Final Reading Project:

Polylogarithmic Approximation Algorithn for Routing with Constant Congestion

Tal Wagner*

December 11, 2014

1 Introduction

The Edge-Disjoint Paths (EDP) problem is a fundamental NP-hard routing problem, in which we are asked to connect given pairs of nodes in a graph with paths not sharing any edges. A natural relaxation is to allow some edges to be shared by a small number of paths, a variant known as EDP with Congestion (EDPwC).[1]

We survey a remarkable sequence of breakthrough results that significantly improved the known approximation bound, achieving the first polylogarithmic approximation with constant congestion. The main algorithm we present is due to Chuzhoy [Chu12]. It relies on much previous work and merges a broad variety of concepts and ideas, and so our goal will be to survey each of the components and explain how they fit together. We will usually not provide full analysis (doing this for each component would easily span dozens of pages), but rather try to extract and convey the main ideas.

1.1 Setting and Notation

We fix the following terminology and notation for the entire report. We are given an instance undirected and unweighted graph $G(V, E)$, with $|V| = n$ nodes and k demand pairs $(s_i, t_i)_{i=1}^{k}$ that are contained in V. Our goal is to connect as many demand pairs as possible with paths, such that each edge in E is used by at most c paths, where c is an integer congestion constant. We can assume w.l.o.g. that each node participates in at most one demand pair. The transformation allowing this can be left as a simple exercise; solution is given in footnote.[2]

A node participating in a demand pair is called a *terminal*, and we denote the subset of terminals by T (so $T = \cup_i \{s_i, t_i\}$). In many parts of the discussion it will convenient to discuss terminals in general, regardless of how they are paired. We will abuse notation by letting k denote also the size of T (rather than $2k$), to avoid confusing constants.

For any disjoint subsets of nodes A and B, we denote by $E(A, B)$ the subset of edges with one endpoint in A and one endpoint in B.

We use $poly \log k$ to denote any polylogarithmic factor in k, i.e. $O(\log^b k)$ for any constant b.

*Email: `talw@mit.edu`

[1]In PSet10 we tackled a related variant of EDP, in which had to route all the pair while minimizing the maximum congestion on any edge. We achieved $O(\log n)$ congestion; the best known bound is $O(\log n / \log \log n)$ ([RT87]).

[2]If s is a terminal participating in two demand pairs (s, t_1) and (s, t_2), we add two new vertices s_1, s_2 and edges s_1-s and s_2-s. We then replace the demand pairs (s, t_1) and (s, t_2) with (s_1, t_1) and (s_2, t_2).

2 Preprocessing

We start with a preprocessing phase that reduces the general case to instances that have better guarantees on the connectivity between the terminals. The latter instances will be the focus of Section 3.

2.1 Routing on Expanders

It is instructive to start by thinking what are the sources of difficulty in our problem. We can identify two obstacles: First, since we are not going to route all the pairs (due to NP-hardness), we wish to somehow choose the ones that are easier to connect (in order to route as many as possible). Second, having decided which terminal pairs to focus on, we need to figure out how to route them through the non-terminal vertices. We will start by considering a simple setting that avoids both these hurdles: all the vertices are terminals ($V = T$), and the entire graph is very well connected. To formalize the discussion we should define well connected graphs, which are known as *expanders*. There are several ways to define them, and the variant that we use here is called edge-expansion.

Definition 2.1. *A graph $G(V, E)$ is an α-expander if for every cut $(S, V \setminus S)$,*

$$|E(S, V \setminus S)| \geq \alpha \cdot \min\{|S|, |V \setminus S|\}.$$

Definition 2.1 evaluates cuts by the ratio of crossing edges to the size of the smaller side, rather than by the absolute number of edges (as in the Minimum Cut problem). We will simply call a graph an *expander* if it is an α-expander for a constant $\alpha > 0$, that may be arbitrarily small (so long as it is $\Omega(1)$ and does not diminish as n grows). These graphs, particularly when sparse (say with constant vertex degrees), have an overwhelming variety of applications in Mathematics and Computer Science. They display connectivity properties comparable to those of the complete graph K_n (in a sense that we will not define here), and manage to do it with much fewer edges – for example, 3-regular[3] expanders are known to exist.

As explained above we expect routing through expanders with $V = T$ to be easier than the general case. For illustration, suppose we have a 3-regular expander in which the n nodes are paired into demand pairs $(s_1, t_1), \ldots, (s_{n/2}, t_{n/2})$. Let us show an $O(1)$-approximation: Consider the cut $(S, V \setminus S)$ with all the s_i's on side S and all the t_i's on the other side. By the expansion property we have $\frac{1}{2}\alpha n$ edges crossing the cut. Since the degrees in S are bounded by 3 we have at least $\frac{1}{6}\alpha n$ nodes in S incident on crossing edges. Since the degrees in $V \setminus S$ are also bounded by 3, we can match at least $\frac{1}{18}\alpha n$ vertices in S to unique nodes in $V \setminus S$. We have thus routed $\frac{1}{18}\alpha n$ pairs on disjoint paths of length 1 each, and since we have $|V| = n$ terminals, this yields $O(1)$-approximation.

For less restrictive bounds on the degree, there are known routing algorithms on expanders with $V = T$ that achieve approximation of roughly $\Omega(1/\log n)$ (see [Fri00] for a survey of some results in this vein). The specific result we will use is due to Rao and Zhou [RZ10] and is summarized in the next theorem (some details are omitted for simplicity). Importantly, it has the added property that the routing is not just edge-disjoint but also *node-disjoint*, which will be required for the analysis we present. Observe that it is a stronger property since node-disjoint paths must be edge-disjoint, but not vice-versa.

Theorem 2.2 (informal)**.** *If G is an expander with its nodes paired into $n/2$ demand pairs, then we can efficiently find a node-disjoint routing of $\Omega(n/\log n)$ demand pairs.*

[3]Recall that a graph is d-regular if the degree of each vertex is d.

The algorithm achieving this is the natural greedy one: Find the the shortest path connecting any demand pair, remove the path from the graph (all edges and nodes), and iterate. The resulting routing is then clearly node-disjoint. We leave he analysis of the approximation guarantee out of scope for this report, in the interest of focusing on the general case.

2.2 Well-Linkedness

Let us now remove the simplifying assumption $V = T$. This reinstates the first hurdle mentioned above of having to choose the best demand pairs to route. Some pairs may be better connected than others, and intuitively we wish to focus on them, and declare the poorly connected pairs "lost causes". To formalize this we need a notion of good *terminal* connectivity, as opposed to the overall connectivity that was formalized by expanders. We use the following.

Definition 2.3. *In a graph $G(V, E)$ with terminals T, a subset $W \subset V$ is α-well-linked if for every $S \subset W$,*

$$|E(S, W \setminus S)| \geq \alpha \cdot \min\{|S \cap T|, |(W \setminus S) \cap T|\}.$$

Compare this definition to Definition 2.1. It as a generalization in two senses: it applies to subsets $W \subset V$, and more importantly, it evaluate cuts by the *number of terminals* in the smaller side, rather than by the total size of the smaller side. Put differently, if $V = T$, then V being α-well-linked is equivalent to the graph being an α-expander.

We will say W is *well-linked* (with respect to a given set of terminals) if it is α-well-linked for some $\alpha \geq 1/poly \log k$. We will also slightly abuse terminology and say that the terminals T are well-linked in W. The following lemma demonstrates how well-linkedness can be exploited for low-congestion routing. We will use it repeatedly in the sequel.

Lemma 2.4. *Let W be an α-well-linked subset of nodes, and let A, B be subsets of terminals in W with equal sizes $|A| = |B| = q$. There is a 1-1 map $\sigma : A \to B$ such that each $v \in A$ is connected with a path to $\sigma(v)$, and those paths create congestion at most $1/\alpha$ with each other (meaning each edge is used by at most $1/\alpha$ paths), and they can be found efficiently.*

Proof. We prove the lemma by solving standard Maximum Flow. Put capacity $1/\alpha$ on each edge. By the α-well-linkedness of the terminals in W, any cut in W with one side containing A and the other containing B is crossed by at least αq edges, and hence its capacitated value is at least q. Now add a new source-sink pair s^*, t^* to W, connecting s^* to each node in A and connecting t^* to each node in B, with unit capacity edges. Noting that $|A| = |B| = q$, one can easily verify that the capacitated value of the minimum s^*t^*-cut is at least q (details in footnote).[4] By the Min-Cut-Max-Flow theorem there is a flow with value q that we can find efficiently by solving for the maximum flow, and recall we can assume w.l.o.g. it is intergal. The integrality means it decomposes into q paths; the $1/\alpha$ capacity constraints imply that each edge participates in at most $1/\alpha$ of the paths; and the unit capacity constraints on the edges connecting s^* to A and t^* to B imply that each node in A and in B participates in exactly one path. The paths induce the desired map σ: it maps each $v \in A$ to the unique node in B that is on the same path as v. □

Lemma 2.4 is not directly applicable to our setting for two reasons: first it does not let us choose the pairing σ (recall we have specified demand pairs in the input), and second, more importantly,

[4]Consider an s^*t^*-cut in which side s^* contains only ϵq of the nodes in A, for some $0 \leq \epsilon \leq 1$. By α-well-linkedness and the $1/\alpha$ edge capacities, side s^* sends ϵq capacity crossing the cut. Also s^* sends $(1 - \epsilon)q$ unit-capacity edges crossing the cut, and together the cut value is at least q. A symmetric argument applies to cuts that contain only a subset of B. All other s^*t^*-cuts are covered by the argument given before adding s^* and t^* to the graph.

it requires α-well-linkedness with constant α in order to guarantee the constant congestion we are after, while the terminals in our input graph may not be well-linked at all. Our next wish is to focus on well-linked subsets of them, and we realize it with a divide-and-conquer approach: partition the vertices to subsets $V_1, V_2, \ldots$ such that the terminals T_i in each subset V_i are well-linked, and route the demand pairs in each V_i separately. The demand pairs lost in the partition, i.e. those with endpoints in different $V_i's$, are the "lost causes" that we will not even try to connect. This procedure is called *well-linked decomposition*, and will be performed using the following theorem due to Chekuri, Khanna and Shepherd [CKS04, CKS05].

Theorem 2.5. *There is an efficient algorithm that given an instance of EDPwC, outputs a partition of the nodes to subsets $V_1, V_2, \ldots$*[5] *such that the terminals contained in each V_i are well-linked in V_i, and the total number of demand pairs contained in the V_i's (meaning not lost in the partition) is $\Omega(k/\log^2 k)$.*

Recall that when we say *well-linked* in the theorem statement we mean α-well-linked with $\alpha \geq 1/poly \log k$ (as remarked after Definition 2.3); the theorem is not strong enough to guarantee constant α. Since we aim for a $poly \log k$-approximation, the theorem's guarantee that a $1/\log^2 k$-fraction of the demand pairs survive the decomposition is acceptable for us. Theorem 2.5 is the main result of [CKS05] and its proof is quite long, so again, we will just sketch the algorithm and leave the analysis out of scope. For each demand pair s_i-t_i denote by P_i the set of paths connecting the pair, and $P = \cup_i P_i$. We set up the following linear program with a variable x_p for each path p in the graph, and a variable y_i for each demand pair (s_i, t_i). Recall that c is our target constant congestion.

$$\begin{aligned}
\max \quad & \sum_{i=1}^{k} y_i & & \qquad \text{(LP1)}\\
s.t. \quad & \sum_{p\in P: e\in p} x_p \leq c & & \forall e \in E\\
& \sum_{p\in P_i} x_p \geq y_i & & \forall i = 1,\ldots,k\\
& 0 \leq x_p & & \forall p \in P_i,\ i = 1,\ldots,k\\
& 0 \leq y_i \leq 1 & & \forall i = 1,\ldots,k
\end{aligned}$$

Observe that if we take this as an integer program then it is an exact formulation of our EDPwC problem: y_i is an indicator for whether pair demand i is routed; x_p is an indicator of whether path p is used in the routing (there is no gain in setting it to more than 1); and the first set of constraints are congestion constraints for each edge. The relaxation LP1 formulates a problem known as *multi-commodity flow*. We interpret it as having k types of flow, where one unit of each type i wants to be shipped from s_i to t_i, and our goal is to ship as many types as possible under shared edge capacities c.[6] This is indeed a very natural relaxation of EDPwC. Much like the usual (single-commodity) flow problem, LP1 can be reformulated in polynomial size by associating variables to edges instead of to paths, and hence it can be solved efficiently.

[5] *Partition* means that the subsets are pairwise disjoint and their union equals V.

[6] Recalling that the multi-source multi-sink case of the usual Maximum Flow problem is reducible to the single-source single-sink case (as we have done for example in the proof of Lemma 2.4), one may ask why is multi-commodity different. The answer is that we do not gain anything from shipping more than a single unit of any given type.

The decomposition algorithm of Theorem 2.5 starts by solving LP1. The resulting total flow on each edge interpreted as a measure of how globally important is the edge for our routing task. It then proceeds to recursively solve the same LP, but with a different objective function: $\max \min_{i=1,\dots,k} y_i$.[7] In words, we now aim to maximize the minimum flow of any type (whereas in multi-commodity flow we could still do well even if some demand pair was shipping no flow at all). This new problem is called *maximum concurrent flow*. Since it requires all the demand pairs to be connected to some extent, we interpret it as a measure of terminal well-linkedness.

If the concurrent flow is small, the algorithm partitions the graph along a cut carrying just a small amount of flow and recurses on the two sides. The recursion terminates on a subgraph either when the multi-commodity flow (which was computed only once in the beginning) is small, meaning the subgraph is not too important globally, or when the concurrent flow (computed on the current subgraph) is large, meaning the terminals in the subgraph are well-linked. It now remains to route demand pairs inside the resulting well-linked partition subsets, which is our task for the next section.

3 Routing in Well-Linked Instances

By the preprocessing of the previous section, we assume from now on that the terminals T are well-linked in our input graph G. As discussed above, well-linkedness roughly means that G is an expander with respect to the terminals, but not necessarily in the full sense of Definition 2.1. We therefore wish to explicitly identify in G an expander-like structure over the terminals, that we can exploit in the fashion outlined in Section 2.1. We formalize this as *embedding* the expander in G.

Definition 3.1. *Let $H(T, F)$ be a graph on the terminal nodes T with an arbitrary edge set F. Denote by P the set of all simple*[8] *paths in the graph G. An* embedding *of H in G is a map $\eta : F \to P$ that maps each edge t-$t' \in F$ to a path in G with endpoints t and t'.*

Inspect Figure 1 for illustration. The graph H is embedded in G_1: for example, the edge a-c in H can be mapped to a path in G_1 from a to c going through the two leftmost black nodes. It can also be mapped to the longer path in G_1 that passes through b. The graph H is embedded in G_2 too: each edge in H can be mapped to a path in G_2 that passes through the black node. In this embedding, the paths share the same edges, and more precisely each edge is shared by 2 paths.

Using the notation of Definition 3.1, we define the *η-congestion* of an edge e in G as

$$c_\eta(e) = |\{f \in F : e \in \eta(f)\}|,$$

the number of the paths going through e that participate in the embedding η. The *congestion* of the embedding η is $\max_{e\in E} c_\eta(e)$, the maximum congestion over the edges. In Figure 1, H embeds into G_1 with congestion 1, and embeds into G_2 with congestion 2 but not 1.

Our plan is now to find an expander H embedded in G, route the demand pairs in H using Theorem 2.2, and carry over the routing to G via the embedding. Since the routing on H will be edge-disjoint (and even node-disjoint, which we recall is an added property of Theorem 2.2), the congestion of the routing in G will be precisely the congestion of the embedding. Hence we are looking for an expander embedding with constant congestion. To find one, we appeal to the following attractive result.

[7]This can be formulated in LP by introducing a new variable y for the minimum.

[8]Recall that a path is *simple* if it never visits the same node twice.

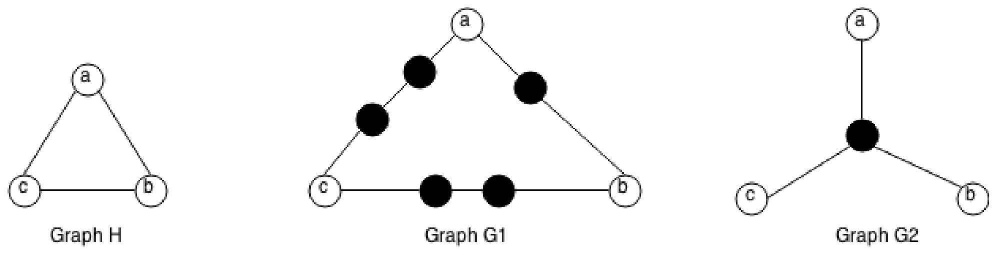

Figure 1: Graph embedding illustration. White nodes are terminals, black are non-terminals.

3.1 Expander Embedding by the Cut-Matching Game

Khandekar, Rao and Vazirani [KRV09] described the following game of two players. They start with a graph on k nodes and no edges, and gradually construct an expander. In each round, the *cut player* chooses a cut (A, B) with equally sized sides $|A| = |B| = \frac{1}{2}k$, and the *matching player* answers with a perfect matching M of the two sides of the cut. The edges M are then added to the graph. The game terminates when the graph becomes an expander. The goal of the cut player is to minimize the number of rounds and the goal of the matching player is to maximize it. [KRV09] proved the following theorem:

Theorem 3.2. *The cut player has an efficient strategy that ends the game after $O(\log^2 k)$, regardless of the choices of the matching player.*

By *strategy* we mean a sequence of moves that the cut player can play, possibly depending on previous moves of the matching player. By *efficient strategy* we mean that the cut player can compute the next move of the winning strategy in polynomial time. We remark that as one could expect, the winning strategy of the cut player is to find a cut violating Definition 2.1 and completing the small side arbitrarily to make the cut equally sized. The cut is found by solving standard Maximum Flow.

We now demonstrate how to embed an expander on the terminals into G and use it to solve EDPwC. Set up a cut-matching game on the vertex set T and simulate the efficient cut player from Theorem 3.2. It chooses a partition (A, B) of the terminals with equal sizes $|A| = |B| = \frac{1}{2}k$. We apply Lemma 2.4 to produce a perfect matching $\sigma : A \to B$ realized by paths in G that create polylogarithmic congestion – say $\log^b k$ – and return σ as the answer of the matching player. When the game terminates it outputs an expander H on the vertex set T, on which we can solve EDP with Theorem 2.2. Since by construction each edge in H corresponds to a path in G, we have an embedding of H in G, and the routing on H translates to a routing on G. The congestion of this routing is the congestion of the embedding, which is $\log^{b+2} k$ because the paths used in each round created congestion $\log^b k$ with each other, and we played for $\log^2 k$ rounds.

This is insufficient since we are out for constant congestion. We will need to remedy both of the failures: route each rounds with only constant congestion, and prevent congestion from accumulating across the rounds.

3.2 Routers

Our improved expander embedding will rely on identifying subsets of non-terminal nodes in the graph that have good internal connectivity and also good connectivity to the terminals. This should be seen as our way of coping with the second hurdle mentioned in the beginning, of how to route well-linked terminal pairs through the non-terminal nodes – and the answer is through these special subsets. We call them *routers* and define them below. We use the following terminology: for a subset $S \subset V$, the *interface nodes* are the nodes in S that have neighbors outside S, and we denote them by $int(S)$.[9]

Definition 3.3. *A subset of nodes $S \subset V$ is a* router *if (i) S contains no terminals, (ii) the interface nodes of S are $\frac{1}{2}$-well-linked in it, (iii) there are $k' = k/poly \log k$ edge-disjoint paths connecting interface nodes of S to terminals in T, and these paths have disjoint endpoints. (This means each path has an interface node of its own on one end and a terminal on its own on the other end.)*

(We remark that the actual definition of routers in [Chu12] is weaker but qualitatively similar, and we use this version for clarity.) Note that in this definition we lose yet another $poly \log k$ factor in the number of terminals, settling for k'. This is required because asking for part (iii) to hold with k it too strong – we will not be able to find such good routers in G. In Section 3.4 we will deal with finding the routers and see where the loss is incurred. For the present section and the next one we assume we have r pairwise-disjoint routers $S_1, \ldots, S_r$, where $r = \Theta(\log^2 k)$ is the maximum number of rounds in the cut-matching game, and focus on using them to obtain a constant-congestion expander embedding.

Each router is well connected to a subset of terminals, and the terminals are well connected within themselves (namely they are well-linked). Therefore by transitivity the routers should be well connected to each other, and this is formalized in the next lemma.

Lemma 3.4. *For every pair S_i, S_j of routers, there are $k'/polylogk$ edge-disjoint and endpoint-disjoint paths connecting interface nodes of S_i to interface nodes of S_j.*

Proof. By definition of routers we have a collection P_i of edge-disjoint and endpoint-disjoint paths connecting nodes from $int(S_i)$ to a subset T_i of terminals. Similarly we have paths P_j and terminals T_j, and $|T_i| = |T_j| = k'$. Since our goal in the current proof is to connect S_i to S_j, we benefit from intersection between T_i and T_j and therefore we consider now the worst case $T_i \cap T_j = \emptyset$. Recalling that the terminals are α-well-linked with $\alpha \geq 1/poly \log k$, we can apply Lemma 2.4 on T_i, T_j and obtain k' paths connecting them while creating congestion $poly \log k$ with each other. We interpret these paths as a flow from T_i to T_j with value k' under uniform capacities of $poly \log k$. Here we mean, similarly to the proof of Lemma 2.4, standard s^*t^*-flow with a super-source s^* connected to all the nodes in T_i and super-sink t^* connected to all the nodes in T_j. Rescaling all capacities to unit, we can ship $k'/poly \log k$ flow from T_i to T_j and assume w.l.o.g. it is integral, yielding $k'/poly \log k$ edge-disjoint and endpoint-disjoint paths running between T_i and T_j (but no longer covering all the terminals in T_i and T_j). Denote this collection of paths by Q_{ij}.

Take any path $q_{ij} \in Q_{ij}$, with endpoints $t_i \in T_i$ and $t_j \in T_j$. Recall that t_i (by definition of T_i) is the endpoint of a unique path $p_i \in P_i$, which has a unique other endpoint in $int(S_i)$. Similarly, we have a unique $p_j \in P_j$ with one endpoint in t_j and the other a unique endpoint in $int(S_j)$. Seaming the paths p_i, q_{ij}, p_j together gives a path from $int(S_i)$ to $int(S_j)$. Performing this on each path in Q_{ij} yields $|Q_{ij}| = k'/poly \log k$ edge disjoint and endpoint disjoint paths between $int(S_i)$ and $int(S_j)$, as needed. □

We can now begin constructing the embedding. Let T_1 denote the subset of k' terminals that are connected to the interface of the router S_1 with disjoint paths, from part (iii) in Definition 3.3. Label the terminals in T_1 as $t_1, t_2, \ldots, t_{k'}$. With each $t_i \in T_1$ we will associate a tree τ_i which is

[9]In other contexts they are frequently called *boundary nodes*.

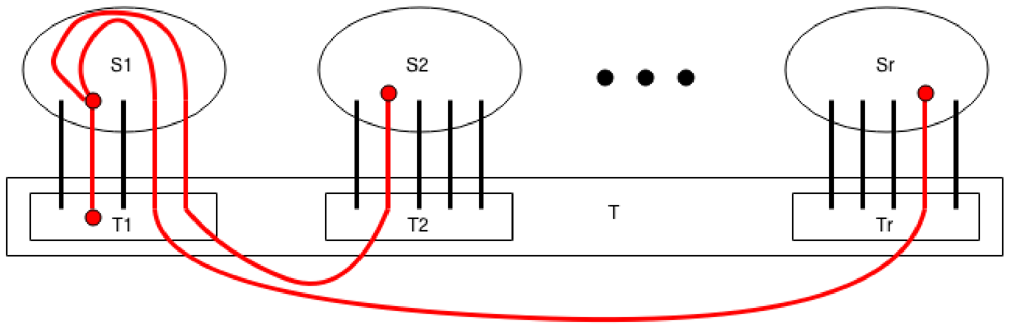

Figure 2: Each router S_j has edge-disjoint endpoint-disjoint paths from some of its interface nodes to a subset T_j of terminals. In red, an illustration of a tree τ_i associated with a terminal $t_i \in T_1$. The tree is rooted in t_i and has a leaf in each router interface. The path from t_i to $int(S_1)$ is used in many root-to-leaf paths, but within the same tree τ_i. We construct such tree for each terminal in T_1 and require that each edge is present in only a constant number of different trees, and that the leaves of the trees are disjoint.

rooted in t_i and has exactly r leaves, one in each router interface. Meaning, τ_i would have one leaf in $int(S_1)$, one in $int(S_2)$, and so on until $int(S_r)$. Each edge in G will participate in only a constant number of trees $\{\tau_i\}$, which we will leverage to achieve constant embedding congestion, and the leaves of the trees will be disjoint.

This raises the question of how to connect the root t_i with a leaf in, say, $int(S_2)$. We know how to route t_i to $int(S_1)$ (since $t_i \in T_1$ and by definition of T_1), but Definition 3.3 might route $int(S_2)$ to a subset T_2 of terminals which is completely disjoint of T_1, yielding no apparent path from t_i to $int(S_2)$. (We know that such a path exists since we naturally assume G is connected, but we need a concrete path to reason about.) We resolve this using Lemma 3.4, which tells us that we can route $k'/poly \log k$ edge-disjoint paths from $int(S_1)$ to $int(S_2)$. Since we can route t_i to $int(S_1)$, and $int(S_1)$ is $\frac{1}{2}$-well-linked in S_1, we get a path from t_1 to $int(S_2)$. We can repeat this argument for every tree τ_i and router S_j, and see that we can indeed build each tree such that it has a leaf in each router interface. See Figure 2 for clarification. (We emphasize that S_2 was taken as an example, and we apply the same argument to every S_j, but the initial choice of T_1 and S_1 is fixed. That is, we associate trees only with the terminals in T_1. It happens that we need to focus on an arbitrary subset of k' terminals – due to the loss in Definition 3.3 – and T_1 is a choice as good as any.)

However, we also need to trees the have small overlap, and more precisely to have each edge of G participate in only a constant number of trees. The analysis achieving this requires lengthy details, so again we suppress it and just present an intuition of why this should be possible. We go by double-counting. First, we count the number of paths required for building all the trees: we build k' trees, and each one has r root-to-leaf paths (one leaf in each router), so together $k'r$ paths make up all the trees. Second, we count the number of paths available to us for the construction: we have two types of paths,

Type I: Paths from router interfaces to terminals, which are given by part (iii) of Definition 3.3. We have r routers and k' such paths per router, so together $k'r$ paths.

Type II: Paths between terminals, which are given by Lemma 2.4 (when invoked in the proof of Lemma 3.4). We have $k'/poly \log k$ such paths between each pair T_i, T_j, so altogether $k'r^2/poly \log k$ paths.

Note that the proof of Lemma 3.4 constructs paths between router interfaces by concatenating Type I paths (denoted P_i, P_j in the proof) and Type II paths (denoted Q_{ij} in the proof). From the double counting, we see that we roughly $k'r$ Type I paths and $k'r^2/poly\log k$ Type II paths available for the construction, and only $k'r$ paths are needed. Hence the average number of time a Type I path is required is 1, and the average number of times a Type II path is required is even smaller, $1/poly\log k$. This coarse description overlooks many non-trivial parts of the argument, and hides several constants, but the key takeaway is that it seems we have sufficiently many paths to construct the trees that we need, and we have edge-disjointness guarantees on large subsets of them (by Definition 3.3 and Lemma 3.4), so there is room for hope we can devise a construction in which every edge is used in only a constant number of trees. And here we again choose to leave further details out of scope.

3.3 Playing the Game

With the trees $\tau_1, \ldots, \tau_{k'}$ at our disposal we are ready to play against the cut player from Theorem 3.2 (who is ourselves) and construct the expander embedding. Each τ_i has a leaf in each router interface, and we think of that leaf as a *representative* of t_i (the terminal associated to τ_i). So each terminal in T_1 has a representative in each router interface, and each node in the router interface represents at most one terminal (since the trees have disjoint leaves).

The embedded expander will be on the vertex set T_1 (and not on all the terminals T). In each round of the game we will use a different router – by definition of r we have sufficiently many routers for that purpose. In round j, the cut player queries a cut $(X, T_1 \setminus X)$ of T_1 with equal side sizes $|X| = |T_1 \setminus X| = \frac{1}{2}k'$. Let A be the representatives of X in $int(S_j)$, and let B be the representatives of $T_1 \setminus X$ in S_j. We thus have $|A| = |B| = \frac{1}{2}k'$. We apply Lemma 2.4 to produce a perfect matching of A an B and answer with it to the cut player.

After r rounds we get an expander H on the nodes T_1, and we route the demand pairs in it by applying Theorem 2.2. Denote this routing on H by f_H. We now show how f_H embeds into G with constant congestion. Let t_a-t_b be an edge used by f_H, with endpoints $t_a, t_b \in T_1$. This edge was added to H because in some round j of the game, Lemma 2.4 matched their respective representatives v_a, v_b in $int(S_j)$. We thus map the edge t_a-t_b in H to the path between t_a and t_b in G which is the concatenation of the following three paths:

- the root-to-leaf path from t_a to v_a in the tree τ_a;
- the path from v_a to v_b inside S_j which was found by Lemma 2.4;
- the root-to-leaf path from t_b to v_b in the tree τ_b.

Let us analyze the congestion of this embedding. For brevity, we will refer to the root-to-leaf paths (the first and third portions of the concatenation) as Type III paths, and to router internal paths (the second portion of the concatenation) as Type IV paths. In each round of the game we use a different router, and Lemma 2.4 guarantees that the paths within each router only create congestion 2 with eachother (recall that the interface nodes are $\frac{1}{2}$-well-linked in their router). Altogether, the Type IV paths create congestion of only 2 with each other. The Type III paths may appear more problematic, since they are not contained in routers and supposedly they can be used in several rounds. However, here we exploit the crucial fact that Theorem 2.2 produces a *node-disjoint* routing in H, and not just edge-disjoint. This means that in f_H each terminal $t_i \in T$ has at most two incident edges, so the embedding of f_H into G uses at most two root-to-leaf paths from the tree τ_i. To make this clearer, revisit Figure 2: the path from the root of the red tree into $int(S_1)$ appears in many root-to-leaf paths, but f_H uses at most two of them. In other words, each tree creates only congestion 2 with itself.

Recalling that each edge in G belongs to only a constant number of trees, and seeing that we use each tree at most twice to embed f_H into G, we conclude that the Type III paths create only constant congestion with eachother. Finally, the Type III paths may go through routers and thus intersect Type IV paths, but this can only add 2 to the congestion on each edge. Rephrasing, we have established that each edge has only a constant Type-III-congestion and only 2 Type-IV-congestion, so the total congestion on the edge is at most their sum, a constant. This concludes the congestion analysis of embedding f_H into G, and it routes $k' = k/poly \log k$ terminal pairs, which is our desired approximation factor.

One point that may appear as a problem is that by routing only pairs inside T_1 we inherently assumed it contains demand pairs. What if all the terminals T_1 are sources? Recall, however, that the choice of T_1 was arbitrary. We could instead choose a subset k' terminals paired into demand pairs; details omitted.

3.4 Finding a Family of Routers

To complete the presentation of the algorithm, it remains to show how to find routers $S_1, \dots, S_r$. To convey the main ideas we will only sketch how to find a single router. We will need to notion of *edge contraction.*

Definition 3.5. *Given graph G on n nodes with an edge e, the contraction of e is the operation of merging its two endpoint into a single node, thus obtaining a modified graph G' on $n-1$ nodes.*

Note that by iterative contractions we can contract many nodes, as long as they are connected, into a single node. Intuitively this allows us to simplify the graph by suppressing portions of it that are irrelevant to our current task.

In order to find a router in our input graph G we first remove all the terminals T (with their incident edges), since a router is not allowed to contain any of them. The term "well-linked" will refer throughout this section to well-linkedness with respect to the interface nodes. For a subset of nodes $S \subset V$, we call the edges crossing from S to $V \setminus S$ the *boundary edges* of S.

We describe an iterative process that stores in memory a graph H which is the result of contraction operations on G. We are only allowed to contract a subset S if it is well-linked, and has only a few boundary edges, namely at most $k/poly \log k$.

Initially we perform the well-linked decomposition from Theorem 2.5 (or more precisely, a suitable variant) on G,[10] and contract each partition subset into a node to obtain H. Then, in each iteration, the number of edges in H strictly decreases. We do this as follows.

Let H_i be H in iteration i, and suppose it has m edges at that point. Choose a uniformly random partition of H to equally sized subsets A, B. Each edge has probability $\frac{1}{2}$ to cross the partition, and hence with some constant probability, we get roughly (up to a constant) the same number of edges contained in A as the number of edges on the boundary of A. For simplicity say we have $\frac{1}{2}m$ edges inside A and $\frac{1}{2}m$ on its boundary, crossing to B. Recalling that each node on the interface of A is incident to a boundary edge, we have $|int(A)| \le \frac{1}{2}m$.

Un-contract all nodes in A, recovering the original nodes of G (only on side A). Apply on the un-contracted A the well-linked decomposition from Theorem 2.5. Recall that currently we are dealing with well-linkedness of the interface nodes, so the "terminals" in the well-linked decomposition are the interface nodes $int(A)$. Theorem 2.5 produces a partition of A into well-linked subsets

[10]This is where we lose the $poly \log k$ factor in part (iii) of Definition 3.3 – the loss is by Theorem 2.5.

$A_1, A_2, \ldots$, and guarantees that the total number of edges crossing between them is at most $\frac{1}{2}m - m/poly\log m$ (recall they correspond to "lost" demand pairs). We now have two cases:

- Case 1: Some A_j has many boundary edges, more than $k/poly\log k$. In this case we found a subset A_j which is well-linked (by the decomposition), contains no terminals (since we priorly removed them from G), and has many outgoing edges.
- Case 2: All A_j's have few boundary edges, at most $k/poly\log k$. In this case we are allowed to contract them, thus obtaining a new subset A' and letting $H_{i+1} = A' \cup B$. We have shown that A' has strictly less ($m/poly\log m$ less) internal edges than A, and moreover A and A' have exactly the same set of boundary edges, since contraction and un-contraction operations do not effect the boundary. This implies that A' has less edges than A, and hence H_{i+1} has less edges than H_i, as desired for iteration i.

Since the number of edges in H cannot decrease infinitely, Case 1 must occur after a finite number of steps (polynomial in the size of G). We then have a subset A_j which we argue is a router. It has some gaps from the requirements of Definition 3.3, which we now loosely describe how overcome.

(i) A_j needs to contain no terminals, which holds, since we have removed all of them from G before running the algorithm to produce A_j.

(ii) A_j needs to be α-well-linked with $\alpha = \frac{1}{2}$. The well-linked decomposition of Theorem 2.5 only guarantees $\alpha \geq 1/poly\log k$. This is the gap mentioned immediately after Definition 3.3: In [Chu12], routers are in fact defined with $\alpha \geq 1/poly\log k$, but the well-linkedness parameter can then be "boosted" to $\frac{1}{2}$ by routing more paths between interface nodes through the terminals T (in a way very similar to Lemma 3.4, only from a router to itself instead of between two routers). We omit details.

(iii) A_j needs to have good connectivity to the terminals (namely, k' edge-disjoint and endpoint-disjoint paths from $int(A_j)$ to terminals). The above algorithm guarantees that A_j has many boundary edges. The bridging argument can be sketched as follows: We can assume w.l.o.g. that the edge set of G is inclusion-minimal for which the terminals T are well-linked (otherwise remove edges from G until this holds). If A_j has many boundary edges but poor connectivity to T, then intuitively not all of its boundary edges can be used to connect terminals in T, and hence one of them can be removed without harming the well-linkedness of T in G – a contradiction to minimality. This argument can be formalized to show that any subset with many boundary edges has good connectivity to T.

This completes the description of how to find one router. Finding a family of r routers is done using similar considerations, by partitioning H to r parts instead of 2 parts.

4 Conclusion

To conclude, we review how the pieces of the algorithm and its analysis (which may appear fragmented in our presentation) fit together.

The algorithm:

1. On the input graph, perform the well-linked decomposition from Theorem 2.5. From now work on each well-linked subset of the partition as a separate instance.

2. In a well-linked instance, find a family of routers using Section 3.4.

3. Use the routers so find an embedded expander H in G, by simulating both the cut-player and the matching-player, as described in Sections 3.2 and 3.3.

4. Find a routing on H using Theorem 2.2, and carry it over to G with constant congestion, as analyzed in Section 3.3.

5. Return the union of the routings found on the portions of the well-linked decomposition from Step 1.

Approximation guarantee: We start with k demand pairs, and need to make sure we route no less than $\Omega(k/poly \log k)$ demand pairs. We review the places where terminal loss was incurred:

- The initial well-linked decomposition from Step 1 already decreases the number of terminals we are trying to route to $O(k/poly \log k)$.

- In order to find routers, we tolerate another $poly \log k$ loss (where we introduce k' in their definition).

- Theorem 2.2, which is invoked in Step 5 to find a routing on the embedded expander H, loses yet another $\log k$ factor in the number of terminals.

Altogether we do remain with $\Omega(k/poly \log k)$ demand pairs routed.

Congestion: The routing on H by Theorem 2.2 is without congestion. Embedding it into G entails only constant congestion, as shown in Section 3.3.

Concluding remarks and subsequent work. The algorithm surveyed here is the main result of [Chu12], building on previous work in [CKS05] and [RZ10]. The analysis in [Chu12] shows it achieves congestion $c = 14$. Subsequently Chuzhoy and Li [CL12] sharpened the same techniques and attained congestion $c = 2$, which is the best possible (recalling that the $c = 1$ case is NP-complete). As another consequence, Chekuri and Ene [CE13] could obtain a polylogarithmic approximation with constant congestion for the technically harder variant node-disjoint paths (NDP).

Notice that the starting point of the algorithm is Theorem 2.5, which starts by solving LP1. Therefore the entire algorithm is in fact a (very involved) rounding procedure for LP1. The *integrality gap* of this LP is known to be $\Omega(poly \log n)$, which means that no rounding procedure for this LP can achieve a better approximation factor. Hence this algorithm is optimal with respect to LP1 up to the degree of the $poly \log k$ factor (and has the added value that the approximation is w.r.t. k and not n).

Remarkably, the techniques presented here were found applicable to Graph-Minor Theory, a framework known to yield deep graph-theoretic structural results with yet more algorithmic applications. Most notably, Chuzhoy and Chekuri [CC14] significantly improved the famous Grid-Minor Theorem of Robertson and Seymour [RS86], proving the first polynomial lower-bound. The proof relies on the tight relation between terminal well-linkedness and the notion of treewidth due to Reed [Ree97], and on the fact that many parts of the algorithm presented here are remeniscent of graph minors. Loosely speaking, the Grid-Minor Theorem requires finding a specific and much more constrained type of graph embedding, but allows a much greater loss in the number of terminals (the proof in [CC14] ends with about $k^{1/98}$; compare this to the $k/poly \log k$ achieved here). This result was intended to be surveyed here but this part is deprecated due to space limitations.

References

[CC14] Chandra Chekuri and Julia Chuzhoy, *Polynomial bounds for the grid-minor theorem*, Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014, 2014, pp. 60–69.

[CE13] Chandra Chekuri and Alina Ene, *Poly-logarithmic approximation for maximum node disjoint paths with constant congestion*, Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013, 2013, pp. 326–341.

[Chu12] Julia Chuzhoy, *Routing in undirected graphs with constant congestion*, Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012, 2012, pp. 855–874.

[CKS04] Chandra Chekuri, Sanjeev Khanna, and F. Bruce Shepherd, *The all-or-nothing multicommodity flow problem*, Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004, 2004, pp. 156–165.

[CKS05] ———, *Multicommodity flow, well-linked terminals, and routing problems*, Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005, 2005, pp. 183–192.

[CL12] Julia Chuzhoy and Shi Li, *A polylogarithmic approximation algorithm for edge-disjoint paths with congestion 2*, 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012, 2012, pp. 233–242.

[Fri00] Alan M. Frieze, *Edge-disjoint paths in expander graphs*, SIAM J. Comput. **30** (2000), no. 6, 1790–1801.

[KRV09] Rohit Khandekar, Satish Rao, and Umesh V. Vazirani, *Graph partitioning using single commodity flows*, J. ACM **56** (2009), no. 4.

[Ree97] Bruce A. Reed, *Tree width and tangles: A new connectivity measure and some applications*, Surveys in Combinatorics (1997), 87–162.

[RS86] Neil Robertson and P D Seymour, *Graph minors. v. excluding a planar graph*, J. Comb. Theory Ser. B **41** (1986), no. 1, 92–114.

[RT87] Prabhakar Raghavan and Clark D. Thompson, *Randomized rounding: a technique for provably good algorithms and algorithmic proofs*, Combinatorica **7** (1987), no. 4, 365–374.

[RZ10] Satish Rao and Shuheng Zhou, *Edge disjoint paths in moderately connected graphs*, SIAM J. Comput. **39** (2010), no. 5, 1856–1887.