is in G. Hence the bold algorithm is correct.*

Proof. The reason why U could not be extreme in G is that there is an extreme set of G, $U' \subset U$ with $d_G(U') \le d_G(U)$. Hence the number of new edges in the boundary is strictly greater for U' than for U. Also, the number of such edges is at most two for any extreme set of G, in particular for U'. Hence there is at most one new edge leaving U. There are two cases when this is possible: either

(1) U contains all vertices of the bold cycle, or
(2) U crosses a maximal extreme set W of G and the two endvertices added to W are in $W - U$ and $W \cap U$.

In case (1), U intersects or contains all maximal extreme sets with demand at least 2. Let G be c-connected; then $d_{G'}(U) \le c+1$ by definition of extreme sets. Since in case (1) $d_G(U) = d_{G'}(U)$, there is a $d_G(U)$-extreme set $A' \subseteq \overline{U}$. Let $A \supseteq A'$ be maximal extreme in G. If $A \subseteq \overline{U}$, then there is no new edge added to A, whence dem$(A) = 1$ but $d_G(A) \le c+1$, contradicting the assumption that the demand was at least 3 before the bold cycle step. If $A \not\subseteq \overline{U}$, then $A \ne A'$ and hence $d_G(A) = c$, $d_G(U) = c+1$. By the definition of extreme sets $d_G(A-U) > c$ and $d_G(U-A) = d_{G'}(U-A) > c+1$, contradicting submodularity. Finally in case (2) by a similar reference to the definition of extreme sets, $d_{G'}(U-W) > d_{G'}(U)$, $d_{G'}(W-U) = d_G(W-U)+1 > d_G(W)+1 = d_{G'}(W)-1$, contradicting submodularity in G'. □

3.2 Unbalanced trees: the basic parallelization idea.

The basic algorithmic structure in computing the demands and implementing `endvertex placement` is a recursive evaluation on the extreme tree, as follows. Let us fix a node of the extreme tree. At one step if the computation on each of its children is ready, we perform the computation on this node. One processor can be assigned to each node; then the number of parallel steps this algorithm takes equals to the depth of the tree. Unfortunately, this can be $\Omega(n)$, if the extreme tree is unbalanced. We propose the following algorithm to overcome this difficulty. At a step we partially do the computation for all nodes which have *at most one* child where the computation is not completed. Also, if there is a subpath of the tree consisting of in the above sense partially computed nodes, we evaluate the algorithm on this path in one step. To show that a particular application of this scheme is an NC algorithm, we use the following fact, which can be proved by simple induction.

Lemma 3.2. *In step i of the above algorithm, computation terminates at each node which has at most $2^i - 2$ successors.* □

It is obvious that (by summing numbers in NC) the demand of the extreme tree can be computed by this algorithm. In the next subsection we show how endvertex placement can be done in such a way, in NC.

3.3 Sketch of the Bold NC algorithm.

By using bold cycles and `endvertex placement`, from Section 2.2 we know that we have to add $O(n)$ different augmenting edge sets to the graph. Now we aim to compute in parallel all these edge sets and the first and last step of augmentation when they are to be added. In order to do this, we have to know, after adding i bold cycles, which sets are maximal extreme and where to place the next endvertex inside them. We compute the following data for each extreme set U:

(1) For each $v \in U$, the first time v is chosen in a sequence of `endvertex placement` steps to this set. Note that bold placement always selects the same vertex in a consecutive sequence of steps.

(2) The sequence of configurations of maximal extreme subsets of U, together with the number of steps when configurations change. Here the placement of two endvertices to all current maximal extreme subsets of the set is considered as one step.

For the computation of (1) we assign one processor for each extreme set–vertex pair. Then the algorithm of Section 3.2 can be applied in a straightforward way with the following modification of the `endvertex placement`: whenever we have to *partially* compute (1) for an extreme set, we assume that the unique child with unknown data is selected *last*. And for (2), we have to determine the first time when the top extreme set U becomes non-extreme and copy the data of sets which are maximal extreme at that time. Extra care is needed when evaluating a path of extreme sets $U_1 \subset U_2 \subset \ldots \subset U_s$ in one step: we have to determine the number of edges which can be added to each U_i before it gets destroyed. By the modified `endvertex placement`, this will happen either when all demands in $U_{i+1} - U_i$ are satisfied or when U_i is top extreme. We omit further details. Finally when (2) is known for all initially maximal extreme sets, we can sort the sequence of times a change happens and compute the $O(n)$ augmenting edge sets in parallel.

4 Extreme tree and cactus algorithms

Theorem 4. *Both the balanced and bold algorithm runs in randomized $O(n^3 \log^3 n)$ and deterministic $O(n^4)$ time. The bold algorithm is in RNC.*

To prove this theorem, we shall provide deterministic, randomized and parallel algorithms for computing extreme trees and cactus representations. Let us summarize the algorithms we present for these tasks. Naor and Vazirani [16] describe an RNC algorithm for computing the cactus. In Section 4.2 we show how to compute the extreme tree in RNC. In Section 4.4 we extend the algorithm of Karger and Stein [12] to compute a cactus in time $\tilde{O}(n^2)$. Then in Section 4.5 we show that the extreme tree $\mathcal{F}(G)$ can be computed by determining $O(n)$ times the connectivity in the presence of a low degree node and computing cactus representations. Finally for the deterministic version, Naor et al. show how the extreme tree is described by the *Gomory–Hu tree* [9] of the graph; computing the Gomory–Hu tree requires $n-1$ maxflow computations. As for the cactus representation, with the algorithm of Karzanov and Timofeev it can also be found by maxflow computations. These flows can be pipelined together to a single maxflow algorithm as described by Hao and Orlin [10]. Since we have to compute flows on augmented graphs where the number of edges can increase up to $\Omega(n^2)$, the flow algorithm of Goldberg and Tarjan [8] has running time $O(n^3)$. This gives the deterministic running time.

4.1 The contraction algorithm and extreme sets.

Karger [11] describes the **contraction algorithm** as follows. We repeatedly pick a random edge of the existing graph and contract its endvertices. The procedure stops when the graph has 2 remaining vertices. A cut **survives the contractions** if no cut edge is contracted, i.e. if the two vertices of the final graph are the contracted components of the cut. In [11] it is shown that a fixed *mincut* survives contractions with probability $\Omega(n^{-2})$.

We show that the contraction algorithm finds not only mincuts, but extreme sets as well. Let U be an arbitrary extreme set. If we contract $\overline{U}$ to a single vertex, $(U|\overline{U})$ becomes the (unique) mincut of the graph. If we apply the contraction algorithm to this graph, U survives contractions with probability $\Omega(|U|^{-2}) = \Omega(n^{-2})$. 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 $\Omega(n^{-2})$.

4.2 RNC algorithm for extreme trees.

In the previous subsection we showed that in a sequence of random edge contractions each extreme set has an $\Omega(n^{-2})$ probability of being contracted to a single vertex at some step. Karger [11] present a RNC algorithm to find all *mincuts* by repeating this procedure $O(n^2 \log n)$ times in parallel, stopping contractions with two vertices remaining. It is possible to modify this RNC algorithm to produce all sets contracted to a single vertex at some intermediate step. There will be $n-2$ such sets for each single contraction procedure, excluding vertices of the original graph. With $O(\mathrm{poly}^{-1}(n))$ error probability, this system contains all extreme sets. Next we sketch an NC algorithm to select extreme sets from them and build the extreme tree.

1. For all $v \in V$, consider all sets which contain v. Sort these sets by size.
2. For each value of the size $(1, \ldots, n)$, sort sets with given size by degree. Build an increasing chain by removing non-minimal sets.
3. Remove all sets from the chain built in Step 2 where the degree does not decrease. Now all sets are either extreme or

contain no extreme set with strictly smaller degree. These latter sets are to be removed in the last step.

4. Build the extreme tree by the algorithm of Section 3.2: in step i add all sets to the tree with size at most 2^i.

4.3 The recursive contraction algorithm.

The most efficient algorithm to find all mincuts is the recursive contraction algorithm of Karger and Stein [12], with running time $O(n^2 \log^3 n)$. Since the number of mincuts can be as large as $\Omega(n^2)$ ([2] and [12]), which means $O(n^3)$ data, what the algorithm produces is some kind of representation with $\tilde{O}(n^2)$ size. Let us denote by $\texttt{contract}\,(G,k)$ the output of the contraction algorithm (Section 4.1) stopped when a graph is contracted to k vertices. The **recursive contraction algorithm** builds a binary tree such that a graph corresponds to each vertex of this tree. Each edge (G_0, G_1) of this tree is given by an operation $G_1 = \texttt{contract}\,(G_0, n/\sqrt{2})$. Leaves of this tree are graphs with two vertices, i.e. can be considered as outputs of a contraction algorithm. As Karger and Stein [12] show, a fixed mincut survives all contractions up to at least one of the leaves with probability $\Omega(1/\log n)$.

The difficulty in accessing data produced by the recursive contraction algorithm is that cuts are found *implicitly* only. The reason is the following. Let n_0 be the size of the vertex set of an intermediate contracted graph G_0; n_0 may be $o(n)$. It is crucial for the efficiency of the recursive contraction algorithm that $\texttt{contract}\,(G_0, k)$ spends $O({n_0}^2)$ time on G_0. But in this case we are not allowed to carry information on the original graph.

In the next applications we need to decide whether $v \in A$ for a vertex v and cut $(A|\overline{A})$. 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 recursive contraction algorithm a single contraction can spend $O(n_0)$ time on G_0 by using incidence 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.

We note that although the recursive contraction algorithm can also run in RNC, it cannot be applied to find the extreme tree as described in Section 4.2. If it were possible to modify the recursive contraction algorithm to find all extreme sets, it would not only make the RNC augmentation algorithm more efficient, but also the sequential one. Our sequential extreme tree algorithm (presented in the last subsection) will use $O(n)$ calls of the recursive contraction algorithm, which will be the bottleneck in the performance of the *bold* algorithm.

Figure 4: *Left*: initial cactus of the phase, $\{v_1, \dots, v_{i+1}\}$ is contained by the vertex denoted u; *Middle*: cycles incident to u; *Right*: final cactus of the phase, new edges denoted thick.

4.4 Accessing data of the recursive contraction algorithm: the cactus representation of mincuts.

We show how to access data produced by the recursive contraction algorithm of Karger and Stein [12], within a time not exceeding its running time. As we described in the previous subsection, the recursive contraction algorithm produces some representation of all mincuts; however this representation is impractical in the sense that it takes too long to perform operations such as finding minimal cuts. Another representation of all mincuts is the cactus of Dinits et al. [2] we described in Section 1.1 (see also Fig. 2). In this subsection we give an algorithm for computing the cactus representation in $O(n^2 \log n)$ time, provided all mincuts are found by the algorithm of Karger–Stein. The cactus represenation is required on one hand as part of the augmentation algorithm, on the other hand we use it to access mincuts for the extreme tree computation in the next subsection.

The only known sequential algorithm for computing the cactus is that of Karzanov and Timofeev [13]. We show how to modify this algorithm so that it will not require reading all the possible $\Omega(n^2)$ cuts. Instead, for the majority of cuts we only need to test whether two given vertices are separated by them or not; we showed in the previous subsection that such a test can be done in $O(\log n)$ time. We shall find the cactus by using at most $O(n^2)$ such tests.

The high-level approach of the Karzanov-Timofeev algorithm [13] is the following: we start with a single node labeled $\{v_1, \dots, v_n\}$ and one-by-one add cuts $(A|\overline{A})$ to the already defined cactus. Let $\{v_1, \dots, v_n\}$ be a numbering of V such that there is always an edge connecting v_{i+1} to $\{v_1, \dots, v_i\}$. Let $\mathcal{S}_i$ be the set of mincuts $(A|\overline{A})$ with $\{v_1, \dots, v_i\} \subseteq A$ but $v_{i+1} \in \overline{A}$. The important property is that $\mathcal{S}_i$ is a **chain of increasing sets** and hence $|\mathcal{S}_i| = O(n)$ ([13]). Assume we have added all cuts of $\mathcal{S}_j$ to the cactus for $j > i$. Let u be the node of the cactus containing $\{v_1, \dots, v_{i+1}\}$. Now we pick a cut $(A|\overline{A}) \in \mathcal{S}_i$. It is possible that $(A|\overline{A})$ separates incident vertices of *no* cycle containing u; then we have to add a new length-2 cycle to the cactus. Otherwise for a cycle C of the existing cactus, $w \in A$, $w' \in \overline{A}$ for two neighbors w and w' of u in C. If this holds, we say that $(A|\overline{A})$ **divides the cycle** C, and we shall have to add an extra edge to this cycle (see Fig. 4). For details on how cuts should be added see [16].

The naive implementation of adding cuts requires checking almost all cut–cycle pairs whether the cycle is divided by the cut. It is easy to see that this could require $\Omega(n^3)$ such tests altogether, for example it is possible that there are $\Omega(n)$ short cycles incident to u and one long cycle which gradually increases by an edge in $\Omega(n)$ steps.

Box 2: **The cactus algorithm**

```
for i = n − 1 downto 1 do
  until there exists (A|Ā) ∈ S_i do
    add (A|Ā) to K,
    if the (∗)-test shows that (A|Ā)
          divides a cycle C then
      let w and w′ be incident to u in C,
      for all (B|B̄) ∈ S_i do
        if by the (∗∗)-test w ∈ B and w′ ∈ B
         then let S_i = S_i − (B|B̄),
      end
    end
  end
  remove redundant nodes from cactus,
end
```

We give a more sophisticated way of adding the new cuts of $\mathcal{S}_i$ to the existing cactus $\mathcal{K}(G_{i+1})$. In Box 2 we describe the algorithm with the following basic structure. First we determine A for an $(A|\overline{A}) \in \mathcal{S}_i$. If we are lucky, it divides a cycle C. Then we remove A from $\mathcal{S}_i$ and test each cut of $\mathcal{S}_i$ whether it divides cycle C. We may and shall remove all cuts from $\mathcal{S}_i$ which divides C, since they are all represented already. If it turns out that $(A|\overline{A})$ divides no cycle, we create a new cycle of length 2. Then repeat this procedure until $\mathcal{S}_i = \emptyset$. Information on cuts are used in two different ways. When checking whether A divides a cycle C, we need

$(*)$ the list of vertices on both sides of the cut,

but for checking whether a cut divides the fixed cycle C, we want to know

$(**)$ whether $(A|\overline{A})$ separates two given vertices.

Lemma 4.1. *Provided a list of mincuts is given, the cactus representaton can be built by using $O(n^2)$ $(**)$-tests. Hence our algorithm finds the cactus with high probability in $O(n^2 \log^3 n)$ time.*

Proof. We estimate the number of $(*)$-tests first; such a step can be described as n $(**)$-steps. Assume $\mathcal{S}_i$ added no redundant edges to the cactus. Then the number of cuts $(A|\overline{A})$ investigated in the main loop would be equal to the number of edges of the cactus. And if there are redundant edges, the number of them is at most twice the size of the cactus. Hence the number of $(*)$-steps is bounded by a constant times the size of the cactus, $O(n)$ ([13]). As for the number of $(**)$-steps, such steps are performed with all remaining elements of $\mathcal{S}_i$ only for those cycles which are divided by some cut. Hence their number is bounded by the number of newly created edges times $|\mathcal{S}_i|$. Altogether this means $O(n^2)$ steps are performed, since both the size of $|\mathcal{S}_i|$ and the cactus is $O(n)$. □

4.5 The sequential extreme tree algorithm.

We compute the extreme tree recursively. Assume we have found an extreme set U and some of its children $U_1, \ldots, U_s$. Then we search for extreme sets with next minimum degree inside U by contracting $U_1 \cup \ldots U_s \cup \overline{U}$ to a single vertex s. In this graph $G' = (V', E')$, extreme sets are cuts different from $(s|V' - s)$ with minimum degree. By reasoning similar to that in the correctness proof of the recursive contraction algorithm, we can show that all such cuts can be found if we stop the recursive contraction algorithm when graphs are contracted to *three* vertices. Note that finding extreme sets inside similar contracted graphs is the high-level approach of the algorithm of Gabow [6] for finding extreme sets as well. Our algorithm also works for large edge capacities.

The algorithm is based on the simple fact [15] that extreme sets with maximum demand are the degree-2 nodes of the cactus representation. Below we describe the algorithm; proof of correctness (Lemma 4.2) is omitted. The only point in the proof is that if W is extreme but not a degree-2 node of $\mathcal{K}_1$, then by minimality of extreme sets, W is a subset of the node of sv_1 in $\mathcal{K}_1$.

1. Find all cuts $(A|\overline{A})$ of G' different from $(s|V' - s)$ with $d(A)$ minimum.

2. Find a vertex v_1 not separated from s by at least one mincut and compute the cactus $\mathcal{K}_1$ of G', after contracting v_1 to s.

3. Compute another cactus $\mathcal{K}_2$ after contracting to s a vertex not in the same node of $\mathcal{K}_1$ as s.

4. Finally, extract extreme sets from the set system given in the next lemma.

Lemma 4.2. *Extreme sets are included in the system consisting of the degree-2 nodes of $\mathcal{K}_1$ and $\mathcal{K}_2$ and U itself. Thus the extreme tree can be found in $O(n^3 \log^3 n)$ time.* □

Acknowledgment.

The author would like to thank András Frank for the discussions, questions and inspiration that led to finding the grouping technique. Michel Goemans, Jon Kleinberg, David Shmoys and Éva Tardos were particularly helpful with reading and commenting various versions of this paper.

References

[1] Cai, G–P. and Y–G. Sun, The minimum augmentation of any graph to a k-edge-connected graph, *Networks*, **19** (1989), pp. 151-172.

[2] Dinits, E.A., A.V. Karzanov and M.L. Lomonosov, On the structure of a family of minimal weighted cuts in a graph, *Studies in Discrete Optimization* (in Russian), A.A. Fridman (Ed), Nauka, Moscow (1976), pp. 290–306.

[3] Frank, A., Augmenting graphs to meet edge connectivity requirements, *Proc. 31st Annual Symp. on Found. of Comp. Sci.* (1990), and *SIAM J. Discr. Math.* **5** (1992) No. 1, pp. 25–53.

[4] Gabow, H.N., A matroid approach to finding edge connectivity and packing arborescences, *Proc. 23rd Annual ACM Symp. on Theory of Comp.* (1991), pp. 112–122.

[5] Gabow, H.N., Applications of a poset representation to edge connectivity and graph rigidity, *Proc. 32nd Annual Symp. on Found. of Comp. Sci.* (1991), pp. 812–821.

[6] Gabow, H.N., Applications of a poset representation to edge connectivity and graph rigidity, *Dept. of Comp. Sci., University of Colorado, Techn. Rept.* CU–CS–545–91 (1991).

[7] Gabow, H.N., Efficient Splitting Off Algorithms for Graphs, *these proceedings.*

[8] Goldberg, A.V. and R.E. Tarjan, A new approach to the maximum flow problem, *Journal of the ACM*, **35** (1988), pp. 921–940.

[9] Gomory, R.E. and T.C. Hu, Multi-terminal network flows. *SIAM J. Appl. Math.* **9** (1961), pp. 551–560.

[10] Hao, J. and J. B. Orlin, A faster algorithm for finding the minimum cut in a graph, *Proc. of 3rd ACM–SIAM Symposium on Discrete Algorithms* (1992), pp. 165–174.

[11] Karger, D.R., Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm, *Proc. of 4th ACM–SIAM Symposium on Discrete Algorithms*, (1993), pp. 21–30.

[12] Karger, D.R. and C. Stein, An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts, *Proc. 25th Annual ACM Symp. on Theory of Comp.* (1993), pp. 757–765.

[13] Karzanov, A.V. and E.A. Timofeev, Efficient Algorithms for Finding all Minimal Edge Cuts of a Nonoriented Graph, *Cybernetics* 156–162, translated from *Kibernetika* **2** (1986), pp. 8–12.

[14] Mulmuley, K., U.V. Vazirani and V.V. Vazirani, Matching is as easy as matrix inversion, *Combinatorica* **7**(1) (1987), pp. 105–113

[15] Naor, D., D. Gusfield, Ch. Martel, A fast algorithm for optimally increasing the edge connectivity, *Proc. 31st Annual IEEE Symposium on Foundations of Comp. Sci.* (1990), pp. 698–707.

[16] Naor, D. and V.V. Vazirani, Representing and enumerating edge connectivity cuts in RNC, *Proc. Second Workshop on Algorithms and Data Structures* (1991), Lecture Notes in Computer Science **519**, Springer–Verlag, pp. 273–285.

[17] Watanabe, T. and A. Nakamura, Edge connectivity augmentation problems, *Comp. System Sci.* **35** (1987), pp. 96–144.