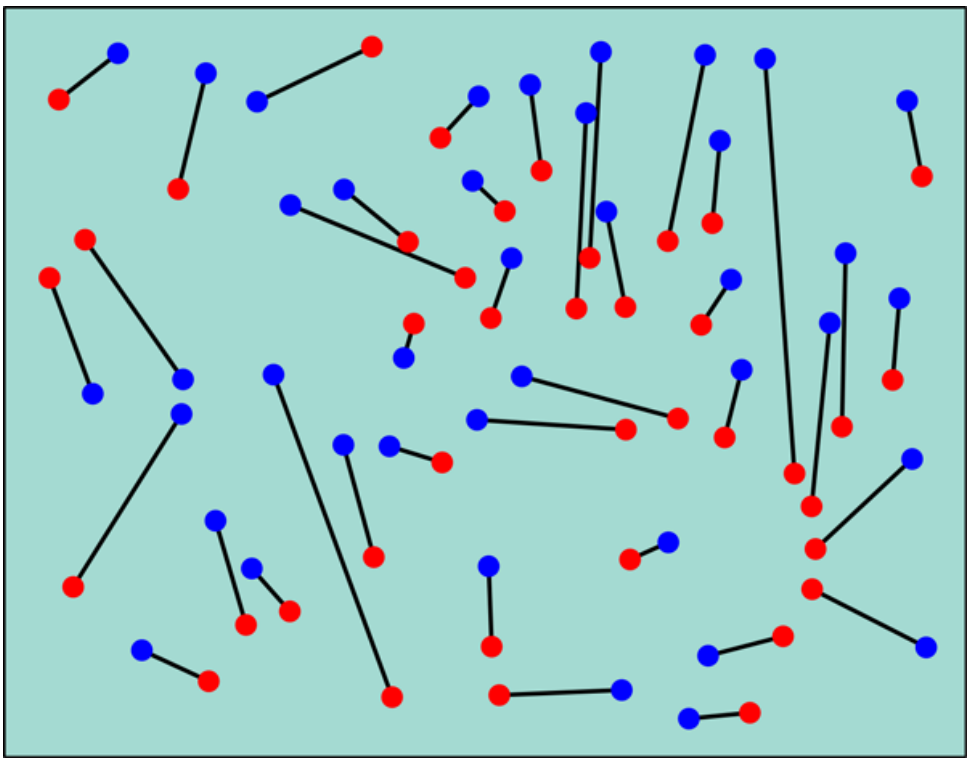
CS598
Topics in Graph Algorithms

Pingbang Hu

November 28, 2024

Abstract

This is an advanced graduate-level graph algorithm course taught by Chandra Chekuri at University of Illinois Urbana-Champaign. This exploratory seminar-style course will cover selection of topics in graph algorithms with an emphasis on recent developments on fast algorithms for a variety of problems such as shortest paths, flows, cuts, and matchings. Structural results and connections to past ideas and results will also be discussed. More information can be found on the course website.



This course is taken in Fall 2024, and the date on the cover page is the last updated time.

Contents

Chapter 1

Introduction

Lecture 1: Overview

Throughout the course, we consider a graph $G = (V, E)$ such that $n \coloneqq |V|$ and $m \coloneqq |E|$. 27 Aug. 11:00

1.1 Minimum Spanning Tree

Finding the minimum cost spanning tree (MST) in a connected graph is a basic algorithmic problem that has been long-studied. We introduce the problem formally.

Definition 1.1.1 (Spanning tree)**.** A *spanning tree* T of a connected graph $G = (V, E)$ is an induced subgraph of G which spans G, i.e., $V(T) = V$ and $E(T) \subseteq E$.

Then, the problem can be formalized as follows.

Problem 1.1.1 (Minimum spanning tree)**.** Given a connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, find the min-cost spanning tree.

Remark. The edge costs need not be positive, but we can make them positive by adding a large number without affecting correctness.

Standard algorithm that are covered in most undergraduate courses are Kruskal's algorithm, Jarnik-Prim's (JP) algorithm, and (sometimes) Borůvka's algorithm. There are many algorithms for MST and their correctness relies on two simple rules (structural properties), for cut and cycle respectively:

Lemma 1.1.1 (Cut rule)**.** If e is a minimum cost edge in a cut $\delta(S)$ for some $S \subseteq V$, then e is in some MST. In particular, if e is the unique minimum cost edge in the cut, e is in every MST.

Definition 1.1.2 (Light)**.** An edge e is *light* or *safe* if there exists a cut $\delta(S)$ such that e is the cheapest edge crossing the cut. Moreover, e is *light* w.r.t. a set of edges $F \subseteq E$ if e is light in (V, F).

Lemma 1.1.2 (Cycle rule)**.** If e is the highest cost edge in a cycle C, then there exists an MST that does not contain e. In particular, if e is the unique highest cost edge in C, e cannot be in any MST.

Definition 1.1.3 (Heavy)**.** An edge e is *heavy* or *unsafe* if there exists a cycle C such that e is the highest cost edge in C. Moreover, e is *heavy* w.r.t. a set of edges $F \subseteq E$ if e is heavy in (V, F).

Corollary 1.1.1. Suppose the edge costs are unique and G is connected. Then the MST is unique and consists of the set of all light edges.

Remark. Without loss of generality, we can assume that the cost are unique by, e.g., perturbation or consistent tie-breaking rule.

1.1.1 Standard Algorithms

Let's review the basic algorithms, the data structures they use, and the run-times that they yield.

Kruskal's Algorithm

Intuitively speaking, Kruskal's algorithm sorts the edges in increasing cost order and greedily inserts edges in this order while maintaining a maximal forest F at each step. When considering the i^{th} edge e_i, the algorithm needs to decide if $F + e_i$ is a forest or whether adding e creates a cycle.

Algorithm 1.1: Kruskal's Algorithm

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MST $T = (V, F)$

1 Sort the edges such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$ // $O(m \log n)$[a]
2 $F \leftarrow \varnothing$ // Initialize the tree
3 **for** $i = 1, \ldots, m$ **do**
4 **if** $e_i + F$ *has no cycle* **then**
5 $F \leftarrow F + e_i$
6 **return** (V, F)

[a] Since the graph is connected, $O(m \log m) = O(m \log n)$ as $n/2 \le m \le n^2$.

Theorem 1.1.1. Kruskal's algorithm takes $O(m \log n)$.

Proof. Sorting takes $O(m \log n)$ time. The standard solution for line 4 is to use a union-find data structure. Union-find data structure with path compression yields a total run time, after sorting, of $O(m\alpha(m, n))$ where $\alpha(m, n)$ is inverse Ackerman function which is extremely slowly growing. Thus, the bottleneck is sorting, and the run-time is $O(m \log n)$. ■

Jarnik-Prim's Algorithm

Jarnik-Prim's algorithm grows a tree starting at some arbitrary root vertex r while maintaining a tree T rooted at r. In each iteration it adds the cheapest edge leaving T until T becomes spanning. Thus, the Jarnik-Prim's algorithm takes $n - 1$ iterations.

Algorithm 1.2: Jarnik-Prim's Algorithm

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MST $T = (V, F)$

1 $r \leftarrow$`uniform(`V`)` // Sample a root
2 $V' \leftarrow \{r\}$, $F \leftarrow \varnothing$ // Initialize the tree
3 **while** $V' \neq V$ **do**
4 $e \leftarrow \arg\min_{e=(u,v)\in\delta(V'),u\in V'} c(e)$
5 $F \leftarrow F + e$, $V' \leftarrow V' + v$ // Update the tree
6 **return** (V, F)

Theorem 1.1.2. Jarnik-Prim's algorithm takes $O(m + n \log n)$.

Proof. To find the cheapest edge leaving T (line 4), one typically uses a priority queue where we maintain vertices not yet in the tree with a key for v equal to the cost of the cheapest edge from v to the current tree. When a new vertex v is added to T the algorithm scans the edges in $\delta(v)$ to update the keys of neighbors of v. Thus, one sees that there are a total of $O(m)$ `decrease-key` operations, $O(n)$ `extract-min` operations, and initially we set up an empty queue. Standard priority queues

implement `decrease-key` and `extract-min` in $O(\log n)$ time each, so the total time is $O(m \log n)$. However, Fibonacci heaps and related data structures show that one can implement `decrease-key` in amortized $O(1)$ time which reduces the total run time to $O(m + n \log n)$. ■

Remark. The Jarnik-Prim's algorithm runs in linear-time for moderately dense graphs!

Borůvka's Algorithm

Borůvka's algorithm seems to be the first MST algorithm, which has very nice properties and essentially uses no data structures. The algorithm works in phases. We describe it recursively to simplify the description, while refer to Algorithm 1.3 for the real implementation. In the first phase the algorithm finds, for each vertex v the cheapest edge in $\delta(v)$. By the cut rule this edge is in every MST.

Note. An edge $e = uv$ may be the cheapest edge for both u and v.

The algorithm collects all these edges, say F, and adds them to the tree. It then shrinks the connected components induced by F and recurses on the resulting graph $H = (V', E')$. It's easy to see that Borůvka's algorithm can be parallelized, unlike the other two algorithms.

Algorithm 1.3: Borůvka's Algorithm

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MST $T = (V, F)$

```
F = ∅                                                  // Initialize the tree
S ← {S_v = {v}}                                        // Collection of all sets
while |S| > 1 do
    S' ← S                                             // Make a copy
    for S ∈ S do
        e_S = (u, v) ← arg min_{e∈δ(S)} c(e)
        S' ← S' − {S_u, S_v} + S_u ∪ S_v [a]            // Merge (i.e., shrink)
        F ← F + e_S                                    // Update the tree
    S ← S'                                             // Update S
return (V, F)
```

[a] Here, S_u and S_v both refer to $S := S_u \cup S_v$ later in the algorithm.

Theorem 1.1.3. Borůvka's algorithm takes $O(m \log n)$.

Proof. The first phase needs $O(m)$ from a linear scan of the adjacency lists, and also computing H (i.e., shrinking) can be done in $O(m)$ time. The main observation is that $|V'| \leq |V|/2$ since each vertex v is in a connected component of size at least 2 as we add an edge leaving v to F. Thus, the algorithm terminates in $O(\log n)$ phases for a total of $O(m \log n)$ time. ■

1.1.2 Faster Algorithms

A natural question is whether there is a linear-time MST algorithm. A brief history of this line:

- Very early on, Yao, in 1975, obtained an algorithm that ran in $O(m \log \log n)$[Yao75], which leverages the idea developed in 1974 for the linear-time Selection algorithm.
- In 1987, Fredman and Tarjan [FT87] developed the Fibonacci heaps and give an MST algorithm which runs in $O(m \log^* n)$.[1] This was further improved to $O(m \log \log^* n)$ [Gab+86].
- Karger, Klein, and Tarjan [KKT95] obtained a linear time randomized algorithm.
- The fastest known deterministic algorithm runs in $O(m\alpha(m, n))$ [Cha00].

[1] Formally, it runs in $O(m\beta(m, n))$, where $\beta(m, n)$ is the minimum value of i such that $\log^{(i)} n \leq m/n$, where $\log^{(i)} n$ is the logarithmic function iterated i times. Since $m \leq n^2$, $\beta(m, n) \leq \log^* n$.

Note. Pettie and Ramachandran gave an optimal deterministic algorithm in the comparison model without known what its actual running time is [PR02]!

Perhaps an easier question is the following.

Problem 1.1.2 (MST verification). Given a graph G and a tree T, decide T is an MST of G or not.

One can always use an MST algorithm to solve the verification problem, but not necessarily the other way around. Interestingly, there is indeed a linear-time MST verification algorithm based on several non-trivial ideas and data structures and was first developed in the RAM model by Dixon, Rauch, and Tarjan [DRT92] with insights from Komlós [Kom85]. Simplification is done by King [Kin97].

Note (RAM model). The RAM model allows bit-wise operation on $O(\log n)$ bit words in $O(1)$ time.

Theorem 1.1.4 (MST verification). There is a linear-time MST verification algorithm in the RAM model. In fact, the algorithm is based on a more general result that we will need: Given a graph $G = (V, E)$ with edge costs and a spanning tree $T = (V, F)$, there is an $O(m)$-time algorithm that outputs all the F-heavy edge of G.

Proof. The original complicated algorithm has been simplified over the years. See lecture notes of Gupta and Assadi for accessible explanation, also the MST surveys [Eis97; Mar08]. ■

Fredman-Tarjan's Algorithm

Here we briefly describe Fredman and Tarjan's algorithm [FT87; Mar08] via Fibonacci heaps, which is reasonably simple to describe and analyze modulo a few implementation details that we will gloss over for the sake of brevity. First, we develop a simple $O(m \log \log n)$ time algorithm by combining Borůvka's algorithm and Jarnik-Prim's algorithm.

As previously seen. Jarnik-Prim's algorithm takes $O(m + n \log n)$ time via Fibonacci heaps where the bottleneck is when $m = o(n \log n)$. On the other hand, Borůvka's algorithm starts with a graph on n nodes and after i^{th} phases, reduces the number of nodes to $n/2^i$; each phase takes $O(m)$ times.

Intuition. Suppose we run Borůvka's algorithm for k phases and then run Jarnik-Prim's algorithm once the number of nodes is reduced. We can see that the total run time is $O(mk)$ for the k phases of Borůvka's algorithm, and $O(m + n/2^k \log n/2^k)$ for the Jarnik-Prim's algorithm on the reduced graph. Thus, if we choose $k = \log \log n$, we obtain a total run-time of $O(m \log \log n)$.

Tarjan and Fredman obtained a more sophisticated scheme based on Jarnik-Prim's algorithm, where the basic idea is to reduce the number of vertices. The algorithm runs again in phases, and we describe the first phase here.

Intuition (First phase). Start growing the tree. If the heap gets too big, we stop.

Consider an integer parameter t such that $1 < t \leq n$. Pick an arbitrary root r_1 and grow a tree T_1 via Jarnik-Prim's algorithm with a Fibonacci heap. We stop the tree growth when the heap size exceeds t for the first time or if we run out of vertices. All the vertices in the tree are marked as visited. Now pick an arbitrary, unmarked vertex as root $r_2 \in V - T$ and grow a tree T_2, and we stop growing T_2 if it touches T_1, in which case it merges with it, or if the heap size exceeds t or if we run out of vertices. The algorithm proceeds in this fashion by picking new roots and growing them until all nodes are marked.

Note. While growing T_2, the heap may contain previously marked vertices. It is only when the algorithm finds one of the marked vertices as the cheapest neighbor of the current tree that we merge the trees and stop.

It's easy to see that the first phase of Fredman-Tarjan algorithm correctly adds a set of MST edges F. After this, we simply shrink these trees and recurse on the smaller graph.

Algorithm 1.4: Fredman-Tarjan's Algorithm

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MST $T = (V, F)$

```
1  V' ← V, F ← ∅                                              // Initialize the tree
2  while |V| > 1 do
3  |  T ← Grow(G)                                             // First phase
4  |  F ← F ∪ E(T)                                            // Update the tree
5  |  Shrink G w.r.t. T, update V and E                       // Second phase
6  return (V', F)
7
8  Grow(G):
9  |  V' ← ∅, F ← ∅, T ← (V', F)                              // Initialize the forest
10 |  while V' ≠ V do
11 |  |  r ← uniform(V − V')                                  // Pick an unmarked vertex
12 |  |  T' ← ({r}, ∅)                                       // Initialize a tree
13 |  |  while |N(T')| < t or V(T') ∩ V' ≠ ∅ do
14 |  |  |  Run one more step of Jarnik-Prim(r, T')          // Starting at r, maintaining T'
15 |  |  V' ← V' ∪ V(T)                                      // Mark
16 |  |  F ← F ∪ E(T')                                       // Update the forest by merging the tree
17 |  return (V, F)                                          // Return a forest of G
```

Note. This can be seen as a parameterized version of Borůvka's algorithm.

The difficult part is to determine its runtime. We have the following.

Theorem 1.1.5. Fredman-Tarjan's algorithm takes $O(m\beta(m, n))$.

Proof. Firstly, the total time to scan edges and insert vertices into heaps and do `decrease-key` is $O(m)$ since an edge is only visited twice, once from each end point. Since each heap is not allowed to grow to more than size t, the total time for all the `extract-min` operations take $O(n \log t)$. With the fact that the initialization of each data structure is easy as it starts as an empty one, hence, the first phase takes $O(m + n \log t)$. We claim that it also reduces the number of vertices to $2m/t$.

Claim. The number of connected components induced by F is $\leq 2m/t$ after the first phase.

Proof. Let $C_1, \ldots, C_h$ be the connected components of F. If for every C_i, $\sum_{v \in C_i} \deg(v) \geq t$,

$$2m = \sum_{v \in V} \deg(v) = \sum_{i=1}^{h} \sum_{v \in C_i} \deg(v) \geq ht \Rightarrow h \leq \frac{2m}{t}.$$

To see why the assumption holds, consider the growth of a tree T' in line 14:

- If we stop T' because heap size $|N(T')|$ exceeds t, then each of the vertex in the heap is a witness to a unique edge incident to T', hence the property holds.
- If T' merged with a previous tree, the property holds because the previous tree already had the property and adding vertices only increases the total degree of the component.

The only reason the property may not hold is if line 17 terminates a tree because all vertices are already included in it, but then that phase finishes the algorithm. ⊛

The question reduces to choosing t.

Intuition. We want linear time in the first phase, i.e., $n \log t$ to be no more than $O(m)$, leading to $t = 2^{2m/n}$. If we do this in every iteration, then this leads to $O(m)$ time per iteration.

We now bound the number of iteration. Consider $t_1 \coloneqq 2^{2m/n}$ and $t_i \coloneqq 2^{2m/n_i}$,[a] where n_i and m_i are the number of vertices and edges at the beginning of the i^{th} iteration, with $m_1 = m$ and $n_1 = n$. From the previous claim, $n_{i+1} \le 2m_i/t_i$, which gives $t_{i+1} = 2^{2m/n_{i+1}} \ge 2^{\frac{2m}{2m_i/t_i}} \ge 2^{t_i}$. Thus, t_i is a power of twos with $t_1 = 2^{2m/n}$, and the Fredman-Tarjan's algorithm stops if $t_i \ge n$ since it will grow a single tree and finish. Thus, the algorithm needs at most $\beta(m,n)$ iterations, giving the total time $O(m\beta(m,n))$. ■

[a] Technically, we need to choose $t_i \coloneqq 2^{\lceil 2m/n_i \rceil}$, but we will be a bit sloppy and ignore the ceilings here.

Lecture 2: MST and Tree Packing

Linear-Time Randomized Algorithm

29 Aug. 11:00

Using randomization, it's possible to derive a linear-time algorithm for MST.

Theorem 1.1.6 ([KKT95]). Karger-Klein-Tarjan's algorithm takes $O(m)$ time that computes the MST with probability at least $1 - 1/\mathsf{poly}(m)$.

Karger-Klein-Tarjan's algorithm relies on the so-called sampling lemma, which we first discussed.

Lemma 1.1.3 (Sampling lemma). Given a graph $G = (V, E)$, and let $E' \subseteq E$ be obtained by sampling each edge e with probability $p \in (0,1)$. Let F be a minimum spanning forest in $G' = (V, E')$ (can be disconnected). Then the expected number of F-light edge in G is less than $(n-1)/p$.

Proof. Let A be the set of F-light edges. Note that both A and F are random sets that are generated by the process of sampling E'. To analyze $\mathbb{E}[|A|]$, we consider Kruskal's algorithm to obtain F from E', where we generate E' on the fly:

Algorithm 1.5: Sampling Process

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, probability $p \in (0,1)$
Result: A minimum spanning forest F and the set of F-light edges A

Sort the edges such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$
$A \leftarrow \varnothing$, $F \leftarrow \varnothing$, $E' \leftarrow \varnothing$
for $i = 1, \dots, m$ **do**
    if `Ber(`p`)=1` **then** `// Toss a biased coin`
        $E' \leftarrow E' + e_i$
        if $F + e_i$ *is a forest* **then**
            $F \leftarrow F + e_i$, $A \leftarrow A + e_i$
    else if e_i *is* F*-light* **then**
        $A \leftarrow A + e_i$
return F, A

The following is exactly the same as the above, but easier to analyze:

Algorithm 1.6: Sampling Process with Tweaks

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, probability $p \in (0,1)$
Result: A minimum spanning forest F and the set of F-light edges A

Sort the edges such that $c(e_1) \le c(e_2) \le \cdots \le c(e_m)$
$A \leftarrow \varnothing$, $F \leftarrow \varnothing$
for $i = 1, \dots, m$ **do**
    if e_i *is* F*-light* **then** `// Sorting implies` $F + e_i$ `is a forest` $\Leftrightarrow$ e_i `is` F`-light`
        $A \leftarrow A + e_i$
        if `Ber(`p`)=1` **then** `// Toss a biased coin`
            $F \leftarrow F + e_i$
return F, A

The second algorithm makes the following observation clear.

Intuition. An edge e_i is added to A implies that it is added to F with probability p.

Hence, $p\mathbb{E}[|A|] = \mathbb{E}[|F|] \leq n-1$, hence $\mathbb{E}[|A|] \leq (n-1)/p$. ■

Remark. This proof is based on the *principle of deferred decisions* in randomized analysis.

With the sampling lemma, we know that when $p = 1/2$, the number of F-light edges from E is at most $2n$. Hence, we can eliminate most of the edges from $E \setminus E'$ from consideration given the fact that we can efficiently compute the F-heavy edges via the MST verification theorem. It's worth noting that to work with the sampling lemma via the natural recursion that it implies means that we need to work with potentially disconnected graph. That is, we will need to consider disconnected graph. Hence, we make the following generalization.

Definition 1.1.4 (Spanning forest). A *spanning forest* T of a graph $G = (V, E)$ (potentially disconnected) is an induced subgraph of G which spans G, i.e., $V(T) = V$ and $E(T) \subseteq E$.

Problem 1.1.3 (Minimum spanning forest). Given a graph $G = (V, E)$ (potentially disconnected) with edge capacity $c\colon E \to \mathbb{R}_+$, find the min-cost spanning forest.

Note. MST and MSF are closely related and one is reducible to the other in linear time, and the cut and cycle rules can be generalized to MSF easily.

Now, consider the following natural recursive divide and conquer algorithm for computing MSF.

Algorithm 1.7: Natural Recursive Algorithm from Sampling Lemma

Data: A graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MSF $T = (V, F)$

```
1 if |V| < n_0 then                                   // n_0 is some constant
2     return Standard-MST(G, c)          // Use a standard deterministic algorithm
3
4 Sample each edge i.i.d. from Ber(1/2) to obtain E_1 ⊆ E
5 (V, F_1) ←Karger-Klein-Tarjan((V, E_1))                  // Recursively compute MSF
6 E_2 ←Light-Edge(G, F_1)          // Compute all F_1-light edges with Theorem 1.1.4
7 (V, F_2) ←Karger-Klein-Tarjan((V, E_2))                  // Recursively compute MSF
8 return (V, F_2)
```

The correctness of Algorithm 1.7 is clear from the cut and cycle rules. The issue is the running time:

Claim. Algorithm 1.7 is not efficient enough.

Proof. The expected number of edges in $G_1 := (V, E_1)$ is $m/2$, and the expected number of edges in $G_2 := (V, E_2)$, via the sampling lemma, is at most $2n$. We see that the algorithm does $O(m+n)$ work outside the two recursive calls (line 5, line 7). Let $T(m, n)$ be the expected running time of the algorithm on an m-edge n-node graph. Informally, we see the following recurrence:

$$T(m, n) \leq c(m+n) + T(m/2, n) + T(2n, n).$$

If we take the problem size to be $n+m$, then Algorithm 1.7 generates two sub-problems of expected size $m/2 + n$ and $2n + n$, with the total size being $4n + m/2$. If $m > 10n$, say, then the total problem size is shrinking by a constant factor, and we obtain a linear-time algorithm. However, this is generally not the case. ⊛

The problem becomes reducing the graph size, which is the trick of Karger-Klein-Tarjan's algorithm: we run Borůvka's algorithm for a few iterations as a preprocessing step, reducing the number of vertices:

Algorithm 1.8: Karger-Klein-Tarjan's Algorithm [KKT95]

Data: A connected graph $G = (V, E)$[a] with edge capacity $c\colon E \to \mathbb{R}_+$
Result: A MSF $T = (V, F)$

```
1  if |V| < n_0 then                                   // n_0 is some large constant
2      return Standard-MST(G, c)                       // Use a standard deterministic algorithm
3
4  G' = (V', E'), T' = (V', F') ←Borůvka(G, c, 2)      // Run two iterations with |V'| ≤ |V|/4.
5
6  Sample each edge in G' i.i.d. from Ber(1/2) to obtain E_1 ⊆ E'
7  (V', F_1) ←Karger-Klein-Tarjan((V', E_1))           // Recursively compute MSF
8  E_2 ←Light-Edge(G_1, F_1)                           // Compute F_2-light edges with Theorem 1.1.4
9  (V', F_2) ←Karger-Klein-Tarjan((V', E_2))           // Recursively compute MSF
10 return (V, F' ∪ F_2)
```

[a] Assume no connected component of G is small.

Now, we provide the proof sketch of Theorem 1.1.6, which can be made precise with expectation.

Proof Sketch of Theorem 1.1.6. The correctness is easy to see as before. As for the running time, we see that Borůvka's algorithm takes $O(m)$ time for each phase, so the total time for the preprocessing (line 4) is $O(m)$. Then, the recurrence for $T(m, n)$ is

$$T(m, n) \le c(m + n) + T(m/2, n/4) + T(2n/4 + n/4),$$

i.e., the resulting sub-problem is of size $n/4 + m/2 + n/4 + n/2 = n + m/2$, which is good enough assuming $m \ge n - 1$.[a] By a simple inductive proof, we can show that $T(m, n) = O(n + m)$. ■

[a] Since we eliminate small components including singletons.

Remark. A more refined analysis of the sampling lemma can be used to show that the running time is linear with high probability as well.

Many properties of forests and spanning trees can be understood in the more general context of *matroids*. In many cases this perspective is insightful and also useful. The sampling lemma applies in this more general context and has various applications [Kar95; Kar98]. Obtaining a deterministic $O(m)$ time algorithm is a major open problem. Obtaining a simpler linear-time MST verification algorithm, even randomized, is also a very interesting open problem.

1.2 Tree Packing

We turn to another interesting problem, tree packing.

Problem 1.2.1 (Tree packing). Given a multigraph $G = (V, E)$, find all the edge-disjoint spanning trees in G. In particular, find the maximum number, $\tau(G)$, of edge-disjoint spanning trees of G

1.2.1 Bound on the Tree Packing Number

There is a beautiful theorem that provides a min-max formula for this. We first introduce some notation.

Notation. Let $\mathcal{P}$ be the collection of partitions of V, and E_P is the edge between connected components induced by a partition $P \in \mathcal{P}$, i.e., $e \in E_P$ if its endpoints are in different parts of P.

It's easy to see that any spanning tree must contain at least $|P| - 1$ edges from E_P. Thus, if G has k edge-disjoint spanning trees, then

$$k \le \frac{|E_P|}{|P| - 1}.$$

More generally, we have the following.

Theorem 1.2.1. The maximum number of edge-disjoint spanning trees in a graph G is given by

$$\tau(G) = \left\lfloor \min_{P \in \mathcal{P}} \frac{|E_P|}{|P| - 1} \right\rfloor.$$

Remark. Theorem 1.2.1 is a special case of a theorem on matroid base packing where it is perhaps more natural to see [Sch+03].

A weaker version of the theorem is regarding fractional packing. In fractional packing, we allow one to use a fraction amount of a tree. The total amount to which an edge can be used is at most 1 (or $c(e)$ in the capacitated case). Clearly, an integer packing is also a fractional packing. The advantage of fractional packings is that one can write a linear program for it, and they often have some nice properties. Let $\tau_{\text{frac}}(G)$ be the fraction tree packing number. Clearly, we have $\tau_{\text{frac}}(G) \geq \tau(G)$.

Corollary 1.2.1. Given a graph G, we have

$$\tau_{\text{frac}}(G) = \min_{P \in \mathcal{P}} \frac{|E_P|}{|P| - 1}.$$

Proof. Assuming Theorem 1.2.1, then with $c \coloneqq |P^*| - 1$ for $P^* = \arg\min_{P \in \mathcal{P}} |E_P|/(|P| - 1)$,

$$\tau(G_c) - \min_{P \in \mathcal{P}} \frac{c|E_P|}{|P| - 1} = \lfloor |E_{P^*}| \rfloor - |E_{P^*}| = 0,$$

where G_c is with edge capacity scaled up by c. This implies that $\tau_{\text{frac}}(G_c) = \tau(G_c)$. As this holds for every c (with different graphs), this can only happen if $\tau_{\text{frac}}(G) = \min_{P \in \mathcal{P}} |E_P|/(|P| - 1)$. ■

The second important corollary that is frequently used is about the min-cut. We see that while the min-cut size $\lambda(G)$ of G is upper-bounding $\tau(G)$, i.e., $\tau(G) \leq \lambda(G)$, this is not tight at all.

Corollary 1.2.2. Let G be a capacitated graph and let $\lambda(G)$ be the global min-cut size. Then

$$\tau_{\text{frac}}(G) \geq \frac{\lambda(G)}{2} \frac{n}{n-1}.$$

Proof. Let P^* be the optimum partition that induces $\tau_{\text{frac}}(G)$. Then, $\tau(G) = |E_{P^*}|/(|P^*| - 1)$. Since for every connected component induced by P^*, at least $\lambda(G)$ edges are going out, hence

$$\tau_{\text{frac}}(G) = \frac{|E_{P^*}|}{|P^*| - 1} \geq \frac{\lambda(G)/2 \cdot |P^*|}{|P^*| - 1} \geq \frac{\lambda(G)}{2} \frac{n}{n-1},$$

where we use the fact that $|P^*| \leq n$ and $i/(i-1)$ is decreasing. ■

We first see a tight example.

Example (Cycle). Consider the n-node cycle C_n. Clearly, $\tau(C_n) = 1$, and $\tau_{\text{frac}}(C_n) \leq n/(n-1)$ since each tree has $n-1$ edges and there are n edges in the graph. Indeed, we have $\tau_{\text{frac}}(C_n) = n/(n-1)$. Finally, we see that $\lambda(G) = 2$.

Proof. Consider n trees in C_n by deleting each of the n edges and assign a fraction value of $1/(n-1)$ for each of them. The corresponding tight partition consists of the n singleton vertices. ⊛

Note. Theorem 1.2.1 and its corollaries naturally extend to the capacitated case. For integer packing, we can assume c_e is an integer for each edge e, and the formula is changed to

$$\tau(G) = \left\lfloor \min_{P \in \mathcal{P}} \frac{c(E_P)}{|P| - 1} \right\rfloor.$$

Corollary 1.2.1 can also be proved in the same way when the edge capacity is rational.

Typically, one uses the connection between tree packing and min-cut to argue about the existence of many disjoint trees, since the global minimum cut is easier to understand than $\tau(G)$. However, we will see that one can use tree packing to compute $\lambda(G)$ exactly which may seem surprising at first due to the approximate relationship Corollary 1.2.2.

1.2.2 Proof of Corollary 1.2.1

Now, we give a different proof for Corollary 1.2.1 via LP duality without relying on Theorem 1.2.1.[2]

Proof of Corollary 1.2.1 [CQ17]. Consider $\mathcal{T}_G \coloneqq \{T \mid T \text{ is a spanning tree of } G\}$. Then, consider the following primal and the dual linear program:

$$\begin{aligned} \max \quad & \sum_{T\in\mathcal{T}_G} y_T & & & \min \quad & \sum_{e\in E} c(e)x_e & & \\ & \sum_{T\ni e} y_T \le c(e) & \forall e \in E; & & & \sum_{e\in T} x_e \ge 1 & \forall T \in \mathcal{T}_G; & (1.1) \\ (\mathrm{P}) \quad & y_T \ge 0 & \forall T \in \mathcal{T}_G; & & (\mathrm{D}) \quad & x_e \ge 0 & \forall e \in E. & \end{aligned}$$

Let y^* and x^* be the optimal solution to the primal and the dual. Then from the strong duality,

$$\sum_{T\in\mathcal{T}_G} y_T^* = \tau_{\text{frac}}(G) = \sum_{e\in E} c(e)x_e^*.$$

We see that if there exists e such that $x_e^* = 0$, then we can just contract all these edges, so without loss of generality, $x_e^* > 0$ for all $e \in E$.

> **Intuition.** If $x_e^* = 0$, we can effectively increase $c(e)$ to ∞ without affecting the value of the dual solution, i.e., e is not a bottleneck in the primal tree packing, hence safe to contract.

> **Claim.** If $x_e^* > 0$ for all $e \in E$, then $\tau_{\text{frac}}(G)$ is achieved via the singleton partition P. In particular,
>
> $$\tau_{\text{frac}}(G) = \frac{\sum_{e\in E} c(e)}{n-1}.$$
>
> **Proof.** From complementary slackness, we know that $\sum_{T\ni e} y_T^* = c(e)$ for all $e \in E$. Hence,
>
> $$(n-1)\sum_{T\in\mathcal{T}_G} y_T^* = \sum_{T\in\mathcal{T}_G}\sum_{e\in T} y_T^* = \sum_{e\in E}\sum_{T\ni e} y_T^* = \sum_{e\in E} c(e),$$
>
> implying that $\sum_{T\in\mathcal{T}_G} y_T^* = \sum_{e\in E} c(e)/(n-1)$. ⊛

Finally, we recall that $\tau_{\text{frac}}(G) \le \min_P |E_P|/(|P|-1)$, hence, the above claim gives us the desired conclusion via induction: this is true if $x_e^* > 0$ for all $e \in E$; otherwise, we contract edges with $x_e^* = 0$ and reduce to this case. ■

> **Remark.** In the above proof, the dual can be interpreted as a relaxation for the min-cut problem. In fact, if $x_e \in \{0, 1\}$, then this is exact.

1.2.3 Finding an Optimum Tree Packing and Approximating Tree Packing

If the linear program in the proof of Corollary 1.2.1 can be solved efficient to get $\tau_{\text{frac}}(G)$, then it will also yield an algorithm for the value of the integer packing $\tau(G)$ since it's just the floor of which. The problem is that while the primal has an exponentially many variables, the dual has an exponentially many constraints. We recall the following fact.

[2] Indeed, this is a hard theorem to prove, so we will not touch on this.

As previously seen. The Ellipsoid method needs a *separation oracle*. For example, applying it to the dual, we need to answer the following question efficiently:

- Given $x \in \mathbb{R}^E$, is it the case that $\sum_{e \in T} x_e \geq 1$ for all $T \in \mathcal{T}_G$?
- If not, find a tree T such that $\sum_{e \in T} x_e < 1$.

We see that this corresponds to solving MST, hence, the dual admits an efficient solution via the Ellipsoid method. One can convert an exact algorithm for the dual to an exact algorithm for the primal.

Remark. There are combinatorial algorithms for solving tree packing (both integer version and fraction versions) in strongly polynomial time [Sch+03].

On the other hand, we're also interested in whether we can find a faster algorithm for tree packing if one allows approximation. With an adaption of the *multiplicative weights update* method and data structures for MST maintenance, there is a near-linear time algorithm.

Theorem 1.2.2 ([CQ17]). There is a deterministic algorithm to compute a $(1-\epsilon)$-approximate fractional tree packing in $\widetilde{O}(m/\epsilon^2)$ time.

Lecture 3: Global Min-Cut with Tree Packing

1.3 Min-Cut and Steiner Min-Cut via Tree Packing

3 Sep. 11:00

Consider the following famous problems about min-cuts.

Problem 1.3.1 (s-t min-cut). Given a graph $G = (V, E)$ with edge capacity $c \colon E \to \mathbb{R}_+$, the *$s$-$t$ min-cut* problem aims to find $\min_{S \subseteq V \colon s \in S, t \in V \setminus S} c(\delta(S))$.

Problem 1.3.2 (Global min-cut). Given a graph $G = (V, E)$ with edge capacity $c \colon E \to \mathbb{R}_+$, the *global min-cut* problem aims to find $\min_{\varnothing \neq S \subsetneq V} c(\delta(S))$.

In what follows, we will simply use *min-cut* to refer to Problem 1.3.2. For convenience, we also introduce its well-known dual, the max-flow problem. We first define the flow formally.

Definition 1.3.1 (Flow). Given a directed graph $G = (V, E)$ with edge capacity $c \colon E \to \mathbb{R}_+$, a *flow* $f \colon E \to \mathbb{R}_+$ from s to t for some pair of $s, t \in V$ satisfies

(a) $0 \leq f(e) \leq c(e)$ for all $e \in E$;

(b) $\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$ for all $v \in V \setminus \{s, t\}$.

Notation (Value). Given a flow, the *value* of f is defined as $|f| \coloneqq \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$.

Then the problem of s-t max-flow can be formulated as follows.

Problem 1.3.3 (s-t max-flow). Given a directed graph $G = (V, E)$ with edge capacity $c \colon E \to \mathbb{R}_+$. The *$s$-$t$ max-flow* problem aims to find $\max_{f \in \mathcal{F}_{s,t}} |f|$ where $\mathcal{F}_{s,t}$ is the set of all s-t flows.

A naive way to solve the min-cut problem is to first fix one end $s \in V$, and compute the s-t min-cut for all $t \in V - s$. Fairly recent work shows how one can do it with only polynomial-logarithmically many max-flow computations (Theorem 1.3.7).

Over the years, several very different algorithmic approaches have been developed for these problems. One of the surprising ones is based on MA-orderings [NI92], which is a combinatorial $O(mn + n^2 \log n)$ time algorithm that does not rely on flow at all.[3] Another approach is to combine several flow com-

[3] This approach generalizes to symmetric submodular functions.

putations together via the push-relabel method [HO94], which also works for directed graphs. Karger developed elegant and powerful random contraction based algorithms for global min-cuts [Kar95], leading to many results. Two notable consequences are the following.

Theorem 1.3.1 ([KS96]). There is a randomized algorithm that runs in $O(n^2 \log n)$ time and outputs the min-cut with high probability.[a]

[a] This is a Monte-Carlo algorithm, so we cannot guarantee that the min-cut found is the correct one.

The following is a consequence of Karger's contraction algorithm [Kar95].

Theorem 1.3.2 (Approximate min-cut [Kar00]). The number of α-approximate min-cuts in a graph is at most $O(n^{2\alpha})$.

Definition 1.3.2 (Approximate min-cut). For $\alpha \geq 1$ an *α-approximate min-cut* is a cut $(S, V \setminus S)$ such that $c(\delta(S)) \leq \alpha\lambda(G)$.

Karger then developed another approach via tree packing to obtain a randomized near-linear time algorithm for min-cut. He also was able to refine the bound on approximate min-cuts via this approach.

Theorem 1.3.3 ([Kar00]). There is a randomized algorithm that runs in time $O(m \log^3 n)$ and outputs the min-cut with high probability.

While the random contraction based algorithm is taught quite frequently due to its elegance and simplicity, the tree packing approach is more technical. More recently, the tree packing approach has led to several new results, which we now discuss.

1.3.1 Tree Packing-Based Algorithm for Min-Cut

Recall Corollary 1.2.2, which gives $\tau_{\text{frac}}(G) \in [\frac{\lambda(G)}{2}\frac{n}{n-1}, \lambda(G)]$. Intuitively, even if we can compute $\tau_{\text{frac}}(G)$ exactly, we have a 2-approximation to $\lambda(G)$. However, this already leads a crucial observation:

Intuition. On average, each tree can't cross the min-cut more than twice.

To formalize the above intuition, consider the following definition.

Definition 1.3.3 (Respecting). Let $T = (V, E_T)$ be a spanning tree and $(S, V \setminus S)$ be a cut. The for an integer $h \geq 1$, we say T is *h-respecting* w.r.t. S if $|E_T \cap \delta(S)| \leq h$.

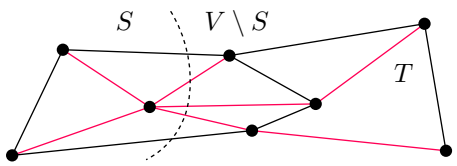


Figure 1.1: The spanning tree T is shown in red edges. T is 3-rspecting the cut $(S, V \setminus S)$.

We can now formalize the intuition in Lemma 1.3.1.

Lemma 1.3.1. Suppose $\{y_T\}_{T \in \mathcal{T}_G}$ is a $(1-\epsilon)$-approximate tree packing of G, and $\delta(S)$ is a min-cut of G. Let $\ell_T := |E_T \cap \delta(S)|$ be the number of edges of T that cross the cut S. Furthermore, let $p_T = y_T / \sum_{T \in \mathcal{T}_G} y_T$ and $q := \sum_{T:\, \ell(T) \leq 2} p_T$. Then,

$$q \geq \frac{1}{2}\left(3 - \frac{2}{1-\epsilon}\left(1 - \frac{1}{n}\right)\right).$$

In particular, if $\epsilon = 0$, then $1 \geq 1/2 + 1/n$, and if $\epsilon < 1/5$, then $q > 1/4$.

Proof. From the assumption, $\sum_{T \in \mathcal{T}_G} y_T \geq (1-\epsilon)\tau_{\text{frac}}(G)$. With Corollary 1.2.2, we have

$$\sum_{T \in \mathcal{T}_G} y_T \geq (1-\epsilon)\frac{n}{n-1}\frac{\lambda(G)}{2}.$$

Let $S \subseteq V$ be a min-cut, we have $1 = \sum_{T \in \mathcal{T}_G} p(T) = \sum_{T\colon \ell(T) \leq 2} p_T + \sum_{T\colon \ell(T) \geq 3} p_T$. Observe that

- each tree T with $\ell(T) \geq 3$ uses up at least 3 edges from $\delta(S)$; while
- each tree T with $\ell(T) \leq 2$ uses up at least 1 edge from $\delta(S)$.

Since the total capacity of $\delta(S)$ is $\lambda(G)$, and the tree packing solution is valid, we have

$$\sum_{T\colon \ell(T) \leq 2} y_T + 3 \sum_{T\colon \ell(T) \geq 3} y_T \leq \lambda(G) \Rightarrow q + 3(1-q) \leq \frac{\lambda(G)}{\sum_{T \in \mathcal{T}_G} y_T} \leq \frac{2}{1-\epsilon}\left(1 - \frac{1}{n}\right),$$

where the last inequality follows from the very first inequality we have derived. ■

Remark. Lemma 1.3.1 states that if the tree packing is sufficiently good, then a constant fraction of the trees in the packing will cross the min-cut at most twice.

Now, we're ready to see Karger's algorithm for min-cut [Kar00]. However, the original algorithm was more involved since at that time, there was no near-linear time approximation algorithm for tree packing, so he used a form of sparsification and then applied an approximation tree packing algorithm on the sparsified graph which is quite a feat. In our case, recall that following.

As previously seen. Theorem 1.2.2 states that we can compute a $(1-\epsilon)$-approximate tree packing of G, given by $\{y_T\}_{T \in \mathcal{T}_G}$, in $O(m \log^3 n/\epsilon^2)$ time.

By black-boxing this near-linear time tree packing algorithm, consider the following.

Algorithm 1.9: Tree Packing-Based Min-Cut Algorithm [Kar00; CQ17]

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, $\epsilon_0 \in < 1/5$
Result: A cut S

1 $\{y_T\}_{T \in \mathcal{T}_G} \leftarrow$ `Approximate-Tree-Packing`(G, c, ϵ_0) // $O(m \log^3 n)$
2 Sample a tree T with probability $p_T = y_T / \sum_{T \in \mathcal{T}_G} y_T$
3 Find the cheapest cut $(S, V \setminus S)$ in G such that T is 2-respecting w.r.t. S
4 **return** S

Firstly, we see that Algorithm 1.9 admits the following.

Lemma 1.3.2. Algorithm 1.9 outputs the min-cut of G with probability at least $1/4$.

Proof. It's immediate from Lemma 1.3.1. ■

To boost the success probability, we can simply repeat the last two steps (line 2, line 3) $\Theta(\log n)$ times, which results in a success probability to at least $1 - 1/n^c$ for any constant c. To analyze the running time, a key ingredient is line 3. Karger showed that one can implement line 3 via a clever dynamic programming coupled with link-cut tree data structure:

Theorem 1.3.4 ([Kar00]). Given a graph $G = (V, E)$ and a spanning tree $T = (V, E_T)$. There is a deterministic algorithm that computes a minimum cut $(S, V \setminus S)$ such that T is 2-respecting w.r.t. S in $O(m \log^2 n)$ time.

We can now prove Theorem 1.3.3.

Proof of Theorem 1.3.3. Since line 1 takes $O(m \log^3 n)$ for ϵ_0 being a constant, and observe that once the approximated tree packing $\{y_T\}_{T \in \mathcal{T}}$ is computed, we can reuse them and apply the repe-

tition for line 2 and line 3 to boost the probability of success. With $\Theta(\log n)$ repetitions, we obtain an $O(m \log^3 n)$ time algorithm as desired with the running time guaranteed by Theorem 1.3.4. ■

1.3.2 Bounding the Number of Approximate Min-Cuts

As hinted in Theorem 1.3.2, we're now interested in how many distinct min-cuts can an undirected graph have. The following theorem was shown a long time ago:

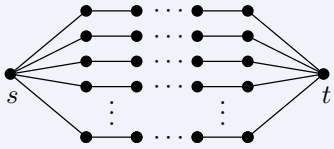
Theorem 1.3.5 ([DKL76]). The number of distinct min-cuts in an undirected graph is at most $\binom{n}{2}$.

Example (Cycle). The worst case example is an n-cycle C_n.

Remark. All the min-cuts of a graph can be represented in a nice and compact data structure called the cactus (cactus representation), which was also shown in [DKL76].

In contrast, for s-t min-cuts, it can be exponentially many in n.

Example. Consider the following multi-highway-like graph, which has exponentially many s-t min-cuts since if we choose one of the road section in each line of the road, it'll be a s-t min-cut.



Recall earlier in Theorem 1.3.2, we stated that Karger used tree packing to prove that the number of α-approximation min-cuts is at most $O_\alpha(n^{\lfloor 2\alpha \rfloor})$. We now prove it formally.

Proof of Theorem 1.3.2 [CQX20]. Consider an optimum fraction tree packing solution $\{(T, y_T^*)\}_{T \in \mathcal{T}_G}$. In the proof of Corollary 1.2.1, where we define the fractional tree packing linear program, we know that there are only m non-trivial constraints, hence there are only m many T's such that $y_T^* > 0$.

> **As previously seen.** A solution x^* to a linear program which has n non-trivial constraints means that the support size of x is at most n, i.e., $x_i > 0$ for at most n many i's.

Consider an α-approximate min-cut $S \subseteq V$, and let $h = \lceil 2\alpha \rceil$. Now, let $q_{h,\alpha}$ be the fraction of tree packing that h-respects $S \subseteq V$, i.e., $q_{h,\alpha} \coloneqq \sum_{T\colon \ell(T) \leq h} p_T$. Using a similar analysis as the one in Lemma 1.3.1, we can argue that

$$q_{h,\alpha} \geq \frac{1}{h}(1 - (2\alpha - \lfloor 2\alpha \rfloor)) \left(1 - \frac{1}{n}\right).$$

The main intuition is the following:

> **Intuition.** Say at least one tree in the packing h-respects the cut (which is the case). Then, the total number of α-approximate min-cuts is at most $m \cdot n^h \leq m \cdot n^{\lfloor 2\alpha \rfloor}$.

But we can do better by noticing that $q_{h,\alpha} > 0$ is a fixed constant for any fixed α. Suppose N is the number of α-approximate min-cuts. For any fixed α-approximate min-cut, $q_{h,\alpha}$ fraction of the tree packing is h-respecting w.r.t. the cut. Consider the following question:

> **Problem.** Fix a single tree T, how many distinct cuts are there such that T h-respects w.r.t.?
>
> **Answer.** We can remove at most h edges from T to create at most $h+1$ components and combine these components into two sides of a cut, hence, each tree T correspond to at most $2^{h+1}\binom{n-1}{h} \leq 2^{h+1} n^h$ cuts. ⊛

Thus, the number of α-approximate min-cuts is at most $2^{h+1} n^h / q_{h,\alpha}$. ■

Lecture 4: Steiner Min-Cut with Isolating Cuts

1.3.3 Steiner Min-Cut

5 Sep. 11:00

Consider the following problem that generalizes the *s-t* min-cut and global min-cut.

Problem 1.3.4 (Steiner min-cut)**.** Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and a set $T \subseteq V$ of terminals, the *Steiner min-cut* problem aims to find the min-cut $(S, V \setminus S)$ which separates some pair of terminals, i.e., $S \cap T \neq \varnothing$ and $(V \setminus S) \cap T \neq \varnothing$.

Remark. Steiner min-cut generalizes both *s-t* min-cut and global min-cut.

Proof. *s-t* min-cut corresponds to $T = \{s, t\}$, while global min-cut corresponds to $T = V$. ⊛

A simple algorithm for the Steiner min-cut is the same as the global min-cut by solving *s-t* min-cut: for $T = \{t_1, \ldots, t_k\}$, fix a terminal, say t_1, then compute t_1-t_i min-cut for all $i \geq 2$. This requires $|T| - 1$ max-flow computations. In fact, this is the best known algorithm even for the global min-cut till [NI92].

Quite recently, a simple yet striking approach that computes the Steiner min-cut with high probability using only $O(\log^3 n)$ *s-t* cut computations is developed [LP20], which is based on isolating cut.

Submodular Function

The main interest here, i.e., solving isolating cut, will be essentially based on properties of symmetric submodular functions. Although we can prove various properties by appealing to first principles, it's useful to see the proofs via submodularity. Here, we give some necessarily background.

Definition. Given a finite ground set V, consider a real-valued set function $f\colon 2^V \to \mathbb{R}$.

Definition 1.3.4 (Modular)**.** The function f is *modular* if for all $A, B \subseteq V$,

$$f(A) + f(B) = f(A \cap B) + f(A \cup B).$$

Definition 1.3.5 (Submodular)**.** The function f is *submodular* if for all $A, B \subseteq V$,

$$f(A \cap B) + f(A \cup B) \leq f(A) + f(B).$$

Definition 1.3.6 (Supermodular)**.** The function f is *supermodular* if for all $A, B \subseteq V$,

$$f(A \cap B) + f(A \cup B) \geq f(A) + f(B).$$

Definition 1.3.7 (Posi-modular)**.** The function f is *posi-modular* if for all $A, B \subseteq V$,

$$f(A - B) + f(B - A) \geq f(A) + f(B).$$

We note that perhaps a more common definition of submodularity is *diminishing marginal utility*, i.e., if $f(A + v) - f(A) \geq f(B + v) - f(B)$ for all $A \subseteq B$. Here, we see some examples.

Example (Modular function as weight function)**.** f is modular if and only if there exists some $w\colon V \to \mathbb{R}$ such that $f(A) = \sum_{v \in A} w(v) + c$ for some shift c.

Example. If f and g are submodular, then so is $\alpha f + \beta g$ for some $\alpha, \beta \geq 0$.

One of the reason that submodularity is important for graphs is because of the following.

Example (Cut)**.** Given a graph $G = (V, E)$, the cut size function $|\delta_G(\cdot)|\colon 2^V \to \mathbb{R}_+$ is submodular.

Proof. We simply note that for any $A, B \subseteq V$,

$$|\delta_G(A)| + |\delta_G(B)| = |\delta_G(A \cap B)| + |\delta_G(A \cup B)| + 2|E(A \setminus B, B \setminus A)| \geq |\delta_G(A \cap B)| + |\delta_G(A \cup B)|,$$

where $E(X, Y)$ is the set of edges crossing X and Y for some $X, Y \subseteq V$. ⊛

The above argument extends naturally to non-negative capacitied graphs and directed graphs.

Example. For a directed graph, $|\delta^+(\cdot)|$ is submodular (so does $|\delta^-(\cdot)|$ by symmetry).

We're also interested in the following property.

Definition 1.3.8 (Symmetric)**.** A set function is *symmetric* if $f(A) = f(V \setminus A)$ for all $A \subseteq V$.

Clearly, $|\delta_G(\cdot)|$ is symmetric. However, for directed graph, this is not necessarily the case. Finally, we see that symmetric submodular function satisfies another important property.

Example. A symmetric submodular function is automatically posi-modular.

Now, we discuss uncrossing, a common and powerful technique that is frequently used in working with submodular functions. We illustrate this in the context of min-cuts.

Lemma 1.3.3. Let $G = (V, E)$ be a graph and $(A, V \setminus A)$, $(B, V \setminus B)$ be two s-t min-cuts. Then $(A \cap B, V \setminus (A \cap B))$ and $(A \cup B, V \setminus (A \cap B))$ are also s-t min-cuts.

Proof. From submodularity, we have $|\delta(A)| + |\delta(B)| \geq |\delta(A \cap B)| + |\delta(A \cup B)|$. However, as both $A \cup B$ and $A \cap B$ are themselves s-t cuts, all terms need to be equal. ■

Corollary 1.3.1. For any graph $G = (V, E)$, there is a unique (inclusion-wise) minimal s-t min-cut.[a]

[a] While maybe not that useful, from the same logic, there is a unique maximal s-t min-cut.

Proof. If there are two s-t min-cuts A, B that are both minimal and distinct, then $A \setminus B \neq \varnothing$ and $B \setminus A = \varnothing$ since otherwise one will be contained in the other, contradicting the minimality. From Lemma 1.3.3, $A \cap B$ is also a s-t min-cut and $A \cap B$ is a strict subset of A and B, again contradicting minimality of A and B. ■

The above proof applies to directed graph as well since we only used submodularity.

Remark (Graphic matroid)**.** A second aspect of submodularity in graphs comes via *matroids*. We will not discuss it here but the rank function of a matroid is a special class of submodular functions; and in a formal sense, matroid rank functions are building blocks for all submodular functions. Given an undirected graph $G = (V, E)$ there is a fundamental matroid associated with the edge set of G called the *graphic matroid*.[a] Several properties of trees and forests can be better understood in the context of the graphic matroid including the Tutte-Nash-Williams theorem.

[a] There are other matroids that are also defined from graphs including the dual graphic matroid for instance.

Isolating Cuts via Poly-Logarthmic Many Max-Flow Computations

We can now formally introduce the isolating cut problem.

Problem 1.3.5 (Isolating cut)**.** Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and a set $T \subseteq V$ of terminals. The *t_i-isolating cut* problem aims to find a cut $(S_i, V \setminus S_i)$ such that $t_i \in S_i$ and $t_j \notin S_i$ (i.e., $t_j \in V \setminus S_i$) for all $j \neq i$, i.e., the cut isolates t_i from the rest of the terminals.

The minimum capacity t_i-isolating cut can be found by a single max-flow computation: by shrinking the terminals in $T - t_i$ into a single vertex s and computing the s-t_i min-cut. Thus, naively, computing all isolating cuts require k max-flow computations. The upshot is that this can be done in only $O(\log k)$

max-flow. Before we describe the algorithm, we first note that from submodularity, we also have a similar structural result, just like Corollary 1.3.1.

Lemma 1.3.4. There is a unique minimal t_i-isolating min-cut $(S_i^*, V \setminus S_i^*)$ such that if $(S_i, V \setminus S_i)$ is any t_i-isolating min-cut, then $S_i^* \subseteq S_i$.

We now describe the algorithm for computing the isolating cuts. Basically, we consider h bi-partitions $(A_1, B_1), \ldots, (A_h, B_h)$ with $h = \lceil \log k \rceil$, and compute a cut separating each bi-partition. Then, we take the intersection among the resulting cut sets, which will be isolating cuts as we will see. Finally, with the structural property Lemma 1.3.4, we can then find the minimum isolating cuts from them.

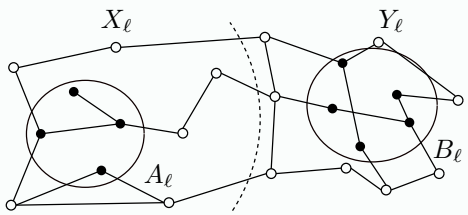
Algorithm 1.10: Isolating Cut [LP20] (also developed independently in [AKT21])

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, terminal $T = \{t_i\}_{i=1}^k$
Result: A set of isolating cuts $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^k$ isolating t_i's

1 **for** $\ell = 1, \ldots, \lceil \log k \rceil$ **do** // $\lceil \log k \rceil$ `bi-partitions` (A_ℓ, B_ℓ) `of` T
2 $A_\ell \leftarrow \{t_i \mid$ binary representation of i has 1 in ℓ^{th} bit$\}$
3 $B_\ell \leftarrow T \setminus A_\ell$
4 $(X_\ell, Y_\ell) \leftarrow$ *s-t*-min-cut$(G,\ c,\ A_\ell,\ B_\ell)$ [a] // $Y_\ell = V \setminus X_\ell$
5 **for** $i = 1, \ldots, k$ **do**
6 $S_i \leftarrow \bigcap_{\ell:\ t_i \in A_\ell} X_\ell \cap \bigcap_{\ell:\ t_i \in B_\ell} Y_\ell$
7 $H_i = (V_i, E_i) \leftarrow$ shrinking $V \setminus S_i$ to a single vertex s_i
8 $(S_i^*, V \setminus S_i^*) \leftarrow$ *s-t*-min-cut$(H_i,\ c,\ t_i,\ s_i)$
9 **return** $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^k$

[a] This can be done via shrinking A_ℓ and B_ℓ in to two separate nodes, and compute the s-t min-cut.

Intuition. The following illustrates line 4, where terminals are black vertices. (A_ℓ, B_ℓ) is a bi-partition of T, while (X_ℓ, Y_ℓ) is a min-cut that separates (A_ℓ, B_ℓ).



Additionally, line 6 is created by considering the intersections of all X_ℓ (or Y_ℓ) that includes t_i.

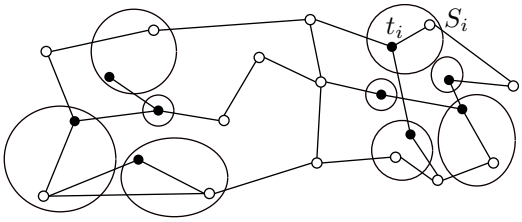
To see the correctness of the Algorithm 1.10, we want to say that s_i-t_i min-cut in H_i is exactly the minimum cost t_i-isolating cut in G. This is due to Lemma 1.3.5.

Lemma 1.3.5. For each i, $(S_i, V \setminus S_i)$ is a t_i-isolating cut. Furthermore, S_i's are pairwise disjoint.

Proof. Firstly, $t_i \in A_\ell \subseteq X_\ell$ or $t_i \in B_\ell \subseteq Y_\ell$, implying $t_i \in S_i$. Consider t_j with $j \neq i$. As $i \neq j$, there is some index ℓ in the binary representation of i and j differ in the bit position. Suppose i has 1 in the ℓ^{th} position and j has 0, then $t_i \in A_\ell$ and $t_j \in B_\ell$, implying $t_i \in X_\ell$ and $t_j \notin X_\ell$ as $t_j \in B_\ell \subseteq Y_\ell$ and $Y_\ell \cap X_\ell = \varnothing$. This means $t_j \notin S_i$.

We now prove that $S_i \cap S_j = \varnothing$ for all $i \neq j$. Firstly, there exists some ℓ such that $t_i \in A_\ell$ and $t_j \in B_\ell$ (or $t_i \in B_\ell$ and $t_j \in A_\ell$). Suppose $v \in X_\ell$, then v can't be in $S_j \subseteq Y_\ell$ and if $v \in Y_\ell$, then v can't be in $S_i \subseteq X_\ell$, hence v can't be in both S_i and S_j. ■

Lemma 1.3.5 gives the following picture, where each t_i lives in exactly one S_i.



Hence, for each i, we have a t_i-isolating cut $(S_i, V \setminus S_i)$. Now, Lemma 1.3.4 states that there is a t_i-isolating min-cut $(S_i^*, V \setminus S_i^*)$ where S_i^* is a subset of any t_i-isolating min-cut, it doesn't say it will be a subset of S_i in particular, as $(S_i, V \setminus S_i)$ is only a t_i-isolating cut. However, we do not lose anything:

Lemma 1.3.6. The minimal t_i-isolating min-cut $(S_i^*, V \setminus S_i^*)$ is in $(S_i, V \setminus S_i)$, i.e., $S_i^* \subseteq S_i$.

Proof. It suffices to prove that if $t_i \in A_\ell$ then $S_i^* \subseteq X_\ell$. Assume not, then $S_i^* \cap (V \setminus X_\ell) \neq \varnothing$. But since $S_i^* \cap X_\ell$ is a t_i-isolating cut, while S_i^* is the minimal t_i-isolating min-cut, $|\delta(S_i^* \cap X_\ell)| > |\delta(S_i^*)|$.

Moreover, it's trivial to see that $S_i^* \cup X_\ell$ is a A_ℓ-B_ℓ cut (not necessarily minimum, just a cut), hence we also have $|\delta(S_i^* \cup X_\ell)| \geq |\delta(X_\ell)|$. From submodularity of $|\delta(\cdot)|$, we have

$$|\delta(S_i^*)| + |\delta(X_\ell)| \geq |\delta(S_i^* \cap X_\ell)| + |\delta(S_i^* \cup X_\ell)|,$$

which is a contradiction. ■

With all the lemmas, it's now easy to see that Algorithm 1.10 is at least correct. Firstly, from Lemma 1.3.6, we know that the optimal t_i-isolating min-cut S_i^* is a subset of S_i (here, S_i^* is not necessary the one found by Algorithm 1.10, we're trying to argue this). As S_i's are disjoint from Lemma 1.3.5, each terminal t_i lives in exactly one S_i, hence computing s_i-t_i min-cut will indeed recover S_i^*.

Intuition. When we contract $V \setminus S_i$, we do not lose the optimal isolating cut S_i^*.

Theorem 1.3.6 ([LP20]). Algorithm 1.10 is a deterministic algorithm that given $G = (V, E)$ and a terminal set $T \subseteq V$ with $|T| = k$, computes all the isolating cuts using $O(\log k)$ max-flow computations on graphs with $|V|$ vertices and $|E|$ edges each.

Proof. We analyze the runtime. It's easy to see that line 1 requires $O(\log k)$ max-flow computations on G. It's also easy to show that computing S_i's in line 6 can be done in $O((m+n)\log k)$ time given (X_ℓ, Y_ℓ) for $\ell \in [\lceil \log k \rceil]$. However, line 7 and line 8 seem to require k max-flow computations.

Claim. In total, line 8 only requires $O(\log k)$ max-flow computations.

Proof. Let us understand the size of H_i. It has $n_i + 1$ vertices where $n_i = |S_i|$, and it has m_i edges where $m_i = |E(S_i)| + |\delta(S_i)|$. Thus, the running time of max-flow on H_i is $T(n_i + 1, m_i)$ where $T(a, b)$ is the running time of max-flow on graph with a nodes and b edges. We observe that $\sum_i (n_i + 1) \leq 2n$ since S_i's are disjoint, while $\sum_i m_i \leq 2m$: consider any edge $uv \in E$. If $uv \in E(S_i)$ for some i, then it does not contribute to any other H_j. If $uv \in \delta(S_i)$ for some i, then it can be in $\delta(S_j)$ for only one more index $j \neq i$.

Thus, the total time to compute all k max-flows is $\sum_i T(n_i, m_i) \leq T(2n, 2m)$ under reasonable assumption, specifically, $T(a, b)$ is super-additive.[a] ⊛

[a] Formally, we first create a single H that includes each H_i as a copy in it, and we can run a single max-flow on H to recover all the max-flow values in each H_i. H will have $O(n)$ vertices and $O(m)$ edges.

With the correctness of Algorithm 1.10, the theorem is proved. ■

We see that this could have been discovered many years ago in terms of its simplicity. Algorithm 1.10 has been very influential in the last few years for a number of problems.

Note. Another perspective of the bi-partitions is that they are a way to *derandomize* a natural randomized algorithm that picks some $O(\log k)$ bi-partitions of T at random and computes the cuts between them. With high probability, every t_i, t_j with $i \neq j$ will be separated in at least on of the random bi-partitions.

Remark. The core idea of isolating cuts relies only on submodularity and symmetry, thus, this applies in much more generality and to several other problems. This is explicitly discussed in [CQ21], though the ideas are implicit in [LP20].

Randomized Algorithm for Steiner Min-Cut via Isolating Cuts

Isolating cut naturally lead to a simple randomized algorithm for Steiner min-cut. The basic idea is quite simple. Consider an optimum Steiner min-cut $(S, V \setminus S)$ and let $T_1 \coloneqq S \cap T$ and $T_2 \coloneqq (V \setminus S) \cap T$, with $k_1 = |T_1|$ and $k_2 = |T_2|$. We may assume that $1 \leq k_1 \leq k_2$.

Note. $(S, V \setminus S)$ is a t_i-t_j min-cut for any $t_i \in T_1, t_j \in T_2$ since otherwise, a lower-cost cut exists.

The basic intuition is the following.

Intuition. If we can sample exactly one terminal in one side of the Steiner min-cut, then we can simply use the isolating cut to recover the Steiner min-cut.

Say we know k_1. We can sample each terminal in T independently with probability $1/k_1$ to obtain $T' \subseteq T$ such that with constant probability, $|T' \cap T_1| = 1$ and $|T' \cap T_2| \geq 1$ (recall $k_1 \leq k_2$). Suppose T' satisfies these properties and let $T' \cap T_1 = \{t_i\}$. Then, $(S, V \setminus S)$ is a minimum cost t_i-isolating cut w.r.t. T'. Hence, by computing t_i-isolating cuts for all $t_i \in T'$ and choosing the cheapest one identifies the Steiner min-cut for T. The problem is that we don't know k_1, and trying all possible values for k_1 (from 1 to $k/2$) will be too expensive.

Intuition. The above sampling procedure is robust: if we sample with probability, say, $1/2k_1$, everything still happens with constant probability. Hence, we only need to try $k_i = 2^i$, i.e., $O(\log k)$ different sampling probabilities.

We formally describe this algorithm as follows.

Algorithm 1.11: Steiner Min-Cut

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, terminal $T = \{t_i\}_{i=1}^k$
Result: A possible Steiner min-cut $(S^*, V \setminus S^*)$

$S^* \leftarrow \varnothing$ // Initialize Steiner min-cut
for $i = 0, \ldots, \lceil \log k \rceil$ **do**
  $T' \leftarrow$ `Sample`$(T,\ 1/2^i)$ // Sample each terminal in T with probability $1/2^i$
  $\{(S_i^*, V \setminus S_i^*)\}_{i=1}^{|T'|} \leftarrow$ `Isolating-Cut`$(G,\ c,\ T')$
  $S^* \leftarrow$ `Min-Cost`$(\{(S_i^*, V \setminus S_i^*)\}_{i=1}^{|T'|} \cup \{(S^*, V \setminus S^*)\})$ // Update minimum cost cut
return $(S^*, V \setminus S^*)$

We now formally prove the robustness we have mentioned.

Lemma 1.3.7. Algorithm 1.11 finds the Steiner min-cut for T with a constant probability.

Proof. We see that for $k_1 = 1$, Algorithm 1.11 is correct (deterministically) since i can only be 0 and $T' = T$. Hence, let $k_1 > 1$. Consider the case that $1/2^{i+1} < 1/k_1 \leq 1/2^i$, where i will be tried at some point during $i = 0$ to $\lceil \log k \rceil$ since $1 \leq k_1 \leq k/2$. Let $\ell = 2^i$, i.e., $\ell \leq k_1 \leq 2\ell$.

Now, let $\mathcal{E}_1$ be the event that $|T_1 \cap T'| = 1$, i.e., exactly one terminal from T_1 is chosen. Then

$$\Pr(\mathcal{E}_1) = k_1 \cdot \frac{1}{\ell} \cdot \left(1 - \frac{1}{\ell}\right)^{k_1 - 1} \geq \left(1 - \frac{1}{\ell}\right)^{2\ell} \geq \frac{1}{e^2}.$$

On the other hand, let $\mathcal{E}_2$ be the event that $T_2 \cap T' \neq \varnothing$. We see that

$$\Pr(\mathcal{E}_2) \geq 1 - \left(1 - \frac{1}{\ell}\right)^{k_2} \geq \left(1 - \frac{1}{\ell}\right)^{\ell} \geq 1 - \frac{1}{e}.$$

Since T_1 and T_2 are disjoint, $\mathcal{E}_1$ and $\mathcal{E}_2$ are independent, we have

$$\Pr(\mathcal{E}_1 \cap \mathcal{E}_2) \geq \left(1 - \frac{1}{e}\right) \cdot \frac{1}{e^2},$$

which is a constant. ■

Theorem 1.3.7. There is a randomized algorithm that given $G = (V, E)$ and terminal set $T \subseteq V$ of size k, outputs the Steiner min-cut with high probability using $O(\log^2 k \log n)$ max-flow computations on graphs with $|V|$ vertices and $|E|$ edges.

Proof. From Lemma 1.3.7, Algorithm 1.11 successes with a constant probability. We further boost the overall success probability by rerunning Algorithm 1.11 $\Theta(\log n)$ times. With Theorem 1.3.6, this requires $O(\log^2 k \log n)$ max-flow computations. ■

Remark (Deterministic algorithm). Li and Panigraphy [LP20] also developed deterministic min-cut and Steiner min-cut algorithms using additional ideas based on expander decomposition.

Lecture 5: Metric Embedding and Multi-Cut Problem

1.3.4 Max-Flow Min-Cut Theorem

10 Sep. 11:00

Finally, we conclude this section by proving the well-known max-flow min-cut theorem. Consider the following linear program relaxation of the s-t min-cut where s, t are two distinct vertices:

$$\begin{aligned}
&\min \sum_{e \in E} c(e) x_e && && \max \sum_{P \in \mathcal{P}_{s,t}} y_P && \\
&\sum_{e \in P} x_e \geq 1 && P \in \mathcal{P}_{s,t}; && \sum_{P \ni e} y_P \leq c(e) && \forall e \in E; \\
\text{(P)}\quad & x_e \geq 0 && \forall e \in E; && \text{(D)}\quad y_P \geq 0 && \forall P \in \mathcal{P}_{s,t},
\end{aligned} \tag{1.2}$$

where $\mathcal{P}_{s,t}$ is the set of all s-t paths. The integer version is with constraints $x_e \in \{0, 1\}$.

Remark. An s-t cut is often also defined as $\delta^+(S)$ for some $S \subseteq V$ where $s \in S$ and $t \in V \setminus S$.

Proof. Suppose E' is an s-t cut and S is the set of nodes reachable from s in $G - E'$. Then, $\delta(S) \subseteq E'$ and $\delta(S)$ is an s-t cut. Hence, it suffices to focus on such limited type of cuts.[a] ⊛

[a] In some more general settings, it is useful to keep these notions separate.

It is well-known that s-t min-cut can be computed efficiently via s-t max-flow, establishing the max-flow min-cut theorem. This fundamental theorem in combinatorial optimization has many applications, and is typically established via the augmenting path algorithm. Here, we give another proof.

Theorem 1.3.8 (Max-flow min-cut). Let $G = (V, E)$ be a directed graph with rational edge capacities $c \colon E \to \mathbb{Q}_+$ and let $s, t \in V$ be two distinct vertices. The s-t max-flow value in G is equal to the s-t min-cut value, and both can be computed in strongly polynomial time. Furthermore, if c is integer valued, then there exists an integer-valued max-flow as well.

Proof. To start, we observe that the primal linear program assigns lengths to edges such that the s-t shortest path according to which is at least 1. This is a fractional relaxation of the cut. We claim that it's possible to round the fractional solution of the primal to the exact s-t min-cut without any

loss. Consider the following rounding algorithms for the primal linear program.

Algorithm 1.12: θ-Rounding Algorithm

Data: A directed graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, $s, t \in V$
Result: A s-t min-cut F

```
{x_e}_{e∈E} ← LP-Solve(Min-Cut-LP(G, c, s, t))        // Solve the primal
θ ← Uniform((0, 1))
for v ∈ V do
    d_x(s, v) ← Shortest-Path-Dist(s, v, G, x)
return F = δ⁺(B_x(s, θ))                  // B_x(s, θ) = {v ∈ V | d_x(s, v) ≤ θ}
```

Firstly, Algorithm 1.12 will output a valid s-t cut since $d_x(s,t) \geq 1$ by feasibility of the linear program solution x and hence $t \notin B_x(s,\theta)$ for any $\theta < 1$.

Claim. For any $e \in E$, $\Pr(e \text{ is cut by Algorithm 1.12}) \leq x_e$.

Proof. The edge $e = (u, v)$ is cut if and only if $d_x(s,u) \leq \theta < d_x(s,v)$. Hence, the edge is not cut if $d_x(s,v) \leq d_x(s,u)$. If $d_x(s,v) > d_x(s,u)$, we have $d_x(s,v) - d_x(s,u) \leq x_{(u,v)}$. Since θ is chosen uniformly at random from $(0,1)$, the probability that θ lies in the interval $[d_x(s,u), d_x(s,v)]$ is at most $x_{(u,v)}$. ⊛

With this claim, from linearity of expectation, we see that $\mathbb{E}[c(\delta^+(B_x(s,\theta)))] \leq \sum_{e\in E} c(e)x_e$. As $B_x(s,\theta)$ will always be a valid s-t cut, this implies that there is an integral cut whose cost is at most that of the linear program relaxation, implying that the linear program relaxation yields an optimum solution.

Finally, observe that the dual is the *path version* of the s-t max-flow. Hence, from strong duality, the optimal value of s-t min-cut is the same as the s-t max-flow. Moreover, we note that the primal is strongly polynomial-time solvable if we have a separation oracle. In this case, given $\{x_e\}_{e\in E}$, we need to answer either this is a feasible solution, or outputs some path p such that $\sum_{e\in P} x_e < 1$, which is exactly the shortest s-t path algorithm and can be solved efficiently. ■

Intuition (Line embedding). The rounding can be thought as putting every vertex on a line from s to t, sorting by their distances given by x_e's. Then, observe that on that line, any two vertices $u, v \in V$ has distance at most $x_{(u,v)}$, and picking θ corresponds to picking a threshold on the line.

The above intuition not only helps the analysis, but also gives a way to derandomize Algorithm 1.12. Basically, we can try all possible θ's, and if we adapt this line embedding viewpoint, the only interesting θ's are given by the n values $d_x(s,v)$.

Chapter 2

Metric Methods

In this chapter, we will see a series of techniques that are based on the structure of the metric spaces underlying the graphs.

2.1 Multi-Cut via Metric Decomposition

Recall that s-t min-cut problem we just saw. Now, consider a more general problem called multi-min-cut.

Problem 2.1.1 (Multi-min-cut). Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and k pairs of vertices $\{(s_i, t_i)\}_{i=1}^k$, the *multi-min-cut problem* aims to find a minimum capacity cut that separates all pairs.

Multi-min-cut is NP-hard even on trees. In general, it is a NP-complete problem. Hence, we ask for an approximation algorithm instead. An $O(k)$-approximation algorithm is trivial by simply outputting the union of all s_i-t_i min-cuts. The goal is an $O(\log k)$-approximation algorithm, which also proves the multi-commodity flow-cut gap.

Note. It turns out that $O(\log k)$ is tight in general graphs. For planar graphs, one can get an $O(1)$-approximation and flow-cut gap. These results are only for undirected graphs since the situation is more complicated in directed graphs, and we will discuss that later.

Again, we can write the following linear program relaxation for the multi-min-cut problem:

$$\begin{aligned}
&\min \sum_{e\in E} c(e)x_e && && \max \sum_{i=1}^{k} \sum_{P\in\mathcal{P}_{s_i,t_i}} y_P && \\
&\sum_{e\in P} x_e \geq 1 && P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}; && \sum_{i=1}^{k} \sum_{P\ni e} y_P \leq c(e) && \forall e \in E; \qquad (2.1)\\
&\text{(P)}\quad x_e \geq 0 && \forall e \in E; && \text{(D)}\quad y_P \geq 0 && \forall P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}.
\end{aligned}$$

The primal assigns distance labels x_e to edges so that, on each path P between s_i and t_i, the distance labels of these edges on P sum up to at least one, just like the s-t min-cut.

Remark. The primal (with exponentially many constraints) is efficiently solvable.

Proof. With the ellipsoid method, we just need a separation oracle. Consider setting the length of each edge to x_e and for each pair (s_i, t_i), compute the length of the shortest path between s_i and t_i and check whether it is at least 1. This only takes k many times compared to the previous separation oracle for the s-t min-cut linear program relaxation, hence it's still polynomial time. ⊛

On the other hand, the dual variable can be interpreted as the amount of flow between s_i and t_i that is routed along the path P. This is called the *maximum throughput multi-commodity flow* problem,

where we don't care about individual demands, but only the overall flow. The dual tries to assign an amount of flow y_P to each path P so that the total flow on each edge is at most the capacity of the edge.

Note. The flow conservation constraints are automatically satisfied and the endpoints of the path P determine which kind of commodity is routed along the path.

2.1.1 Approximation via Randomized Decomposition

The first algorithm [GVY93] (see [Vaz01; WS11]) that achieved an $O(\log k)$-approximation for multi-min-cut is based on the region growing technique [LR99]. Here, we present the randomized rounding algorithm due to its future application for metric embedding [CKR05], which in particular also shows the integrality gap of the primal linear program to be $O(\log k)$. The goal is to find a procedure to cut (i.e., decompose) the graph into components that satisfies the requirement of being a multi-cut, i.e.,

- each component has diameter at most 1 when the edge capacity is induced by the primal solution;
- the probability that edge e is cut is at most αx_e,

then we will get an α-approximation algorithm. To do this, we consider the following algorithm.

Algorithm 2.1: Random Partition [CKR05]

Data: A connected graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, $\{(s_i, t_i) \mid s_i, t_i \in V\}_{i=1}^k$
Result: A multi-min-cut F

$\{x_e\}_{e\in E} \leftarrow$ `LP-Solve(Multi-Min-Cut-LP(`G, c, $\{(s_i,t_i)\}_{i=1}^k$`))` `// Solve the primal`
$\theta \leftarrow$ `Uniform(`$[0, 1/2)$`)`
$\sigma \leftarrow$ `Uniform(`$S([k])$`)` `// Random permutation from permutation group` $S([k])$
for $i = 1, \dots, k$ **do**
  $V_{\sigma(i)} \leftarrow B_x(s_{\sigma(i)}, \theta) \setminus \bigcup_{j<i} V_{\sigma(j)}$
$F \leftarrow \bigcup_{i=1}^k \delta(V_i)$
return F

Intuition. It essentially reduces to *low-diameter decomposition* in a metric space.

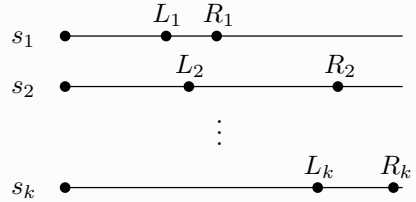
Lemma 2.1.1. Algorithm 2.1 outputs a feasible multi-cut for the given instance.

Proof. Suppose not, then there exists a pair (s_i, t_i) are still connected in $G - F$. This can only happen if s_i is "grabbed" by some terminals s_j which is proceeded before s_i, i.e., there exists some $V_j \subseteq B_x(s_j, \theta)$ that contains both s_i and t_i. However, if s_j grabs both s_i and t_i, it means the distance between s_i and t_i is at most $2\theta < 1$, a contradiction to the feasibility of $\{x_e\}_{e\in E}$. ■

Lemma 2.1.2. For any $e \in E$, $\Pr(e = uv \text{ is cut by Algorithm 2.1}) \le 2H_k x_e \le O(\log k) x_e$.[a]

[a] H_k is the k^{th} harmonic number.

Proof. Let $L_i := \min(d_x(s_i, u), d_x(d_i, v))$ and $R_i := \max(d_x(s_i, u), d_x(s_i, v))$. Without loss of generality, we can renumber s_i's such that $L_1 \le L_2 \le \cdots \le L_k$.

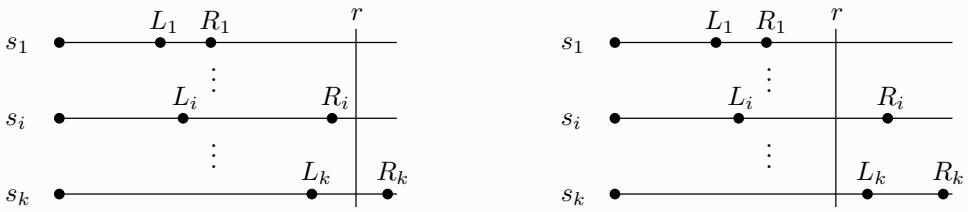


Let A_i be the event that s_i cut $e = uv$ *first*, i.e., A_i is the event that $|V_i \cap \{u, v\}| = 1$ and $|V_j \cap \{u, v\}| = 0$ for all j such that $\sigma(j) < \sigma(i)$, where $|V_i \cap \{u, v\}| = 1$ simply says that s_i *cuts* the edge e. If A_i happens, then for all j that come before i in σ, neither u nor v can be in V_j since:

- if only one of u and v is in V_j, then s_j cuts e;
- if both u and v are in V_j, s_i can't cut e as the cut set only grabs the leftover vertices (line 5).

Let A be the event that e is cut, which is the union of the disjoint events A_i's, hence $\Pr(A) = \sum_{i=1}^{k} \Pr(A_i)$. Now, for any fixed $r \in [0, 1/2)$, we see that

- $r \notin [L_i, R_i)$: This is easy to understand as if $r \notin [L_i, R_i)$, s_i is impossible to cut e;
- $r \in [L_i, R_i)$: Consider some $j < i$ and suppose j comes before i in the permutation (i.e., $\sigma(j) < \sigma(i)$). Since $j < i$, $L_j \le L_i \le r$. Hence, at least one of u and v is inside the ball of radius r centered at s_j. Consequently, s_i can't be the first to cut e, resulting in the fact that s_i is the first to cut the edge e if $\sigma(i) < \sigma(j)$ for all $j < i$.



Since σ is a random permutation, i appears before j for all $j < i$ with probability $1/i$. Hence,

$$\begin{cases} \Pr(A_i \mid \theta = r) = 0, & \text{if } \theta \notin [L_i, R_i); \\ \Pr(A_i \mid \theta = r) \le 1/i, & \text{if } \theta \in [L_i, R_i). \end{cases}$$

As θ is independent of σ, we have

$$\Pr(A_i) \le \frac{1}{i} \cdot \mathbb{P}(\theta \in [L_i, R_i)) = \frac{2}{i}(R_i - L_i) \le \frac{2x_e}{i}$$

from the triangle inequality $R_i \le L_i + x_e$. This finally leads to

$$\Pr(A) = \sum_{i=1}^{k} \Pr(A_i) \le \sum_{i=1}^{k} \frac{2x_e}{i} \le 2H_k x_e,$$

and we conclude the theorem by noting that $H_k = O(\log k)$. ■

Theorem 2.1.1. Algorithm 2.1 is an $O(\log k)$-approximation (in expectation) algorithm for the multi-min-cut problem. Furthermore, the integrality gap of the multi-min-cut linear program is $O(\log k)$.

Proof. Let F be the set of edges outputted by Algorithm 2.1. For each edge e, let $\xi_e = \mathbb{1}_{e \in F}$ in an indicator random variable. Hence, we have $\mathbb{E}[\xi_e] = \mathbb{P}(\xi_e = 1) \le 2H_k x_e$ from Lemma 2.1.2. This leads to

$$\mathbb{E}[c(F)] := \mathbb{E}\left[\sum_{e \in F} c(e)\right] = \mathbb{E}\left[\sum_{e \in E} c(e)\xi_e\right] = \sum_{e \in E} c(e) \Pr(\xi_e) \le 2H_k \sum_{e \in E} c(e)x_e = 2H_k \, \mathsf{OPT}_{\mathrm{LP}}$$

where $\mathsf{OPT}_{\mathrm{LP}}$ is the optimal value of the linear program. Since $\mathsf{OPT}_{\mathrm{LP}} \le \mathsf{OPT}$ where OPT is the optimum value of the multi-min-cut problem, we have

$$\mathbb{E}[c(F)] \le 2H_k \, \mathsf{OPT}_{\mathrm{LP}} \le 2H_k \, \mathsf{OPT} = O(\log k) \, \mathsf{OPT}\,.$$

This also implies that there exists a set of edges F such that the total capacity of edges in F is at most $2H_k \, \mathsf{OPT}_{\mathrm{LP}}$, i.e., $\mathsf{OPT}_{\mathrm{LP}} \le \mathsf{OPT} \le 2H_k \, \mathsf{OPT}_{\mathrm{LP}}$, which proves the integrality gap result. ■

The expected cost analysis can be used to obtain a randomized algorithm via repetition that outputs an $O(\log k)$-approximation with high probability. The algorithm can also be derandomized, but it's not straight forward.

Remark (Flow-cut gap). Recall that when $k = 1$, we have the max-flow min-cut theorem. The integrality gap of the standard linear program for multi-min-cut is the same as the relative gap between flow and cut when k is arbitrary. The upper bound on the integrality gap gives an upper bound on the gap.

Lecture 6: Low-Diameter Decomposition and Tree Embeddings

2.1.2 Low-Diameter Decomposition

12 Sep. 11:00

Given a graph $G = (V, E)$ and edge length $\ell \colon E \to \mathbb{R}_+$, which define a metric space (V, d) where $d(u, v)$ is the shortest path distances between u and v in G as before. As we have seen, a useful notion is to decompose or partition the graph into subgraphs (or clusters) of small diameter. More precisely, given a graph $G = (V, E)$, we would like to partition V into clusters with vertex sets $\{V_i\}_{i=1}^h$ such that each V_i has diameter at most some given parameter δ.

Example (Singleton). It's trivial to consider the singleton partition, where $V_i = \{v\}$ for all $v \in V$.

However, the goal in partitioning is to ensure that two vertices $u, v \in V$ that are close to each other, say $d(u, v) < \delta$, should ideally not be split apart into different clusters. However, as the graph (or metric space) is connected, it's impossible to do this deterministically.

Example (Line graph). Considered a line graph L_n. The most natural randomized algorithm is to shift the line by $\theta \in [0, \delta)$, and then we separate the line graph by δ-length clusters. In this way, the probability that any pair $u, v \in V$ is cut is at most $d(u, v)/\delta$.

This is the best we can hope for, i.e., u, v are separated only with probability proportional to $d(u, v)/\delta$.

Definition 2.1.1 (Low-diameter decomposition). Let $G = (V, E)$ be a graph with edge lengths $\ell \colon E \to \mathbb{R}_+$, which induces a distance metric $d \colon V \times V \to \mathbb{R}_+$. Let Δ be the diameter of the metric space (V, d). For a given $\delta \in [0, \Delta]$, a *low-diameter decomposition* with cutting probability parameter α is a probability distribution $\mathcal{D}$ over the set $\mathcal{P}$ of all partitions of V such that

- for any partition $P = (V_1, \ldots, V_h) \in \mathcal{P}$ in $\operatorname{supp}(\mathcal{D})$, $\operatorname{diam}((V_i, d)) \le \delta$;
- for all $u, v \in V$, $\Pr(u, v \text{ are separated}) \le \alpha d(u, v)/\delta$.

Given a partition $P \in \mathcal{P}$, we write E_P be the set of edges (pairs) that are separated by P. Hence, in Definition 2.1.1, the second point is equivalent to $\Pr((u, v) \in E_P) \le \alpha d(u, v)/\delta$.

Definition 2.1.2 (Low-diameter decomposition scheme). A *low-diameter decomposition scheme* with parameter α is a family of algorithms that given any $\delta \in [0, \Delta)$, generates a low-diameter decomposition with cutting (separation) probability parameter at most α.

Notation (Strong v.s. weak diameter guarantee). A low-diameter decomposition is said to have the *strong diameter* guarantee if the diameter of each cluster V_i in the induced graphs $G[V_i]$ is at most δ. Note that Definition 2.1.2 does not require that because it is based on the metric closure (V, d) of the given graph. The standard definition is called the *weak diameter* guarantee. Some applications require the strong diameter guarantee. We will, by default, work with weak diameter guarantee and mention strong diameter guarantee when needed.

Note (Padded decomposition). Definition 2.1.1 is often strengthened to require more. Given a point u, let $B_d(u, r) = \{v \in V \mid d(u, v) \le r\}$. In *padded decomposition*, we require that for each u, and for each $r \le \delta$, the probability of $B_d(u, r)$ being contained in the same part is at least $e^{-\beta r}$.

Remark (Sparse cover). A *sparse cover* consists of several clusters $\{V_i\}_{i=1}^{h}$. Each cluster should have weak/strong diameter at most δ. For each u, v with distance at most δ, there must be some cluster V_i that contains both u and v, and no vertex u must be in more than some number s of clusters. The techniques underlying sparse covers and low-diameter decomposition are related though we will mostly work only with the latter.

The main question here for the low-diameter decomposition is the smallest α that one can obtain. It turns out that for general metric spaces, $\alpha = O(\log n)$ is a tight bound for both strong and weak diameter guarantee [Bar96]. For planar graph metrics, $\alpha = O(1)$ is achievable, where weak diameter guarantee was shown in [KPR93], and the strong diameter guarantee was more difficult and was shown later. See [Fil24] for some recent work and pointers to literature on these topics.

Now, we present the algorithm for weak diameter guarantee. Let (V, d) be a metric space with $|V| = n$. Borrowing ideas from Algorithm 2.1, we see that implicitly this is a metric partitioning scheme. We simply modify the algorithm to ensure that the weak diameter of each cluster is at most a given parameter δ.

Algorithm 2.2: Random Partition [CKR05]

Data: A metric space (V, d), δ
Result: E_P for the partition P

$\theta \leftarrow$ `Uniform`$([0, \delta/2))$
$\sigma \leftarrow$ `Uniform`$(S(V))$ `// Random permutation from permutation group` $S(V)$
for $i = 1, \ldots, n$ **do**
  $V_{\sigma(i)} \leftarrow B_d(v_{\sigma(i)}, \theta) \setminus \bigcup_{j<i} V_{\sigma(j)}$
return $\bigcup_{i=1}^{n} \delta(V_i)$

From the same analysis as in Lemma 2.1.2, claim the following.

Claim. Algorithm 2.2 correctly outputs a partition of V into clusters, each of which has weak diameter at most δ.

Furthermore, the probability guarantee can also be stated.

Theorem 2.1.2. The probability that u and v are in different clusters outputted by Algorithm 2.2 is at most $2H_n d(u, v)/\delta$, i.e., $\alpha = H_k = O(\log n)$.

Proof. We see that $\Pr(A_j) \le 2d(u, v)/\delta \cdot 1/j$, hence $\Pr(A) \le 2H_n d(u, v)/\delta$. ■

Finally, we consider a not so apparent modification of Algorithm 2.2: we sample θ from $[\delta/4, \delta/2)$ instead of $[0, \delta/2)$.

Algorithm 2.3: Refined Random Partition [CKR05]

Data: A metric space (V, d), δ
Result: E_P for the partition P

$\theta \leftarrow$ `Uniform`$([\delta/4, \delta/2))$
$\sigma \leftarrow$ `Uniform`$(S(V))$ `// Random permutation from permutation group` $S(V)$
for $i = 1, \ldots, n$ **do**
  $V_{\sigma(i)} \leftarrow B_d(v_{\sigma(i)}, \theta) \setminus \bigcup_{j<i} V_{\sigma(j)}$
return $\bigcup_{i=1}^{n} \delta(V_i)$

Intuition. Intuitively, this will preserve closer points.

It's clear that the guarantee about the diameter remains the same.

Claim. Algorithm 2.3 correctly outputs a partition of V into clusters, each of which has weak diameter at most δ.

The main difference is in the probability guarantee which is refinement of the previous bound.

Theorem 2.1.3. The probability that u and v are in different clusters outputted by Algorithm 2.3 is at most $\frac{4d(u,v)}{\delta}\log\frac{|B(u,\delta/2)|}{|B(u,\delta/8)|}$, i.e., $\alpha = \alpha(\delta) = 4\log\frac{|B(u,\delta/2)|}{|B(u,\delta/8)|}$.

Proof. We sketch the proof based on the proof of Lemma 2.1.2. Assuming the exact same notation, and we fix $u, v \in V$ and think of V as $v_1, \ldots, v_n$. If $d(u,v) \geq \delta/8$, then the edge is going to get cut with constant probability, and the bound is not giving anything interesting, so we are primarily interested in the case when $d(u,v) < \delta/8$.

We consider the event A_i which is that v_i is the first vertex to separate the pair u, v. We can argue as before that $\Pr(A_i) \leq 1/i \cdot \Pr(\theta \in [L_i, R_i])$, and this is at most $4d(u,v)/\delta i$ since we're choosing the radius from $[\delta/4, \delta/2)$. The new twist is that since we choose $\theta \in [\delta/4, \delta/2)$ and $d(u,v) < \delta/8$, no vertex $v_j \in B(u, \delta/8)$ can separate u, v because $L_j \leq d(v_j, u) < \delta/8$ and $R_j \leq L_j + d(u, v_j) \leq \delta/8 + \delta/8 = \delta/4$. Any such vertex will capture both u, v if they are not already separated. Similarly, any vertex $v_j \notin B(u,\delta)$ can cut the pair because $L_j \geq \delta - d(u,v) \geq \delta - \delta/8 \geq \delta/2$. Therefore, if A is the event of u, v being cut, then

$$\Pr(A) \leq \sum_{j \in B(u,\delta)\setminus B(u,\delta/8)} \Pr(A_j) \leq \frac{4d(u,v)}{\delta} \sum_{|B(u,\delta/8)| < j \leq |B(u,\delta)|} \frac{1}{j} \leq \frac{4d(u,v)}{\delta}\log\frac{|B(u,\delta)|}{|B(u,\delta/8)|},$$

proving the result. ■

2.2 Dominating Tree Metrics Embedding

Using tree representations of graphs is a powerful tool in algorithm design. Here, we are interested in representing the distances in an undirected graph via distances in a spanning tree. Let $G = (V, E)$ be a graph with edge length $\ell \colon E \to \mathbb{R}_+$, which induces a metric space (V, d) via shortest path distances. The main question is the following:

Problem 2.2.1 (Tree embedding). Given a graph $G = (V, E)$ with edge length $\ell \colon E \to \mathbb{R}_+$, the *tree embedding* problem aims to find a spanning tree $T = (V, E_T)$ of G such that for any $u, v \in V$, $d_T(u,v) \leq \alpha d_G(u,v)$ where α is called the *distortion* or *stretch*.[a]

[a] Clearly, $d_T(u,v) \geq d_G(u,v)$.

Example (Cycle). Consider a cycle C_n. We see that for a fixed edge (u,v), there exists one spanning tree T such that $d_T(u,v) = n - 1$.

Motivated by applications of spanning tree based metric approximations, we observe that if we are allowed to pick a probability distribution over spanning trees, then the expected distance for any pair of vertices can be much better than the above worst-case example.

Example (Cycle). Again, consider a cycle C_n. If we allow randomization (picking trees randomly),

$$\mathbb{E}[d_T(u,v)] = \frac{n-1}{n} \cdot 1 + \frac{1}{n} \cdot (n-1) \leq 2.$$

It's showed that [Alo+95] for any weighted graph $G = (V, E)$, there is a distribution $\mathcal{D}$ over spanning trees of G such that for any $u, v \in V$,

$$\mathbb{E}_{T \sim \mathcal{D}}[d_T(u,v)] \leq \exp\left(\sqrt{\log n \log\log n}\right) \cdot d_G(u,v) < n^{o(1)} d_G(u,v).$$

Intuition. This is a probabilistic approximation of a graph metric by spanning tree metrics.

This can also be viewed as a metric *embedding* result. In keeping with metric embedding terminology, we're interested in the worst-case guarantee of how much the expected distance for any pair increases, i.e., minimizing α. In the above example $\alpha \leq \exp\left(\sqrt{\log n \log\log n}\right)$.

Note (Lower bound). A lower bound of $\alpha = \Omega(\log n)$ is required for probabilistic tree approximation, and it's conjectured that this is tight [Alo+95].

2.2.1 Dominating Tree Metric

It turned out that this is quite difficult to obtain even a poly-logarithmic bound [Elk+05], and currently the best known bound is $O(\log n \log\log n)$ [ABN08]. To make the problem easier, [Bar96] proposed to forget about the graph topology and just focus on (V, d), i.e., consider the *metric embeddings* instead of spanning tree embeddings. More generally, we work with the metric completion (V, d) and view it as a complete graph on V, where any spanning tree of the complete graph is now allowed. Moreover, we allow additional vertices. This is formalized as follows.

Definition 2.2.1 (Dominating tree metric). A tree $T = (V_T, E_T)$ with edge length $\ell_T \colon E_T \to \mathbb{R}_+$ is a *dominating tree metric* for a finite metric space (V, d) if $V \subseteq V_T$ and for all $u, v \in V$, $d_T(u, v) \geq d_G(u, v)$.

Then, we're interested in approximating metrics probabilistically by dominating tree metrics:

Definition 2.2.2 (Probabilistic approximation). A *probabilistic approximation* of a metric space (V, d) by dominating tree metrics is a probability distribution $\mathcal{D}$ over a collection of trees $\{T_i\}_{i=1}^h$ if each T_i is a dominating tree metric for (V, d).

Furthermore, we say that the probabilistic approximation $\mathcal{D}$ has *stretch* α if for all $u, v \in V$,

$$\mathbb{E}_{T\sim\mathcal{D}}[d_T(u, v)] \leq \alpha d(u, v).$$

2.2.2 Tree Embedding with Low-Diameter Decomposition

One can use low-diameter decomposition to efficiently sample from a distribution that has stretch $\alpha = O(\log^2 n)$ [Bar96], and this was subsequently improved to $O(\log n \log\log n)$ [Bar98], and finally improved to the optimal $O(\log n)$ [FRT03] using Algorithm 2.3.

Intuition. Recursively decompose (V, d) using low-diameter decomposition.

Specifically, let (V, d) be a metric space with diameter Δ. We use low-diameter decomposition with parameter $\delta = \Delta/2$ to randomly partition V into clusters $\{V_i\}_{i=1}^h$ of diameter at most $\Delta/2$, then we recursively find a tree for each of the V_i, with root r_i. We create a new dummy root r and connect each r_i to r with an edge of length Δ.

Algorithm 2.4: Tree Embedding

Data: A metric space (V, d), diameter D
Result: A rooted tree (T, r)

if $|V| = 1$ **then**
  $T \leftarrow (V, \varnothing)$
  return (T, v) // $V = \{v\}$

Create a tree T with root r
$\{V_i\}_{i=1}^h \leftarrow$ `Low-Diameter-Decomposition`$((V, d),\ D/2)$
for $j = 1, \ldots, h$ **do**
  $(T_j, r_j) \leftarrow$ `Tree-Embedding`$((V_j, d),\ D/2)$
  Connect T_j to T by adding edge (r, r_j) of length D
return (T, r)

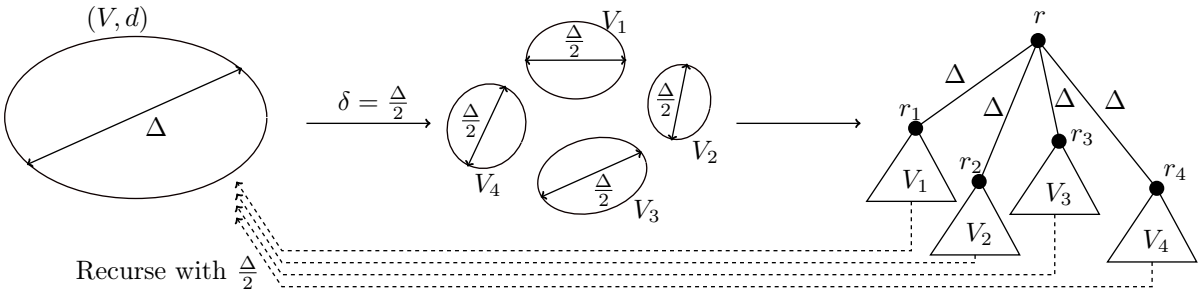


Figure 2.1: Illustration of Algorithm 2.4.

Notation. We say $\delta = \Delta/2^i$ at level i to make the analysis cleaner.

We will now assume that the minimum distance is at least 1 by scaling, and let Δ be the diameter of the metric space with this assumption.

Remark. If the minimum distance is at least 1, Algorithm 2.4 yields a stretch of $O(\log n \log \Delta)$.

Theorem 2.2.1. Let Δ be the diameter of (V, d). Algorithm 2.4 outputs a random dominating tree metric $T = (V_T, E_T)$ with length ℓ_T such that for each $u, v \in V$, $\mathbb{E}_{T\sim\mathcal{D}}[d_T(u, v)] \leq O(\alpha \log \Delta) d(u, v)$ where α is the cutting probability of the low-diameter decomposition algorithm used.

Proof. We prove this by induction. It's clear that the base case is trivial. If we start with $D = \Delta$, then at depth i of the recursion, the parameter is $\Delta/2^{i-1}$, and it is the upper-bound on the diameter of the metric space in that recursive call. We note the following claims.

Claim. The length of the root to leaf path of a tree created at level i of the recursion is at most $\sum_{j\geq i} \Delta/2^{j-1} \leq 2\Delta/2^{i-1}$.

Suppose u and v are first separated at level i of the recursion. Then, $d_T(u, v) \leq 4\Delta/2^{i-1}$ from the above claim. We see that if u and v are separated in the first level of the recursion due to the low-diameter decomposition algorithm, its probability is at most $\alpha d(u, v)/(\Delta/2) \leq 2\alpha d(u, v)/\Delta$, in which case their distance in the tree is at most 4Δ. Otherwise, they are in the same cluster, and we can apply induction. Note that u and v are definitely separated by level t where t is the smallest integer such that $\Delta/2^{t+1} < d(u, v)$. Hence, the depth of the recursion is at most $1 + \lceil \log \Delta \rceil \leq 2 \log \Delta$. It's easy to unroll the induction and use the preceding claim to obtain

$$\mathbb{E}_{T\sim\mathcal{D}}[d_T(u, v)] \leq \sum_{i=0}^{t+1} 2\alpha \frac{d(u, v)}{(\Delta/2^{i-1})} \cdot \left(4\frac{\Delta}{2^{i-1}}\right) \leq O(\alpha \log \Delta) d(u, v) \tag{2.2}$$

since the depth of the recursion is $O(\log \Delta)$. ■

If we use Algorithm 2.2 as the low-diameter decomposition algorithm, then we have $\alpha = O(\log n)$. From Theorem 2.2.1, we see that the tree may require depth $\log \Delta$ to provide a good approximation, and in general $\log \Delta$ can be as large as n, so we get an $O(n \log n)$ approximation.

Example. Consider the metric induced by a path with n edges and edge lengths are 2^i for all $i = 1, \ldots, n$. In such cases, the dependence of the stretch on $\log \Delta$ is undesirable.

One can alter Algorithm 2.4 to make the stretch bound $O(\log^2 n)$: in applying the low-diameter decomposition algorithm with parameter δ, we ensure that any pair u, v such that $d(u, v) \leq \delta/n^2$ is not cut during the procedure. We can do this by contracting all such pairs without changing the diameter of the resulting metric space too much. This will ensure that in the tree construction process, a pair u, v participates in only $O(\log n)$ levels and hence the expected stretch can be bounded by $O(\alpha \log n)$.

Remark (Hierarchically well-separated tree)**.** The trees constructed by Algorithm 2.4 have an additional strong property: the edge lengths at each level are the same and the length from the root to the leaf go down by a factor of 2 at each level. A tree metric with such a property is called *hierarchically well-separated* tree metric and this additional property can be exploited in algorithms and comes for free in the construction.

Finally, we note that if we choose Algorithm 2.3 as the low-diameter decomposition algorithm specifically, the expected stretch is actually $O(\log n)$, which is optimal [FRT03].

Theorem 2.2.2. Let Δ be the diameter of (V, d). When Algorithm 2.3 is used as the low-diameter decomposition algorithm in Algorithm 2.4, Algorithm 2.4 outputs a random dominating tree metric $T = (V_T, E_T)$ with length ℓ_T such that for each $u, v \in V$, $\mathbb{E}_{T\sim\mathcal{D}}[d_T(u, v)] \leq O(\log n)d(u, v)$.

Proof. Firstly, we recall that from Theorem 2.1.3, the probability that u and v are in different clusters outputted by Algorithm 2.3 is at most $\frac{4d(u,v)}{\delta} \log \frac{|B(u,\delta/2)|}{|B(u,\delta/8)|}$. Thus, the guarantee $\alpha(\delta)$ from the low-diameter decomposition algorithm is no longer uniform but depends on the diameter. Plugging this to Equation 2.2, we have

$$\begin{aligned}\mathbb{E}_{T\sim\mathcal{D}}[d_T(u, v)] &\leq \sum_{i=0}^{t+1} 2 \log \frac{|B(u, \Delta/2^i)|}{|B(u, \Delta/2^{i+3})|} \frac{d(u, v)}{\Delta/2^{i-1}} \cdot \left(4\frac{\Delta}{2^{i-1}}\right) \\ &\leq 8d(u, v) \left(\log|B(u, \Delta/2)| + \log|B(u, \Delta/4)| + \log|B(u, \Delta/8)|\right) \leq O(\log n)d(u, v),\end{aligned}$$

proving the desired result. ■

Remark (Lower bound)**.** $O(\log n)$ bound for low-diameter decomposition algorithm and tree embeddings are near optimal (modulo precise constant factors).

Proof. We can use the existence of low-girth graphs which are closely tied to expanders in a direct fashion. Another way is via indirection. Previously, in the lecture note, we saw that the integrality gap of the linear program relaxation for multi-min-cut is $\Omega(\log k)$ (the upper-bound is proved in Theorem 2.1.1 via a low-diameter decomposition algorithm). In fact, if α is the factor for the low-diameter decomposition algorithm, then we get an $O(\alpha)$-approximation for multi-min-cut via the linear program. Thus, one see that $\alpha = \Omega(\log n)$ for general metrics (in the integrality gap example for multi-min-cut $k = \Omega(n^2)$). One can use similar approaches to prove for tree embedding. ⊛

Note (Efficient algorithms)**.** While we focus on the quality of low-diameter decomposition algorithms and tree embedding but not so much on the running times, it is easy to see that the algorithms themselves can be implemented in polynomial time. The main computation is about the shortest paths. If one computes all pairs shortest paths (APSP), then the algorithms are pretty simple. However, APSP is slow, which takes $O(mn)$ times. It is possible to compute the low-diameter decomposition algorithm and metric tree embeddings in close to linear time on a weighted graph with m edges. This involves computing approximate shortest paths and several tricks, and sometimes we give up on the quality of the approximation by logarithmic factors.

Lecture 7: Linear Programming for Sparsest Cut

2.3 Sparsest Cut

17 Sep. 11:00

Consider building a network of n vertices. The best network might be the complete graph, which can do everything and is robust. However, the problem is that the degree is too high $(n - 1)$.

Example. For degree equal to 2, the best we can hope for is a cycle.

The magic happens whenever the degree goes up to 3.

Intuition. If we can down-weight edges in a complete graph K_n by $3/(n-1)$, then any cut S has

$$\delta(S) = |S| \cdot |V \setminus S| \cdot \frac{3}{n-1} \approx c|S|$$

for $|S| \ll |V \setminus S|$ and some $c \geq 0$.

This notion can be formalized as the so-called expander [HLW06]. We postpone the formal introduction of expander, and first focus on a closely related problem, the sparsest cut problem, specifically, the non-uniform version. It turns out that solving this helps us answer various questions for expanders.

2.3.1 Uniform and Non-Uniform Sparsest Cut Problem

To introduce the problem, we first define the sparsity for a cut.

Definition 2.3.1 (Sparsity). Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and a demand graph $H = (V, F)$ with demand capacity $D\colon F \to \mathbb{R}_+$, for any cut $S \subseteq V$, its *sparsity* is defined as

$$\frac{c(\delta(S))}{\sum_{i\colon |S \cap \{s_i, t_i\}|=1} D_i}.$$

That is, the sparsity of a cut is the ratio of the capacity of the cut and the total demand of the pairs separated by S. Now, we can introduce the non-uniform sparsest cut problem.

Problem 2.3.1 (Non-uniform sparsest cut). Given a supply graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and k pairs of vertices $\{(s_i, t_i)\}_{i=1}^k$ along with non-negative demand values $D_1, \ldots, D_k$.[a] The *non-uniform sparsest cut problem* aims to find a cut S with minimum sparsity.

[a] If G is undirected, then the demand pairs are unordered, i.e., we do not distinguish (s_i, t_i) from (t_i, s_i).

Here, demands forms a demand graph $H = (V, F)$ with edges (s_i, t_i) and demand capacity $D\colon F \to \mathbb{R}_+$ such that $D((s_i, t_i)) = D_i$. With this representation, the sparsity of cut S is simply $c(\delta_G(S))/D(\delta_H(S))$ where $\delta_G(S)$ (respectively, $\delta_H(S)$) represents the supply (respectively, demand) edges crossing S.

Intuition. We're trying to find the best "bang per buck" cut, i.e., how much capacity do we need to remove per amount of demand separated to satisfy the demand?

Remark. We can define a cut as removing a set of edges, leading to more than two components. In the case of sparsest cut in undirected graphs, it suffices to restrict attention to cuts of the form $\delta(S)$ for some $S \subseteq V$. This is not necessarily true for directed graphs or even in undirected graphs with node-weights or in hypergraphs.

Problem 2.3.2 ((Uniform) sparsest cut). The *sparsest cut* problem is the same as Problem 2.3.1 with all $D(u, v) = 1$ for each unordered pair of vertices (u, v).[a]

[a] That is, $\{(s_i, t_i)\}_{i=1}^k$ is the set of all unordered pairs of vertices.

We see that in the uniform sparsest cut problem, the sparsity of a cut S is given by $c(\delta_G(S))/|S||V \setminus S|$. In this case, the demand graph H is a complete graph with unit demand values on each edge.

Finally, to further motivate the problem, we see that the uniform sparsest cut helps us directly and indirectly solve the graph bisection, a central problem is graph algorithm:

Problem 2.3.3 (Graph bisection). Given a graph $G = (V, E)$, the *graph bisection problem* aims to find a partition of G into $(S, V \setminus S)$ such that $|S| = |V \setminus S|$ while minimizing $|\delta(S)|$.

2.3.2 Linear Program Relaxation and Maximum Concurrent Flow

It's not clear how to start solving the uniform sparsest cut problem as writing a linear program relaxation for which is not obvious, compared to multi-min-cut and other cut problems where we have explicit terminal pairs that we wish to separate. Hence, to write an integer program for which, we let $y_i \in \{0,1\}$ being the indicator variable of whether we want to separate (s_i, t_i). Moreover, let $x_e \in \{0,1\}$ for all $e \in E$ to be the cut indicator variables.

> **Intuition.** If we decide to separate (s_i, t_i), then for every path between s_i and t_i we should cut at least one edge on the path.

Hence, a natural integer program of non-uniform sparsest cut and its relaxation is given by:

$$\begin{aligned}
&\min \frac{\sum_{e\in E} c(e)x_e}{\sum_{i=1}^{k} D_i y_i} && && \min \sum_{e\in E} c(e)x_e && \\
&\sum_{e\in P} x_e \geq y_i && \forall P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}; \quad \to && \sum_{i=1}^{k} D_i y_i = 1 && \\
&x_e \in \{0,1\} && \forall e \in E; && \sum_{e\in P} x_e \geq y_i && \forall P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}; \\
&y_i \in \{0,1\} && \forall i = 1,\dots,k; && x_e \geq 0 && \forall e \in E; \\
& && && \text{(P)} \quad y_i \geq 0 && \forall i = 1,\dots,k,
\end{aligned} \tag{2.3}$$

where we use the standard trick, i.e., linearization to normalize the denominator to be 1, to make the ratio of the integer program into a linear program. With the dual variable z_P for each path such that it indicates the amount of "flow" sent on the path P, the dual of the above is

$$\begin{aligned}
&\max \lambda && \\
&\sum_{P\in\mathcal{P}_{s_i,t_i}} z_P \geq \lambda D_i && \forall i = 1,\dots,k; \\
&\sum_{i=1}^{k} \sum_{\substack{P\in\mathcal{P}_{s_i,t_i} \\ P\ni e}} z_P \leq c(e) && \forall e \in E; \\
&z_P \geq 0 && \forall P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}; \\
&\text{(D)} \quad \lambda \geq 0, &&
\end{aligned} \tag{2.4}$$

which is a multi-commodity flow. In particular, it solves the *maximum concurrent multi-commodity flow* problem for the given instance, i.e., it finds the largest value of λ such that there is a feasible multi-commodity flow for the given pairs in which the flow routed for pair (s_i, t_i) is at least λD_i.

> **Notation** (Concurrent flow). It is called *concurrent flow* since we need to route all demand pairs to the same factor which is in contrast to the dual of multi-min-cut, which corresponds to the maximum throughput multi-commodity flow.[a]
>
> [a] Recall that in this case, some pairs may have zero flow while others have a lot of flow.

We note that this dual can be solved efficiently via ellipsoid method since the separation oracle is just the shortest path problem. One can also write a compact linear program via distance variables as

$$\begin{aligned}
&\min \sum_{uv\in E} c(uv)d(uv) \\
&\sum_{i=1}^{k} D_i d(s_i,t_i) = 1 \\
&d \text{ is a metric on } V.
\end{aligned}$$

In this context, we can understand the *flow-cut gap* of non-uniform sparsest cut by the following equivalent way of thinking about the problem:

Intuition (Cut-condition). Given a multi-commodify flow instance on G. A necessary condition to route all the demand pairs is that if G satisfies the *cut-condition*, i.e., for every $S \subseteq V$, the capacity $c(\delta(S))$ is at least the demand separated by S. However, the converse it not necessarily true, i.e., the cut-condition is not sufficient.

The cut-condition is sufficient when $k = 1$ but is not true in general even for $k = 3$ in undirected graphs. The question is the maximum value of λ such that we can route λD_i for each pair i. The worst-case integrality gap of the proceeding linear program is precisely the flow-cut gap.

2.3.3 Rounding Linear Program via ℓ_1 Embeddings

It's known that there is an $O(\log n)$-approximation algorithm and the flow-cut gap for uniform sparsest cut, together with a lower bound of $\Omega(\log n)$ on the flow-cut gap for uniform sparsest cut via expanders [LR99]. This leads to an $O(\log^2 n)$-approximation for non-uniform sparsest cut, and it was an open problem to obtain a tight conjectured bound of $O(\log n)$. It turns out to be possible via the optimal rounding algorithm for the linear program relaxation. This goes via metric embedding theory [LLR95; AR98], hence we need some basics in metric embeddings to point out the connection and rounding.

Note. Even though the metric embedding machinery is powerful, it can seem like magic. The more basic ideas for uniform sparsest cut based on region growing is useful to know [WS11].

Remark (Rounding via multi-min-cut). There are also close connections between sparsest cut and multi-min-cut. In particular, suppose there is an $\alpha(k, n)$-approximation for non-uniform sparsest cut, then we have an $O(\alpha(k, n) \ln k)$-approximation for multi-min-cut. The converse is also true.[a]

[a] See the note for a reference.

Before we start, consider the following simple setting when G is a tree $T = (V, E)$.

Example (Tree). Given a tree $G = T = (V, E)$, for each edge $e \in T$, we can associate a cut S_e which is one side of the two components in $T - e$. The capacity of the cut $\delta(S_e)$ is $c(e)$. Let $D(e) = \sum_{i:\ |S_e \cup \{s_i, t_i\}|=1} D_i$ be the demand separated by e. The sparsity of the cut S_e is simply $c(e)/D_e$, hence finding the sparsest cut in this case is easy. Interestingly, the linear program relaxation give an optimum solution on a tree.

Proof. Let (x, y) be a feasible solution to the dual of the linear program relaxation with objective value λ. We want to prove that if G is a tree T, then there is an edge $e \in T$ such that $c_e/D_e \leq \lambda$. We note that by considering the compact linear program with distance variables, we have

$$\lambda = \frac{\sum_{e \in E} c(e) x_e}{\sum_{i=1}^{k} D_i d_x(s_i, t_i)}$$

where $d_x(s_i, t_i)$ is the shortest path distance between s_i and t_i induced by x. Since there is a unique path P_{s_i,t_i} from s_i to t_i in T such that $d_x(s_i, t_i) = \sum_{e \in P_{s_i,t_i}} x_e$, we have

$$\lambda = \frac{\sum_{e \in E} c(e) x_e}{\sum_{i=1}^{k} D_i d_x(s_i, t_i)} = \frac{\sum_{e \in E} c(e) x_e}{\sum_{i=1}^{k} D_i \sum_{e \in P_{s_i,t_i}} x_e} = \frac{\sum_{e \in E} c(e) x_e}{\sum_{e \in E} x_e \sum_{i:\ e \in P_{s_i,t_i}} D_i} = \frac{\sum_{e \in E} c(e) x_e}{\sum_{e \in E} D(e) x_e}.$$

Finally, the result follows from $\sum_i a_i / \sum_i b_i \geq \min_i a_i / b_i$ for positive a_i and b_i's. ⊛

Example (Ring). For a ring graph, the same technique works where we need to remove two edges.

The reason why the above proof works for trees is because of a more general phenomenon: the shortest path distances are ℓ_1 metrics, or equivalently, cut metrics.

Cut, Line, and ℓ_1 Metrics, and Metric Embedding

To explain the above phenomenon, consider a finite metric space (V, d) with following metrics:

Definition 2.3.2 (Cut metric)**.** Let (V, d) be a finite metric space. The metric d is a *cut metric* if there is a set $S \subseteq V$ such that $d = d_S$, where d_S associated with the cut S is defined as

$$d_S(u, v) = \begin{cases} 1, & \text{if } |S \cap \{u, v\}| = 1; \\ 0, & \text{otherwise.} \end{cases}$$

The cut-cone consists of non-negative combination of cut metrics:

Definition 2.3.3 (Cut cone)**.** Let (V, d) be a finite metric space. The metric d is in the *cut cone* if there exist non-negative scalars y_S where $S \subseteq V$ such that for all $u, v \in V$,

$$d(u, v) = \sum_{S \subseteq V} y_S d_S(u, v).$$

Beside the cut metric, another useful metric is the line metric and the well-known ℓ_1 metric:

Definition 2.3.4 (Line metric)**.** Let (V, d) be a finite metric space. The metric d is a *line metric* if there is a mapping $f \colon V \to \mathbb{R}$ such that for all $u, v \in V$,

$$d(u, v) = |f(u) - f(v)|.$$

Definition 2.3.5 (ℓ_1 metric)**.** Let (V, d) be a finite metric space. The metric d is an ℓ_1 *metric* if there is some integer d and a mapping $f \colon V \to \mathbb{R}^d$ such that for all $u, v \in V$,

$$d(u, v) = \|f(u) - f(v)\|_1$$

It might not be too surprising that the following holds.

Lemma 2.3.1. A metric d of a metric space (V, d) is an ℓ_1 metric if and only if it is a non-negative combination of line metrics (in the cone of the line metrics).

Proof. If d is an ℓ_1 metric then each dimension corresponds to a line metric. Conversely, any non-negative combination of line metrics can be made into an ℓ_1 metric where each line metric becomes a separate dimension (scalar multiplication for a line metric is also a line metric). ■

A more interesting observation is that any cut metric d_S is a simple line metric: map all vertices in S to 0 and all vertices in $V \setminus S$ to 1. This leads to the following.

Lemma 2.3.2. A metric d is an ℓ_1 metric if and only if d is in the cut cone.

Proof. If d is in the cut cone, then it is a non-negative combination of the cut metrics, hence it is a non-negative combination of line metrics by the above observation, hence an ℓ_1 metric.

For the converse, it suffices to argue that any line metric is in the cut cone. Let $V = \{v_i\}_{i=1}^n$ and let d be a line metric on V. Without loss of generality, assume that the coordinates x_i of the points for each v_i corresponding to the line metric d are $x_1 \leq x_2 \leq \cdots \leq x_n$ on the real line. For $1 \leq i < n$, let $S_i = \{v_1, v_2, \ldots, v_i\}$. It is not hard to verify that $\sum_{i=1}^{n-1} |x_{i+1} - x_i| d_{S_i} = d$. ■

Now we have introduced all the necessary metrics we will use. Consider a finite metric space (V, d).

Claim. Any finite metric space can be viewed as one that is derived from the shortest path metric induced on a graph with some non-negative edge lengths.

If $G = (V, E)$ is a simple graph and $\ell \colon E \to \mathbb{R}_+$ are some edge-lengths, the metric induced on V depends both on the *topology* of G and the lengths as well, i.e., finite metrics can encode graph structure, hence it can be diverse. When trying to round we may want to work with simpler metric spaces.

Intuition (Embedding)**.** Embed a given metric space (V, d) into a simpler host metric space (V', d') via an embedding $f \colon V \to V'$.

Note. Even though we may be interested in finite metric spaces, the host metric space can be continuous or infinite such as the $\mathbb{R}^h$ for dimension h.

As embedding typically distorts the distances, thus, we want to find embeddings with small distortion.

Definition 2.3.6 (Distortion). Let (V, d) and (V', d') be two metric spaces and let $f\colon V \to V'$ be an embedding. The *distortion* of f is given by[a]

$$\max_{\substack{u,v\in V\\ u\neq v}} \left(\frac{d'(f(u), f(v))}{d(u,v)}, \frac{d(u,v)}{d'(f(u), f(v))} \right).$$

[a] Additive version are also explored, although they are very restrictive due to lack of scale invariance.

Additionally, we're interested in the following kind of embeddings:

Definition. Let (V, d) and (V', d') be two metric spaces and let $f\colon V \to V'$ be an embedding.

Definition 2.3.7 (Isometric embedding). The embedding f is *isometric* if for all $u, v \in V$,

$$d(u,v) = d'(f(u), f(v)).$$

Definition 2.3.8 (Contraction). The embedding f is a *contraction* if for all $u, v \in V$,

$$d(u,v) \geq d'(f(u), f(v)).$$

Definition 2.3.9 (Non-contracting). The embedding f is *non-contracting* if for all $u, v \in V$,

$$d(u,v) \leq d'(f(u), f(v)).$$

Of particular importance are embeddings of finite metric spaces into $\mathbb{R}^h$, where the distance in the host space is measured under a norm such as ℓ_p norm. The dimension h is also important in various applications but in some settings like with non-uniform sparsest cut, it is not. We assume the following:

Theorem 2.3.1 (Bourgain [Bou85; LLR95]). Any n-point finite metric space can be embedded into ℓ_2 (hence also ℓ_1) with distortion $O(\log n)$. Moreover, the embedding is a contraction and can be constructed in randomized polynomial time and embeds points into $\mathbb{R}^h$ where $h = O(\log^2 n)$.

In fact, one can obtain a refined version of Theorem 2.3.1 that is useful for non-uniform sparsest cut.

Theorem 2.3.2 (Bourgain [LLR95]). Let (V, d) be an n-point finite metric space and let $S \subseteq V$ with $|S| = k$. Then there is a randomized polynomial time algorithm to compute an embedding $f\colon V \to \mathbb{R}^{O(\log^2 n)}$ such that the embedding is a contraction[a] and for every $u, v \in S$, $\|f(u) - f(v)\|_1 \geq cd(u,v)/\log k$ for some universal constant c.

[a] I.e., $\|f(u) - f(v)\|_1 \leq d(u,v)$ for all $u, v \in V$

By utilizing Theorem 2.3.2 with the previous insight on tree, we can provide a general guarantee.

As previously seen. The integrality gap of the linear program is 1 on trees since the shortest path metric on trees is in the cut cone, i.e., ℓ_1-embeddable.

One can prove that if the shortest path metric on a graph G embeds into ℓ_1 with distortion α, then the integrality gap of the linear program is at most α. This will imply an $O(\log n)$-integrality gap via Theorem 2.3.2 since any n-point finite metric space embeds into ℓ_1 with distortion $O(\log n)$.

Lecture 8: Randomized Rounding for Sparsest Cut and Expanders

Randomized Rounding Algorithm with Metric Embeddings

19 Sep. 11:00

Now, we see how to utilize Theorem 2.3.2 to design a randomized rounding algorithm for the non-uniform sparsest cut problem. The main theorem is the following.

Theorem 2.3.3. Let $G = (V, E)$ be a graph. Suppose any finite metric induced by edge lengths on E can be embedded into ℓ_1 with distortion α, then the integrality gap of the linear program for the non-uniform sparsest cut is at most α for any instance on G.

Proof. Let (x, y) be a feasible fraction solution of the linear program relaxation, and let d be the metric induced by edge lengths given by x. Let λ be the value of the solution, i.e.,

$$\lambda = \frac{\sum_{uv \in E} c(uv) d(u, v)}{\sum_{i=1}^{k} D_i d(s_i, t_i)}.$$

Since d can be embedded into ℓ_1 with distortion at most α, and any ℓ_1 metric is in the cut cone from Lemma 2.3.2, it implies that there are scalars z_S, $S \subseteq V$, such that for all $u, v \in V$,

$$\frac{1}{\alpha} \sum_{S \subseteq V} z_S d_S(u, v) \le d(u, v) \le \sum_{S \subseteq V} z_S d_S(u, v).$$

Without loss of generality, we assume that the embedding is a contraction. Then, we have

$$\begin{aligned} \lambda = \frac{\sum_{uv \in E} c(uv) d(u, v)}{\sum_{i=1}^{k} D_i d(s_i, t_i)} &\ge \frac{1}{\alpha} \frac{\sum_{uv \in E} c(uv) \sum_{S \subseteq V} z_S d_S(u, v)}{\sum_{i=1}^{k} D_i \sum_{S \subseteq V} d_S(s_i, t_i)} \\ &= \frac{1}{\alpha} \frac{\sum_{S \subseteq V} z_S c(\delta_G(S))}{\sum_{S \subseteq V} z_S D(\delta_H(S))} \ge \frac{1}{\alpha} \min_{S \subseteq V} \frac{c(\delta_G(S))}{D(\delta_H(S))}. \end{aligned}$$

Hence, there is a cut whose sparsity is at most $\alpha\lambda$. ■

Theorem 2.3.3 shows that one of the cuts with $z_S > 0$ has sparsity at most $\alpha\lambda$. Now, suppose we have an ℓ_1 embedding into h-dimensions, i.e., $\mathbb{R}^h$. Observe the following.

Intuition. First, each dimension in $\mathbb{R}^h$ corresponds to a line embedding, and each line embedding is in the cut cone with only $n - 1$ cuts used to express it (recall Lemma 2.3.2). Thus, given an ℓ_1 embedding into $\mathbb{R}^h$ with distortion α, we only need to try $d(n - 1)$ cuts and one of them will be guaranteed to have sparsity at most $\alpha\lambda$.

Algorithm 2.5 exploits this intuition.

Algorithm 2.5: Non-Uniform Sparsest Cut via Embedding

Data: A supply graph $G = (V, E)$ with edge capacity $c \colon E \to \mathbb{R}_+$, demand graph $H = (V, F)$ with demand capacity $D \colon F \to \mathbb{R}_+$
Result: The sparsest cut $S \subseteq V$

```
(\{x_e\}_{e∈E}, \{y_i\}_{i=1}^k) ← LP-Solve(Non-Uniform-Sparsest-Cut-LP(G, c, H, D))
f ← Bourgain-Embedding((V, d_x))                    // f: V → ℝ^h, h = O(log² n)
for ℓ = 1, ..., h do
    Sort ℓ^th coordinate of {f(v)}_{v∈V} as x_1^(ℓ) ≤ x_2^(ℓ) ≤ ··· ≤ x_n^(ℓ)    // f(v)_ℓ = x_i^(ℓ) for some i
    for j = 1, ..., n − 1 do
        S_j^(ℓ) ← {v ∈ V | f(v)_ℓ ≤ x_j^(ℓ)}
S ← arg min_{ℓ∈[h], j∈[n−1]} c(δ_G(S_j^(ℓ)))/D(δ_H(S_j^(ℓ)))
return S
```

The guarantee of Algorithm 2.5 can be derived from Theorem 2.3.3.

Theorem 2.3.4. Algorithm 2.5 outputs cuts of sparsity at most $\alpha\lambda^*$,[a] in particular, it's a randomized $O(\log k)$-approximation algorithm for non-uniform sparsest cut.

[a] λ^* is the optimal solution of the dual linear program.

Proof. From Theorem 2.3.2, f is a contraction and with distortion $\alpha = O(\log k)$. From a similar argument as in Theorem 2.3.3, we see that for some z_S, $S \subseteq V$, we have

$$\begin{aligned}
\lambda^* &= \frac{\sum_{e\in E} c(e)x_e}{\sum_{i=1}^k D_i d_x(s_i,t_i)} \\
&\geq \frac{\sum_{uv\in E} c(uv)\|f(u)-f(v)\|_1}{\alpha\sum_{i=1}^k D_i\|f(s_i)-f(t_i)\|_1} \\
&= \frac{1}{\alpha}\frac{\sum_{uv\in E} c(uv)\sum_{\ell=1}^h |f(u)_\ell - f(v)_\ell|}{\sum_{i=1}^k D_i\sum_{\ell=1}^h |f(s_i)_\ell - f(t_i)_\ell|} \\
&= \frac{1}{\alpha}\frac{\sum_{\ell=1}^h\sum_{j=1}^{n-1}|x_{j+1}^{(\ell)}-x_j^{(\ell)}|c(\delta_G(S_j^{(\ell)}))}{\sum_{\ell=1}^h\sum_{j=1}^{n-1}|x_{j+1}^{(\ell)}-x_j^{(\ell)}|D(\delta_H(S_j^{(\ell)}))} = \frac{1}{\alpha}\frac{\sum_{S\subseteq V} z_S c(\delta_G(S))}{\sum_{S\subseteq V} z_S D(\delta_H(S))} \geq \frac{1}{\alpha}\min_{S\subseteq V}\frac{c(\delta_G(S))}{D(\delta_H(S))},
\end{aligned}$$

where the second last equality follows from the fact that $s_i, t_i \in V$ as well. ■

2.3.4 Line, ℓ_1, and Tree Embeddings, and State-of-the-Art

Theorem 2.3.2 shows that any finite metric space on n points embeds into ℓ_1 with distortion $O(\log n)$. Here, we hint on the underlying algorithm of the construction: from Lemma 2.3.1, ℓ_1 embeddings are a non-negative combination of line embeddings. A particular type of line embedding is the following.

Definition 2.3.10 (Fréchet embedding)**.** Let (V,d) be a metric space and let $S \subseteq V$. The *Fréchet embedding* is a contraction $f\colon V \to \mathbb{R}$ such that $f(v) = d(S,v)$.

Many results in embeddings into ℓ_p spaces are based on using Fréchet embeddings in various clever and often highly non-trivial ways. In particular, Theorem 2.3.2 is based on picking many random sets and combining the resulting Fréchet embeddings.

Now, we note that Theorem 2.3.2 can also be derived via probabilistic tree embeddings because every tree metric embeds into ℓ_1 isometrically. For general metrics, tree embeddings provide a more constrained space while yielding the same worst-case distortion. However, one can ask if ℓ_1 embeddings provide better distortion for concrete graph classes. This is indeed the case.

Example (Ring)**.** Consider a ring graph (a cycle with capacities). One can prove that tree embeddings require a distortion 2, while the ring metric can be isometrically embedded into ℓ_1. Thus, the flow-cut gap on ring is 1 which is not obvious.

Rather than looking at distortion, one can ask about the flow-cut gap obtained via different embeddings for a particular graph class. First, recall the followings for the non-uniform sparsest cut.

As previously seen (Theorem 2.3.4)**.** The flow-cut gap in general undirected graphs is $O(\log k)$.

Additionally, for general graph, a lower bound is also known.

Remark (Lower bound)**.** Expanders give a lower bound on the flow-cut gap to be $\Omega(\log k)$ even for uniform sparsest cut [LR99].

With these general bounds in mind, we now consider the flow-cut gap for planar graphs in particular.

Example (Planar graph)**.** There is a famous conjecture that the flow-cut gap in planar graphs is $O(1)$ [Gup+04]. Interestingly, for tree embeddings, there is a lower bound of $\Omega(\log n)$ even on the special case of planar graphs called series parallel graphs. Hence, tree embeddings are not powerful

enough to prove the conjecture.

The best flow-cut gap so far is $O(\sqrt{\log n})$ via ℓ_1 embeddings, thus separating the general graph case from the planar graph case. For series parallel graphs, we know that the flow-cut gap is a tight bound of 2 and establishing this tight bound took a fair amount of work.

For uniform sparsest cut, the flow-cut gap in planar graphs is $O(1)$ [KPR93]. One can show a tight connection between embeddability into ℓ_1 and flow-cut gap [Gup+04].

On the other hand, we can ask for a better approximation guarantee. First, note the following.

Note. Approximating the non-uniform sparsest cut problem is not the same as establishing the flow-cut gap as the flow-cut gap relies on the linear program relaxation.

Problem. Can we obtain a better approximation than $O(\log k)$ for non-uniform sparsest cut?

Answer. Yes! By using semi-definite programming based relaxation, one can obtain an $O(\sqrt{\log n})$-approximation for uniform sparsest cut [ARV09] and also the product instances. Based on this, an $O(\sqrt{\log n}\log\log n)$-approximation for non-uniform sparsest cut is achieved [ALN05; ALN07].[a] ⊛

[a] There was a conjecture that the SDP based relaxation would yield an $O(1)$-approximation, but it was shown that the integrality gap is essentially close to $\Omega(\sqrt{\log n})$.

2.3.5 Node Capacities[1]

A slight generalization of the uniform sparsest cut problem is obtained by considering demands induced by weights on vertices, which we refer to product instance.

Problem 2.3.4 (Product instance of sparsest cut). Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and vertex weight $\pi\colon V \to \mathbb{R}_+$. The *product instance of sparsest cut problem* is the non-uniform sparsest cut problem with demand $D(u, v)$ for edge $uv \in E$ setting to be $\pi(u)\pi(v)$.

Notation. In this case, the dual flow instances are called *product multi-commodity flow*.

We see that the product instances indeed generalizes uniform sparsest cut: If $\pi(u) = 1$ for all u, then this reduces to the uniform sparsest cut problem.

Remark (Subset sparsity). If $\pi(u) \in \{0, 1\}$ for all u, then we are focusing our attention on the sparsity w.r.t. the set $V' = \{v \in V \mid \pi(v) = 1\}$. Since vertices with $\pi(u) = 0$ play no role.

2.4 Expander and Well-Linked Set

We now introduce expanders [HLW06], which relate to the non-uniform sparsest cut in an intricate way.

Definition 2.4.1 (Expander). An *expander* with parameter α is a graph $G = (V, E)$ such that for all $S \subseteq V$ with $|S| \le |V|/2$ such that $|\delta(S)| \ge \alpha|S|$.

Definition 2.4.2 (Expansion). The *expansion* of a graph $G = (V, E)$ is $\min_{S\colon |S|\le|V|/2} |\delta(S)|/|S|$.

Another related notion called conductance also has a nice connection to uniform sparsest cut.

Definition 2.4.3 (Conductance). Given a graph $G = (V, E)$ and a cut $S \subseteq V$, the *conductance* $\phi(G)$ of G is defined as $|\delta(S)|/\operatorname{vol}(S)$ where $\operatorname{vol}(S) = \sum_{v\in S} \deg(v)$.

It's clear that G is an α-expander if the expansion of G is at least α. Initially, expansion arose from the graph bisection problem as we have seen before. However, as we will soon see, expander itself is quite

[1] We refer to the note for further information on further generalizations to node capacitated graph and directed graph.

interesting. Firstly, we note that expanders do exist, and they are quite common.

Lemma 2.4.1. There exists expanders with degree 3 with expansion $\alpha = \Omega(1)$. More specifically, a random 3-regular graph is an expander with high probability.

Hence, with Lemma 2.4.1, a natural way to generate an expander is to first sample a random regular graph, then check its expansion. However, computing the expansion is coNP-hard.

2.4.1 Expansion and Conductance via Sparsest Cut

The first connection of expander to the uniform sparsest cut is that the latter can be used to find the expansion of a graph. In particular, we see that when $|S| \leq |V|/2$, we have

$$\frac{1}{|V|}\frac{|\delta(S)|}{|S|} \leq \frac{|\delta(S)|}{|S||V \setminus S|} \leq \frac{2}{|V|}\frac{|\delta(S)|}{|S|}.$$

Remark. The expansion and the uniform sparsest cut's sparsity are within a factor of 2 of each other. Hence, while determining the expansion exactly is coNP-hard, we can use uniform sparsest cut to certify the expansion of a graph within a factor of 2.

Sometimes it is useful to consider expansion with vertex weights $w\colon V \to \mathbb{R}_+$ as well. In this case, the expansion is defined as $\min_{S\colon w(S)\leq w(V)/2} |\delta(S)|/w(S)$.

Note. It's dual corresponds to the *product multi-commodity flow instances* where $\pi(v) = w(v)$.

As for conductance, it's easy to see that one can capture it by expansion via setting weights on vertices with $w(v) = \deg(v)$, which further reduces to the product instance of sparsest cut.

Claim. For regular graphs, expansion and conductance are the same.

2.4.2 Spectral Relaxation for Conductance

In several applications it is important to obtain constant-degree expanders with constant expansion.

Intuition. The $O(\sqrt{\log k})$-approximation algorithms we saw are not useful in this regime.

It turns out that there is a very different method based on spectral graph theory that helps in this regime. For an undirected graph on n vertices, consider the Laplacian $\mathcal{L}_G := D - A$, where D is the diagonal degree matrix and A is the adjacent matrix. In particular, we have

$$(\mathcal{L}_G)_{ij} := \begin{cases} \deg(v_i), & \text{if } i = j; \\ -1, & \text{if } i \neq j \text{ and } A_{ij} = 1; \\ 0, & \text{otherwise.} \end{cases}$$

Since $\mathcal{L}_G$ is symmetric, all its eigenvalues are real. Moreover, this matrix is also positive semi-definite, hence all its eigenvalues are actually non-negative. Let $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ be its eigenvalues, then a well-known and famous result in spectral graph theory is the following.

Theorem 2.4.1 (Cheeger's inequality)**.** Given a graph G with conductance $\phi(G)$,

$$\frac{\lambda_2}{2} \leq \phi(G) \leq \sqrt{2\lambda_2}.$$

Remark. λ_2 provides a constant factor approximation for the conductance when it is a constant!

Since the expansion and conductance are related by the maximum degree, when the degree is a small constant, one can use λ_2 to certify expansion. Due to its importance for certifying expansion/conductance, some use λ_2 as the definition of expansion since it is computable and also helps in construction of expanders.

2.4.3 Expander Decomposition

Even if a graph is not an expander at first, it is possible to decompose a graph into smaller subgraphs such that each of them has good expansion/conductance. More explicitly, the goal is to remove as few edges as possible such that the graph decomposes into expanders. This is useful since expander leads to good algorithms. The question is the trade-off between the number of edges that we remove and the expansion that we can guarantee for the pieces.

> **Notation.** Technically the process works with conductance, but we use the terminology of expander decomposition for historical reasons.

Since one is often interested in finding fast algorithms for expander decomposition, it is common to explore trade-offs between the quality of the conductance of the pieces and the number of edges.

Definition 2.4.4 (Expander decomposition). A (ϕ, ϵ)-*expander decomposition* for some $\epsilon \in (0,1)$ is a partition of the (connected) graph $G = (V, E)$ into vertex induced subgraphs $\{G_i = G[V_i]\}_{i=1}^{h}$ such that each G_i has conductance at least ϕ, and the number of inter-cluster edges is at most ϵm, i.e.,

$$\frac{1}{2}\sum_{i=1}^{h}|\delta(V_i)| \leq \epsilon m.$$

Lecture 9: Expander Decomposition and Well-Linked Sets

We first see an algorithm that compute the expander decomposition. 24 Sep. 11:00

Algorithm 2.6: Expander Decomposition

Data: A connected graph $G = (V, E)$, base graph edge set size M,[a] parameter $\epsilon \in (0,1)$
Result: An $(\Omega(\frac{\epsilon}{\alpha \log m}), \epsilon)$-expander decomposition $\{G_i\}_{i=1}^{h}$

```
if m ≤ 10 log M/ε then                                   // Base case
    return G

(S, V \ S) ←α-Uniform-Sparsest-Cut(G) [b]                // vol(S) ≤ vol(V \ S)

if |δ(S)|/vol(S) > c/10 log M then                       // Check sparsity
    return G
else
    {G_i^(1)}_{i=1}^{h1} ←Expander-Decomposition(G[S], M, ε)
    {G_i^(2)}_{i=1}^{h2} ←Expander-Decomposition(G[V \ S], M, ε)
    return {G_i^(1)}_{i=1}^{h1} ∪ {G_i^(2)}_{i=1}^{h2}
```

[a] In the initial call, it's m.
[b] Recall that this is for conductance, which can be formalized as a product instance.

> **Note.** Algorithm 2.6 is intuitive and simple in retrospect: we simply compute the sparsest cut and recurse. The caveat is to keep track of the final termination condition is based on the original graph parameter $M = m$.

We now prove that Algorithm 2.6 indeed computes the expander decomposition.

Theorem 2.4.2. Let $G = (V, E)$ be a graph and $\epsilon \in (0,1)$. Suppose there is an α-approximation for the uniform sparsest cut, then there is an efficient algorithm that outputs an $(\Omega(\frac{\epsilon}{\alpha \log m}), \epsilon)$-expander decomposition of G.

Proof. We prove that Algorithm 2.6 achieves the desired properties.

Claim. The conductance of each subgraph output by Algorithm 2.6 is at least $\epsilon/10\alpha \log M$.

Proof. For the base case, if G is connected and has at most $10 \log M/\epsilon$ edges, then the conductance of G is at least $\epsilon/10 \log M$ since at least one edge crosses any cut, and the volume of the smaller side is at most $10 \log M/\epsilon$.

On the other hand, if the α-approximation algorithm of the uniform sparsest cut for conductance outputs a cut $(S, V \setminus S)$ with sparsity at least $\epsilon/10 \log M$, we know that the actual sparsity of G is at least $\epsilon/10\alpha \log M$ as desired. ⊛

Next, we analyze the total number of edges cut, which needs to be at most ϵm.

Intuition. If G is of constant size (and connected) or it does not have a sparse cut, Algorithm 2.6 does not cut any edges.

Let $T(m)$ be the total number of edges cut by Algorithm 2.6 on a graph with m edges. Algorithm 2.6 removes edges between S and $V \setminus S$ only when $|\delta(S)| \leq \epsilon \operatorname{vol}(S)/10 \log M$ where $\operatorname{vol}(S) \leq \operatorname{vol}(V \setminus S)$. Let $m' := |\delta(S)|$, $m_1 := |E(G[S])|$, and $m_2 := |E(G[V \setminus S])|$, then

$$m' \leq \frac{\epsilon}{10 \log M}(2m_1 + m') \Rightarrow m' \leq (1 - o(1))\frac{\epsilon}{5 \log M} m_1 \leq \frac{\epsilon}{4 \log M} m_1.$$

With $m_1 \leq m_2$, the recurrence can be written as

$$T(m) \leq T(m_1) + T(m_2) + \frac{\epsilon}{4 \log M} \min(m_1, m_2) = T(m_1) + T(m_2) + \frac{\epsilon}{4 \log M} m_1$$

where $m_1 + m_2 \leq m$, which gives $T(m) \leq \epsilon m$. ■

If we don't care about efficiency, we can set $\alpha = 1$ and solve the uniform sparsest cut exactly. In particular, Theorem 2.4.2 guarantees that the decomposed pieces have conductance $\Omega(1/\log m)$ while cutting only a constant fraction of the edges.

Note. The bound $\Omega(1/\log m)$ is tight as shown by the hypercube [Ale+17].

We can rephrase Theorem 2.4.2 in a different form where we want a lower bound on the conductance of the pieces and express the number of edges cut as a function that parameter:

Corollary 2.4.1. Let $G = (V, E)$ be a graph and ϕ be a parameter. Suppose there is an α-approximation for the uniform sparsest cut, then there is an efficient algorithm that computes a $(\phi, O(\alpha \cdot \phi \cdot \log m))$-expander decomposition.

Note. Number of edges cut is less than m only if $\alpha\phi \log m < 1$, so one should think of $\phi \leq 1/\alpha \log m$.

Remark. Theorem 2.4.2 is phrased in terms of m, the number of edges. Capacitated graphs can be handled by scaling since we do not assume that G is simple. However, the dependence on $\log m$ means that when capacities are large, we are not guaranteed a strongly polynomial bound. One can handle this issue in various ways depending on the application. In most applications of expander decomposition, it is the case that the total capacity of the edges can be assumed to be polynomially bounded in n and in this case, the $\log m$ factor is typically replaced with $\log n$.

We remark that Algorithm 2.6 is based on sparsest cut algorithms. Traditionally, these algorithms were quite slow. There have been several developments in the last few years which enabled sparsest cut to be reduced to a poly-logarithmic number of *s-t* flows via the so-called *cut-matching game* [KRV09; Ore+08], which in turn enabled faster flow algorithms. There are now near-linear time randomized algorithms for expander decomposition (with slightly weaker parameters than the ideal one) for the regimes of interest [SW19]. In some applications the randomized algorithm is not adequate and there has been considerable effort to obtain deterministic algorithms. There are now almost-linear time deterministic algorithm [Chu+20; SL21].

2.4.4 Well-Linked Set

Consider the following generalization of Definition 2.4.1, where we only care about expansion of a subset.

Definition 2.4.5 (Well-linked). A set $X \subseteq V$ is *well-linked* in a graph $G = (V, E)$ if for all $S \subseteq V$,

$$|\delta(S)| \geq \min(|S \cap X|, |S \cap (V \setminus X)|).$$

On the other hand, recall that α-expansion means that for all sets $S \subseteq V$ with $|S| \leq |V|/2$, $|\delta(S)| \geq \alpha|S|$. This is a cut condition. Suppose A, B are two disjoint sets of vertices of equal size $|A| = |B|$, clearly we have $|A|, |B| \leq |V|/2$. We can ask for a similar guarantee as the same cut condition. This turns out to be another generalization of expander:

Definition 2.4.6 (Linkage). Let $A, B \subseteq V$, $A \cap B = \varnothing$, and $|A| = |B|$. An A-B *linkage* is a set of edge-disjoint paths connecting A to B with each vertex in $A \cup B$ in exactly one path. In this case, we say A and B are *linked* in G.

Note. We do not have to insist on $A \cap B = \varnothing$. If not and we allow each vertex in $A \cap B$ to connect to itself via an empty path, then it is the same as asking $A \setminus B$ and $B \setminus A$ to be linked. Thus, requiring $|A| = |B|$ suffice.

We can view linkage as sending flows:

Definition 2.4.7 (Fractional linkage). An A-B linkage is *fractional* if there is a flow in G with that satisfies demand of 1 on each vertex in A and a demand of -1 on each vertex of B.[a] In particular, we say that A, B are α-*linked* for some parameter α if there is a flow in G that satisfies the demand of α on each vertex in A and a demand of $-\alpha$ on each vertex of B.

[a] Note that this corresponds to a single-commodity flow.

Lemma 2.4.2. Suppose G is an α-expander with $\alpha \geq 1$. Then there is an A-B linkage in G for every pair of disjoint equal sized sets A, B.

Proof. Let $|A| = |B| = k$. To check whether there are k desired edge-disjoint paths, consider creating a flow problem by adding two new vertices s and t such that s and t are connected to all vertices in A and B of capacity 1, respectively. Let $H = (V \cup \{s, t\}, E_H)$ be this new graph. If there is an s-t flow of value k in H, then these correspond to the desired paths. Suppose otherwise, i.e., the flow is strictly less than k. Then by max-flow min-cut theorem, there is a set $S' \subseteq V_H$ with $s \in S'$ and $t \notin S'$ such that $|\delta_H(S')| < k$. Let $S \coloneqq S' - s$ be the corresponding set of vertices in G. It's easy to see that

$$|\delta_H(S')| = |\delta_G(S)| + |A \cap (V \setminus S)| + |B \cap S|.$$

This implies that $S \neq \varnothing$ since otherwise $|A \cap (V \setminus S)| = |A| = k$, and the above will imply $|\delta_H(S')| \geq k$, a contradiction. Similarly, $V \setminus S \neq \varnothing$. Now, suppose $|S| \leq |V \setminus S|$, i.e., the smaller side. Then by the expansion guarantee, $|\delta_G(S)| \geq |S| \geq |A \cap S|$, implying

$$|\delta_H(S')| = |\delta_G(S)| + |A \cap (V \setminus S)| + |B \cap S| \geq |A \cap S| + |A \cap (V \setminus S)| = k,$$

which is again a contradiction. The same proof applies if $|V \setminus S| \leq |S|$. ■

One can scale capacities or directly prove the following.

Corollary 2.4.2. Suppose G is an α-expander. Then if A, B are disjoint vertex sets with $|A| = |B|$, then A, B are α-linked.

Interestingly, the converse is also true.

Lemma 2.4.3. Suppose G is a graph and for any two disjoint set A, B of equal size, A, B are α-linked. Then G is an α-expander.

Proof. We cannot have $|\delta_G(A)| < \alpha|A|$ due to the max-flow min-cut theorem. ■

The following shows the connection of linkage and well-linked.

Claim. A set X is well-linked if for all $A, B \subseteq X$ and $|A| = |B|$, A and B are linked.

Definition 2.4.8 (Fractinoal well-linked)**.** A set X is *α-well-linked* in a graph G is for any two $A, B \subseteq X$ with $|A| = |B|$, the sets A, B are α-linked.

More generally, we have the following.

Lemma 2.4.4. A set $X \subseteq V$ is α-well-linked in G if and only if for any set $S \subseteq V$, $|\delta(S)| \geq \alpha \min(|S \cap X|, |S \cap (V \setminus X)|)$.

Corollary 2.4.3. A graph $G = (V, E)$ is an α-expander if and only if V is α-well-linked in G.

Thus, the notion of well-linked sets extends the definition of expansion to subsets of the graph. This is very useful in a number of settings.

Example (Star)**.** A start on n vertices is an expander and has a well-linked set of size n. This strange artifact is because of the large degree of the center vertex.

This artifact disappears if we ask for constant degree graphs or if we insist on node linkage, i.e., we now want *node-disjoint paths*.

Definition 2.4.9 (Node linkage)**.** Let $A, B \subseteq V$, $A \cap B = \varnothing$, and $|A| = |B|$. An A-B *node linkage* is a set of node-disjoint paths connecting A to B with each vertex in $A \cup B$ in exactly one path.[a]

[a] We also skip the definition of α-linkage for now. The definition is basically the same where we want flow with node capacities rather than edge capacities.

Definition 2.4.10 (Node well-linked)**.** A set X is *α-node-well-linked* in G if for any two $A, B \subseteq X$ with $|A| = |B|$, A, B are α-linked.

Intuition. If X is node-well-linked in a graph, then X cannot have a sparse node separator, i.e., if S separates $G \setminus S$ into components and S does not have any vertices of X then no component of $G \setminus S$ can have more than $|S|$ vertices of X.

In graph theory literature on treewidth $\mathrm{tw}(G)$ (formally introduced in Definition 5.3.3), the notion of linkages is defined primarily via node-disjoint paths. We will not use this very often in this course hence we overload edge and node well-linked notations. In particular, its connection to node-well-linkedness is the following.

Theorem 2.4.3 ([Ree97])**.** Let k be the cardinality of the largest node-well-linked set in a graph G. Then $k \leq \mathrm{tw}(G) \leq 4k$.

Remark. In fact, most algorithmic approaches to computing $\mathrm{tw}(G)$ are based on algorithms for sparse node separator computations.

Node-well-linkedness is connected to vertex-expanders. Sometimes people do not distinguish between these two notions too much in the expansion literature because of the following.

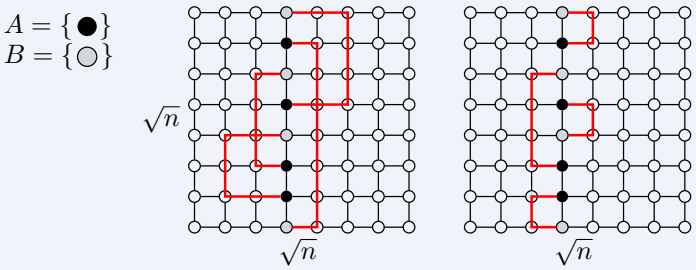
Claim. If G is an α-edge-expander with maximum degree d, then G is an $\Omega(\alpha/d)$-vertex-expander.

Thus, if one is working with constant degree graphs, the two notions are not very far.

Example (Star). Consider the start graph again. One can see that it is an edge-expander, but it is very far from being a vertex expander. In fact, the largest node-well-linked set in a star is of size 2.

On the other hand, the grid is not only edge-well-linked, but also node-well-linked.

Example (Grid). A $\sqrt{n} \times \sqrt{n}$ grid is a planar graph with n vertices. It has a bisection with $O(\sqrt{n})$ edges hence it is at best a $1/\sqrt{n}$-expander (which in fact it is). It has a well-linked set of size $\Omega(\sqrt{n})$, i.e., rows or columns X of $\sqrt{n}$ vertices. We see that although G is not a good expander, but it has good expansion w.r.t. X. Actually, it's even node well-linked (right).



In some sense, grid is the best planar graph in terms of node-well-linkedness. First, we need to understand the following:

Definition 2.4.11 (Balanced separator). Given a graph $G = (V, E)$, a vertex subset S is a *balanced separator* if every connected component of $G - S$ is of size less than $2n/3$.

Theorem 2.4.4. Every planar graph has a balanced separator of size $O(\sqrt{n})$. Hence, no planar graph on n vertices have a node-well-linked set of size more than $c\sqrt{n}$ for some fixed constant c.

2.4.5 Well-Linked Decomposition

Finally, we note that the same decomposition can be done for well-linked sets, just as the expander decomposition. This is a generalization since well-linkedness generalizes the notion of expansion and conductance from the whole vertex set to subset of vertices. To state the desired result, we need to introduce the notion of weighted well-linked:

Definition 2.4.12 (Weighted-well-linked). Let $\pi \colon X \to \mathbb{R}_+$ be non-negative weights on $X \subseteq V$. We say that X is *π-weighted well-linked* in G if for any $S \subseteq V$, $|\delta_G(S)| \geq \min(\pi(S \cap X), \pi(S \cap (V \setminus X)))$.

Note that if $\pi(v) = 1$ for all $v \in X$, we get back to Definition 2.4.5. If $\pi(v) = \deg(v)$ for each $v \in V$, we obtain the conductance.

Remark. Typically, it is assumed that $\pi(v) \leq c\pi(V)$ for some constant $c < 1$, e.g., $c = 1/3$. That is, no single vertex is too heavy compared to the total weight.

Now, suppose we have a multi-commodity flow between vertices in V that is routed in G. Let $D(uv)$ be the flow routed for the unordered pair uv. For each $u \in V$, let $\pi(u) = \sum_{v \in X, v \neq u} D(uv)$ be the total flow from u to some vertex in X.

Intuition. We would like the terminals (vertices) to be π-wel-linked, but if they are not, we would like to do a decomposition into vertex induced subgraphs such that they are.

Furthermore, we would like to do it in such a way that we lose as little amount of the original flow as possible. Since it is difficult to bound the total flow lost, so we will upper bound it by the total number of edges crossing the partitions.

Note. Consider each edge $uv \in E$ and send one unit of flow along that edge for the pair uv, then $\pi(u) = \deg(u)$ for each $u \in V$. This recovers the expander decomposition that we saw earlier.

Theorem 2.4.5 (Well-linked decomposition). Let $G = (V, E)$ be a graph and $\epsilon \in (0, 1)$ be a parameter. Suppose there is an α-approximation algorithm for the uniform sparsest cut. Let $\pi : V \to \mathbb{R}_+$ be induced by a multi-commodity flow in G, where $\pi(u)$ is the total flow originating at u. Then there is an efficient algorithm that decomposes G into vertex induced subgraphs $\{G_i = G[V_i]\}_{i=1}^h$ for $V_i \subseteq V$ such that in each G_i, V_i is $\beta\pi'$-well-linked where $\beta = \Omega(\frac{\epsilon}{\alpha \log m})$ and $\pi'(u)$ is the total flow incident to u in G_i (that remains from the original flow). Furthermore, the number of inter-cluster edges is at most $\epsilon\pi(V)$, i.e.,

$$\frac{1}{2}\sum_{i=1}^{h} |\delta(V_i)| \leq c\pi(V).$$

Proof. The corresponding algorithm and the proof is essentially the same as in Theorem 2.4.2. Let $M = \pi(V)$ to keep it fixed throughout the algorithm as before. We use the α-approximation algorithm on V with weights π. It outputs a cut $(S, V \setminus S)$ with $\pi(S) \leq \pi(V \setminus S)$. If $|\delta(S)| \geq \epsilon\pi(S)/(10 \log M)$, we output G, and we are guaranteed that V is $\pi/(10\alpha \log M)$-well-linked in G since we used an α-approximation algorithm for uniform sparsest cut. Otherwise, we remove the edges in $\delta(S)$ and recurse on $G[S]$ and $G[V \setminus S]$ after throwing out all the flow that is lost by the edges in $\delta(S)$. The recursion analysis is the same as before, and we can guarantee that the total number of edges cut is at most $c\pi(V)$. ■

Remark. In routing problems, it is more useful to work with the notion of flow-well-linkedness rather than cut-well-linkedness. Theorems similar to Theorem 2.4.5 can be shown via the known flow-cut gap results [CKS05].

The notion of well-linked decomposition can be generalized to node-capacitated setting naturally. It can also be generalized to directed graphs (only symmetric demands) [CE15], which has found several applications recently in fast algorithms [BGS20]. Via duality, they are related to low-diameter decomposition in directed graphs, which have played a key role in recent fast algorithms for shortest paths with negative lengths. We will see this later (Section 3.3).

Lecture 10: Tree-Based Oblivious Routing

2.5 Oblivious Routing

26 Sep. 11:00

Consider routing demands between source-sink pairs in a network $G = (V, E)$ with capacity $c : E \to \mathbb{R}_+$. In particular, the actual demands are not known in advance and come and go in an online fashion. Hence, the problem is, which routes should the demand for some specific pair (s, t) be routed on?

Intuition. To feasibly route a given set of demands, we need to essentially solve a multi-commodity flow, which helps to balance the network's capacity among many competing pairs.

This is a non-trivial problems and leads to lots of the breakthroughs. Oblivious routing is one of which that we will be interested in. The goal in oblivious routing, in some sense, is to bridge the static setting where the demands are all fully known to the fully online setting where we specify a route to a new demand when it arrives.

Problem 2.5.1 (Oblivious routing). Given a directed graph $G = (V, E)$ with edge capacity $c : E \to \mathbb{R}_+$, the goal of *oblivious routing* is to output a distribution of paths $\mathcal{P}_{u,v}$ between every pair of $u, v \in V$ such that when the demand $D : V \times V \to \mathbb{R}_+$ come, specifically, $D(u, v)$ for a pair $u, v \in V$, a flow is routed on a path $P \in \mathcal{P}_{u,v}$ according to the distribution.

Alternatively, we can also think of sending $D(u, v)$ demand fractionally along the paths in proportion to the probability. Clearly, Problem 2.5.1 is oblivious since the probability distribution over $\mathcal{P}_{u,v}$ is specified before knowing any of the actual demands. In particular, if we know the entire set of demands in advance, we can compute an optimal routing via some sort of multi-commodity flow computation. The question now is that, how well can an oblivious routing do when compared to a static routing, and how to measure the quality. More fundamentally, do good oblivious routing even exists?

The initial work that motivated the problem of oblivious routing is that good deterministic oblivious routing exist in special classes of graphs such as hypercubes [VB81]; later the need for randomization in the sense of picking paths randomly as we described was realized.

2.5.1 Routable Demands and Congestion

We first digress towards defining routable demands and some properties.

Definition 2.5.1 (Routable). A demand matrix $D \in \mathbb{R}_+^{n \times n}$ is *routable* in a directed graph $G = (V, E)$ with capacity $c \colon E \to \mathbb{R}_+$ if there exists a feasible multi-commodity flow for D in G.

It's easy to see that $\mathcal{D}_G := \{D \in \mathbb{R}_+^{n \times n} \mid D \text{ is routable } \}$ is a convex set. Moreover, given a demand matrix D, we can efficiently check if D is routable in G via solving a linear program for multi-commodity flow. Thus, we have a membership oracle for the convex set $\mathcal{D}_G$.

Intuition. The flow linear program is nice because it is also linear in $D(u, v)$ values, hence we can claim something stronger.

Suppose we have $w(u, v)$ for all $u, v \in V$. We ask the following question.

Problem 2.5.2 (Optimiznig demand polytope). Can we solve $\max_{D \in \mathcal{D}_G} \sum_{u,v} w(u, v)D(u, v)$?

Answer. By the equivalence of optimization and separation, we have an efficient separation oracle over $\mathcal{D}_g$! That is, there is an efficient algorithm that given $D \in \mathbb{R}_+^{n \times n}$, either outputs that $D \in \mathcal{D}_G$ or outputs a separating hyperplane that separates D from the convex set $\mathcal{D}_G$. ⊛

The above is a convoluted but easy way of deriving a useful fact. A more direct way is often referred to as the Japanese theorem on multi-commodity flow, which can be derived from linear program duality, as we saw in non-uniform sparsest-cut.

Lemma 2.5.1 (Japanese theorem). The demand D is routable if and only if for all $\ell \colon E \to \mathbb{R}_+$,

$$\sum_{e \in E} c(e)\ell(e) \geq \sum_{u,v \in V} D(u, v)d_\ell(u, v)$$

where d_ℓ is the shortest path distance induced by ℓ.

Hence, to prove that D is not routable in G, it suffices to produce one length function $\ell \colon E \to \mathbb{R}_+$ that violates the inequality in Lemma 2.5.1.

Intuition. One can view $\mathcal{D}_G$ as being defined by an infinite set of linear inequalities, one for each non-negative length function.

Next, to measure the quality of oblivious routing, the central notion is the congestion.

Definition 2.5.2 (Congestion). A demand matrix $D \in \mathbb{R}_+^{n \times n}$ is *routable with congestion* $\rho > 0$ in a directed graph $G = (V, E)$ with capacity $c \colon E \to \mathbb{R}_+$ if D is routable in G where edge capacities are multiplied by ρ.

When $\rho = 1$, it corresponds to the routable case. Moreover, the set of all routable demand metrics in G with a fixed congestion ρ is also convex.

2.5.2 Oblivious Routing to Minimize Congestion

In oblivious routing, we want to specify a probability distribution over $\mathcal{P}_{u,v}$ for each pair (u, v). Naively, since there are exponential many paths, specifying a probability on each path can be tricky.

Intuition. One can view such a probability distribution as a flow of one unit from u to v where the flow on a path $P \in \mathcal{P}_{u,v}$ is equal to the probability of P.

Thus, one compact way to represent a probability distribution over paths is via a unit flow via flow values on edges, which can be specified using only m numbers. Formally, consider the following:

Definition 2.5.3 (Oblivious routing scheme)**.** Given a graph $G = (V, E)$, consider the following.

> **Definition 2.5.4** (Edge-based oblivious routing)**.** An *edge-based oblivious routing* is a collection of unit flows in G, one for each $(u, v) \in V \times V$, specified via edge-based flows $f_e^{(u,v)}$.

> **Definition 2.5.5** (Path-based oblivious routing)**.** A *path-based oblivious routing* is similar to edge-based where the unit flow is specified via a path based flow.

However, edge-based flow does not uniquely specify a path-based flow, so we are losing information by using the compact representation, but it is helpful in some computational situations.

> **Note.** $f_e^{(u,v)}$ is a number in $[0, 1]$, and whether the oblivious routing is specified via edge-based or path-based, this quantity is well-defined.

Now, for oblivious routing, we can define its congestion as follows.

Definition 2.5.6 (Congestion of oblivious routing)**.** Let D be a demand matrix. The *congestion* incurred for D via the oblivious routing specified by $f_e^{(u,v)}$ for $e \in E$ and $(u, v) \in V \times V$ is

$$\max_{e \in E} \frac{\sum_{u,v \in V} D(u, v) f_e^{(u,v)}}{c(e)}.$$

Moreover, the *congestion* of an oblivious routing is the smallest $\rho \geq 1$ such that every routable demand $D \in \mathcal{D}_G$ is routed with congestion at most ρ by the oblivious routing.

We say that an optimal oblivious routing for G is the one which achieves the smallest ρ among all oblivious routings. In other words, we are asking how much should we scale up the capacities of G so that any demand matrix D that is routable in G can be routed in G via the oblivious routing, i.e.,

$$\max_{D \in \mathcal{D}_G} \max_{e \in E} \frac{\sum_{u,v \in V} D(u, v) f_e^{(u,v)}}{c(e)}.$$

Claim. Every graph on n vertices has an oblivious routing scheme with congestion n^2.

Proof. Consider computing the max-flow for every pair and then scale it down to a unit flow. ⊛

We give some pointers to the fundamental results on oblivious routing in a chronological order:

- Harald Räcke proved that every undirected graph admits an oblivious routing with congestion $O(\log^3 n)$ [Räc02]. The initial proof is based on an optimal algorithm for non-uniform sparsest cut, hence does not lead to an efficient algorithm to construct the oblivious routing. Soon after, efficient algorithms are known with the congestion being $O(\log^2 n \log \log n)$ [BKR03; HHR03]. These results are based on the hierarchical expander decomposition of the graph which results in a compact cut representation of every graph.
- Via a simple idea in retrospect, that the optimal oblivious routing can be computed efficiently via linear program techniques even for directed graphs [Aza+03]. Note that being able to compute an optimum oblivious routing for any given graph does not tell us an easy way to understand a universal bound. We will see this next.
- Oblivious routing in directed graphs requires congestion $\Omega(\sqrt{n})$ [Aza+03]; this lower bound holds even for the restricted case of single-source demands. A similar lower bound was shown to hold also for undirected graphs with node capacities [Haj+07]. Recently, the lower bound for directed graphs has been improved to $\Omega(n)$ [Ene+16].

- In another breakthrough, Harald Räcke made a beautiful and surprising connection between oblivious routing and probabilistic tree embeddings for metric distortion [Räc08] and obtain an optimal $O(\log n)$-congestion tree-based oblivious routing scheme. Via this algorithm, an $O(\log n)$-approximation for the minimum bisection problem in graphs is developed.

2.5.3 Efficient Algorithm for Optimal Oblivious Routing

In this section, we will prove that one can find an optimal edge-flow based oblivious routing in polynomial time [Aza+03]. In particular, we will consider directed graphs since it is easier to define edge-based flows. To find an edge-flow based oblivious routing with minimum congestion, we write the following linear program which essentially follows from the definition:

$$\begin{aligned} \min\ & \rho \\ & f_e^{(u,v)} \text{ defines a unit flow from } u \text{ to } v \quad \forall e \in E, \forall (u,v) \in V \times V; \\ & \sum_{(u,v)\in V\times V} D(u,v) f_e^{(u,v)} \le \rho c(e) \qquad \forall e \in E, \forall D \in \mathcal{D}_G; \\ & f_e^{(u,v)} \ge 0 \qquad \forall e \in E, \forall (u,v) \in V \times V. \end{aligned} \tag{2.5}$$

Note. We omit writing down the linear constraints for the unit flow from u to v since it's easy.

The only catch in solve Equation 2.5 is that it has an infinite number of constraints, with polynomially many (mn^2) variables. Hence, one can use the ellipsoid method if we have an efficient separation oracle.

Claim. There is an efficient separation oracle for Equation 2.5.

Proof. Given $f_e^{(u,v)}$ and ρ, we see that checking unit flow is easy. For the second constraint, given any fix edge e, we simply maximize $\sum_{u,v\in V} D(u,v) f_e^{(u,v)}$ via Problem 2.5.2 (efficiently!) and compare it with $\rho c(e)$. Hence, by iterating through all $e \in E$, we're done. ⊛

While the resulting algorithm is not very efficient in practice, but it shows the power of the ellipsoid method and working with large "implicit" linear programs (Definition 4.1.5) fearlessly. One can use multiplicative weight updates and other techniques to obtain fast approximation algorithms for some of these problems.

Remark. This technique to design optimum oblivious routing is fairly general. We do not need to consider the full set of demand matrices $\mathcal{D}_G$, as long as we have a nice convex set of demand matrices that we can optimize over, we can find an optimum oblivious routing when restricted to those demand matrices.

2.5.4 Tree-Based Oblivious Routing via Duality and Low Stretch Trees

We now prove that in undirected graphs, there is always an oblivious routing with congestion $O(\log n)$, which is an optimal bound. Räcke prove this result via an elegant connection to tree embeddings for distance preservation [Räc08]. The bound, the connection, and the tree-based aspect are all important.

As previously seen. Probabilistic approximation of a graph by spanning trees for distances incurs an $O(\log n \log\log n)$ distortion [ABN08] while we can obtain an optimal $O(\log n)$ distortion if we allow dominating tree metrics (recall the differences).

It is easier to understand the ideas, and in particular the notation, by working with spanning trees than with general hierarchical tree representations (Theorem 3.1.1) of graphs (in some cases, the spanning tree result is useful and needed). The difference in the distortion is insignificant for our purposes here, and we will use these results in a black-box fashion.

Now, let $\alpha(n)$ denote the best bound that we can obtain for approximating the distance in an n-node graph $G = (V, E)$ with edge length $\ell\colon E \to \mathbb{R}_+$ by a probability distribution over spanning tree $p\colon \mathcal{T}_G \to [0,1]$ such that $\sum_{T\in\mathcal{T}} p_T = 1$. A simple implication of $\alpha(n) = O(\log n \log\log n)$ is the following.

Lemma 2.5.2. Let $G = (V, E)$ be a graph with non-negative edge lengths $\ell \colon E \to \mathbb{R}_+$. Let $w \colon E \to \mathbb{R}_+$ be any set of no-negative edge weights. Then there is a spanning tree $T \in \mathcal{T}_G$ such that

$$\frac{\sum_{uv \in E} w(uv) d_T(u,v)}{\sum_{e \in E} w(e)\ell(e)} \leq \alpha(n),$$

where $d_T(u,v)$ is the length of the unique path from u to v in T.

Proof. Recall that there is a probability distribution $p \colon \mathcal{T}_G \to [0,1]$ such that for every pair of vertices $u, v \in V$, $\mathbb{E}[d_T(u,v)] \leq \alpha(n) d_G(u,v)$ where distances are induced by the edge lengths $\ell \colon E \to \mathbb{R}_+$. Now, fix any weight function w over pairs of vertices. Then by linearity of expectation,

$$\mathbb{E}\left[\sum_{u,v \in V} w((u,v)) d_T(u,v)\right] \leq \alpha(n) \sum_{u,v \in V} w((u,v)) d_G(u,v).$$

Thus, there exists a tree T such that $\sum_{u,v \in V} w((u,v)) d_T(u,v) \leq \alpha(n) \sum_{u,v \in V} w((u,v)) d_G(u,v)$. Now, consider the case when the support of the weights w is only on the edges of G, i.e., $w((u,v)) = 0$ when $uv \notin E$. In this case, we write $w(uv)$ for $uv \in E$. Furthermore, we let $d_T(u,v)$ is the shortest path length along the path in T. Since $d_G(u,v) \leq \ell(u,v)$,

$$\sum_{uv \in E} w(uv) d_T(u,v) \leq \alpha(n) \sum_{uv \in E} w(uv) d_G(u,v) \leq \alpha(n) \sum_{uv \in E} w(uv)\ell(uv),$$

which gives the desired result. ■

We now introduce a specific type of oblivious routing. Consider any probability distribution p over the spanning trees of G. Observe that this distribution induces an oblivious routing since each tree $T \in \mathcal{T}_G$ gives a unique path between any pair $(u,v) \in V \times V$. We sample a tree from the distribution and whenever a demand arrives, we simply route it along the unique path in the tree.

Notation (Tree-based oblivious routing)**.** The above scheme is what we called *tree-based oblivious routing*, which is indeed path-based.

Note. The distribution over $\mathcal{T}_G$ induces a distribution for every pair of vertices simultaneously.

While this is a restricted class of oblivious routing, but it's particularly nice, at least form a theoretical point of view. Two natural questions arise.

Problem. How good is the best tree-distributions-based oblivious routing?

Problem. Can the best tree-distributions-based oblivious routing be efficiently computed?

Räcke showed that tree-based oblivious routing have a nice structural property that allows one to characterize the congestion in a simple way [Räc08]. Let T be a spanning tree and consider an edge $e \in E_T$. $T - e$ induces a partition of the vertex set of $G = (V, E)$ into two sets $(S_e, V \setminus S_e)$. We define the load $L(T, e)$ on edge e to be $c(\delta(S_e))$. If e is not in T, then we let $L(T, e) = 0$.

Intuition. Think of all the edges crossing the cut $(S_e, V \setminus S_e)$. For each of those edges $e' = (s,t) \in \delta(S_e)$, the path from s to t in T has to go via e.

Hence, if we want to route demands corresponding to edges, then the congestion on e will be $L(T,e)/c(e)$. Formally, since we now have a probability distribution over $\mathcal{T}_G$, hence we consider the expected load and expected congestion:

Definition. Let $p \colon \mathcal{T}_G \to [0,1]$ be a probability distribution that induces a tree-based oblivious routing. Consider a given edge $e \in E$.

Definition 2.5.7 (Expected load)**.** The *expected load* on e is $L(e) = \sum_{T \in \mathcal{T}_G} p(T) L(T, e)$.

Definition 2.5.8 (Expected congestion)**.** The *expected congestion* on e is $\rho(e) = L(e)/c(e)$.

A simple yet important observation is the following, which characterizes the quality of the tree-based oblivious routing based on the expected congestion of the edges.

Lemma 2.5.3. Given $p \colon \mathcal{T} \to [0, 1]$, the maximum congestion of the tree-based oblivious routing induced by p is at most $\max_{e \in E} \rho(e)$.

Proof. Fix a demand matrix $D \in \mathcal{D}_G$. Congestion of e is less than $\rho(e)$ since

$$\frac{\sum_{T \in \mathcal{T}_G, T \ni e} p_T \sum_{|S_e \cap \{u,v\}|=1} D(u, v)}{c(e)} \leq \frac{\sum_{T \in \mathcal{T}_G, T \ni e} p_T L(T, e)}{c(e)}.$$

Taking maximum over $e \in E$ gives the result. ∎

Lemma 2.5.3 allows us to write an linear program to find the best tree-based oblivious routing. Let x_T to denote the probability that $T \in \mathcal{T}_G$ is chosen in the probability distribution, and ρ to denote the congestion that we wish to minimize. We write down constraints that express x as a probability distribution and to express the load on each edge being bounded by $\rho c(e)$:

$$\begin{aligned}
&\min \rho && && \max \beta \\
&\sum_{T \in \mathcal{T}_G} x_T = 1 && && \sum_{e \in E} c(e) z_e = 1 \\
&\sum_{T \in \mathcal{T}_G} x_T L(T, e) \leq \rho c(e) && \forall e \in E; && \sum_{e \in T} L(T, e) z_e \geq \beta && \forall T \in \mathcal{T}_G; \\
\text{(P)}\quad & x_T \geq 0 && \forall T \in \mathcal{T}_T; && \text{(D)}\quad z_e \geq 0 && \forall e \in E.
\end{aligned}$$

We see that the dual is equivalent to

$$\max_{z \colon E \to \mathbb{R}_+} \min_{T \in \mathcal{T}_G} \frac{\sum_{e \in T} L(T, e) z_e}{\sum_{e \in E} c(e) z_e}. \tag{2.6}$$

The observation is that Lemma 2.5.2 implies that the optimal dual value is $\alpha(n)$, which corresponds to the bound for tree-based distance approximation! Suppose this was true then we have shown that there exists a tree-based oblivious routing with congestion $O(\log n \log \log n)$. We now prove this formally.

Theorem 2.5.1. The optimal dual value β is at most $\alpha(n)$.

Proof. Observe that by the definition of $L(T, e)$, after interchanging the order of summation,

$$\sum_{e \in T} L(T, e) z_e = \sum_{uv \in E} c(uv) \sum_{e \in P_T(u,v)} z_e,$$

where $P_T(u, v)$ is the unique path from u to v in T. Note that $\sum_{e \in P_T(u,v)} z_e$ can be thought of as the length of the path from u to v in T according to lengths given by z, which we denote as $d_T(u, v)$. Now, think of $c(e)$ as a weight $w(e)$ and think of z_e as length $\ell(e)$, Equation 2.6 can be written as

$$\max_{\ell \in \mathbb{R}^m_+} \min_{T \in \mathcal{T}_G} \frac{\sum_{uv \in E} w(uv) d_T(u, v)}{\sum_{uv \in E} w(uv) \ell(uv)},$$

and by Lemma 2.5.2, this is at most $\alpha(n)$. ∎

Theorem 2.5.1 implies that we have $\alpha(n) = O(\log n \log \log n)$ expected congestion for tree-based oblivious routing.[2] Since Theorem 2.5.1 is based on duality, it is not immediately clear that it leads to an efficient algorithm. As one would expect, we need an efficient algorithm for the dual separation

[2] See note for how to obtain the optimum bound $O(\log n)$.

oracle. This is the problem of approximating distances in the graph by spanning trees, and we have seen efficient algorithms for it. However, it is still not obvious that we can use such approximation algorithms in the dual, but there are standard techniques via the multiplicative weight update method and related ideas [Räc08].

Chapter 3

Some Cool Stuffs

Lecture 11: Expander Hierarchy and Its Sufficiency

In this chapter, we present some recent non-trivial results. The first one is a hierarchical decomposition, which is also known as *expander hierarchy*. The second and the third are about single-source shortest path, but with negative lengths. 1 Oct. 11:00

3.1 Cut-Based Hierarchical Decomposition

Räcke proved the existence of a cut-based hierarchical decomposition of undirected graphs as a tool to prove the existence of good oblivious routings [Räc02], which is different from the tree-based oblivious routing construction we saw last time. This cut-based hierarchical decomposition has found several applications outside the original motivation for oblivious routing, and is also a fundamental structural result in graph theoretic terms.

> **Note.** The construction and proof are technical, and this is the first attempt to teach it and provide some additional commentary along the way, which we hope is of pedagogical value for those interested in understanding the details of a construction from [BKR03].

We first set up notations to state the result where the statement is tailored towards cut-approximation rather than the oblivious routing aspect.

3.1.1 Statement of the Hierarchical Decomposition

Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{Z}_+$, consider the cut-tree.

> **Definition 3.1.1** (Laminary). A family of subsets is *laminary* if no two sets A, B in the family cross, i.e., $A \cap B = \varnothing$ or $A \subseteq B$ or $A \supseteq B$.

> **Definition 3.1.2** (Hierarchical decomposition). Given a graph $G = (V, E)$, a *hierarchical decomposition* is a laminary family of subsets of the vertex set V.

Any laminary family over V, augmented with the entire set V if needed, defines a way to decompose V, and hence implicitly the graph, in a recursive fashion. Each such decomposition has an associated rooted tree T, where the leaves of T are labeled with the vertex set V and each internal node v_t corresponds to a subset S_{v_t} of the vertices of G which are the leaves in the subtree T_{v_t}. We associate the graph $H_{v_t} = G[S_{v_t}]$ with each v_t. Hierarchical decompositions of graphs are used in several settings and the meaning and applications of them depend on the application and context. Here, we will be interested in decompositions that preserve cuts of the graph, in particular for routing demands in G.

> **Definition 3.1.3** (Cut-tree). A hierarchical decomposition T is a *cut-tree* if for each edge $(v_t, v_{t'})$ of T where $v_{t'}$ is the parent of v_t, we assign a capacity equal to $c(\delta_G(S_{v_t}))$.

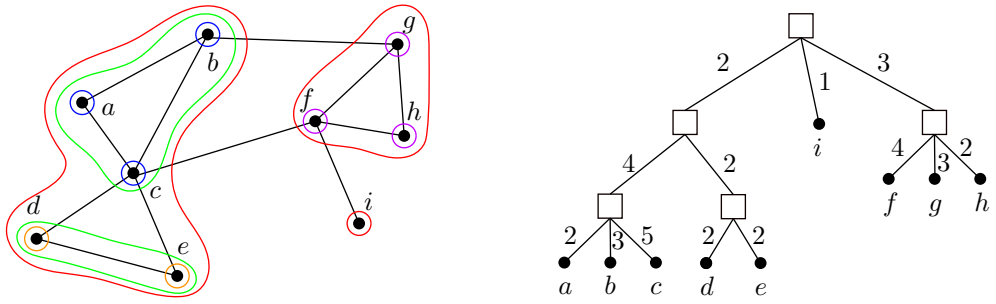


Figure 3.1: A graph with a hierarchical decomposition and its representation as a cut tree.

Remark. A cut-tree has exactly n edges, so it's keeping track only a very small number of cut values.

Surprisingly, every graph admits a cut-tree which can approximate all sparse cuts in the graph. To formalize the statement, we start with the following simple observation.

Claim. Let $G = (V, E)$ be a graph with edge capacities $c\colon E \to \mathbb{R}_+$. Suppose $D \in \mathbb{R}_+^{V \times V} \in \mathcal{D}_G$ is a non-negative demand matrix that is routable in G. Then, it is routable in any cut-tree T of G (D is now demand matrix over the leaves of T).

Proof. Recall that a demand matrix is routable in a capacitated tree if and only if it satisfies the cut condition. The only sparse cuts of interest in a tree are the single edge cuts. If D is routable in G, then for any cut $(S, V \setminus S)$ we have $D(S, V \setminus S) \leq c(S, V \setminus S)$ in G. Since T is a cut-tree, and D is only between leaves of T, each edge $v_t, v_{t'}$ in E_T corresponds to the cut $(S_{v_t}, V \setminus S_{v_t})$ in G and the capacity of the edge in $tisequal \to fmc(S, V \setminus S)$. Thus, D satisfies the cut condition in T, hence is routable in T. ⊛

Now, we're ready to state the result.

Theorem 3.1.1 (Hierarchical expander decomposition [Räc02]). Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, there is a cut-tree $T = (V_T, E_T)$ such that any demand matrix $D \in \mathbb{R}_+^{V \times V}$ that is routable in T is also routable in G with congestion $O(\log^3 n)$.

Note. Hierarchical expander decomposition has found many applications in fast graph algorithms recently [Gor+21].

In other words, there is a very compact data structure, i.e., the cut-tree, that captures *all* the relevant cuts for routing (in particular, the sparse cuts) in G approximately within a poly-logarithmic factor. The first proof was based on finding sparsest cuts and hence did not lead to an efficient algorithm [Räc02]. Soon after, efficient algorithm [BKR03] with an improved congestion bound of $O(\log^2 n \log\log n)$ [HHR03], which is still the beset known.

We will give a proof outline of the construction of [BKR03] that achieves a weaker congestion bound of $O(\log^4 n)$. If one wants to approximate only cuts but not the routing aspect, then an improved bound of $O(\log^{3/2} n \log\log n)$ is known [RS14].

3.1.2 Intuition, Connection to Expanders and Well-Lined Decomposition

Firstly, we say that a cut-tree is ρ-approximate if any demand matrix D routable in T can be routed in G with congestion ρ. Moreover, throughout we will be working with induced subgraphs of the original graph that arise as the algorithm constructs the hierarchical decomposition. Given $S \subseteq V$, its induced subgraph $G[S]$ can also be referred as a cluster, and we associate it with a node of the cut-tree T. We will be interested in how well can S interface with the rest of the graph in terms of the edges between S and $V \setminus S$. Since we will be interested in G and various induced subgraphs, consider the following.

Notation. The set of all edges with one end point in A and the other in B is denoted as $E(A, B)$.[a]

[a] Note that A and B need not be disjoint.

Thus, to represent $\delta_G(S)$, we will often use $E(S, V \setminus S)$. Finally, we will use $c(A, B)$ as a shortcut for the total capacity of edges in $E(A, B)$:

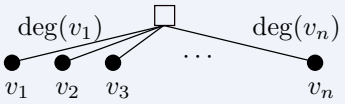
Notation. We let $c(A, B) := \sum_{e \in E(A,B)} c(e)$, and will refer to $c(\delta_G(S))$ sometimes as $\text{out}(S)$ for compactness.

Most of the time, we are interested in conductance, but we use the word expansion. Suppose $G = (V, E)$ has good conductance $\phi(G) = \Omega(1)$, i.e., for all $S \subseteq V$, $c(\delta(S)) \geq \phi \,\text{vol}(S)$. We know the following from the flow-cut gap.

Claim. Suppose G has conductance ϕ. Let D be any demand matrix such that the following holds for each $u \in V$: $b(u) = \sum_{v \in V} D(u, v) \leq c(\delta(u))$. Then, G satisfies the cut condition for ϕD and hence D is routable in G with congestion $O(\sigma(n)/\phi)$ where $\sigma(n)$ is the flow-cut gap for product multi-commodity flow in graphs of at most n nodes. In particular, $\sigma(n) = O(\log n)$ in general graphs and is $O(1)$ in planar graphs.

Thus, if $\phi = \Omega(1)$, the set of routable demand matrices can be approximately characterized by stating that they should respect the trivial cuts at the vertices.

Example (Expander). If G is a graph with conductance ϕ, then the star graph with V as the leaves is a cut-tree with approximation $O(\log n/\phi)$.

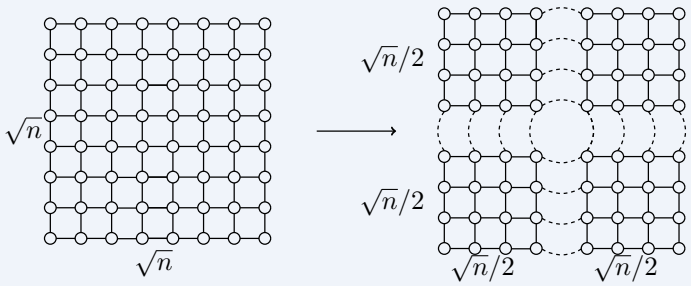


What should one do if G does not have good conductance?

Intuition. A natural approach is to decompose the graph into well-connected pieces, i.e., subgraphs with high conductance, along sparse cuts.

The question is how to. It is also not difficult to see that decomposing the graph into one or a few levels is not sufficient by looking at the grid graph.

Example (Grid). Consider a planar graph such as the $\sqrt{n} \times \sqrt{n}$ grid. Then no subgraph has good conductance unless it is essentially of constant size. Thus, the only reasonable thing is to decompose the graph recursively into several levels. For example, the grid can be decomposed naturally into say four equal sized grids and then recursively to obtain a $O(\log n)$ depth cut-tree which one can prove has poly-logarithmic approximation.



Although grids are easy to decompose due to their nice symmetric properties, it is still non-obvious to formally understand why it yields a good cut-tree.

Intuition. A key is to see that the boundary of the grid is well-linked.

How should one do this for a general graph whose structure is not apparent to us? The key concepts

that one needs are being able to understand how a given cluster $H = G[S]$ interfaces with the rest of the graph through its boundary edges (those edges that cross S) and how a given cluster is decomposed into sub-clusters. We formalize these notations as follows. They are the refined versions of well-linkedness.

Intuition. Since well-linkedness allows us to use the notion of expansion/conductance for subsets of vertices and with arbitrary weights so that we can use it in a refined way.

We start from the following.

Definition 3.1.4. Let $G = (V, E)$ and $\pi : V \to \mathbb{R}_+$ be bounds on vertices. Let $\alpha > 0$ be a scalar.

Definition 3.1.5 (Cut well-linked)**.** G is (π, α)-*cut-well-linked* if for all $S \subseteq V$, $c(\delta(S)) \geq \alpha \cdot \min(\pi(S), \pi(V \setminus S))$.

Definition 3.1.6 (Flow well-linked)**.** G is (π, α)-*flow-well-linked* if for any demand matrix D such that $\sum_{v \in V} D(u, v) \leq \pi(u)$ for all $u \in V$ is routable in G with congestion $1/\alpha$.

We define flow-well-linkedness in a stronger form since this is useful for the application in this topic. It can be defined in a slightly different form as we only consider the demand matrix D_π being $D_\pi(u, v) = \pi(u)\pi(v)/\pi(V)$. The advantage of this definition is that we can check whether G satisfies this requirement efficiently via a multi-commodity flow computation. Indeed, these two do not differ much:

Claim. Let $\pi : V \to \mathbb{R}_+$ be non-negative bounds on vertices of $G = (V, E)$. Suppose G can route the product multi-commodity flow induced by π, then G can route any demand matrix D with congestion 2 if $\sum_{v \in V} D(u, v) \leq \pi(u)$ for all $u \in V$.

From the flow-cut gap, we obtain the following.

Claim. IF G is (π, α)-cut-well-linked, then G is $(\pi, \alpha/\sigma(n))$-flow-well-linked.

Note. If G has conductance ϕ then it is (π, ϕ)-cut-well-linked where $\pi(u) = c(\delta(u))$ is the capacitated degree of u in G. As we saw, it also implies that G is $(\pi, \phi/\sigma(n))$-flow-well-linked.

From the above discussion, we see that the key advantage of allowing arbitrary π is the following:

Intuition. Allowing π to be arbitrary allows us to generalize the notation of conductance and routability to subsets of vertices and to control the amount of flow incident to a vertex.

IN the context of cut-trees, we are interested in the following two types of well-linkedness:

Definition 3.1.7 (Boundary-well-linked)**.** Let $G = (V, E)$ be a graph and $H = G[S]$ be an induced subgraph (cluster) $S \subseteq V$. We say S is α-*cut/flow-boundary-well-linked* if H is (π, α)-cut/flow-well-linked where $\pi(u) = c(u, V \setminus S) = c(\delta(u) \cap \delta(S))$ for all $u \in S$.

Note that we require $H = G[S]$ to be well-linked as a separate graph, implying the following:

Intuition. If a cluster is boundary-well-linked, then it acts as a good router as far as its external interface is concerned.

This is an important property that is essentially required of any cluster in a cut-tree since the capacity of the edge from a cluster to its parent is equal to $c(S, V \setminus S)$, the boundary capacity. Another important property that is needed for a cluster is with respect to its children in the cut-tree, i.e., how it is partitioned into sub-clusters for the next level. This motivates the following:

Definition 3.1.8 (Partition-and-boundary-well-linked)**.** Let $G = (V, E)$ be a graph and $H = G[S]$ be an induced subgraph (cluster) $S \subseteq V$. Let $\mathcal{D} = \{S_i\}_{i=1}^r$ be a partition of S into sub-clusters

$\{H_i = G[S_i]\}_{i=1}^r$. We say S is *α-cut/flow-partition-and-boundary-well-linked* if H is (π', α)-cut/flow-well-linked where $\pi'(u)$ is the capacity of edges going outside its sub-cluster in $\mathcal{D}$ for each $u \in S$.

Intuition. Given a cluster S in the cut-tree and its children $\{S_i\}_{i=1}^r$, the edge connecting S_j to S in the tree has capacity equal to $c(\delta_G(S_j))$. In a sense the cluster S acts as a single node in the tree connecting the children together. Thus, S should be able to route any set of demands that go between the children as long as the total demand leaving each S_j is at most $c(\delta_G(S_j))$.

Consider the same example as in Figure 3.1, we see that for $S = \{f, g, h\}$, their corresponding π values are $2, 1, 0$ when considering boundary-well-linkedness, while they are $4, 3, 2$ when considering partition-and-boundary-well-linkedness (i.e., π'):

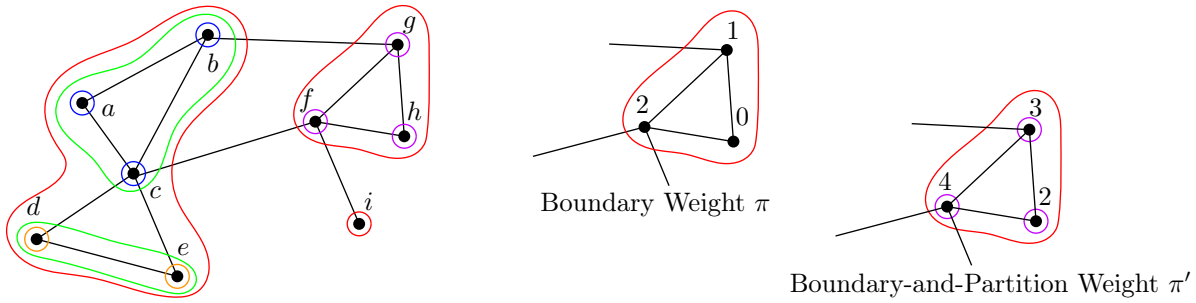


Figure 3.2: Corresponding weight values when considering Definition 3.1.7 ($\pi(u)$) and Definition 3.1.8 ($\pi'(u)$) for $u \in S = \{f, g, h\}$.

To simplify the notation, consider the following:

Notation (Partition-well-linked). We say that S is *α-partition-well-linked* w.r.t. $\mathcal{D}$ and omit $\mathcal{D}$ when it is implicit in the context.

It is sometimes useful to view the boundary-well-linkedness and partition-well-linkedness by sticking dummy vertices as leaves and saying that the dummy vertices are well-linked (with weight 1). E.g., in Figure 3.2, imagine adding a dummy vertex to edge sticking out of the cluster.

3.1.3 Sufficient Condition of Being a Good Cut-Tree

Now that we have set up the definitions, the first part of the proof is to understand what conditions should the tree T satisfy such that it has good properties. Interestingly, only two properties are required: low depth/height and partition-well-linkedness of each cluster in the decomposition.

Lemma 3.1.1. Suppose T is a cut-tree for G of height h. Suppose each cluster S of T (corresponds to an internal node of T) is α-flow-partition-well-linked, then T is $O(h/\alpha)$-approximate.

Proof. Recall that each cluster $H = G[S]$ in the tree T is α-flow-partition-well-linked. For each S:

- π_S: the weights on vertices of S which correspond to the out-degree of the vertices;
- π'_S: the weights corresponding to the partition $\mathcal{D}$ of S induced by its children in T.

Essentially, this is Figure 3.2 where we specify S. Clearly, $\pi'_S(u) \geq \pi(u)$ for all $u \in S$. Also, note that $\pi_S(S) = c(S, V \setminus S)$ is the capacity of the edges leaving S. When S is clear from the context, we simply use π and π'. Now, suppose D is a demand matrix that is routable in T. We need to show that it is routable in G with congestion $O(h/\alpha)$.

Intuition. The height h comes into the picture since T is a hierarchical decomposition, each edge $e = uv \in E$ can be in up to h clusters along a path from the root of T until u and v get separated from the first time. The proof works by constructing a series of multi-commodity flows that can be stitched together to route all the demands.

We assume that each cluster S in T has its own private copy of each edge $e \in E[S]$ when we compute multi-commodity flows in the cluster. We will bound the congestion of routing per cluster and then use the fact that each edge is in at most h clusters to derive the final congestion. Consider routing the demands from bottom up. Let (a, b) be a vertex-pair with demand $D(a, b)$. Let $H = G[S]$ be the cluster which corresponds to the last common ancestor of a and b in T. We call the cluster H the meetup cluster for the pair (a, b). Let $\{H_i\}_{i=1}^r$ be the children of H in T. Without loos of generality, let $a \in H_1 = G[S_1]$ and $b \in H_2 = G[S_2]$. Consider any cluster $H' = G[S']$ along the path from a to H not including H itself. Such a cluster is called a transition cluster for pair (a, b). Then, a will distribute its $D(a, b)$ flow such that each node $u \in S'$ receives a fraction $\pi_{S'}(u)/\pi_{S'}(S')$ of the flow.

Remark. Recall that $\pi_{S'}(S') = c(S', V \setminus S')$ is the boundary capacity of the cluster S' and $\pi_{S'}(u)$ is the amount of that capacity incident to $u \in S'$. So for nodes u that is not on the boundary, $\pi_{S'}(u) = 0$, i.e., *we're sending flow to the boundary of* S'.

Intuition. All of a's flow has to leave the cluster S' to eventually reach b, so we spread the flow in proportion to the boundary capacity.

Note that the actual flow from a for the pair (a, b) that reaches $u \in S'$ is $D(a, b)\pi_{S'}(u)/\pi_{S'}(S')$. For pair (a, b) this stops as H_1 for a and at H_2 for b, before the meetup cluster $H = G[S]$. It then comes the responsibility of the meetup cluster H to ensure that these flows match up in S. Thus, the flow for pair (a, b) is fully satisfied in the clusters of the subtree at their least common ancestor which corresponds how it is routed in the tree T.

To complete the description, since the flow is sent bottom up, we need to describe how a cluster $H = G[S]$ with children $\{H_i\}_{i=1}^r$ maintains the invariant. It has two parts:

- For all demand pairs (a, b) such that H is the meetup cluster, H has to ensure that the flow for the pairs which have reached the children's boundary are exchanged in the graph $G[S] = H$.
- For all demand pairs (a, b) for which H is a transition cluster (meaning that their least common ancestor is above H in T), their flow has reached exactly one of the children of H inductively. H needs to redistribute this flow using edges only in $G[S] = H$ such that for each such demand pair (a, b), the flow is distributed over nodes $u \in S$ in proportion to $\pi_S(u)$.

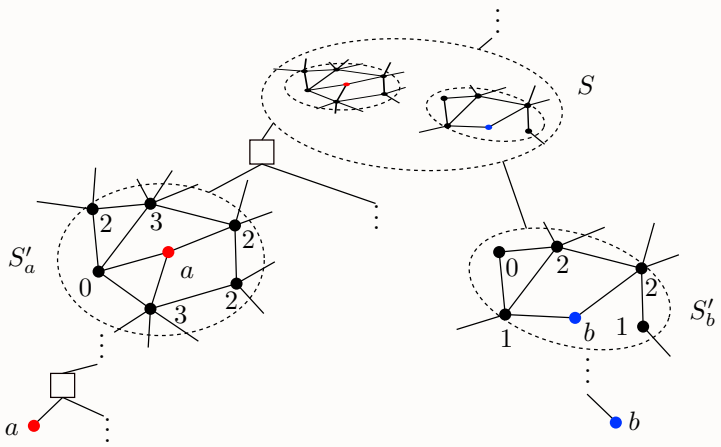


Figure 3.3: The cluster S'_a is on the path from a in T to the least common ancestor S with b, where a distributes $D(a, b)$ to each node $u \in S'_a$ in proportion to their boundary capacity $\pi_{S'_a}$. Same for b.

We show that each of the above two steps can be accomplished in $H = G[S]$ with congestion $O(1/\alpha)$ by using the α-flow-partition-well-linkedness property. It turns out to be reasonably simple because we have set up the definitions properly. Consider the first part. Fix a pair (a, b) for which H is the meetup cluster. As we discussed before, say $a \in H_1 = G[S_1]$ and $b \in H_2 \in G[S_2]$ where H_1 and H_2 are children of H, a has already sent its flow to vertices in S_1 where $u \in S_1$ has received a flow of $D(a, b)\pi_{S_1}(u)/\pi_{S_1}(S_1)$, and similarly, b has sent its flow to S_2 in proportion to π_{S_2}. We exchange this flow by setting up a demand matrix. It is not hard to see that there is one which will ensure that the flow is matched up. We do this for every pair for which H is the meetup cluster we can exchange the flows that have reached the children clusters by setting up their own demand matrix. Routing the union of these demand matrices in H would suffice to exchange the flow, and the problem now is whether we can do it and what is the congestion incurred.

Claim. All these demand matrices can be routed in H with congestion $1/\alpha$.

Proof. Consider any child cluster $H_i = G[S_i]$ of H. Since D is routable in T, the total demand that needs to meetup in H and which originates in H_i is at most the capacity of the edge (H, H_i) in T, which has capacity $c(S_i, V \setminus S_i) = \pi_{S_i}(S_i)$, i.e., the boundary capacity of S_i. This means that for each $u \in S_i$, the total flow that has reached u for all the demands that need to be matched up in H is at most $\pi_{S_i}(u)/\pi_{S_i}(S_i) \cdot c(S_i, V \setminus S_i) = \pi_{S_i}(u)$. Thus, for exchanging the meetup flow, we have set up many demand matrices, but the total demand from these matrices that originate at $u \in S_i \in S$ is at most $\pi_{S_i}(u)$. This is true for all the vertices in S. But recall that any demand matrix that satisfies this degree condition, by definition of the α-flow-partition-well-linkedness, can be routed in $H = G[S]$ with congestion $1/\alpha$.[a] ⊛

[a] Indeed, this is the reason we required this condition on the clusters.

Hence, we're done for the first part. The second part is similar. Consider any demand pair (a, b) for which H is a transition cluster. This means that one of a, b is in H and the other is not. Say $a \in H$, and it has already distributed the flow to the boundary of the child H_i of H where $a \in H_i$, i.e., for each $u \in S_i$ in proportion to $\pi_{S_i}(u)/\pi_{S_i}(S_i)$. To maintain the invariant, in cluster H we need to distribute a's flow to nodes in S such that each $v \in S$ gets flow in proportion to $\pi_S(v)/\pi_S(S)$. We can set up a demand matrix to exchange this flow again. For any node $u \in S_i$, the total demand originating at u is at most $\pi_{S_i}(u)$ since the total demand crossing S_i is at most $c(s_i, V \setminus S_i)$, and for each node $v \in S$, the total demand that it receives is at most $\pi_S(v) \le \pi'_S(v)$. Thus, the union of the demand matrices respect the total bounds imposed by π' at each node of S, and hence by the α-flow-partition-well-linkedness, can be routed in S with congestion $1/\alpha$.

For the base case of the induction, we see that the flow originates at the singleton leaves which is its own boundary and hence there is nothing to do there. As we argued earlier, the total congestion on an edge is the union of the congestions of all the clusters it participates in, and an edge participates in at most h clusters, which proves the result. ■

Remark. We can also construct an oblivious routing from Lemma 3.1.1 as well, but we will not do so to keep the presentation simpler.

The power of Lemma 3.1.1 is that it has essentially captures what we need from the hierarchical decomposition. We will see a construction that guarantees that $h = O(\log n)$ and $\alpha = \Omega(1/\log^3 n)$, and this gives us a cut-tree that is $O(\log^4 n)$-approximate, proving (a weaker version of) Theorem 3.1.1.

Lecture 12: Construction of Expander Hierarchy

3.1.4 Top-Down Algorithm for Constructing a Cut-Tree

3 Oct. 11:00

We now describe a top-down divide-and-conquer algorithm that starts with V and creates a cut-tree satisfying the conditions (i.e., the partition-well-linkedness property) of Lemma 3.1.1 with $h = O(\log n)$ and $\alpha = \Omega(1/\log^3 n)$, which gives an $O(\log^4 n)$-approximation, as we promised.

The algorithm does not backtrack, i.e., once it partitions a cluster S into its children, it simply recurses on the children. For this to work, the algorithm requires that each cluster satisfy an additional property before it can be successfully partitioned into sub-clusters that satisfy the partition-well-linkedness

property. This requires something called precondition [BKR03].

Notation (Parameter)**.** We will use some parameters for formal proofs and to help see the ideas.

- σ: the upper bound on the flow-cut gap in an n-vertex graph for product multi-commodity flow instances, where we know that $\sigma = O(\log n)$.[a]
- λ: $64\sigma \log n$ with $\lambda = \Theta(\log^2 n)$ for general graph.
- α: $1/24\sigma\lambda = \Omega(1/\log^3 n)$, which is the goal. Note that $1/\alpha$ is a log factor smaller than λ.

[a] Recall that we can get an $O(\sigma(n))$ approximation for non-uniform sparsest cut when needed.

We will assume that the capacities are integer valued. The algorithm is polynomial in the sum of the capacities, so technically we can only apply it for capacities that are poly-bounded, but we will ignore this issue since we are more interested in the proof of the existence for now.

Definition 3.1.9 (Precondition)**.** Let $H = G[S]$ for $S \subseteq V$ be a cluster. Let $\pi \colon S \to \mathbb{R}_+$ be boundary capacity weights, i.e., $\pi(u)$ is the capacity of the edges leaving S that are incident to u. We say S satisfies the *precondition* if for all $U \subseteq S$ such that $|U| \leq 3|S|/4$,

$$c(U, S \setminus U) \geq \frac{\pi(U)}{\lambda}.$$

Intuition. We want each cluster S to be boundary-well-linked. On the other hand, to ensure the recursion depth, the algorithm will obtain a tree of height $O(\log n)$ by ensuring that each cluster S is decomposed into sub-clusters such that no sub-cluster S_i has more than $2|S|/3$ vertices.

Definition 3.1.9 resembles a cut-well-linkedness condition, but it is not quite the same since the numerator is about π while the denominator is about the cardinality of U. As we will see, this actually corresponds to sparsity of a related demand matrix.

The motivation for this precondition comes from the eventual divide-and-conquer algorithm for constructing a cut-tree which requires each part to be a constant factor smaller than its parent cluster. Note that the entire vertex set V will trivially satisfy this precondition and hence the algorithm can get started. To enable recursion, the algorithm will ensure that when it partitions S into sub-clusters, each of the resulting sub-clusters also satisfies this precondition.

Intuition. We need this precondition because it turns out that not every cluster can be successfully partitioned to satisfy the partition-well-linkedness property.

However, since it's not feasible to efficiently check whether a given cluster satisfies the precondition, so we will use approximation algorithms via flow-cut gap to indirectly guarantee that the precondition is satisfied. Now, to ensure precondition, we consider something like well-linked style decomposition. Specifically, during the algorithm, we will need to ensure that the sub-clusters that are created satisfy the precondition for the recursion. As we remarked, we cannot directly guarantee it. For this reason, we will use a decomposition procedure called assure precondition, that will decompose *any* given cluster S such that the resulting pieces after the decomposition satisfy the precondition. We first see the lemma.

Lemma 3.1.2. Let $S \subseteq V$ with $|S| \geq 2$ be any cluster. Then assure precondition partitions S into sub-clusters $\{S_i\}_{=1}^{p}$ for some $p \geq 1$ such that

(a) each S_i satisfies the precondition, and

(b) $\sum_{i=1}^{p} c(\delta_G(S_i)) \leq 2c(\delta_G(S))$.

We postpone the proof of Lemma 3.1.2, and note that the algorithm to achieve the above is very similar to the expander decomposition, but not exactly the same due to the nature of the condition.

As previously seen. In expander decomposition, we wish to decompose a given graph into pieces such

that each piece has good conductance while cutting as few edges as possible. From Theorem 2.4.2, we know that each piece has conductance $\Omega(1/\sigma \log n)$ via an $O(\sigma)$-approximation algorithm for non-uniform sparsest cut based on flow-cut gap, while cutting only a constant fraction of edges.

Specifically, assure precondition algorithm first sets up a demand matrix D:[1] for each vertex $u \in S$, it creates a demand $D(u, v)$ for each v where $D(u, v) = \pi(u)/|S|$.

Intuition. u wants to distribute $\pi(u)$ uniformly among all vertices. The final demand matrix is then the union of these demand matrices, one for each u.

From Lemma 3.1.2 (b), the increase in the number of cut edges is upper bounded by $2c(\delta_G(S))$ since the sum counts each such newly cut edge twice. But we are only interested in each piece being boundary-well-linked, so this is similar to well-linked decomposition with initial weight equal to $\pi_S(u)$.

Algorithm 3.1: Assure Precondition

Data: A connected graph $G = (V, E)$, cluster $S \subseteq V$
Result: A partition $\{S_i\}_{i=1}^{p}$ of S satisfying Lemma 3.1.2

```
1  for (u, v) ∈ V × V, u ≠ v do                                  // Ordered pair
2      D(u, v) ← π_S(u)/|S|
3
4  ((A, B), ψ) ←α-Non-Uniform-Sparsest-Cut(G[S], H = (V, D))[a]   // sparsity ψ of (A, B)
5
6  if ψ > 4σ/λ then                                              // If not sparse
7      return {S}
8  else
9      {S_i^(1)}_{i=1}^{p_1} ←Assure-Precondition(G, A)
10     {S_i^(2)}_{i=1}^{p_2} ←Assure-Precondition(G, B)
11     return {S_i^(1)}_{i=1}^{p_1} ∪ {S_i^(2)}_{i=1}^{p_2}
```

[a] We assume that this is a flow-cut gap based α-approximation algorithm.

Proof of Lemma 3.1.2. Our goal is to relate the sparsest cut for the special demand matrix we design to the precondition so that we can guarantee that S will satisfy the precondition if there is no sparse cut with respect to D. Let ϕ be the overall minimum sparsity according to D. Via the flow-cut gap, we can find a cut (A, B) such that its sparsity according to D is at most $\sigma \cdot \phi$.

Claim. Suppose S does not satisfy the precondition, then $\phi \leq 4/\lambda$.

Proof. If S does not satisfy the precondition, then there is a partition $(U, S \setminus U)$ of S such that

$$c(U, S \setminus U) \leq \frac{\pi(U)}{\lambda}$$

from the definition (where $|U| \leq 3|S|/4$). Consider the demand according to D crossing this partition, which is

$$\pi(U) \cdot \frac{|S \setminus U|}{|S|} + \pi(S \setminus U) \cdot \frac{|U|}{|S|} \geq \pi(U) \cdot \frac{|S \setminus U|}{|S|},$$

which is greater than $\pi(U)/4$ since $|S \setminus U| \geq |S|/4$. Thus, the sparsity of $(U, S \setminus U)$ is at most $4c(U, S \setminus U)/\pi(U) \leq 4/\lambda$ by our assumption. Hence, $\phi \leq 4/\lambda$. ⊛

This ensures that assure precondition guarantees that each cluster in the final partition satisfies the precondition because of the termination condition and the recursion. The main issue is why it does not cut too many edges in creating the partition.[a] We will write a recursion to understand the total number of edges cut. If S satisfies the minimum sparsity condition (line 6), i.e., we return S and don't cut any edges, then (b) is trivial. Now, suppose we find a partition (A, B) of S with

[1] This is very different from a product multi-commodity flow!

sparsity $\psi < 4\sigma/\lambda < 1/16 \log n$, then we recurse on A and B. The total demand crossing (A, B) is

$$\pi(A)\frac{|B|}{|S|} + \pi(B)\frac{|A|}{|S|} \leq \pi(A) + \pi(B) = \pi(S),$$

implying

$$c(A, B) \leq \frac{\pi(S)}{16 \log n},$$

resulting in the total edges cut is at most $\pi(S)$ with a careful analysis, hence we're done. ■

[a] The analysis is similar to Theorem 2.4.2, bit a bit more involved.

Remark. Unlike the recursion analysis in Theorem 2.4.2, these new cut-edges $c(A, B)$ create two new problems whose total size in terms of outgoing edges is slightly larger compared to the original problem. Nevertheless, $\log n$ is sufficiently large that this increase can also be absorbed.

We now describe the main algorithm called partition, which achieves the following guarantee:

Lemma 3.1.3. Let $S \subseteq V$ with $|S| \geq 2$ with $|S| \geq 2$ be any cluster that satisfies the precondition. Then partition algorithm partitions S into sub-clusters $\{S_i\}_{i=1}^{p}$ for some $p > 1$ such that

(a) $|S_i| \leq 2|S|/3$ for each sub-cluster S_i,

(b) each S_i satisfies the precondition,

(c) S satisfies the α-flow-partition-well-linkedness with respect to the decomposition.

Suppose we have such an algorithm. Then the top-down recursive algorithm is straightforward. In each step, it ensures that the sizes of the sub-clusters goes down by a constant factor, and they satisfy the precondition to enable recursion. Moreover, the cluster itself satisfies the desired partition-well-linkedness property with respect to the decomposition.

As previously seen. V trivially satisfies the precondition because it has no outgoing edges as we mentioned, hence the algorithm can get started. Moreover, given a cluster S and its decomposition into sub-clusters, we can check efficiently whether it satisfies the partition-well-linkedness property.

Problem. How should partition work?

Answer. Suppose $S = V$ and G has good conductance. Then, as we argued, the right decomposition is to simply take all the singleton vertices, and it satisfies the desired properties. Hence, naturally, the algorithm starts with an *optimistic* guess: the trivial singletons' decomposition of S. The nice aspect of this starting decomposition is that the properties (a) and (b) are automatically satisfied.

Note. Property (c) can be checked efficiently, and if we got lucky to satisfy it, we're done!

Of course this is too optimistic. What the algorithm actually does is to maintain a current partition that satisfies the properties (a) and (b). Let this partition be $\mathcal{P} = \{H_i\}_{i=1}^{p}$. It first checks if S satisfies property (c) for $\mathcal{P}$. If it does, we're done. Otherwise, it finds a sparse cut (A, B) of S w.r.t. π' as S is not partition-well-linkedned. It then uses the partition (A, B) to compute another partition $\mathcal{P}'$ such that once again, $\mathcal{P}'$ satisfies the properties (a) and (b).

Note. We make sure it's making progress by ensuring that the total number of inter-cluster edges in $\mathcal{P}'$ is strictly less than the inter-cluster edges in $\mathcal{P}$.[a] This is the crux of the proof.

[a] The algorithm is only guaranteed to reduce the number of inter cluster edges by one, so this is the reason for the dependence of the running time on $\sum_{e \in E} c(e)$.

Thus, the algorithm can repeat this process, and it is guaranteed to terminate with a partition that satisfies all three properties. ⊛

Now, we need to describe how to compute the new partition $\mathcal{P}'$ from $\mathcal{P}$. We develop some intuition before giving a formal description.

Intuition. Given that (A, B) is a sparse cut for the partition-well-linkedness condition, it means that the number of edges crossing A is too small in comparison to $\pi'(A)$, i.e., the total number of inter-cluster edges in the current decomposition and the total number of external edges. Since S satisfy the precondition, it is already $1/\lambda$-boundary-well-linked. Thus, the reason we have a sparse cut is that there are too many inter-cluster edges inside A due to the current partition $\mathcal{P}$.

Suppose we get lucky and A is the union of some sub-collection of clusters from $\mathcal{P}$. Then it makes sense to merge all the clusters in A and make A the new cluster since it reduces the number of inter-cluster edges. But this new cluster A may not satisfy the precondition, but we can use assure precondition on A to partition it into clusters that satisfy the precondition.

Remark. We will introduce new inter-cluster edges in this process, but not too many.

Since A may not be a clean union of current clusters, the algorithm employs a simple heuristic to find a union of existing clusters: it takes the union of all clusters that have at least 3/4 of their vertices inside A.

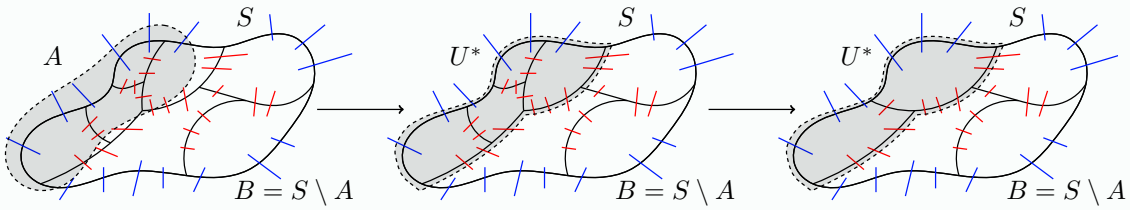


Figure 3.4: The algorithm should first find a sparse cut (A, B) if the current decomposition does not satisfy the desired partition-well-linkedness property. It converts A to a set U^* that is a union of existing clusters. It then uses assure precondition to find new clusters within U^* that satisfy the precondition. Note how new inter-cluster edges are introduced in the last step.

Such a heuristic is called the *rounding step*, which we denote as $U^* = \text{round}(A)$ [ABN08]. While intuitive, the main technical difficulty is to show that this works.[a]

[a] It may not be clear at this stage that round(A) is non-empty!

We now formally describe the algorithm.

Algorithm 3.2: Partition

Data: A connected graph $G = (V, E)$, cluster $S \subseteq V$ satisfying precondition
Result: A partition $\mathcal{P}$ of S satisfying Lemma 3.1.3

$\mathcal{P} \leftarrow \{\{v\} : v \in S\}$

for $(u, v) \in V \times V$, $u \neq v$ **do** // Ordered pair
  $D(u, v) \leftarrow \pi'_S(u)/|S|$

while S *is not* α*-partition-well-linkedness w.r.t.* $\mathcal{P}$ **do** // Via flow computation
  $((A, B), \psi) \leftarrow \alpha$-Non-Uniform-Sparsest-Cut$(G[S], H = (V, D))$ // $|A| \leq |B|$, $\psi \leq \sigma\alpha$
  $U^* \leftarrow \varnothing$
  for $R_i \in \mathcal{P}$ **do** // Compute round(A)
    if $|R_i \cap A| > 3|R_i|/4$ **then**
      $U^* \leftarrow U^* \cup R_i$
      $\mathcal{P} \leftarrow \mathcal{P} - \{R_i\}$
  $\mathcal{Q} \leftarrow$Assure-Precondition(G, U^*)
  $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{Q} - \{U^*\}$
return $\mathcal{P}$

To summarize, Partition starts with the partition consisting of the singletons and either terminates or updates the partition. When Algorithm 3.2 does not terminate, it then computes U^* by taking the union of some existing clusters and then uses assure precondition to re-cluster it. We now prove Lemma 3.1.3, i.e., the formal guarantee of the partition algorithm.

Proof of Lemma 3.1.3. Firstly, when partition terminates, the output clearly satisfies property (c) as well since it explicitly checks for it. Hence, we will show that it maintains the properties (a) and (b) for the partition at the start of each iteration. The initial partition satisfies the properties (a) and (b), so we only need to show that this is maintained. First, we observe the following.

Claim. Suppose $\mathcal{P}$ fails the partition-well-linkedness property. Let U^* be the union of clusters that have large intersection with A. Then $|U^*| \leq 2|R|/3$, i.e., property (a) is maintained by partition.

Proof. Since $|A| \leq |R|/2$, and we only include in U^* clusters that have large intersection with A, and they satisfy property (a) previously. ⊛

Next, since we run assure precondition on U^* (line 13), the resulting sub-clusters $\mathcal{Q}$ satisfy the precondition and their size can only decrease. Since the clusters outside U^* are not touched, and they already satisfied the properties, thus, partition indeed maintains the invariant for $\mathcal{P}$ such that each cluster is not too big and that is satisfies the precondition, i.e., properties (a) and (b). Hence, the remaining part is to ensure that partition is efficient.

The main technical part that ensures termination is the following, which relates the sparsity of the cut (A, B) with the sparsity of the cut U^*.

Claim. Given a sparse cut (A, B) w.r.t. π' for the current $\mathcal{P}$ and U^* outputted by partition,

$$\frac{c(\delta_G(U^*))}{\pi'(U^*)} \leq 4\lambda \frac{c(A,B)}{\pi'(A)}.$$

In other words, converting A to a union of the existing clusters costs a factor 4λ in the sparsity.

Proof. ⊛

Not done yet

Now, since (A, B) is a sparse cut with sparsity at most $\sigma\alpha$,

$$\frac{c(A,B)}{\pi'(A)} \leq \sigma\alpha \leq \frac{1}{24\lambda}.$$

Combined with the claim, we have $c(\delta_G(U^*)) \leq \pi'(U^*)/6$. We now see how to use this to prove termination of partition. In particular, we prove that partition terminates in time polynomial in $|S|$ and the maximum integer edge capacity $\max_{e \in E} c(e)$.

Let $E(\mathcal{P}) := \{uv \in E \mid u, v \in S \text{ and } u, v \text{ in different clusters}\}$ be the set of inter-cluster edges induced by the partition $\mathcal{P}$. Observe that since we count each inter-cluster edge twice and the edges leaving S to outside once,

$$\pi'(S) = 2c(E(\mathcal{P})) + c(\delta_G(S))$$

Let $\mathcal{P}'$ be the partition at the end of the iteration (line 14). It suffices to prove that the total capacity of $E(\mathcal{P}')$ is strictly less than that of $E(\mathcal{P})$. We first recall how $\mathcal{P}'$ differs from $\mathcal{P}$:

As previously seen. We remove all clusters of $\mathcal{P}$ contained inside U^* and add clusters of the new partition $\mathcal{Q}$ of S, obtained by running assure precondition on U^*.

Claim. Indeed, $E(\mathcal{P}') < E(\mathcal{P})$.

Proof. From Lemma 3.1.2 (b), we have

$$\sum_{S' \in \mathcal{Q}} c(\delta_G(S')) \leq 2c(\delta_G(U^*)).$$

Let $\pi'' \colon S \to \mathbb{R}_+$ be the new weights induced by the partition $\mathcal{P}'$ that counts the inter-cluster and outgoing edges of $\mathcal{P}'$. We see that by counting, we have

$$\pi''(S) = \pi'(S) - \pi'(U^*) + \sum_{S' \in \mathcal{Q}} c(\delta_G(S')) \leq \pi'(S) - \pi'(U^*) + 2c(\delta_G(U^*)) < \pi'(S),$$

where the last inequality follows from the implication of the previous claim, i.e., $\pi'(U^*) \geq 6c(\delta_G(U^*))$, and $c(\delta_G(U^*)) > 0$. Finally, from the same counting argument as before, $\pi''(S) = 2c(E(\mathcal{P}')) + c(\delta_G(S))$, we see that $E(\mathcal{P}') < E(\mathcal{P})$. ⊛

Hence, the number of inter-cluster edges reduces in each iteration, and we're done. ■

Lecture 13: Real Negative Weights Shortest Path

3.2 Single-Source Shortest Path with Negative Real Weight[2]

8 Oct. 11:00

In the following two lectures, we will discuss a recent breakthrough of SSSP with negative length [HJQ24], which can now be solved in $\widetilde{O}(mn^{4/5})$. This is built upon the recent work that first break the $\widetilde{O}(mn)$ bound that achieves $\widetilde{O}(mn^{8/9})$ [Fin24]. Before we start, we first formally introduce the single-source shortest path problem.

Problem 3.2.1 (Single-source shortest path). Given a graph $G = (V, E)$ with edge capacity $w \colon V \to \mathbb{R}$ and a source vertex $s \in V$, the *single-source shortest path* problem, or *SSSP*, aims to find the shortest path from a source $s \in V$ to t for all $t \in V - s$.

3.2.1 Johnson's Algorithm

There are several classical algorithms for solving the SSSP, including Dijkstra's algorithm (which runs in $O(m + n \log n)$ with Fibonacci heap) and Bellman-Ford algorithm (which runs in $O(mn)$). However, we know that Dijkstra's algorithm can only handle the case when edge lengths are all positive, and hence, for general real edge lengths that are potentially negative, Bellman-Ford algorithm remains the state-of-the-art for decades. The key ingredient of the breakthrough is the classical Johnson's potential re-weighting algorithm. To be precise, let's give a quick review.

As previously seen (Johnson's algorithm). A natural strategy to achieve re-weighting is to consider a *potential* $\phi \colon V \to \mathbb{R}$ and re-weight a (directed) edge $u \to v$ to be $w'(u, v) \coloneqq w(u, v) + \phi(u) - \phi(v)$.

Intuition. Reweigh edge lengths to preserve shortest paths, and also make weights non-negative.

For any u-v path $u = v_0 \to v_1 \to \cdots \to v_k = v$, the new weight of this path is

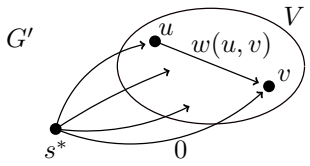
$$\begin{aligned} &w'(v_0, v_1) + w'(v_1, v_2) + \cdots + w'(v_{k-1}, v_k) \\ &= w(v_0, v_1) + \phi(v_0) - \phi(v_1) + w(v_1, v_2) + \phi(v_1) - \phi(v_2) + \cdots + w(v_{k-1}, v_k) + \phi(v_{k-1}) + \phi(v_k) \\ &= w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{k-1}, v_k) + \phi(v_0) - \phi(v_k). \end{aligned}$$

Hence, all u-v paths change by $\phi(v_0) - \phi(v_k)$, i.e., the structure of the shortest paths are preserved. Furthermore, consider adding a dummy source vertex s^* with edge weight 0 for all new edge (s^*, v) for all $v \in V$. Then, consider $\phi(v) \coloneqq d_{G'}(s^*, v)$. We see that for any $u \to v \in E$,

$$w'(u, v) = w(u, v) + \phi(u) - \phi(v) = w(u, v) + d_{G'}(s^*, u) - d_{G'}(s^*, v) \geq 0$$

[2] This section is taught by Kent Quanrud. Guest lectures!

if and only if $w(u,v) + d_{G'}(s^*, u) \geq d_{G'}(s^*, v)$, which is obviously true.



More generally, if we set $\phi(v) \coloneqq d_G(s, v)$ from any source $s \in V$ rather than that dummy extra source, the above still holds. Hence, distance is a good potential function; on the other hand, good potential also helps computing distances: if $w_\phi(e) \geq 0$ for all $e \in E$, then we can use Dijkstra's algorithm on $G_\phi = (V, E)$ with edge weight w_ϕ and solve SSSP.

With the notion of potential, the goal is to find such a "good" potential ϕ such that $w_\phi(e) \geq 0$ for all $e \in E$. We call this neutralization.

Definition 3.2.1 (Neutralize)**.** A potential ϕ *neutralize* a set of negative length edges $F \subseteq E$ if $w_\phi(e) \geq 0$ for all $e \in F$.

With potential, a crucial observation is the following.

Lemma 3.2.1. A graph G has no negative weight cycle if and only if there is a potential ϕ such that $w_\phi(e) \geq 0$ for all $e \in E$.

Thus, the goal is to either detect that G has a negative weight cycle or find a potential that neutralizes all the negative weight edges while not introducing any new negative weight edges. We see that finding good potential and SSSP is pretty much the same problem, and we will switch our perspective between these two. Finally, we make the following remark.

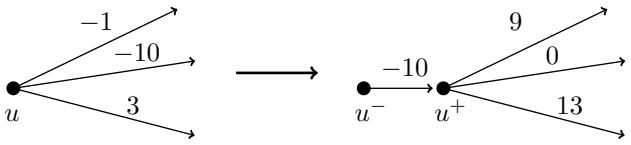
Claim (Verification)**.** One of the useful facts about single-source shortest path trees and distances is that we can check in linear time whether a given tree or set of distances are valid or not.

Proof. We simply relax (update rule of Dijkstra's algorithm) all edges and see if any of the distances change. If they do not change then the distances are valid, otherwise they are obviously not. ⊛

This verification is useful when running randomized algorithms that may work only under the assumption that G has no negative cycle but may still produce some (potentially incorrect) output when G has a negative cycle. Hence, in the following sections about SSSP with negative edge weights, we can assume that there is no negative weight cycle in the graph.

3.2.2 Some Preprocessing

We first note that without loss of generality, we can make sure that the number of negative weight edges is $O(n)$ by splitting each vertex u into u^- and u^+ and making $w(u^-, u^+)$ equal the weight of the most negative weight out-going edges of u (0 if there is no negative weight edge), and adjusting the weights going out of u (now u^+) appropriately, which now are all non-negative.



3.2.3 Preliminary

Before we get started, we need to establish some common terminologies used in this line of works.

Notation (Hop)**.** A *hop* in a walk means a negative edge. Hence, we will say an h-hop shortest path to indicate that this path contains h negative edges.

Lecture 14: Real Negative Weights Shortest Path Continued

Next, we need to deal with the so-called *betweenness reduction*. 10 Oct. 11:00

I'm being lazy. Refer to the slides.

Lecture 15: Scaling Algorithm for Negative Integral SSSP

3.3 Single-Source Shortest Path with Negative Integral Weight 15 Oct. 11:00

As we have seen how to deal with real weights, we consider the case of integer weights. The goal is to check if a graph G has a negative weight cycle or to output shortest path distances from a source s to any other vertex $v \in V$ (which are well-defined when there is no negative weight cycle). As we have mentioned, developed from 50's and 60's, the classical Bellman-Ford algorithm runs in $O(mn)$ time and the Floyd-Warshall runs in $O(n^3)$ time. Similar to the real weighted case, the general strategy of Johnson's algorithm still applies, i.e., we want to find potentials $\phi(v)$ such that each edge (u, v) has non-negative modified weight $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v) \geq 0$.

> **As previously seen.** This can be done by adding a dummy source s^* in the new graph G' and compute $\phi(v) \coloneqq d_{G'}(s^*, v)$ for all $v \in V$.

Unlike the real weighted case, a trick called *scaling* can be exploited, where we don't need to obtain a potential that makes every edge non-negative exactly.[3] Goldberg described such a scaling algorithm for integer weighted case that runs in $O(m\sqrt{n}\log W)$, where $W = \max_{e\in E, w(e)<0}|w(e)|$ is the absolute value of the most negative weight [Gol95]. In a recent breakthrough, a randomized near-linear time scaling algorithm is developed [BNW22; BCF23], which we will now discuss.

3.3.1 Scaling Algorithm

The high-level intuition of the scaling trick is the following.

> **Intuition** (Scaling). When considering integer weights, paths between s and v are more structured in terms of *separation*: the weights of two paths are either the same or differ by at least 1.

In particular, given a graph $G = (V, E)$ with edge weights $w\colon V \to \mathbb{Z}$, consider the scaled weight $w'(e) \coloneqq 2n \cdot w(e)$ for all $e \in E$. Note that the graph with w' will have a negative weight cycle if and only if the graph with the original weight w has.

> **Claim.** Let P and Q be any two s-v paths. Then either $|w'(P) - w'(Q)| = 0$ or $|w'(P) - w'(Q)| \geq 2n > n$. This is also true for any potential ϕ, i.e., either $|w'_\phi(P) - w'_\phi(Q)| = 0$ or $|w'_\phi(P) - w'_\phi(Q)| \geq n$.
>
> **Proof.** If two s-v paths P and Q have different weights, since $w(P), w(Q) \in \mathbb{Z}$, we know that they differ by at least 1. By scaling, $w'(P), w'(Q)$ differ by at least $2n$. ⊛

With this simple observation, we can iteratively apply the following to increase the edge lengths in each iteration with suitable potentials:

> **Theorem 3.3.1** (Scale down). Let $G = (V, E)$ be a graph with edge lengths $w\colon V \to \mathbb{R}$ such that $w(e) \geq -2B$ for all $e \in E$. There is a randomized algorithm called *scale down* such that:
>
> - If it terminates, it outputs a *halving potential* $\phi\colon V \to \mathbb{R}$ such that $w_\phi(e) \geq -B$ for all $e \in E$.
> - If G does not contain a negative weight cycle, then the expected runtime is $O(m\log^4 n)$ and the output is correct. If G has a negative weight cycle, then the algorithm may not terminate.

[3] We should note once again that, if we solve SSSP, a good potential can be obtained. So we can still find a good potential at the end, although our original goal is to solve SSSP.

Note. Even it there is a negative weight cycle, the subroutine from Theorem 3.3.1 can still terminate and will be a valid halving potential.

Assuming we have the subroutine from Theorem 3.3.1, furthermore, if we assume that it will either detect that there is a negative weight cycle, or outputting a halving potential, we can solve SSSP:

Algorithm 3.3: Scaling Algorithm for SSSP

Data: A directed graph $G = (V, E)$ with integral edge length $w \colon V \to \mathbb{Z}$, source $s \in V$
Result: SSSP from the source s or $\perp$, indicating a negative weight cycle

$w' \leftarrow 2nw$
$W' \leftarrow \max_{e \in E \colon w(e) < 0} |w'(e)|$
$\phi \leftarrow 0$ `// Initialize potential for all` $v \in V$

for $i = 1, \dots, O(\log W')$ **do**
  $\phi' \leftarrow$ `Scale-Down`(G, w'_ϕ)
  if $\phi' = \varnothing$ **then** `// Check negative weight cycle`
    return $\perp$
  $\phi \leftarrow \phi + \phi'$ `// Also update` w'_ϕ

$w'' \leftarrow \max(0, w'_\phi)$
$T \leftarrow$`Dijkstra`(G, w'', s) `// Compute shortest path tree`
$d(s, \cdot) \leftarrow$`Shortest-Path-Distance`(T, w) `// Trivial to compute`
return $\{d(s, v)\}_{v \in V}$

Theorem 3.3.2 ([BNW22]). There is a randomized algorithm (Algorithm 3.3) that takes as input a graph $G = (V, E)$ with integral edge weights $w \colon V \to \mathbb{Z}$ and a source s such that

- if G contains a negative cycle, it does not terminate;
- otherwise, it returns valid SSSP distances for s in expected time $O(m \log^4 n \log(nW))$.

Proof. Let $B \coloneqq 2^{\lceil \ln W' \rceil}$.[a] We see that if we run the scale down subroutine $O(\log W') = O(\log nW)$ times, we will find a potential $\phi \colon V \to \mathbb{Z}$ such that $w'_\phi(e) \geq -1$ for all $e \in E$ (if there is a negative weight cycle, it'll be detected by running this subroutine as well). Then, by considering $w'(e) \coloneqq \max(0, w'_\phi(e))$ for all $e \in E$, running the Dijkstra's algorithm on G with edge length w', we can find the correct shortest path in G since w' only perturbs shortest path lengths by at most $n - 1$, while we know that any two paths will have difference at least n if they're not of the same length. This is exactly Algorithm 3.3. ∎

[a] Hence, $w'(e) = 2nw(e) \geq -2B$ for all $e \in E$.

With some slight tweaks, we obtain the following.

Theorem 3.3.3. There is a randomized Monte-Carlo algorithm that takes as input a graph $G = (V, E)$ with integral edge weights $w \colon V \to \mathbb{Z}$ and a source $s \in V$ such that

- if G contains a negative weight cycle, it returns an error message;
- if G does not contain a negative weight cycle, then it returns valid SSSP distances for s with high probability. It may return an error message even if there is no negative weight cycle.

The algorithm runs in $O(m \log^5 n \log(nW))$ time.

Proof. We first consider the following intermediate algorithm. It runs Algorithm 3.3 on input (G, w, s); if it terminates before twice of its expected runtime and returns SSSP distances, then this intermediate algorithm returns the same output. If Algorithm 3.3 takes more than twice of its expected runtime, then the intermediate algorithm is stopped and returns error. By Markov's inequality, this intermediate algorithm has the following properties:

- if G has no negative weight cycle, it returns a correct output with probability $1/2$; and
- if G has a negative weight cycle, it always returns error; and
- it may return error even if G has no negative cycle with probability at most $1/2$.

Then, we can simply run this intermediate algorithm $O(\log n)$ times independently and returning error if all invocations return error. It is easy to see that it has the desired properties. The runtime is then $O(\log n)$ times the expected run-time of Algorithm 3.3, which is $O(m \log^5 n \log(nW))$. ■

Remark. The Monte-Carlo algorithm described in Theorem 3.3.3 can be viewed as almost what we would like because it returns valid SSSP distances with high probability when they exist. But it may also return error with a small probability even when there is no negative weight cycle. Ideally, we would also like an algorithm that can detect and output a negative cycle with high probability when G has one. Suppose we had such an algorithm, then we can run the two algorithms and get what we want. We will discuss such an algorithm that finds a negative weight cycle at the end.

Hence, the only difficult (and remaining) part is to prove Theorem 3.3.1.

3.3.2 Preliminary Tools

Now, we can start building up toward proving Theorem 3.3.1. This requires some preliminary facts. Firstly, the intuition is to reduce the problem to directed acyclic graph (DAG).

Intuition. DAG has an equivalent role for directed graphs as tree for undirected graphs in some sense. In particular, we know that for shortest path, DAG is easy.

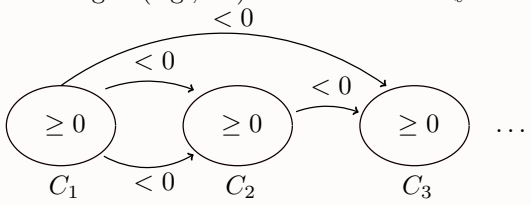
To further motivate making a directed graph a DAG, consider the following.

Problem 3.3.1 (Feedback arc set). Given a directed graph $G = (V, E)$ with positive edge weight $w \colon V \to \mathbb{R}_+$. The *feedback arc set* problem asks for removing the least weight set of edges such that the resulting graph is a DAG.

The first fact is that if the graph induces a nice DAG, then we can find a good potential efficiently.

Lemma 3.3.1 (Fix DAG edge). Suppose $G = (V, E)$ is a directed graph with edge weight w such that in each strongly connected component (SCC), the edge weights are non-negative. Then one can compute a potential $\phi \colon V \to \mathbb{R}$ such that $w_\phi(e) \geq 0$ for all $e \in E$ in $O(m + n)$ time.

Proof. Consider the DAG[a] G^{SCC} associated with contracting all the SCCs $\{C_i\}_{i=1}^{\ell}$ to nodes $\{v_i\}_{i=1}^{\ell}$. Then by sorting v_i's in G^{SCC} in the topological order in $O(m+n)$, we can neutralize edges between C_i's by adding $-(\ell - 1)L$ for some big L (e.g., W) in the ℓ^{th} SCC C_ℓ.



This works since for any $u \to v$ in C_i, change of u and v's potential are the same, hence $w(u, v) = w_\phi(u, v)$. On the other hand, for edges crossing clusters, say from C_i to C_j for $i < j$, w_ϕ will increase by $(j - i)L$. Hence, if L is big enough, these negative edges can be all made non-negative. ■

[a] It's a fact that this will form a DAG.

Another useful tool turns out to be the low-diameter decomposition. We have seen low-diameter decomposition in the context of undirected graphs and metrics. Here, we're interested in directed graphs and distances. There were implicit in algorithm for cut and flow problems for symmetric demands [Kle+97; Sey95; Eve+00], but the utility of their randomized variants have only recently been explored.

Intuition. Since reachability is asymmetric in directed graphs, in several problems of interest, we are focus on diameter of SCC.

If we also think of cuts as removing subsets of edges rather than edges leaving a set, then a nature formulation of the low-diameter decomposition of a directed graph with *non-negative* weights $w\colon E \to \mathbb{R}_+$ is to remove a subset of edges $E' \subseteq E$ such that the diameter of each SCC in $G - E'$ is at most some given parameter D. The diameter of an SCC $C \subseteq V$ is $\max_{u,v\in C} d_G(u,v) \le D$ for all $u, v \in C$ in G. This is the notion of "weak diameter" since we are using distances in G.

Notation (Strong diameter). The *strong diameter* guarantee is where $d_C(u,v) \le D$ for all $u, v \in C$ where $d_C(u,v)$ is the distance using only edges in $G[C]$.

The following states that there exists an efficient randomized algorithm to compute the low-diameter decomposition for directed graph.

Theorem 3.3.4 (Low-diameter decomposition for directed graph [BNW22]). There is a randomized algorithm such that given a directed graph $G = (V, E, w)$ where $w\colon E \to \mathbb{R}_+$ and a diameter bound $D \ge 0$, outputs a set of edges $E' \subseteq E$ such that in $O(m \log^2 n + n \log^3 n)$ time,

(a) all the SCCs in $G - E'$ has weak diameter $\le D$, and

(b) $\Pr(e \in E') \le O(w(e) \log^2 n / D + n^{-10})$.[a]

[a] The n^{-10} is negligible and is for technical reasons to obtain a fast algorithm. We will ignore it for convenience.

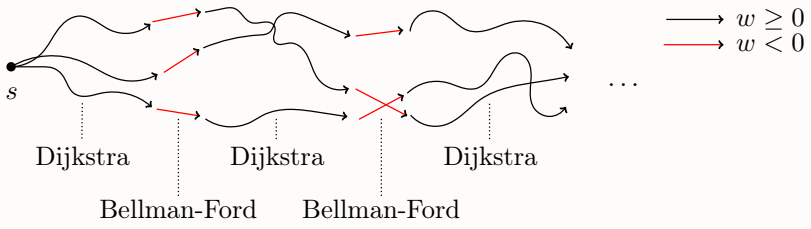
Intuition. The low-diameter decomposition allows us to get to the case of Lemma 3.3.1.

Remark. [BCF23] describes an algorithm with strong diameter guarantee with the cutting probability increased to $O(\log^3 nw(e)/D)$. They also describe a different decomposition procedure which gives a faster algorithm while not necessarily giving a low-diameter decomposition.

Lastly, we note that if we have some bound on the number of negative edges shortest paths will have, we can actually solve SSSP by a combination of Dijkstra's algorithm and Bellman-Ford algorithm.

Lemma 3.3.2 (Fix bad edge [BNW22]). Suppose $G = (V, E)$ is a directed graph with edge weight $w\colon V \to \mathbb{R}$ has no negative weight cycle. Suppose every s-v shortest path is at most κ_v-hops for every $v \in V$ from the source s. Then SSSP (with the corresponding potential) can be solved in $O\big((n + \sum_{v\in V} |\delta^+(v)|\kappa_v) \log n\big)$ time.

Proof. We essentially run Dijkstra and Bellman-Ford alternatively on a modified graph. The intuition is that Dijkstra computes the correct shortest path length for positive edges, while one iteration of Bellman-Ford fix one negative edge.



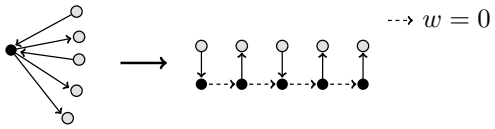
To actually bound the running time to be the sum of $\sum_{v\in V} \kappa_v$, one will need to implement this algorithm carefully with *laziness*, i.e., nodes participate in the computation only when necessary:

Claim. After i iterations, we get correct distance $d(s,v)$ if $\kappa_v \le i$. Hence, v contributes to runtime for κ_v iterations.

This leads to the desired bound.[a] ■

[a] This can be proved with only Dijkstra by the layering graph construction as we have seen in the Homework 1.

Remark. We can make the graph to have in and out-degree $O(1)$ for all v by blow-up n to $\Theta(m)$:



Hence, the algorithm described in Lemma 3.3.2 runs in $O\big((m + \sum_{v \in V} \kappa_v) \log n\big)$ time.

Lecture 16: Low-Diameter Decomposition for Directed Graph

3.3.3 The Scale Down Algorithm via Hop Reduction and Fixing

17 Oct. 11:00

Now, we can start proving Theorem 3.3.1, i.e., implementing the scale down algorithm. Recall that we're given a directed graph $G = (V, E)$ with edge lengths $w \colon V \to \mathbb{R}$ such that $w(e) \geq -2B$ for all $e \in E$. We want to design an algorithm such that it outputs a potential where $w_\phi(e) \geq -B$ for all $e \in E$.

Note. Following the earlier discussion, we will assume that there is no negative weight cycle.

Consider two new graphs $G^B = (V, E, w^B)$ such that

$$w^B(e) = \begin{cases} B + w(e), & \text{if } w(e) < 0; \\ w(e), & \text{if } w(e) \geq 0, \end{cases}$$

and $G^B_+ = (V, E, w^B_+)$ where $w^B_+(e) = \max(0, w^B(e))$. For any subgraph H of G, we will use H^B and H^B_+ to refer to the corresponding subgraphs of G^B and G^B_+.

Claim. If a potential ϕ neutralizes all negative edges in G^B without introducing additional negative edges, then $w_\phi(e) \geq -B$ for all $e \in E$.

Proof. Fix an edge $e = (u, v)$. If $w(e) < 0$, there are two cases:

- If $w(e) \geq -B$: then $w^B(e) \geq 0$ and by our assumption, $w^B_\phi(e) \geq 0$, implying

$$w_\phi(e) = \phi(u) + w(e) - \phi(v) = \phi(u) + w^B(e) - B - \phi(v) = w^B_\phi(e) - B \geq -B.$$

- If $w(e) < B$: then $w^B(e) = w(e) + B < 0$ and $w^B_\phi(e) \geq 0$. It's obvious that $w_\phi(e) \geq -B$.

If $w(e) \geq 0$, then from our assumption this will not be affected since $w^B(e) = w(e)$. ⊛

Hence, our goal now is to neutralize all negative edges in G^B without introducing any new ones. It is not quite obvious why it'll be easier to work with G^B.

Intuition. If G does not contain a negative weight cycle, then G^B is making the problem easier. [BCF23] uses a slightly different scheme which ensures that the minimum mean cycle length is ≥ 1, which formalizes more explicitly the change in the graph by adding B to the negative weight edges.

To neutralize negative edges in G^B, we again take inspiration from Johnson's algorithm: i.e., let s^* be a dummy node with 0 weight edges from s^* to every $v \in V$ in G^B, which we denote this new graph as $G^B + s^*$. Since s^* is connected to each vertex v with edge of weight 0 the shortest path distances from s^* are at most 0. Thus, consider an s^*-v path $P_{s^*,v}$ of non-zero hops $\kappa_v > 0$, the weight of any prefix (say, $P_{s^*,u} \subseteq P_{s^*,v}$) of $P_{s^*,v}$ is at most 0, since otherwise we can go from s^* directly to that intermediate vertex u to reduce the cost.

Now, the key idea is to use low-diameter decomposition for directed graph with an appropriate parameter such that the number of hops needed is reduced, i.e., hop reduction.

Intuition. From Lemma 3.3.2, if the number of hops for s^*-v shortest paths is small in $G^B + s^*$, or even if the average is small, then we can solve the problem in near-linear time. We aim to reduce to this with low-diameter decomposition for directed graph, recursion, and also fixing edges that are easy to fix along the way (i.e., the case of Lemma 3.3.1).

Since low-diameter decomposition requires non-negative edge lengths, we consider applying it to G^B_+.

Lemma 3.3.3. Let $G = (V, E)$ be a graph with edge length $w \colon V \to \mathbb{R}$ such that $w(e) \geq -2B$ for all $e \in E$. Let $\kappa_{G^B}(v)$ to be the maximum number of hops of any s^*-v shortest path in $G^B + s^*$, and $\kappa_{G^B} \coloneqq \max_{v \in V} \kappa_{G^B}(v)$. Suppose $\kappa \geq \kappa_{G^B}$, and we run low-diameter decomposition for directed graph on G^B_+ with $D = \kappa B/2$ to get an output E'. Then,

(a) for all $v \in V$, $\mathbb{E}[|P_v \cap E'|] \leq O(\log^2 n)$, where P_v is a $\kappa_{G^B}(v)$-hop s^*-v shortest path in G^B;

(b) for all SCC H in $G - E'$, $\kappa_{H^B} \leq \kappa/2$, where κ_{H^B} concerns with $H^B + s$, similar to κ_{G^B}.

Proof. We prove (b) first. Assume that H is an SCC in $G - E'$ such that there is one shortest path $P_v^{H^B} = s^* \to u \to \cdots \to v$ contains entirely in H^B using more than $\kappa/2$ negative edges w.r.t. $w^B\big|_H$. Denote the corresponding u-v path as $P \coloneqq P_v^H \setminus \{s^*\}$. Observe that there is a v-u walk Q of length at most D w.r.t. w^B_+ from the low-diameter decomposition guarantee, i.e., $w^B_+(Q) \leq D$. Hence, $w(Q) \leq D$ as well. However, we see that $Q \cup P$ is a cycle, furthermore, its weight is negative:

$$w(Q \cup P) = w(Q) + w(P) \leq D - B \cdot \left(\frac{\kappa}{2} + 1\right) = \frac{\kappa B}{2} - \frac{\kappa B}{2} - B < 0,$$

since we assume that P contains more than $\kappa/2$ negative edges w.r.t. w^B, with the fact that $w^B(P) = w^B(P_v^H) \leq w^B(\{s^* \to v\}) = 0$, hence its original weight is at most $(\kappa/2 + 1) \cdot (-B)$.

We now prove (a), consider any s^*-v shortest path P_v for some v in G^B of hops $\kappa_{G^B}(v)$. From the same reason as above, we have $w^B(P_v) \leq 0$, hence $w^B_+(P_v) \leq \kappa B$. By linear of expectation,

$$\mathbb{E}[|P_v \cap E'|] \leq \frac{w^B_+(P_v)}{D} O(\log^2 n) \leq \frac{\kappa B}{D} O(\log^2 n) = O(\log^2 n)$$

since $\Pr(e \in E') \leq O(w(e) \log^2 n / D)$ from the low-diameter decomposition guarantee. ∎

Intuition. Lemma 3.3.3 guarantees not only the hop-reduction, but also that (in expectation) the number of edges that prevent the reduction to Lemma 3.3.1 is small per shortest path.

Remark. Consider a vertex v and an s^*-v path P_v with κ negative weight edges in G^B. Why does that path not exist in H^B since we proved that in H^B the shortest path to v can only contain $\kappa/2$ negative weight edges? The point is that suppose P_v has (s, u') as its first edge. Then the above proof shows that there cannot be a (v, u') path in G with weight at most $D = \eta B/2$. Thus, u' and v cannot be in the same SCC with diameter bound D. Thus, the shortest path to v inside H^B is not going to be able to use u'.

With Lemma 3.3.3, it's quite natural in retrospect to consider a recursive algorithm that reduces the hop length, recurses and fixes the remaining edges. Recall that the goal is to find a potential ϕ that neutralizes all negative weight edges in G^B without adding new ones. The algorithm starts with an upper bound of κ_{G^B}, say n, and computes a set of edges E' using the low-diameter decomposition for with $D = \kappa B/2$. For each SCC H in $G - E'$, we recurse since $\kappa_{G^B} \leq \kappa/2$. This recursively yields a potential ϕ_1^H on vertices of H^B that neutralizes all negative edges inside H^B (without introducing new ones). Since the potentials are found separately for each SCC, and these SCCs partition the vertex set, we can find an overall potential ϕ_1 that neutralizes all negative edges inside the SCCs. Note that $G - E'$ consists of other edges which are not inside any SCC, and we call them the *DAG edges* since they induce a DAG on the SCCs, as mentioned in Lemma 3.3.1. Then, we simply use Lemma 3.3.1 to fix these DAG edges, leaving edges in E' that are still potentially negative. For those edges, we simply fix them using some sort of brute-force way: with Lemma 3.3.2. But as shown in Lemma 3.3.3 (a), in G^B, there can be at most $O(\log^2 n)$ negative weight edges in expectation, which suffice for efficiency.

Algorithm 3.4: Scale-Down Algorithm

Data: A directed graph $G = (V, E)$ with integral edge length $w \colon V \to \mathbb{Z}$, $\kappa \geq \kappa_{G^B}$
Result: A potential ϕ

1 **if** $\kappa \leq 2$ **then** // Base case
2 $\quad d(s^*, \cdot) \leftarrow$ `Brute-Force`$(G^B + s^*, w^B, s^*)$ // Lemma 3.3.2
3 $\quad \phi \leftarrow d(s^*, \cdot)$
4 $\quad$ **return** ϕ
5
6 $E' \leftarrow$ `Low-Diameter-Decomposition`$(G^B_+, \kappa B/2)$
7 $\{H_i\}_{i=1}^{\ell} \leftarrow$ `Strongly-Connected-Component`$(G - E')$
8
9 **for** $i = 1, \ldots, \ell$ **do** // Recurse
10 $\quad \phi_1^{H_i} \leftarrow$ `Scale-Down`$(H_i, w|_{H_i}, \kappa/2)$
11
12 $\phi_1 \leftarrow \sum_{i=1}^{\ell} \phi_1^{H_i}$
13 $\phi_2 \leftarrow \phi_1 +$ `Fix-DAG-Edges`$((G^B - E')_{\phi_1}, w^B_{\phi_1})$ // Lemma 3.3.1
14 $\phi_3 \leftarrow \phi_2 +$ `Fix-Bad-Edges`$((G^B)_{\phi_2}, w^B_{\phi_2})$ // Lemma 3.3.2
15
16 **if** $w^B_{\phi_3}(e) < 0$ *for any* $e \in E$ **then** // Check validity
17 $\quad$ **while** 1 **do** // Infinite loop
18 $\quad\quad 1 + 1$
19 **else**
20 $\quad$ **return** ϕ_3

We now prove Theorem 3.3.1.

Proof of Theorem 3.3.1. We will show that running Algorithm 3.4 with $\kappa = n$ satisfies the requirement. Firstly, the correctness essentially follows from Lemma 3.3.3, the properties of potentials, and the induction. Note that, under the assumption that G has no negative weight cycle, each step of Algorithm 3.4 correctly computes its output and hence Algorithm 3.4 becomes a correct Las Vegas algorithm. If G has a negative weight cycle, then the algorithm is not guaranteed to terminate. Since we check validity of ϕ_3 before returning it (line 16), we ensure that it returns a valid output if it terminates.

We now analyze the time complexity. Note that Algorithm 3.4 has recursion depth $O(\log n)$. We consider the expected time outside the recursion. In the base case (line 1), it is easy to see that the running time is $O(m \log n)$ via Lemma 3.3.2. The low-diameter decomposition takes $O(m \log^3 n)$ (line 6) as stated in Theorem 3.3.4.[a] When fixing the DAG edges in line 13, we consider the linear time ($O(m + n) = O(m)$) algorithm implied by Lemma 3.3.1. In the last step (line 14), we use the algorithm implied in Lemma 3.3.2. Recall that by Lemma 3.3.3, we have $\mathbb{E}[|P_v \cap E'|] = O(\log^2 n)$. Thus, by linearity of expectation, $\mathbb{E}[\sum_{v \in V} |P_v \cap E'|] = O(n \log^2 n)$. Thus, the expected running time of the algorithm from Lemma 3.3.2 is $O(m \log^3 n)$.

Finally, in each recursive call, we reduce the parameter κ by half and the total sum of the edges inside the SCCs is at most m. Thus, the total expected time of the algorithm is simply the depth of the recursion times the time outside the recursion, i.e., $O(m \log^4 n)$. ■

[a] Note that we have the preprocessing step such that $n = \Omega(m)$.

3.3.4 Finding a Negative Cycle

Finally, we discuss a scaling algorithm to find a negative weight cycle, assuming we have an algorithm $\mathcal{A}$ such that it will detect when G has a negative weight cycle, and otherwise outputs a valid potential that neutralizes all negative weight edges without introducing new ones. While we don't have a deterministic algorithm of $\mathcal{A}$, but we do have the algorithm implied in Theorem 3.3.3 which we can act as a substitute [BNW22; BCF23]. Let $G = (V, E)$ be a directed graph with integral edge weights $w \colon V \to \mathbb{Z}$. Say $\mathcal{A}$ has detected that G contains a negative cycle. We start scaling all weights by n^3, i.e., consider $w_0(e) = n^3 w(e)$ for all $e \in E$. Let G_0 be this new graph.

Note. If G has a negative cycle, then G_0 has a negative cycle with weights $\leq -n^3$.

Given G and an integer $M \geq 0$, we define a graph G^{+M} as the graph obtained by adding M to each edge weight. Note that we add M to all edges, not just negative weight edges as we did earlier in defining G^B. Let M^* be the smallest integer such that $G_0^{+M^*}$ does not have a negative cycle. It means that $G_0^{+(M^*-1)}$ has a negative cycle.

Claim. $M^* \geq n^2$ and $M^* \leq n^3 W$. Given access to $\mathcal{A}$ that can detect a negative cycle using binary search, one can find M^* using $O(\log(nW))$ call to which.

Proof. Since there is a negative weight cycle C in G_0 of weight $\leq -n^3$, to make it positive, we need to add at least n^2 to each edge since $|C| \leq n$. The second part is obvious. ⊛

The following claim is also easy to see.

Claim. Since $G_0^{+M^*}$ does not have a negative weight cycle, there is a potential $\phi\colon V \to \mathbb{Z}$ such that ϕ neutralizes all negative weight edges in $G_0^{+M^*}$ without introducing new ones and this can be computed by $\mathcal{A}$.

The main lemma that leads to the algorithm is the following.

Lemma 3.3.4. Let ϕ be a potential that neutralizes the negative weight edges in $G_0^{+M^*}$, and $E' = \{e \in E \mid \left(w_0^{+M^*}\right)_\phi(e) > n\}$. Then $G_1 \coloneqq (V, E \setminus E')$ contains a cycle, and any cycle in G_1 is a negative cycle in G.

Proof. There is a negative cycle C in $G_0^{+(M^*-1)}$ which clearly is also a negative weight cycle in G. We see that $w_0^{+M^*}(C) = w_0^{+(M^*-1)} + |C|$, thus, $w_0^{+M^*}(C) \leq n$. Note that ϕ neutralizes all negative weights in $G_0^{+M^*}$, hence all edge weights of C become non-negative w.r.t. ϕ, but the total weight of C does not change w.r.t. ϕ as potentials do not change cycle weights. This means that $\left(w_0^{+M^*}\right)_\phi(C) \leq n$, and it contains only non-negative edge weights. All these imply that any edge in C must have weight at most n w.r.t. $\left(w_0^{+M^*}\right)_\phi$, hence $C \cap E' = \varnothing$, which means that all edges of C survives in G_1, hence G_1 must have a cycle.

Now, we argue that any cycle in G_1 is a negative weight cycle in G. Let C be an arbitrary cycle in G_1. Since C does not have any edges from E' , we have $\left(w_0^{+M^*}\right)_\phi(C) \leq n|C| \leq n^2$. Since potentials do not change cycle weights, we therefore have $w_0^{+M^*}(C) \leq n^2$. However, this implies

$$w_0(C) = w_0^{+M^*}(C) - M^*|C| \leq w_0^{+M^*}(C) - 2M^* \leq n^2 - 2n^2 < 0,$$

where we use the fact that $|C| \geq 2$ and $M^* \geq n^2$. Hence, the weight of C in G_0 is negative, which implies that it is also negative in G as well. ■

Based on Lemma 3.3.4, we see that the following algorithm outputs a negative cycle assuming access to $\mathcal{A}$. It invokes $\mathcal{A}$ only $O(\log(nW))$ times.

Algorithm 3.5: Find Negative Weight Cycle

Data: A directed graph $G = (V, E)$ with integral edge length $w\colon V \to \mathbb{Z}$ containing a negative weight cycle
Result: A negative weight cycle C

$w_0 \leftarrow n^3 w$
$G_0 \leftarrow (V, E, w_0)$
Find the smallest $M^* > 0$ such that $G_0^{+M^*}$ has no negative weight cycle `// Binary search`
$\phi \leftarrow \mathcal{A}(G_0^{+M^*}, w_0^{+M^*})$ `// Neutralize` $G_0^{+M^*}$
$E' \leftarrow \{e \in E \mid \left(w_0^{+M^*}\right)_\phi(e) > n\}$
$C \leftarrow$ `Find-Cycle`$(G - E')$
return C

If we use the algorithm implies in Theorem 3.3.3 in place of $\mathcal{A}$, the resulting algorithm may make

mistakes and fail, but we can detect failure in finding a negative cycle, and thus we can obtain a Monte Carlo algorithm that can find a negative cycle with high probability if G has one.

Chapter 4

Multiplicative Weight Update

Lecture 17: Approximating Linear Programs

22 Oct. 11:00

In this chapter, we try to bridge the field of combinatorial (discrete) optimization and continuous optimization from the lens of approximating positive linear programs using multiplicative weight update. In particular, we will look into the class of *positive* linear programs, which is a large and interesting class of linear programs that arise in combinatorial optimization.

> **Intuition.** Some useful properties that can be exploited algorithmically via iterative methods coming from first order optimization methods and this leads to fast approximation solutions.

Although these methods are old in optimization, the formal analysis with provable worst-case guarantees for concrete problems can be traced to the work of efficient algorithms for multi-commodity flows [SM90] which was later abstracted to general classes of linear programs [PST95; GK94]. Parallel algorithms were also derived via these methods [LN93], and there have been many developments since then.

4.1 Positive Linear Program

Positive linear programs, as the name suggests, are linear programs where the input to the linear program consists of positive numbers. This allows one to discuss relative approximations.

4.1.1 Packing, Covering and Mixture of Both

There are three types of positive linear programs that we commonly work with.

Definition 4.1.1 (Packing linear program)**.** Given data $c \in \mathbb{R}^n_+$, $A \in \mathbb{R}^{m\times n}_+$, and $b \in \mathbb{R}^m_+$, a *packing linear program* with n variables and m non-trivial constraints is of the form

$$\begin{aligned} \max\ & c^\top x \\ & Ax \le b; \\ & x \ge 0. \end{aligned}$$

Definition 4.1.2 (Covering linear program)**.** Given data $c \in \mathbb{R}^n_+$, $A \in \mathbb{R}^{m\times n}_+$, and $b \in \mathbb{R}^m_+$, a *covering linear program* with n variables and m non-trivial constraints is of the form

$$\begin{aligned} \min\ & c^\top x \\ & Ax \ge b; \\ & x \ge 0, \end{aligned}$$

Oftentimes, we have a mixture of both.

Definition 4.1.3 (Mixed packing and covering linear program)**.** Given data $A \in \mathbb{R}_+^{m_1 \times n}$, $b \in \mathbb{R}_+^{m_1}$, $C \in \mathbb{R}_+^{m_2 \times n}$, and $d \in \mathbb{R}_+^{m_2}$, a *mixed packing and covering linear program* with n variables and $(m_1 + m_2)$ non-trivial constraints is in the form of feasibility problem of inequalities[a]

$$\begin{aligned} \max\ & 0 \\ & Ax \leq b; \\ & Cx \leq d; \\ & x \geq 0. \end{aligned}$$

[a] This is general enough since we can put the objective (either maximization or minimization) into the constraints.

4.1.2 Explicit and Implicit Linear Program

Definition 4.1.4 (Explicit linear program)**.** An *explicit linear program* is one in which all the data is specified in terms of the variables, constraints and the non-negative entries in the inequalities.

For an explicit linear program, we will assume that the representation is given in the sparse form, and we will usually let n denote the number of variables, m the number of constraints, and N the number of non-zeroes in the constraint matrices.

Remark. Hence, the input size of an explicit specified positive linear program is $\Theta(N + m + n)$.

On the other hand, we also have the so-called implicit linear program.

Definition 4.1.5 (Implicit linear program)**.** An *implicit linear program* is one in which we have an underlying data such as a graph or geometric object and a linear program formulation that is specified implicitly based on that data.

We note that for implicit linear programs, they can have exponentially many variables or constraints, and often they can be solved efficiently or approximately via Ellipsoid method. In such settings, we will not write down the explicit linear program, and we will seek iterative methods that have a running time that depends on the size of the *original input* that defines the linear program.

Note. The disadvantage of the explicit linear program is that it has many variables and constraints and typical explicit linear program solvers use memory that is quadratic in the number of variables to do matrix operations while solving the linear program via the simplex or interior point method, and this makes the memory a bottleneck.

4.1.3 Examples

Let's see some examples of packing linear programs.

Example (Maximum weight bipartite matching)**.** Given a bipartite graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, the following linear program gives an exact solution to the maximum weight bipartite matching:

$$\begin{aligned} \max\ & \sum_{e \in E} c(e) x_e \\ & \sum_{e \in \delta(u)} x_e \leq 1 \quad \forall u \in V; \\ & x_e \geq 0 \qquad\quad \forall e \in E. \end{aligned}$$

One can see that this is a explicit packing linear program with $N = 2|E|$, $n = |E|$, and $m = |V|$.[a] Although this linear program is mainly used for bipartite matching, it is sometimes useful even for non-bipartite graphs. One can show that its integrality gap is $2/3$ for general graphs.

[a] Note that the notation of n, m is different from the usual graph usage.

Example (Tree packing). Recall the tree packing linear program, where the goal is to pack spanning trees into the capacity of a given graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$:

$$\begin{aligned} \max \quad & \sum_{T \in \mathcal{T}_G} y_T \\ & \sum_{T \ni e} y_T \le c(e) \qquad \forall e \in E; \\ & y_T \ge 0 \qquad\quad \forall T \in \mathcal{T}_G. \end{aligned}$$

This is an implicit packing linear program with an exponential number of variables and $m = |E|$ constraints. As mentioned in Theorem 1.2.2, this can be (approximately) solved efficiently without the Ellipsoid method (which runs in polynomial time, but it is considered impractical).

Example (Maximum multi-commodity flow). We considered the maximum multi-commodity flow problem as a relaxation for the multi-min-cut problem. Given a graph $G = (V, E)$ with non-negative edge capacities $c\colon E \to \mathbb{Q}_+$ and k source-sink pairs $\{(s_i, t_i)\}_{i=1}^k$. Let $\mathcal{P}_{s_i,t_i}$ denote the set of all s_i-t_i paths in G. We have a variable x_P for each path $P \in \bigcup_{i=1}^k \mathcal{P}_i$ to denote the amount of flow that is being routed on path P, and we want to maximize the total flow sent between all pairs:

$$\begin{aligned} \max \quad & \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_{s_i,t_i}} x_P \\ & \sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_{s_i,t_i} \\ P \ni e}} x_P \le c(e) \qquad\quad \forall e \in E; \\ & x_P \ge 0 \qquad\qquad P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}. \end{aligned}$$

This is also an implicit packing linear program. We can also write it as an explicit linear program with variables $f(e, i)$, which is the amount of flow on e for pair i:

$$\begin{aligned} \max \quad & \sum_{i=1}^{k} \left[\sum_{e \in \delta^+(s_i)} f(e,i) - \sum_{e \in \delta^-(s_i)} f(e,i) \right] \\ & \sum_{i=1}^{k} f(e,i) \le c(e) \qquad\qquad \forall e \in E; \\ & \sum_{e \in \delta^-(u)} f(e,i) = \sum_{e \in \delta^+(u)} f(e,i) \qquad u \ne s_i, t_i \text{ for } i \in [k]. \end{aligned}$$

We see that the flow conservation constraints make this linear program not positive anymore.

As for the covering linear programs, we first introduce the well-known set cover problem and its special case called vertex cover.

Problem 4.1.1 (Set cover). Given a collection of n subsets $\mathcal{S} = \{S_i\}_{i=1}^n$ of S such that each $\mathcal{S}_i \subseteq S$ is of size m and has a cost c_i. The *set cover* asks to find a minimum cost sub-collection of $\mathcal{S}$ whose union is S.

Problem 4.1.2 (Vertex cover). Given a graph $G = (V, E)$ with edge capacity $w\colon E \to \mathbb{R}_+$, the *vertex cover* is a special case of set cover with $S = E$ and $\mathcal{S} := \{S_v = \delta(v)\}_{v \in V}$ with unit cost for each S_v.

Example (Set cover). Define an $m \times n$ matrix A where $A_{ij} = 1$ if $j \in S_i$, 0 otherwise. Then, the

linear program relaxation for set cover is of the form

$$\begin{aligned} \min\ & c^\top x \\ & Ax \geq 1; \\ & x_v \geq 0 \qquad \forall v \in V. \end{aligned}$$

This is clearly a covering linear program. It is known that the integrality gap is $O(\log m)$, and the integrality gap for vertex cover is 2.

Example (Covering integer program)**.** The *covering integer program* (CIP) generalizes set cover. A CIP is essentially an integer version of a covering linear program:

$$\begin{aligned} \min\ & c^\top x \\ & Ax \geq b; \\ & x \in \{0,1\}^n. \end{aligned}$$

We work with the linear program relaxation which is a covering problem. When A, b are arbitrary, the integrality gap of the natural linear program for CIP can be as large as m. One needs to add knapsack cover inequalities to strengthen the linear program and then the linear program becomes much more complex but nevertheless one can solve it fast using the multiplicative weight update methods. See [CQ19] and references to work on CIPs.

Finally, we see some examples on the mixed packing and covering linear programs.

Example (Covering integer program with bounded constraints)**.** One can also add bound constraint on CIPs where a variable x_i can be at most d_i. Then we get packing constraints and the linear program becomes a mixed packing and covering linear program, albeit in a simple form

$$\begin{aligned} \min\ & c^\top x \\ & Ax \geq b; \\ & x \leq d; \\ & x \in \mathbb{Z}_+^n. \end{aligned}$$

This naive formulation is bad in terms of integrality gap; one needs knapsack cover inequalities.

Example (Maximum concurrent flow)**.** We have seen the maximum concurrent flow linear program in the context of the sparsest cut problem. If we guess λ, then it becomes a constant, and we get a mixed packing and covering linear program. To do this, we can do a binary search for λ.

$$\begin{aligned} \max\ & \lambda \\ & \sum_{P \in \mathcal{P}_{s_i,t_i}} y_P \geq \lambda D_i \qquad \forall i = 1, \dots, k; \\ & \sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_{s_i,t_i} \\ P \ni e}} y_P \leq c(e) \qquad \forall e \in E; \\ & y_P \geq 0 \qquad \forall P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}; \\ & \lambda \geq 0, \end{aligned}$$

Example (Densest subgraph)**.** Another interesting mixed packing and covering linear program comes up when solving the densest subgraph problem [BGM14].

4.1.4 Known Approximation Results for Explicit Packing Linear Programs

Finally, we list out several results known for explicit positive linear programs. We refer to [Qua19; Wan17] for a good overview. Here, we state a few high-level results for sequential models.

- For mixed packing and covering linear programs, there is a multiplicative weight update based algorithm that runs in deterministic $O(N \log N/\epsilon^2)$ time and outputs a $(1-\epsilon)$-approximate feasible solution [You14]. This implies a $(1-\epsilon)$-approximation algorithm for packing and covering in $O(N \log N/\epsilon^2)$ time.
- For packing, there is a randomized algorithm that yields a $(1-\epsilon)$-approximation feasible solution in $O(N \log N \log(1/\epsilon)/\epsilon)$ time [AO15]. A similar time bound for covering is also known [WRM15].

4.2 Multiplicative Weight Update Method

The *multiplicative weight update* (MWU) is a meta-algorithm that has many applications and arises in a number of areas, even though the central analysis is very similar. The survey of Arora, Hazan, and Kale [AHK12] outlines the utility of seeing the general approach. Here, we are interested in applications of MWU to solving positive linear programs. We will follow the ideas and exposition from [CJV15].

> **Note.** Even within offline optimization, the use of MWU is varied, and it is not always easy to figure out the similarities and differences even among closely related papers. For example, although one can derive some of these via the expert framework outlines in [AHK12], there are some limitations as well as the overhead of introducing the expert framework.

4.2.1 Convex Optimization, Lagrangian Relaxation, and Soft-Max Trick

While our goal is to solve linear programs, we start by considering convex programs.

> **Intuition.** It makes some ideas and notations cleaner and more general, and also indicates where we use linearity and positivity when we ant refined results.

Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be a concave function and let $g_i\colon \mathbb{R}^n \to \mathbb{R}$ for all $i \in [m]$ be a set of convex functions, and let P be a convex set in $\mathbb{R}^n$.[1]

> **Note.** We will assume that all functions are continuous and bounded in the domain of interest, and ignore the technical difficulties that arise when considering unbounded functions and domains.

Consider the following optimization problem which generalizes the setting of packing linear programs:

$$\begin{aligned} \max\ & f(x) \\ & g_i(x) \le 1 \quad \forall i \in [m]; \\ & x \in P. \end{aligned}$$

This is a constrained convex program. Unconstrained convex minimization is itself a non-trivial problem and typically addresses it via gradient descent and other methods. Here, we use it as a modeling tool for explanation purposes and eventually work with linear functions. The first idea is to view the multiple constraints as a single constraint $h(x) \le 1$ where $h(x) = \max_{i\in[m]} g_i(x)$:

$$\begin{aligned} \max\ & f(x) \\ & \max_{i\in[m]} g_i(x) \le 1; \\ & x \in P; \end{aligned} \quad \Leftrightarrow \quad \begin{aligned} \max\ & f(x) \\ & h(x) \le 1; \\ & x \in P. \end{aligned}$$

From convex analysis, we know that $h(x) = \max_{i\in[m]} g_i(x)$ is still convex, but not smooth anymore. A heuristic way is to make it smooth by considering the soft-max, a smooth-out version of max. Specifically,

[1] Even though we typically use P as the non-negative orthant in $\mathbb{R}^n$, we can usually restrict the domain to be a box of sufficiently large size in the non-negative orthant. We will come back to this later.

for a parameter $\eta > 0$, the soft-max function is defined as

$$g_\eta(x) \coloneqq \text{soft-max}_\eta(\{g_i(x)\}_{i=1}^m) \coloneqq \frac{1}{\eta} \ln\left(\sum_{i=1}^m \exp(\eta g_i(x))\right).$$

Again, from convex analysis, $g_\eta(x)$ is convex.

Intuition. The soft-max function tries to blow-up the maximum value among its argument by some (large) positive number η and applying the exponential function to which. By doing so, the maximum among its arguments will stand out and dominate others.

It turns out that the error incurs by replacing maximum with the soft-max function is small:

Claim. For all $x \in \mathbb{R}^n$, we have

$$\max_{i\in[m]} g_i(x) \le g_\eta(x) \le \max_{i\in[m]} g_i(x) + \frac{\ln m}{\eta}.$$

Then, if we choose $\eta = \ln m/\epsilon$, we get a smooth proxy for $h(x)$ that has an additive error of at most ϵ. Specifically, considering

$$\begin{aligned} \max\ & f(x) \\ & g_\eta(x) \le 1; \\ & x \in P, \end{aligned}$$

we see that since $h(x) \le g_\eta(x) \le h(x) + \ln m/\eta = h(x) + \epsilon$, we have $g_\eta(x) - \epsilon \le h(x) \le g_\eta(x)$.

Remark. We can compress multiple constraints into one with a small loss of factor $(1 + \epsilon)$.

Another standard approach in constrained continuous (convex and non-linear) optimization is to replace a constrained problem by its Lagrangian relaxation. In the context of the above, we would replace the constrained optimization problem by the following concave optimization problem:

$$\max_{x\in P} f_w(x) \coloneqq \max_{x\in P} f(x) - \sum_{i=1}^m w_i(g_i(x) - 1),$$

where $w_1, w_2, \ldots, w_m \ge 0$ are Lagrange multipliers.

Intuition. We penalize the objective for violating the constraints.

The following weak duality claim is easy to check.

Claim. For any non-negative weights $w_1, w_2, \ldots, w_m \ge 0$, $\max_{x\in P} f_w(x) \ge \mathsf{OPT}$ where OPT is the optimum value of the original optimization problem. Thus, $\min_{w\ge 0} \max_{x\in P} f_w(x) \ge \mathsf{OPT}$.

For convex optimization problems, the duality theorem says that under some mild conditions (e.g., Slater conditions), strong duality holds, i.e., $\min_{w\ge 0} \max_{x\in P} f_w(x) = \mathsf{OPT}$.

Although Lagrangian relaxation is nice, it is not obvious how to find good weights that lead to good upper and lower bounds. One can combine the idea of soft-max function and Lagrangian relaxation to reduce the problem to have a single constraint, namely $g_\eta(x) \le 0$. Thus, we want to solve the problem $\max_{x\in P} f(x)$ such that $g_\eta(x) \le 0$. One can use some gradient descent-like approaches for solving this problem and the gradient of $g_\eta(x)$ is relevant in this context. We will elaborate on this later.

Finally, motivated by the Lagrangian relaxation approach, we assume that we have a black-box oracle for the following problem for any given non-negative weights $w_1, w_2, \ldots, w_m \ge 0$:

$$\begin{aligned} \max\ & f(x) \\ & \sum_{i=1}^m w_i g_i(x) \le \sum_{i=1}^m w_i; \\ & x \in P. \end{aligned} \tag{4.1}$$

The upshot is the following:

Remark. When f and g_i's are positive linear functions and P is the non-negative orthant, the above oracle can be easily obtained.

4.2.2 Continuous Time Multiplicative Weight Update Algorithm

We consider an adaptive iterative algorithm that uses a notion of time that goes from 0 to 1.

Intuition. We're essentially solving a sequence of Lagrange relaxation and penalize the violated constraints more along the way.

From a discrete point of view, we can think of taking T steps of size $\delta = 1/T$ where δ is small. Thus, after ℓ steps, the time is $t = \ell\delta$. Later, for ease of analysis, we will make this a continuous process.

The algorithm will maintain a set of non-negative weights $w_i(t)$, one for each constraint. The non-negative weights represent the relative importance of each constraint at time t. Initially, we have no information, so all weights are equally important, hence we set $w_i(0) = 1$.

Intuition. An alternative view is to think of the normalized weights $w_i(t)/\sum_{k=1}^m w_k(t)$ as a probability distribution on the constraints.

In each step, we will use the oracle to compute a solution $v(t)$ for the Lagrangian relaxation defined by the weights, i.e.,

$$\begin{aligned} v(t) \coloneqq \arg\max\ & f(y) \\ & \sum_{i=1}^m w_i(t) g_i(y) \le \sum_{i=1}^m w_i(t); \\ & y \in P. \end{aligned}$$

Clearly, $f(v(t)) \ge \mathsf{OPT}$ since we are solving a relaxation. At the end, we will take an average of convex combination of these solutions and concavity of f will guarantee that we are doing well on the objective value. We think $x(t)$ the current solution as $\int_0^t v(t)\,\mathrm{d}t$ as the accumulation of the solutions computed so far with initial point at $t = 0$ being $x(0) = 0$.

The main question is to deal with the constraint violation. For this, we will maintain the invariant that $w_i(t) = \exp\left(\eta \int_0^t g_i(v(t))\,\mathrm{d}t\right)$, which tries to keep track of the violation of constraint i.

Algorithm 4.1: Multiplicative Weight Update Uniform Step

Data: An objective f, constraints $\{g_i\}_{i=1}^m$, feasible set P, step size δ, parameter η
Result: A solution x_{out}

for $i = 1, \dots, m$ **do**
  $w_i(0) \leftarrow 1$
$x(0) = 0$ // $x \in \mathbb{R}^n$
$t = 0$

while $t < 1$ **do**
  $v(t) \leftarrow$ `Oracle`$(f, \{g_i\}_{i=1}^m, \{w_i(t)\}_{i=1}^m, P)$
  for $i = 1, \dots, m$ **do**
    $w_i(t+\delta) \leftarrow w_i(t) \cdot e^{\eta\delta g_i(v(t))}$
  $x(t+\delta) \leftarrow x(t) + \delta v(t)$
  $t \leftarrow t + \delta$
return $x(1)$

By taking $\delta \to 0$, we obtain a continuous time algorithm whose analysis can be done simply and is illuminating. Specifically, we will get a differential equation on the evolution of the weights: from

$$w_i(t+\delta) = w_i(t)\exp(\eta\delta g_i(v(t))),$$

we see that

$$\frac{\mathrm{d}w_i(t)}{\mathrm{d}t} = \lim_{\delta\to 0}\frac{w_i(t+\delta) - w_i(t)}{\delta} = \lim_{\delta\to 0}\frac{w_i(t)(\exp(\eta\delta g_i(v(t))) - 1)}{\delta} = w_i(t)\eta g_i(v(t))$$

by using the fact that $e^x - 1 \to x$ when $x \to 0$. Finally, we also see that the final return is

$$x_{\text{out}} = \int_0^1 v(t)\,\mathrm{d}t.$$

This results in the following continuous version of Algorithm 4.1.

Algorithm 4.2: Multiplicative Weight Update Continuous

Data: An objective f, constraints $\{g_i\}_{i=1}^m$, feasible set P, parameter η
Result: A solution x_{out}

for $i = 1, \ldots, m$ **do**
  $w_i(0) \leftarrow 1$

while $t = 0, \ldots, 1$ *continuously* **do**
  $v(t) \leftarrow$ `Oracle`$(f, \{g_i\}_{i=1}^m, \{w_i(t)\}_{i=1}^m, P)$
  for $i = 1, \ldots, m$ **do**
    Evolve $w_i(t)$ as $\mathrm{d}w_i(t)/\mathrm{d}t = w_i(t)\eta g_i(v(t))$

return $\int_0^1 v(t)\,\mathrm{d}t$

The following is one reason to choose the time interval to be $[0, 1]$ to naturally obtain a convex combination of solutions as the output.

Lemma 4.2.1. In Algorithm 4.2, if f is concave, then $f(x_{\text{out}}) \geq \mathsf{OPT}$.

Proof. From the concavity of f, we have

$$f(x_{\text{out}}) = f\left(\int_0^1 v(t)\,\mathrm{d}t\right) \geq \int_0^1 f(v(t))\,\mathrm{d}t \geq \int_0^1 \mathsf{OPT}\,\mathrm{d}t = \mathsf{OPT},$$

where $f(v(t)) \geq \mathsf{OPT}$ since $v(t)$ is an optimum solution to a Lagrangian relaxation. ■

We now express the constraint values in terms of the weights.

Lemma 4.2.2. In Algorithm 4.2, For each $i \in [m]$, we have $g_i(x_{\text{out}}) \leq \ln w_i(1)/\eta$.

Proof. By concavity, we have

$$\begin{aligned} g_i(x_{\text{out}}) = g_i\left(\int_0^1 v(t)\,\mathrm{d}t\right) &\leq \int_0^1 g_i(v(t))\,\mathrm{d}t \\ &= \int_0^1 \frac{1}{\eta}\frac{1}{w_i(t)}\frac{\mathrm{d}w_i(t)}{\mathrm{d}t}\,\mathrm{d}t = \frac{1}{\eta}\left(\ln w_i(1) - \ln w_i(0)\right) = \frac{\ln w_i(1)}{\eta}, \end{aligned}$$

where we know that $\mathrm{d}w_i(t)/\mathrm{d}t = \eta w_i(t) g_i(v(t))$ in Algorithm 4.2. ■

While we do not have a direct way to bound $w_i(1)$, but we can bound $w_i(1)$ by $\sum_{i=1}^m w_i(1)$. This is a very crude bound, but since we will see how the total sum of weights evolves, it turns out to be enough.

Lemma 4.2.3. In Algorithm 4.2, for all t, $\sum_{i=1}^m w_i(t) \leq e^{\eta t}\sum_{i=1}^m w_i(0)$. Hence, $\sum_{i=1}^m w_i(1) \leq me^{\eta}$.

Proof. Here is where we use the fact that we solve the Lagrangian relaxation. Note that $v(t)$ satisfies the constraint that $\sum_{i=1}^m w_i(t)g_i(v(t)) \leq \sum_{i=1}^m w_i(t)$. Moreover, we have $\mathrm{d}w_i(t)/\mathrm{d}t = \eta w_i(t)g_i(v(t))$ for $i \in [m]$. Hence,

$$\frac{1}{\eta}\sum_{i=1}^m \frac{\mathrm{d}w_i(t)}{\mathrm{d}t} \leq \frac{1}{\eta}\sum_{i=1}^m \eta w_i(t) g_i(v(t)) \leq \sum_{i=1}^m w_i(t).$$

Thus, the total weight $w(t) \coloneqq \sum_{i=1}^m w_i(t)$ satisfies the differential equation $\mathrm{d}w(t)/\mathrm{d}t \leq \eta w(t)$, which implies that $w(t) \leq e^{\eta t}w(0) = me^{\eta}$. ■

Putting everything together, we have the following.

Theorem 4.2.1. The output x_{out} of Algorithm 4.2 satisfies the following properties:

(a) $f(x_{\text{out}}) \geq \mathsf{OPT}$;

(b) $g_i(x_{\text{out}}) \leq 1 + \ln m/\eta$ for all $i \in [m]$.

If there is a point $x_0 \in P$ such that $g_i(x_0) = 0$ for all $i \in [m]$, then $x'_{\text{out}} = \theta x_{\text{out}} + (1-\theta)x_0$ is feasible, in that $g_i(x'_{\text{out}}) \leq 1$ for all $i \in [m]$ and $x'_{\text{out}} \in P$.

Proof. Firstly, (a) is shown in Lemma 4.2.1. For (b), by combining Lemma 4.2.2 and Lemma 4.2.3,

$$g_i(x_{\text{out}}) \leq \frac{1}{\eta}\ln(w_i(1)) \leq \frac{1}{\eta}\ln\left(\sum_{i=1}^{m} w_i(1)\right) \leq \frac{1}{\eta}\ln(e^{\eta}m) \leq 1 + \frac{\ln m}{\eta}.$$

The second part is easy to verify. ■

We see that to obtain a $(1+\epsilon)$-approximation in terms of the constraint violation, we can choose $\eta = \ln m/\epsilon$. We note that the second part of Theorem 4.2.1 allows us to obtain a feasible solution x'_{out} since x_{out} from Algorithm 4.2 might be infeasible.

Intuition. The impact of using x'_{out} instead of x_{out} in terms of the objective function f is less easy to see. However, for non-negative linear objectives, we see that $f(x_{\text{out}})' \geq \theta f(x_{\text{out}})$, and thus we get an approximation to the objective while making the solution feasible.

Remark. The analysis in Theorem 4.2.1 does not use any property of f, g_i other than continuity. In particular, it does not use non-negativity of g_i's. Hence, the analysis can also be applied to have constraints of the form $g_i(x) \geq 1$ where g_i is concave, which is equivalent to $-g_i(x) + 2 \leq 1$.

Note (Potential approach). Our analysis relied on keeping track of the total weight $w(t)$ carefully. In some works, the potential function $\phi(t) = \ln w(t)/\eta$ or proxies for it are used. Potential functions can be powerful tools that can yield strong results though they can sometimes be mysterious [BG17].

Lecture 18: Discrete Multiplicative Weight Update Algorithm

4.2.3 Discrete Time Multiplicative Weight Update Algorithm

24 Oct. 11:00

We want to obtain an algorithm that terminate in a few iterations.

As previously seen. Recall that we're working with a convex program

$$\begin{aligned} \max\ & f(x) \\ & g_i(x) \leq 1 \quad \forall i \in [m] \\ & x \in P, \end{aligned}$$

where f is concave and g_i's are convex. The underlying linear programs we're interested in are

$$\begin{aligned} \max\ & c^\top x \\ & Ax \leq b \\ & x \geq 0, \end{aligned}$$

where $c \in \mathbb{R}^n_+$, $A \in \mathbb{R}^{m\times n}_+$, and $b \in \mathbb{R}^m_+$.

For this, we will work with the extra condition that $g_i(x) \geq 0$ for all $x \in P$. This models the packing constraints of interest and is needed for the algorithms and analysis, unlike the continuous algorithm.

Firstly, consider the simple uniform step size algorithm Algorithm 4.1. For simplicity, δ is chosen that there is an integer T such that $T\delta = 1$. Thus, the algorithm runs for T time steps, and the goal

is to minimize T such that the discrete implementation ensures that the constraints are approximately satisfied. It turns out that T is related to a parameter called the width of the given instance.

Definition 4.2.1 (Width). The *width* ρ of the given instance is defined as $\rho := \sup_{x \in P, i \in [m]} g_i(x)$.

In other words, it is asking how much can some arbitrary point in P violate a specific constraint in a *relative sense*.[2] Recall that we solve a Lagrangian relaxation in each step and find a point $v(t)$, which can violate some specific constraint i by significant factor since we are only solving the relaxation by taking weighted sum of the constraints. Hence, ρ allows us to bound the worst-case violation.

It may seem that ρ is unbounded for even simple instances because g_i may be an increasing function and P is typically the non-negative orthant. However, for many problems of interest, there are implicit bounds on the variables and P can be implicitly taken to be a bounding box that restrict ρ. We will give some examples later on to illustrate this. Suppose for now that ρ is some bounded number.

Theorem 4.2.2. Suppose $\eta = \ln m/\epsilon$, $\delta = \epsilon/\eta\rho$, and $\epsilon \in (0, 1/2)$. Then in $O(\rho \ln m/\epsilon^2)$ iterations, Algorithm 4.1 will output a point x_{out} such that

(a) $f(x_{\text{out}}) \geq \mathsf{OPT}$, and

(b) $g_i(x_{\text{out}}) \leq 1 + 2\epsilon$ for all $i \in [m]$.

Instead of proving Theorem 4.2.2, we consider the non-uniform step size case, which turns out to be stronger and Theorem 4.2.2 can be reduced to which. Specifically, Garg and Konemann [GK07] obtained a width-independent algorithm and analysis by a seemingly simple but important tweak to Algorithm 4.1:

Intuition. We choose the maximum step size at any point greedily so that the analysis goes through.

In particular, all we need is that we want to find δ such that $w_i(t + \delta) \leq w_i(t) \cdot e^{\epsilon}$, i.e., δ is the maximum value such that $\eta\delta g_i(v(t)) \leq \epsilon$.

Note. The choice of the step size requires $g_i \geq 0$ over the domain P, otherwise, it is not well-defined.

Surprisingly, this yields an algorithm that bounds the number of iterations to $O(m \log m/\epsilon^2)$, which depends only on the number of constraints and ϵ and is width-independent. We see the algorithm directly.

Algorithm 4.3: Multiplicative Weight Update Non-Uniform Step

Data: An objective f, constraints $\{g_i\}_{i=1}^m$, feasible set P, parameter η
Result: A solution x_{out}

for $i = 1, \ldots, m$ **do**
  $w_i(0) \leftarrow 1$
$x(0) = 0$ // $x \in \mathbb{R}^n$
$t = 0$

while $t < 1$ **do**
  $v(t) \leftarrow$ `Oracle`$(f, \{g_i\}_{i=1}^m, \{w_i(t)\}_{i=1}^m, P)$
  $\delta \leftarrow \max \delta'$ such that $\max_{i \in [m]} \delta' \eta g_i(v(t)) \leq \epsilon$ `// Max step size`
  $\delta \leftarrow \min(\delta, 1 - t)$ `// Handle the last iteration`
  for $i = 1, \ldots, m$ **do**
    $w_i(t + \delta) \leftarrow w_i(t) \cdot e^{\eta\delta g_i(v(t))}$
  $x(t + \delta) \leftarrow x(t) + \delta v(t)$
  $t \leftarrow t + \delta$
return $x(1)$

We will mimic the continuous time algorithm (Algorithm 4.2) analysis in some ways and point out where the discretization matters and how we bound the errors. First, let's set up some notations.

[2] Since we have normalized the right-hand-side of each constraint to 1 by considering $g_i(x) \leq 1$ as the constraint.

Notation. The algorithm runs for some number T of iterations, and we use j to index the iterations and i as before to index the constraints. We think of the time before the start of iteration $j+1$ as t_j, and the step size in that iteration as δ_j. We have $t_0 = 0$ and $t_{j+1} = t_j + \delta_j$ for $j = 0, 1, \ldots, T-1$ such that $\sum_{j=0}^{T-1} \delta_j = 1$. Note that in iteration $j+1$, we compute $v(t_j)$ and δ_j w.r.t. weights $w_i(t_j)$'s. In other words, t_{j+1} is the time at the end of iteration $j+1$.

The following is easy to prove even with a discrete step size (compared to Lemma 4.2.1).

Lemma 4.2.4. In Algorithm 4.3, if f is concave, then $f(x_{\text{out}}) = f(x(1)) \geq \mathsf{OPT}$.

Proof. From the definition and concavity of f, we have

$$f(x_{\text{out}}) = f\left(\sum_{j=0}^{T-1} \delta_j v(t_j)\right) \geq \sum_{j=0}^{T-1} \delta_j f(v(t_j)) \geq \sum_{j=0}^{T-1} \delta_j \,\mathsf{OPT} \geq \mathsf{OPT},$$

where we again use the fact that $v(t_j)$ is the solution of the Lagrangian relaxation. ■

A similar claim to Lemma 4.2.2 still holds in the discrete setting, but the proof is useful to see.

Lemma 4.2.5. In Algorithm 4.3, for each $i \in [m]$, we have $g_i(x_{\text{out}}) \leq \ln w_i(1)/\eta$.

Proof. We have $x_{\text{out}} = \sum_{j=0}^{T-1} \delta_j v(t_j)$, and hence by convexity of g_i, $g(x_{\text{out}}) \leq \sum_{j=0}^{T-1} \delta_j g(v(t_j))$. We see that in iteration $j+1$, the weight of constraint i is updated as

$$w_i(t_{j+1}) = w_i(t_j)\exp(\eta\delta_j g_i(v(t_j))) \Rightarrow \delta_j g_i(v(t_j)) = \frac{\ln w_i(t_{j+1}) - \ln w_i(t_j)}{\eta}.$$

Summing up two sides over j and using telescoping, we have

$$g(x_{\text{out}}) \leq \sum_{j=0}^{T-1} \delta_j g(v(t_j)) \leq \frac{\ln w_i(1) - \ln w_i(0)}{\eta},$$

with $w_i(0) = 1$, we're done. ■

The main technical lemma is the following, which bounds the evolution of the total weight and this is where the choice of the step size is crucial.

Lemma 4.2.6. In Algorithm 4.3, for all t_j, $\sum_{i=1}^m w_i(t_j) \leq e^{(1+\epsilon)\eta t_j} \sum_{i=1}^m w_i(0)$. Hence, $\sum_{i=1}^m w_i(1) \leq me^{(1+\epsilon)\eta}$.

Proof. Consider iteration $j+1$. Since g_i's are all positive on the domain, all numbers involved are positive. We see that

$$\sum_{i=1}^m w_i(t_j + \delta_j) = \sum_{i=1}^m w_i(t_j)\exp(\eta\delta_j g_i(v(t_j))),$$

where we ensure that $\eta\delta_j g_i(v(t_j)) \leq \epsilon \leq 1/2$ in Algorithm 4.3. Observe that it suffices to prove

$$\sum_{i=1}^m w_i(t_j + \delta_j) \leq e^{(1+\epsilon)\delta_j\eta}\left(\sum_{i=1}^m w_i(t_j)\right).$$

Claim. If $a \in (0, 1/2)$, we have $e^a \leq 1 + a + a^2$.

Proof. This is immediate from Taylor's expansion. ⊛

Hence, by writing $\eta\delta_j g_i(v(t_j)) = a_i$, we have

$$\begin{aligned}\sum_{i=1}^{m} w_i(t_j)\exp(\eta\delta_j g_i(v(t_j))) &\le \sum_{i=1}^{m} w_i(t_j)\exp\left(1+a_i+a_i^2\right)\\ &\le \sum_{i=1}^{m} w_i(t_j)\left(1+a_i(1+\epsilon)\right)\\ &\le \sum_{i=1}^{m} w_i(t_j) + (1+\epsilon)\eta\delta_j \underbrace{\sum_{i=1}^{m} w_i(t_j)g_i(v(t_j))}_{\le \sum_{i=1}^{m} w_i(t_j)}\\ &\le \left(\sum_{i=1}^{m} w_i(t_j)\right)(1+(1+\epsilon)\eta\delta_j) \le \left(\sum_{i=1}^{m} w_i(t_j)\right)\exp((1+\epsilon)\eta\delta_j)\end{aligned}$$

since $1+b \le e^b$ for all $b \ge 0$. Hence, we're done. ■

Remark. The proof of Lemma 4.2.6 will go through easily for a fixed step size as long as the step size guarantees that no weight of a constraint goes up by more than an e^ϵ factor. This is satisfied by the choice of the step size based on the width! The rest of the analysis is the same.

Theorem 4.2.3. If $\eta \ge \ln m/\epsilon$, then the output x_{out} of Algorithm 4.3 satisfies the following:

(a) $f(x_{\text{out}}) \ge \mathsf{OPT}$;

(b) $g_i(x_{\text{out}}) \le 1+2\epsilon$ for all $i \in [m]$.

Moreover, Algorithm 4.3 terminates in $O(m\log m/\epsilon^2)$ iterations.

Proof. Firstly, (a) is shown in Theorem 4.2.3. For (b), from Lemma 4.2.5, we have $g_i(x_{\text{out}}) \le \ln w_i(1)/\eta$ and that $\ln\sum_{i=1}^m w(1) \le (1+\epsilon)\eta\ln m$ from Lemma 4.2.6. This implies

$$g_i(x_{\text{out}}) \le \frac{1}{\eta}\ln(w_i(1)) \le \frac{1}{\eta}\ln\left(\sum_{i=1}^{m} w_i(1)\right) \le \frac{1}{\eta}\ln\left(me^{(1+\epsilon)\eta}\right) = 1+\epsilon+\frac{\ln m}{\eta} \le 1+2\epsilon.$$

Next, we switch to exponential weights instead of reasoning with logarithms since it will be useful later. We will also use a loose argument though one can do a tighter analysis. Let T be the total number of iterations Algorithm 4.3 takes.

Claim. For some fixed small constant c, $T \le cm\ln m/\epsilon^2$.

Proof. Firstly, in the end, the total weight $\sum_{i=1}^m w(1)$ is

$$e^{(1+\epsilon)\eta}m = m\cdot e^{(1+\epsilon)(\ln m)/\epsilon} = m^{1+\frac{1+\epsilon}{\epsilon}} = m^{O(1/\epsilon)}.$$

Observe that since we're being greedy, in each iteration we choose δ_j to be the maximum such that there is some i in that iteration with $\eta\delta_j g_i(v(t_j)) = \epsilon$ exactly. But this implies that for that i, its weight w_i increases by a factor of e^ϵ. Since each weight starts at 1, with the fact that the total weight at the end is $m^{O(1/\epsilon)}$, no weight w_i can have its weight updated by an e^ϵ factor more than $O(\log m/\epsilon^2)$ times. But there are only m constraints and each iteration must increase at least one constraint's weight by a factor of e^ϵ, thus, we cannot have T more than $cm\log m/\epsilon^2$ for some small but fixed constant c. ⊛

This finishes the proof. ■

4.2.4 Scaling Weights

In the analysis so far, we have sued constraints of the form $g_i(x) \leq 1$ which makes the notation clean. Of course, any constraint of the form $g_i(x) \leq b_i$ with $b_i > 0$ can be scaled to achieve the standard form. However, in some combinatorial applications, when A the packing matrix corresponds to an incidence matrix of a combinatorial object and b_i represents some capacity, scaling makes the matrix unnatural.

We can avoid this by modifying Algorithm 4.3 as follows. Suppose we have constraints of the form $g_i(x) \leq b_i$ for all $i \in [m]$ where $b > 0$. we start with $w_i(0) = 1/b_i$ for each i. Then, we do the analysis with $b_i w_i$ instead of w_i. Since the weights are updated in a multiplicative fashion, the whole analysis goes through cleanly.

> **Note.** With weights set up like this, we still solve the oracle in each iteration with the constraint $\sum_{i=1}^{m} w_i g_i(y) \leq \sum_{i=1}^{m} w_i$; the b_i's are taken care of automatically.

4.3 Application to Packing Linear Program

We show how to generic scheme that we have described via MWU can be used to derive fast approximation algorithms for packing linear programs. Recall that the general packing linear program looks like

$$\begin{aligned} \max\ & c^\top x \\ & Ax \leq b; \\ & x \geq 0. \end{aligned}$$

In terms of the generic framework, we have $f(x) = c^\top x$ and $g_i(x) = \sum_{j=1}^{n} A_{ij} x_j$ corresponds to the i^{th} row of A, and P corresponds to $\mathbb{R}^n_+$.

> **Note.** In some settings, we think of P as a bounding box implied by constraints with $0 \leq x_j \leq u_j$ where u_j is an upper bound on x_j implicitly implied by the constraints in $Ax \leq b$. This often helps in figuring out the width of the system.

Firstly, we can interpret the weights as exponential in loads.

> **Intuition.** In the continuous MWU method, we are maintaining $w_i(t)$ as $\exp\left(\eta \int_0^t g_i(v_t)\,\mathrm{d}t\right)$. When g_i is linear and P is the non-negative orthant, $\int_0^t g_i(v_t)\,\mathrm{d}t$ is simply $\sum_{i=1}^{n} A_{ij} x_j(t)$, where $x(t)$ is the current accumulated vector. We think of this as the total load $\ell_i(t)$ on constraint i at time t, and hence $w_i(t) = \exp(\eta \ell_i(t))$.

Next, we note that in the case of linear objective, it is possible to transform between the approximation guarantees for constraints (Theorem 4.2.3) and objective value.

> **Remark.** The MWU method, as described, naturally give an optimum solution that violates the constraints. We can then scale the solution down to obtain a feasible solution while losing a bit in the objective. This is particularly easy for packing problems with linear objectives. Thus, we can obtain a $(1-\epsilon)$-approximation in the objective while obtaining a feasible solution.

4.3.1 Efficient Oracle for Positive Packing Linear Program

Recall that we need an oracle that given weights w_i's, solve

$$\begin{aligned} \max\ & f(x) \\ & \sum_{i=1}^{m} w_i g_i(x) \leq \sum_{i=1}^{m} w_i; \\ & x \geq 0, \end{aligned}$$

In the case of packing linear programs, consider the normalized linear program with constraints $Ax \leq 1$, then the problem becomes

$$\begin{aligned} \max \sum_{j=1}^{n} c_j x_j \\ \sum_{i=1}^{m} w_i \sum_{j=1}^{n} A_{ij} x_j \leq \sum_{i=1}^{m} w_i \\ x \geq 0; \end{aligned} \quad \Leftrightarrow \quad \begin{aligned} \max \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} \left(\sum_{i=1}^{m} w_i A_{ij} \right) x_j \leq \sum_{i=1}^{m} w_i \\ x \geq 0; \end{aligned} \quad =: \quad \begin{aligned} \max \sum_{j=1}^{n} c_j x_j \\ \sum_{j=1}^{n} \alpha_j x_j \leq 1 \\ x \geq 0, \end{aligned} \tag{4.2}$$

where

$$\alpha_j := \frac{1}{\sum_{i=1}^{m} w_i} \sum_{i=1}^{m} w_i A_{ij} \geq 0$$

since all entries are positive.

> **Intuition.** This is a fractional knapsack problem! We know that the optimum solution x^* is just to pick the coordinate $j^* := \arg\max_{j \in [n]} c_j/\alpha_j$ and setting $x^*_{j^*} := 1/\alpha_{j^*}$ and the rest to 0.

We summarize the MWU algorithm for the case of positive packing linear program as follows.

Algorithm 4.4: Multiplicative Weight Update for Positive Packing Linear Program

Data: A cost vector $c \in \mathbb{R}^n_+$, constraint matrix-vector pair $A \in \mathbb{R}^{m \times n}_+$, $b \in \mathbb{R}^n_+$, error ϵ
Result: A solution x_{out}

$\eta \leftarrow \log m/\epsilon$

for $i = 1, \ldots, m$ **do**
  $w_i(0) \leftarrow 1/b_i$
$x(0) = 0$ // $x \in \mathbb{R}^n$
$t = 0$

while $t < 1$ **do**
  $j^* \leftarrow \arg\max_{j \in [n]} c_j/(\sum_{i=1}^{m} A_{ij} w_i)$
  $\delta \leftarrow \epsilon/(\eta \max_{i \in [m]} A_{ij^*})$ // Max step size
  $\delta \leftarrow \min(\delta, 1 - t)$ // Handle the last iteration
  for $i = 1, \ldots, m$ **do**
    $w_i(t + \delta) \leftarrow w_i(t) \cdot e^{\eta \delta A_{ij^*}}$
  $x(t + \delta) \leftarrow x(t) + \delta e_{j^*}$
  $t \leftarrow t + \delta$
return $x(1)$

We note that an important aspect of j^* is that there is an optimum solution x^* to the oracle whose support is a *single coordinate*.

> **As previously seen.** The width-independent analysis (Theorem 4.2.3) shows that the number of iterations depends only on the number of constraints m.

Thus, we can apply the method to implicit linear programs with exponential (in some original problem size) number of variables but only a polynomial number of constraints as long as we can implement the oracle efficiently. Due to this structure, the sequence of oracles can often be maintained dynamically.

> **Remark** (Maintaining the oracle). We see that at each time step t, we solve for $v(t) = x^*$, which is simply $1/\alpha_{j^*}$ at the $j^{*\text{th}}$ coordinate and 0 otherwise. In the next round for time step $t + \delta$, only those w_i's such that $g_i(v(t)) > 0$ needs to be updated, i.e., for those i such that $\sum_{j=1}^{n} A_{ij} x^*_j \neq 0$. Since A is positive, this is equivalent to those i such that $A_{ij^*} > 0$.

Finally, another nice aspect of the MWU method is that it accumulates a series of solutions of Lagrangian relaxations by taking non-negative sums. In the positive settings, this means that we do not

use subtraction at all. This is convenient for approximation: Suppose we only have an α-approximation oracle in the relative sense. Then, the process yields a $(1-\epsilon)\alpha$-approximation solution. This is convenient not only in obtaining approximate solutions, but also in speeding up MWU method-based algorithms substantially by using various tricks and data structures that can avoid updating information at every step since we are allowed to use an approximate oracle.

Note. If $\alpha = (1-\epsilon)$, then $(1-\epsilon)\alpha \geq 1 - 2\epsilon$.

Remark (Explicit packing linear program)**.** From Theorem 4.2.3, the algorithm takes $O(m \log m/\epsilon^2)$ iterations. The oracle for each iteration can be done in $O(N)$ time easily and naively. Thus, we can obtain a $(1-\epsilon)$-approximate solution in $O(mN \log m/\epsilon^2)$ time. By being slightly careful and using a simple approximate oracle, we can reduce the running time to $O(N \log m/\epsilon^2)$, which is near-linear.

Lecture 19: Make Multiplicative Weight Update Faster

4.3.2 Examples

31 Oct. 11:00

It is instructive to see what the oracle corresponds to when considering combinatorial problems. Hence, in this section, we continue on the previous examples we have seen for the packing linear programs, and see what are their corresponding MWU oracle.

Example (Maximum Weight Matching)**.** Continue from the previous example, where we consider the maximum weight matching problem in bipartite graphs with the following linear program:

$$\begin{aligned} \max \sum_{e \in E} c(e) x_e & \\ \sum_{e \in \delta(u)} x_e \leq 1 & \quad \forall u \in V; \\ x_e \geq 0 & \quad \forall e \in E, \end{aligned}$$

where the variables x_e correspond to the edges and the constraints correspond to the vertices. Thus, the MWU method maintains a weight w_u for each $u \in V$. The oracle is that given vertex weights w_u, it finds the edge $e = uv \in E$ that maximizes $c_e/(w_u + w_v)$ in each step.

Intuition. Updating the weights is simple and fast because we are only touching two vertex weights. Hence, the bottleneck is finding the edge with the maximum ratio in each iteration.

We can explore simple tricks that will help speed this up. First, we bucket edges into logarithmic groups by considering weights in powers of $(1+\epsilon)$; this affects the approximation factor only a $(1-\epsilon)$-factor. Within each bucket, we are trying to find the weight with the minimum weight where the weight of an edge is the sum of the weights of its end points. We can keep a priority queue for edges based on their weight. How do we update these weights? Suppose in iteration j of the algorithm, we pick edge $e_j = (u, v)$. Then the algorithm updates the weights of edges that are incident to u and w and these will affect the priority queues. It is $\deg(u) + \deg(v)$. This seems like a lot, but we make a crucial observation.

As previously seen. The width-independent MWU analysis proves that the weight of any specific constraint i is updated only $O(\log m/\epsilon^2)$ times!

In particular, the weight of a vertex u is updated only $O(\log m/\epsilon^2)$ times, where $m = |V|$. Hence, the total number of updates is only $O(\sum_{u \in V} \deg(u) \log m/\epsilon^2) = O(|E| \log |V|/\epsilon^2)$! The priority queue operations have only a logarithmic overhead. Thus, we can see that with some basic observations we can implement the MWU algorithm in $\widetilde{O}(|E|/\epsilon^2)$ time.

Example (Tree packing)**.** Continue from the previous example, where we consider the tree packing problem with the following linear program:

$$\begin{aligned} \max \quad & \sum_{T \in \mathcal{T}_G} y_T \\ & \sum_{T \ni e} y_T \le c(e) \qquad \forall e \in E; \\ & y_T \ge 0 \qquad\qquad \forall T \in \mathcal{T}_G. \end{aligned}$$

This is an implicit linear program with an exponential number of variables but only m constraints where $m = |E|$. Note that the coefficient of each variable y_T in the objective is 1. In this example, the right-hand side of the constraint corresponding to edge e is its capacity $c(e)$ and these are non-uniform. It now makes sense to use weights that start with $w_e = 1/c(e)$.

The oracle is that given weights w_e on edges, we with to find a tree T that maximizes $\frac{1}{\sum_{e \in T} w_e}$, i.e., $\min_{T \in \mathcal{T}_G} \sum_{e \in T} w_e$. This is just the MST problem, which is the separation oracle for the dual tree-packing linear program! Thus, each iteration, we simply solve an MST problem and update the weights of the edges in the chosen MST T^*, which takes $O(m + n)$ time from Theorem 1.1.6. There are only $O(m \log m/\epsilon^2)$ iterations, hence the total time is $O(m^2 \log m/\epsilon^2)$.

Example (Maximum multi-commodity flow)**.** Continue from the previous example, where we consider the maximum multi-commodity flow with the following linear program:

$$\begin{aligned} \max \quad & \sum_{i=1}^{k} \sum_{P \in \mathcal{P}_{s_i,t_i}} x_P \\ & \sum_{i=1}^{k} \sum_{\substack{P \in \mathcal{P}_{s_i,t_i} \\ P \ni e}} x_P \le c(e) \qquad\qquad \forall e \in E; \\ & x_P \ge 0 \qquad\qquad\qquad P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}. \end{aligned}$$

Once again, we have an implicit packing linear program with an exponential number of variables corresponding to the paths for all the commodity pairs, and the number of constraints is m, corresponding to the edges. The weights correspond to exponential of the loads on the edges where the load on an edge is the total flow routed so far on the edge. The oracle explicitly in this case is the same as $\max_{P \in \bigcup_{i=1}^{k} \mathcal{P}_{s_i,t_i}} 1/\sum_{e \in P} w_e$, i.e., finding the shortest path among all the commodity pairs, according to the weights on edges! We can find for each pair (s_i, t_i) a shortest s_i-t_i path and choose the minimum. The algorithm routes some flow on that path and updates the weights along that path and iterates. The bottleneck is to compute the shortest paths.

4.4 Speed Up Multiplicative Weight Update

Finally, we discuss how to combine what we have seen to some specific example of positive packing linear program problem to further speed up Algorithm 4.4.

> **As previously seen.** Naively, each iteration in Algorithm 4.4 requires $O(N)$, hence the overall runtime is $O(Nm \log m/\epsilon^2)$ with Theorem 4.2.3.

4.4.1 General Idea

Without loss of generality, we may assume that $c_j = 1$ for all j via scaling. Hence, solving j^* is equivalent to solving $\min_{j \in [n]} \sum_{i=1}^{m} A_{ij^*} w_i$. Let $\lambda_j \coloneqq \sum_{i=1}^{m} A_{ij} w_i$. The idea of speeding up Algorithm 4.4 is the following:

Intuition. Instead of solving j^* exactly, we allow to return some coordinate such that the returned λ_j is not too far from the optimal value for the particular problem at hand.

Example. Consider divide values into $(1+\epsilon)$-buckets.

4.4.2 Case Study: Tree Packing

As an example, consider the tree packing again:

$$\begin{aligned} \max \quad & \sum_{T \in \mathcal{T}_G} x_T \\ & \sum_{T \ni e} x_T \le c(e) \qquad \forall e \in E; \\ & x_T \ge 0 \qquad\qquad \forall T \in \mathcal{T}_G. \end{aligned}$$

As we have seen, the optimal solution j^* corresponds to the MST T^* w.r.t. weights w_e maintained by Algorithm 4.4, and the total algorithm takes $O(m^2 \log m/\epsilon^2)$ from the previous example. To speed up the algorithm, as we have hinted on, we avoid computing MST from scratch; instead, we maintain MST via dynamic data structure. The classical dynamic algorithm for maintaining MST takes $\mathsf{poly}\log n$ per edge weight change [HDT01] (specifically, $O(\log^2 n)$), hence the total time is

$$O\left(\frac{m \log m}{\epsilon^2} \cdot (n\mathsf{poly}\log n + n)\right) = O(mn\mathsf{poly}\log n/\epsilon^2)$$

if we implement this naively, i.e., we update $n-1$ edge weights at the end of each iteration and also maintain the corresponding new MST, which takes $O(n) + n \cdot \mathsf{poly}\log n$.

Compared to the original runtime, $O(m^2 \log m/\epsilon^2)$, this is not that much of an improvement. In particular, as we have seen in the very beginning of the class, we know that it is possible to get a near-linear time algorithm:

Theorem 4.4.1 (Theorem 1.2.2 [CQ17]). There is a deterministic $O(m \log^3 n/\epsilon^2)$-time algorithm that gives a $(1-\epsilon)$-approximation for fractional tree packing in a capacitated undirected graph with m edges. Hence, a $(1-\epsilon)$-approximate packing can be described in $\widetilde{O}(m/\epsilon^2)$ space.

The key idea of Theorem 4.4.1 is twofold. Firstly, as suggested before, we can delay the update of edge weight w_e for *only* computing MST:

Intuition (Lazy update). Report the update to the MST dynamic structure [HDT01] only if weight increases by $(1+\epsilon)$ factor. Hence, now MST is approximate.

Specifically, we can update weight lazily if it does not change much, say, less than an $e^\epsilon \approx 1+\epsilon$ factor. From Theorem 4.2.3, the total number of updates we will report to the dynamic structure is only $O(m \log m/\epsilon^2)$. Hence, now, the total runtime becomes

$$O\left(\frac{m \log m}{\epsilon^2} \cdot (\mathsf{poly}\log n + n)\right) = O(mn\mathsf{poly}\log n/\epsilon^2)$$

due to the fact that we still maintain every actual (ground-truth) edge weight update after each iteration.

Remark. To get rid of n entirely, we cannot afford to update w_e exactly after each iteration, even if we only report the updates lazily.

The second idea is that on top of the lazy update to the dynamic MST data structure, we also update our ground-truth weight lazily if it does not change much:

Intuition. Maintain edge weight w_e within a $(1 \pm \epsilon)$ multiplicative factor throughout.

It turns out that there exists such a data structure that can handle this type of lazy update as well [KY14; You14], but it is designed for explicit problems. By some adaption, the idea is then to combine it with the previous dynamic MST data structures [HDT01], which yields an amortized runtime throughout the algorithm of $O(\mathsf{poly}\log n)$ per iteration, giving the desired near-linear runtime.

The high-level intuition of this adaption is the following.

Intuition. When a tree T is updated, its rate of change of w_e depends on $c(e)$:

- if all $c(e)$ values are *uniform*, then $n-1$ edges increase weight by $(1+\epsilon)$ factor;
- if $c(e)$ values are *non-uniform*, delay updating small weight changes.

The problem is how to handle the second case. If we allow randomization, then the idea is simply to touch (update) w_e *in proportion to* $1/c(e)$. Specifically, when T is the tree added in an iteration, let $c_{\min} \coloneqq \min_{e\in T} c(e)$ be the minimum edge capacity in T. Then, for each edge $e \in E(T)$, we update w_e as

$$w_e \leftarrow \begin{cases} \exp(e) w_e, & \text{if } \theta \le c_{\min}/c(e); \\ w_e, & \text{otherwise,} \end{cases}$$

where $\theta \sim \mathcal{U}([0,1])$. In expectation, this update is correct, although these updates are clearly correlated. The benefit of this is that these updates are easy to implement via data structures due to the correlation, with the catch being we need to prove that MWU analysis works with correlated rounding. This is done by the so-called *drift analysis* [KY14].

Remark (Determinisitic approach). If we insist on a deterministic algorithm, the idea is to bucket $c(e)$ values geometrically and lazily update using amortization. However, this is somewhat involved to describe, and it takes advantage of $\{0,1\}$ structure of A.

Chapter 5

Cut-Matching for Fast Sparsest Cut

Lecture 20: Sparsest Cuts via s-t Flow

In this chapter, we show that one can approximate expansion to within an $O(\log^2 n)$ factor via $O(\log^2 n)$ calls to an approximate single-commodity s-t flow routine. Since we have fairly reliable s-t flow algorithms, both in practice and in theory, this gives us a fast approximation algorithm for computing expansion and other problems such as uniform sparsest cut and conductance. 5 Nov. 11:00

> **Note.** We now have near-linear time algorithms for a *constant factor approximation* to s-t flow in undirected graphs [Pen16], and an almost-linear time *exact* algorithm for s-t flow in directed graphs [Che+22].

Previously, we have discussed how to check if a graph G is an expander; more generally, how to estimate the expansion, conductance, and uniform sparsest cut.[1]

> **As previously seen.** There are three approximation algorithms that we have discussed or mentioned:
>
> - Multi-commodity flow [LR99]: $O(\log n)$-approximation (Theorem 2.3.4).
> - SDP relaxation [ARV09]: $O(\sqrt{\log n})$-approximation (previous problem).
> - Spectral method: $O(\sqrt{\mathsf{OPT}})$-approximation for conductance (Theorem 2.4.1).

The spectral method is very fast but does not yield good approximation in the regime when the conductance is low. The other two methods are based on solving slow (although polynomial) mathematical programming relaxation (e.g., Equation 2.4). While we have discussed the multiplicative weight update-based methods for fast solutions to linear programs, and one of the motivations for these algorithms was to speed up the maximum concurrent flow linear program for uniform sparsest cut. However, even with several ideas, the fastest way we know how to solve the linear programs takes $O(mn)$ time for a constant factor approximation.

5.1 Cut-Matching Game

The algorithm that achieves $O(\log^2 n)$-approximation for expansion using only $O(\log^2 n)$ many s-t-max-flow computations is based on the so-called *cut-matching game* framework [KRV09], which has since become a powerful tool in several applications. The framework has been further improved to yield an $O(\log n)$-approximation [Ore+08]. On another line of work, a more systematic approach via matrix-multiplicative weight updates obtains fast primal-dual algorithms that yield $O(\log n)$ and $O(\sqrt{\log n})$-approximation algorithms, which also use appropriate flow sub-routines [AK07].

5.1.1 Sparse Cut and Expanding Matching

For the sake of discussion, let us consider the decision version in terms of expansion:

[1] The expansion is within a factor of 2 of the sparsity from the previous remark.

Problem 5.1.1 (Expansion). Given a graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$ and a parameter φ, decide if the expansion of G is at least φ/α for some approximation factor $\alpha \geq 1$. If not, then find a corresponding sparse cut, i.e., $S \subseteq V$ such that $c(\delta(S)) \leq \varphi \cdot \min(|S|, |V \setminus S|)$.

Note. We assume that $\varphi = 1$ since we can scale c correspondingly by $1/\varphi$.

Remark. If one can solve expansion, then binary search can find an approximate sparsest cut.

To motivate the cut-matching game, we recall the following.

As previously seen. Recall that we have a "dual" notion of expansion, i.e., well-linked, from Corollary 2.4.2 and Lemma 2.4.3.

We further note that one can check if A, B are linked by an s-t flow computation such that a linkage can be viewed as creating a perfect matching between A and B where the edges of the matching correspond to the paths in the A-B linkage.

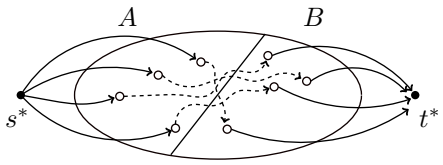


Figure 5.1: Adding a super source s^* to every $s \in A$ and a super sink from $t \in B$ to t^*.

From Figure 5.1, it's clear that the flow paths of integral s^*-t^* flow will implicitly match up vertices in A to vertices in B, hence inducing a perfect matching M_i if such a feasible flow exist.

Problem. What is the advantage of the notion of well-linkedness in this context?

Answer. To prove that G is not an expander, it suffices to exhibit an equal-sized bipartition $(S, V \setminus S)$ with $|S| = n/2$[a] and one can check via an s^*-t^* flow computation if $S, V \setminus S$ are not linked. Of course, it is much easier to exhibit a cut S which is not expanding. ⊛

[a] For simplicity, let n be even throughout this section.

On the other hand, if the s^*-t^* flow does perfectly match S to $V \setminus S$, then this only says that G is not necessarily not an expander. How to proceed?

Intuition. We think of the problem as generating a certificate for the expansion by generating a series of partition of V.

One idea is to keep selecting different sets $S_i \subseteq V$, each of size $n/2$, and using the corresponding s_i^*-t_i^* max-flow $f_{s_i^*, t_i^*}$ to either route a perfect matching M_i between S_i and $V \setminus S_i$, or prove that G has expansion at most a constant.

Intuition. If these matchings behave like *uniformly random matchings*, then after a constant number k (e.g., 3) of sets $S_1, \ldots, S_k$, we will have embedded a random expander with congestion k into G, certifying that G has expansion at least $1/k$ (e.g., Lemma 2.4.1).

The problem becomes that, since we treat the s_i^*-t_i^* flow computation as a black-box, and there is no reason that the matchings should behave like random matchings. The upshot is that Khandekar, Rao, and Vazirani [KRV09] show how to adaptively select sets $S_1, \ldots, S_k$, where S_i depends on the matchings $M_1, \ldots, M_{i-1}$ returned in rounds 1 through $i-1$ such that with high probability, after $k = O(\log^2 n)$ rounds, the multigraph $H = (V, \bigcup_{i=1}^k M_i)$ induced by the union of $M_1, \ldots, M_k$, is a 1-expander by

showing that H can route uniform multi-commodity flow: a demand of $1/n$ between each ordered pair of vertices (u, v), which suffice for H being a 1-expander.

Remark. The above means that the 1-expander H can be embedded with congestion $O(\log^2 n)$ since each M_i is routable in G, implying that G has expansion at least $1/k = \Omega(1/\log^2 n)$.

Our goal is then to minimize k while ensuring that H is an expander without actually manually checking it. Other methods use different ways to certify that H is a 1-expander.

Example. One can also use spectral methods via Cheeger's inequality.

5.1.2 Cut-Matching Game and Results

We now describe the cut-matching game formally.

Definition 5.1.1 (Cut-matching game). Given a graph G with edge capacity $c\colon E \to \mathbb{R}_+$, the *cut-matching game* proceeds in iterations. In iteration i:

1. *cut player* generates a partition $(S_i, V \setminus S_i)$ of V such that $|S_i| = n/2$;
2. *matching player* gives a routable perfect matching M_i in G between S_i and $V \setminus S_i$.

The cut player's goal is to make $H_i := (V, \bigcup_{j \le i} M_i)$ an expander as quickly as possible, while the goal of the matching player is to delay it as much as possible.

We think of ourselves as the cut player, while the matching player is some black-box s-t flow computation algorithms that we do not have control of.

Intuition. In reality, however, we compute the flow ourselves and hence we're essentially also the matching player as well, just that we do not have the control of the final matching M_i (if it exists).

Firstly, if there is no feasible matching flow between S and $V \setminus S$, then this should indicate that G is not a very good expander. Indeed, by multi-commodity flow-cut gap (Theorem 2.3.4), there is a cut with expansion at most $O(\log n)$. However, in this case, we can improve this bound to 1.

Lemma 5.1.1. Suppose there is no feasible matching flow between S and $V \setminus S$ for some $S \subseteq V$ with $|S| = n/2$. Then G has expansion at most 1.

Proof. Since we have an s^*-t^* min-cut of size less than $n/2$, where the graph is now with the super source s^* and the super sink t^* with the corresponding auxiliary edges from s^* to S and $t \in V \setminus S$ to t^*. In this cut, let a be the number of edges from s^* and let b be the number of edges to t^*, and let c be the remaining number of edges. We have $a + b + c < n/2$.

The c non-auxiliary edges is also a cut in G, separates at least $n/2 - a$ vertices in S from $n/2 - b$ vertices in $V \setminus S$. The expansion of this cut is at most

$$\frac{c}{\min(n/2 - a, n/2 - b)} < \frac{n/2 - a - b}{n/2 - \max(a, b)} \le 1,$$

which implies that G has expansion at most 1. ∎

On the other hand, if G has expansion at least 1, every bipartition produces a fractional perfect matching M_i via an s_i^*-t_i^* flow. Repeatedly doing so only implies that G is not necessarily not an expander (which isn't saying much). As a thought experiment, supposed the M_i's behaved like random matchings. As we have seen, a constant number of matchings forms an expander, and embedding each of these matchings in G implies that G is an expander. Unfortunately, we cannot say that M_i's behave like random matchings since ultimately the matchings are out of our control. However, the following theorem says that we can adaptively choose the bipartitions so that the union of M_i's form a matching after $O(\log^2 n)$ rounds.

Theorem 5.1.1 ([KRV09]). Consider the cut-matching game, where the matching player returns a fractional matching $M_i \in [0,1]^{V \times V}$ between the cut $(S_i, V \setminus S_i)$ produced by the cut player in iteration i. There is a randomized adaptive strategy for the cut player such that with high probability, the union of the first k matchings $M_1, \ldots, M_k$[a] forms a 1/2-expander with $k = O(\log^2 n)$. Moreover, the certificate of H's expansion is a feasible routing of the uniform multi-commodity flow over V.

[a]Formally, each matching flows is treated as a weighted, undirected edge set. Then the union is just the sum.

It's clear that by combining Lemma 5.1.1 and Theorem 5.1.1, we can prove the following.

Corollary 5.1.1. There is a randomized algorithm, that given a graph G, either proves that G is not an expander, or with high probability certifies that it is an $\Omega(\frac{1}{\log^2 n})$-expander using $O(\log^2 n)$ single-commodity flow instances on G. Via binary search, there is a randomized algorithm for uniform sparsest cut via $O(\log^3 n)$ single-commodity flow.

Proof. If any $(S_i, V \setminus S_i)$ cannot be feasibly routed, then we have a cut with expansion at most 1 from Lemma 5.1.1. Otherwise, from Theorem 5.1.1, with high probability, we obtain a 1/2-expander $H = (V, \bigcup_{i=1}^{k} M_i)$. Since each matching flow fits in G, so the k matching flows fit in G scaled up by a factor of k, i.e., H can be embedded in G with congestion $k = O(\log^2 n)$, proving the corresponding expansion. The binary search for uniform sparsest cut is trivial. ■

Finally, we note that this also yields an alternate proof of the flow-cut gap, although it is slightly weaker than the $O(\log n)$-bound we saw in Theorem 2.3.4:

Corollary 5.1.2. The multi-commodity flow-cut gap for uniform instances is $O(\log^2 n)$.

5.2 A Randomized Cutting Strategy

Our goal now is to prove Theorem 5.1.1, specifically, come up with a randomized strategy of selecting the cut $(S_i, V \setminus S_i)$ in each iteration i such that the union of the fractional matchings, $H = (V, \bigcup_{i=1}^{k} M_i)$, is a 1/2-expander.

5.2.1 Intuition: Certifying Expansion

But how can we certify that H is a 1/2-expander in the first place?

Problem. This seems circular: our original motivation is to decide whether G is a 1-expander. However, it seems like we still require to answer the same problem for H.

Answer. This is not cyclical because we will do something very specific to the structure of the matchings, such that we do not need to do the actual checking. ⊛

The idea is inspired by random-walk, but we will not explicitly mention the motivation and instead think of routing:

Intuition. Use $H_i := (V, \bigcup_{j \le i} M_j)$ to route a directed multi-commodity flow where we send (about) $1/n$ units of flow between every pair of vertices. This implies that we can embed the weighted clique, K_n/n, into H_i. Since K_n/n has expansion 1, which also certifies that for H_i.

We build out this multi-commodity flow incrementally, one matching at a time. We start with a trivial flow where each vertex sends one unit of flow to itself. Each successive matching M is used to disperse (and hopefully diversify) the flow. Specifically, at the beginning of a generic iteration, we have a multi-commodity flow for again each vertex sending and receiving one unit of flow. Given the new matching M, we update the flow as follows:

- for each pair (u, v), out of the 1 total unit of flow terminating at u from various sources, we send an $M(uv)/2$-fraction of each commodity to v;

- we also do the same for the 1 total unit of flow terminating at v, sending $M(uv)/2$-fraction of each commodity at u.

Intuition. Use the fractional perfect matching to mix up the existing flow, hoping to produce a multi-commodity flow closer to uniform. We then choose the bipartition that maximizes the mixing.

5.2.2 Potential Argument

Mathematically, at the beginning of the iteration i, for each $u \in V$, let $b_u^{(i)} \in [0,1]^V$ be the stochastic vector such that $b_u^{(i)}(v)$ is the amount of flow that u has routed to v after the i^{th} iteration in H_i. Then the above strategy is essentially that for each edge $uv \in M_i$, let

$$b_u^{(i)} = b_v^{(i)} = \frac{1}{2}(b_u^{(i-1)} + b_v^{(i-1)}),$$

where we assume for simplicity that the perfect matching M_i is integral. We note that at the beginning, $b_u^{(0)}(v) = \mathbb{1}_{u=v}$ is the indicator vector for all $u \in V$, i.e., $b_u^{(0)}(u) = 1$ and $b_u^{(0)}(v) = 0$ for all $v \neq u$.

Note. For the case of fractional matching, we have $b_u^{(i)} = b_u^{(i-1)}/2 + \sum_{v \in V} M(uv) b_v^{(i-1)}/2$. This will not make any difference, hence we just focus on the integral case.

We first observe the following, which simply formalizes that the routing is feasible in each iteration.

Claim. For all $u \in V$, $\sum_{v \in V} b_u^{(i)}(v) = 1$ for all i. And If H_{i-1} routes the demand matrix implied by $b_u^{(i-1)}$ for all $u \in V$, then H_i routes the demand matrix implied by $b_u^{(i)}$ for all $u \in V$ as well.

Proof. The first part is trivial. For the second part, since H_i differs from H_{i-1} via the matching M_i. For each $uv \in M_i$, we use the edge to exchange the flow at u and flow at v: $1/2$ capacity to send $b_u^{(i-1)}$ to $b_v^{(i-1)}$ and $1/2$ capacity to send $b_v^{(i-1)}$ to $b_u^{(i-1)}$. ⊛

Ideally, as we have discussed, we want $b_u(v) = 1/n$ for all u and v. In a sense, we are taking the weighted averages of $b_u^{(i)}$'s as dictated by M.

Intuition. In principle, pairing up very "different" $b_u^{(i)}$'s should produce averages closer to uniform.

The problem is again that we do not have the control of how the pairing is done, as it is given by M_{i+1}. We can only choose the next partition $(S_{i+1}, V \setminus S_{i+1})$. To do that, let $x_u^{(i)} = b_u^{(i)} - \vec{1}/n \in \mathbb{R}^V$, and consider the potential $\phi_i = \sum_{u \in V} \|x_u^{(i)}\|_2^2$, i.e.,

$$\phi_i = \sum_{(u,v) \in V \times V} \left(b_u^{(i)}(v) - \frac{1}{n} \right)^2.$$

We note that $\phi_0 = \sum_{(u,v) \in V \times V} (\mathbb{1}_{u=v} - 1/n)^2 = n(1 - 1/n) = n - 1$.

Intuition. ϕ_i measures the difference between the multi-commodity flow routed so far and a fully symmetric multi-commodity flow.

Note that in terms of $x_u^{(i)}$, we now have $\sum_{u \in V} x_u^{(i)} = 0$, and the update rule for $x_u^{(i)}$ becomes

$$x_u^{(i)} = x_v^{(i)} = \frac{1}{2}(x_u^{(i-1)} + x_v^{(i-1)}).$$

This potential is useful due to the following.

Claim. If $\phi_i \leq 1/4n^2$, then $b_u^{(i)}(v) \geq 1/2n$ for all $u, v \in V$, and hence H_i is a 1/2-expander.

5.2.3 Reducing Potential Via Matching

Now the question is, how can we effectively minimize ϕ_i. In this context, we can abstract out the graphs and flows, just focus on analyzing $x_u^{(i)}$ with the abstract update rule $x_u^{(i)} = x_v^{(i)} = (x_u^{(i-1)} + x_v^{(i-1)})/2$. A natural greedy strategy is that at each iteration, choose a new bipartition $(S_i, V \setminus S_i)$ that minimizes the maximum of $\phi_i = \sum_{u \in V} \|x_u^{(i+1)}\|_2^2$ over all fractional perfect matchings M_i of $(S_i, V \setminus S_i)$. We work backwards by first analyzing ϕ_i analytically as a function of M.

Lemma 5.2.1. Let M be an arbitrary perfect matching M on V, and let $x_u \in \mathbb{R}^k$ for some $k \in \mathbb{N}$ for all $u \in V$. Suppose for each matching edge $uv \in M$, $y_u = (x_u + x_v)/2$, then

$$\sum_{u \in V} \|y_u\|_2^2 = \sum_{u \in V} \|x_u\|_2^2 - \frac{1}{2} \sum_{uv \in M} \|x_u - x_v\|_2^2.$$

Proof. For any $u \in V$ and the matching edge $uv \in M$,

$$\begin{aligned}
\|y_u\|_2^2 &= \left\| \frac{1}{2} x_u + \frac{1}{2} x_v \right\|_2^2 \\
&= \frac{1}{4}\|x_u\|_2^2 + \frac{1}{4}\|x_v\|_2^2 + \frac{1}{2}\langle x_u, x_v \rangle \\
&= \frac{1}{4}\|x_u\|_2^2 + \frac{1}{4}\|x_v\|_2^2 + \frac{1}{4}\left(\|x_u\|_2^2 + \|x_v\|_2^2 - \|x_u - x_v\|_2^2\right) = \frac{1}{2}\|x_u\|_2^2 + \frac{1}{2}\|x_v\|_2^2 - \frac{1}{4}\|x_u - x_v\|_2^2.
\end{aligned}$$

By summing up over $u \in V$, since M is a perfect matching, we have the result. ■

Lemma 5.2.1 shows that, specifically for $x_u^{(i-1)} \in \mathbb{R}^V$, under the update rule (which is the same as our mixing rule), we can express the reduction of ϕ_{i-1} explicitly:

$$\phi_i = \phi_{i-1} - \frac{1}{2} \sum_{uv \in M_i} \|x_u^{(i-1)} - x_v^{(i-1)}\|_2^2. \tag{5.1}$$

Hence, the potential reduction for an edge uv is proportional to $\|x_u - x_v\|_2^2$. Thus, the cut player should somehow force the matching player to pair vertices that are *far apart*.

As previously seen. The cut player can only give a partition $(S_i, V \setminus S_i)$ and the matching player has control of the actual matching it generates.

It's quite hard to ensure this in high dimension. The intuition is to look at the easiest case:

Intuition. The key observation is that when $k = 1$, the potential goes down a lot.

We formalize this as follows.

Lemma 5.2.2. For each $u \in V$, let $z_u \in \mathbb{R}$ be a scalar such that $\sum_{u \in V} z_u = 0$. Suppose we partition V to be (A, B) according to the magnitude of z_u, where A and B is the first and the second half, respectively.[a] Then for every perfect matching M between A and B, we have

$$\sum_{uv \in M} \|z_u - z_v\|_2^2 = \sum_{uv \in V} (z_u - z_v)^2 \geq \sum_{u \in V} z_u^2.$$

[a] I.e., $A = \{u \colon z_u \leq \text{median}(\{z_v\}_{v \in V})\}$ and $B = V \setminus A$ such that $|A| = |B| = n/2$, with ties broken arbitrarily.

Proof. Let $\beta \in \mathbb{R}$ to be the median of z_u's. Then, for any perfect matching M between A and B, consider a matched edge $uv \in M$. We have $(z_u - z_v)^2 = ((z_u - \beta) - (z_v - \beta))^2 = (|z_u - \beta| + |z_v - \beta|)^2$, since $z_u \leq \beta$ and $z_v \geq \beta$. Summing up over all pairs in M, we have

$$\sum_{uv \in M} \|z_u - z_v\|_2^2 = \sum_{uv \in M} (|z_u - \beta| + |z_v - \beta|)^2 \geq \sum_{u \in V} z_u^2 - 2\beta \sum_{u \in V} z_u + n\beta^2 \geq \sum_{u \in V} z_u^2,$$

where we use the fact that $\sum_{u \in V} z_u = 0$ in the final inequality. ■

Remark. The potential reduction is huge, indeed, linear, at least for 1 dimension.

Proof. Recall that from Lemma 5.2.1, the potential is reduced by $\sum_{uv\in M}\|x_u - x_v\|_2^2/2$, which in this case is $\sum_{uv\in M}(z_u - z_v)^2/2$. From Lemma 5.2.2, this is at least half of the original potential ($\sum_{u\in V} z_u^2 = \sum_{u\in V}\|z_u\|_2^2$), so we get a linear decay. ⊛

5.2.4 Random Projection Cutting Strategy

A natural idea is then to reduce the original $x_u^{(i)} \in \mathbb{R}^V$ to one-dimensional.

Intuition. The general strategy inspired by Lemma 5.2.2 is then to project onto a random line.

Lecture 21: Finishing Cut-Matching Game

7 Nov. 11:00

Specifically, in iteration i, we have vectors $x_u^{(i)}$ for $u \in V$, where each $x_u \in \mathbb{R}^n$ and $\sum_{v\in V} x_u^{(i)}(v) = 0$. We can then reduce the problem to the one dimensional case by using the well-known idea of random projection that maps points in high dimensions to a line. Recall the following simple facts:

As previously seen. If $Z_i \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mu_i, \sigma_i^2)$ for $i = 1, 2$, then $Z_i + Z_2 \sim \mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$.

Furthermore, we also have the following.

Lemma 5.2.3. Let $g \sim \mathcal{N}(0, I_n)$ and let $x \in \mathbb{R}^n$ and $z := \langle g, x\rangle$. Then:

(a) $\mathbb{E}[z] = 0$;

(b) $\mathbb{E}[z^2] = \sum_{i=1}^k x_i^2 = \|x\|_2^2$;

(c) for any t, $\Pr(z^2 \geq t\|x\|_2^2) \leq e^{-t/2}$.

Proof. The first two are standard properties. We just make some additional remark on the third concentration bound for χ^2. Suppose $Z \sim \mathcal{N}(0, \sigma^2)$. Then, it's well known that we have

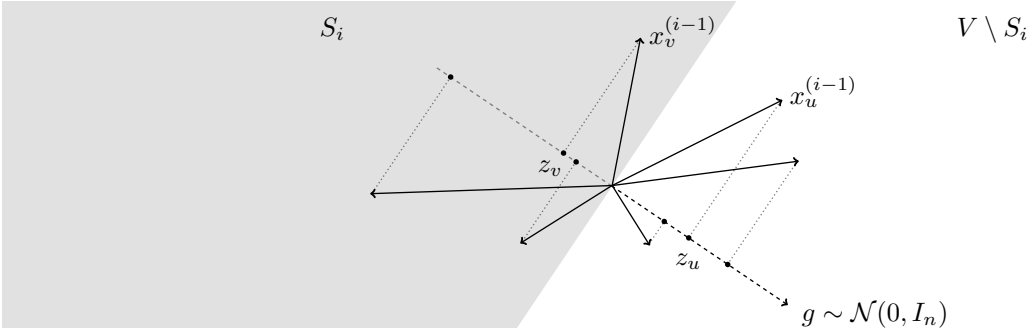
$$\Pr(Z > \beta) \leq e^{-\beta/2\sigma^2}.$$

Now, Z^2, the square of the Gaussian random variable, is a positive random variable following the so-called χ^2 distribution. Note that its mean is σ^2. It also exhibits strong concentration around its mean: specifically, when $Z \sim \mathcal{N}(0, 1)$, we have

$$\Pr(Z^2 \geq 1 + 4t) \geq e^{-t}.$$

Using this, the third concentration bound can be easily proved. ■

Hence, in iteration i, a natural strategy for the cut player is to project $x_u^{(i-1)}$ to one dimension via $z_u^{(i)} = \langle g, x_u^{(i-1)}\rangle$ and partition V into $(S_i, V \setminus S_i)$ by sorting $z_u^{(i)}$ and select the first half to be S_i:



This leads to the following guarantee.

Lemma 5.2.4. Let M_i be any perfect matching between S_i and $V \setminus S_i$ generated by the cut player with the linear projection strategy based on $x_u^{(i-1)}$ for $u \in V$. Then

$$\mathbb{E}\left[\sum_{uv \in M_i} (z_u - z_v)^2\right] \geq \phi_{i-1},$$

where $z_u \coloneqq \langle g, x_u^{(i-1)} \rangle$ for all $u \in V$, where $g \sim \mathcal{N}(0, I_n)$.

Proof. From the proof of Lemma 5.2.2, we have[a]

$$\sum_{uv \in M_i} (z_u - z_v)^2 \geq \sum_{u \in V} z_u^2 - 2\beta \sum_{u \in V} z_u,$$

where β is the median of z_u for $u \in V$. We see that by taking expectation on both sides, with Lemma 5.2.3, we have

$$\mathbb{E}\left[\sum_{uv \in M_i} (z_u - z_v)^2\right] \geq \mathbb{E}\left[\sum_{u \in V} z_u^2\right] - 2\beta \mathbb{E}\left[\sum_{u \in V} z_u\right] = \sum_{u \in V} \mathbb{E}\left[z_u^2\right] = \sum_{u \in V} \|x_u^{(i-1)}\|_2^2 = \phi_{i-1},$$

where we use the fact that the sum of mean zero Gaussian's is still a mean zero Gaussian. ■

[a] Since $\sum_{u \in V} z_u$ is not necessarily 0 as z_u is random.

With Lemma 5.2.4, we can then combine Lemma 5.2.2 and Lemma 5.2.1 to prove the following:

Lemma 5.2.5. Given current flow vectors $b_u^{(i-1)}$ and difference vectors $x_u^{(i-1)}$ for all $u \in V$, there is a randomized algorithm that produces a partition $(S_i, V \setminus S_i)$ such that for any perfect matching M_i between S_i and $V \setminus S_i$, the potential ϕ_i after mixing according to M_i satisfies

$$\mathbb{E}[\phi_i] \leq \left(1 - \frac{1}{c \log n}\right) \phi_{i-1} + O(n^{-10}),$$

where c is some sufficiently large fixed constant.

Proof. Consider the linear projection strategy described above. Since $(z_u - z_v)^2 = \langle g, x_u^{(i-1)} - x_v^{(i-1)} \rangle^2$, then by Lemma 5.2.3, with high probability,

$$(z_u - z_v)^2 = \langle g, x_u^{(i-1)} - x_v^{(i-1)} \rangle^2 \leq c \log n \cdot \|x_u^{(i-1)} - x_v^{(i-1)}\|_2^2. \tag{5.2}$$

Assuming this happens deterministically. Then, from Lemma 5.2.4 and Equation 5.1,

$$\mathbb{E}[\phi_{i-1} - \phi_i] = \frac{1}{2}\mathbb{E}\left[\sum_{uv \in M_i} \|x_u^{(i-1)} - x_v^{(i-1)}\|_2^2\right] \geq \frac{1}{c \log n}\mathbb{E}\left[\sum_{uv \in M_i} (z_u - z_v)^2\right] \geq \frac{1}{c \log n}\phi_{i-1},$$

which implies $\mathbb{E}[\phi_i] \leq (1 - 1/c \log n)\phi_{i-1}$. To handle the case that Equation 5.2 doesn't happen deterministically, we need to do a little of extra work, which is why we get the weaker bound with a slight additive bound. We omit the exact detail for simplicity. ■

Intuition. Lemma 5.2.5 states that using random projection, we only loses a $1/\log n$ factor compared to the true one-dimensional case.

Remark. Lemma 5.2.5 implies that the algorithm uses only the information about the flow routed so far and nothing about the previous matchings.

Lemma 5.2.5 immediately imply Theorem 5.1.1:

Proof of Theorem 5.1.1. From the previous claim, we know that we want to reduce $\phi_0 \le n$ to $1/4n^2$. As $\phi_i \le (1 - 1/c\log n)\phi_{i-1}$ for some constant $c > 0$ from Lemma 5.2.5, we have

$$\phi_i \le \left(1 - \frac{1}{c\log n}\right)^i \phi_0 \le \left(1 - \frac{1}{c\log n}\right)^i n.$$

Setting the right-hand side to be less than $1/4n^2$ and solve for i, we have $i = O(\log^2 n)$. ■

This completes the whole proof. Finally, we describe the whole cut-matching game in Algorithm 5.1 for completeness.

Algorithm 5.1: Cut-Matching Game

Data: A graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, iteration k
Result: A certification graph $H = (V, \bigcup_{i=1}^k M_i)$

for $u \in V$ **do** // Initialization
  $x_u^{(0)} \leftarrow \mathbb{1}_{u=v} - \vec{1}/n$

while $i = 1, \dots, k$ **do**
  $S_i \leftarrow$ `Cut-Player`$(\{x_u^{(i-1)}\}_{u\in V})$
  $M_i \leftarrow$ `Matching-Player`(G, S_i)
  for $uv \in M_i$ **do**
    $x_u^{(i)} \leftarrow (x_u^{(i-1)} + x_v^{(i-1)})/2$
    $x_v^{(i)} \leftarrow (x_u^{(i-1)} + x_v^{(i-1)})/2$

return $(V, \bigcup_{i=1}^k M_i)$

`Cut-Player`$(\{x_u\}_{u\in V})$:
  $g \leftarrow$ `Normal`$(0, I_n)$ // $g \sim \mathcal{N}(0, I_n)$
  for $u \in V$ **do**
    $z_u \leftarrow \langle g, x_u\rangle$
  $\beta \leftarrow$ `Median`$(\{z_u\}_{u\in V})$
  $S \leftarrow \{u \in V : z_u \le \beta\}$ // Break ties arbitrarily such that $|S| = n/2$
  return S

`Matching-Player`(G, S):
  $(G', s^*, t^*) \leftarrow$ `Build-Linkage-Routing-Graph`$(G, S, V \setminus S)$ // Figure 5.1
  $M \leftarrow$ `Single-Commodity-Flow`(G', s^*, t^*) // Compute routable perfect matching
  return M

Note. Explicitly, we run Algorithm 5.1 with $k = O(\log^2 n)$, and whenever M is not a valid perfect matching or is not routable (indicating that S and $V \setminus S$ is not well-linked), we just report that G is not a 1-expander.

5.3 Application to Treewidth

Previously, we have mentioned *treewidth* briefly (e.g., Theorem 2.4.3). Now, we see another good application of the cut-matching game to treewidth.

5.3.1 Tree Decomposition and Treewidth

We care about how "tree-like" does the graph look. One of the (among many) interesting properties of a tree is that we can separate the tree in a balanced way by removing only one node:

Example (Tree). For a tree T, there exists a single vertex such that after removing it, no connect component has more than $2n/3$ vertices, i.e., there exists a singleton balanced separator.

What if we allow removing more vertices?

Example (Series-parallel graph). Series-parallel graph has the same property when allowing removing 2 vertices.

Taking these two examples into account, we consider the following notion of tree decomposition.

Definition 5.3.1 (Tree decomposition). Given a graph $G = (V, E)$, a *tree decomposition* of G is a tree $T = (V_T, E_T)$ and a function $B \colon V_T \to 2^V$ such that

(a) for all $u \in V$, the set of "bags" $B(t)$ that contains u form a connected sub-tree of T;

(b) for all $uv \in E$, there exists some $t \in V_T$ such that both $u, v \in B(t)$.

Definition 5.3.2 (Width). The *width* of a tree decomposition T of G with the function $B \colon V_T \to 2^V$ is defined as $\text{width}(T) = \max_{t \in V_T} |B(t)|$.

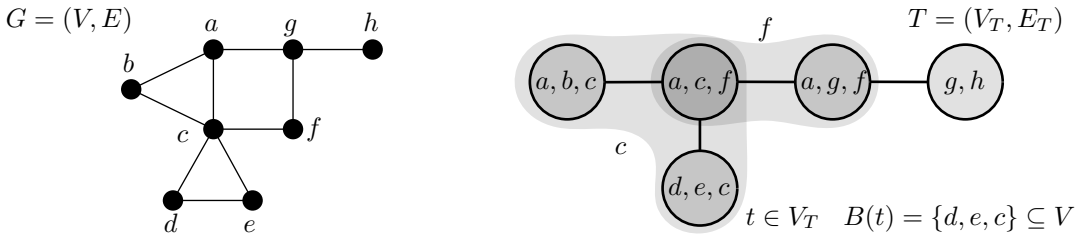


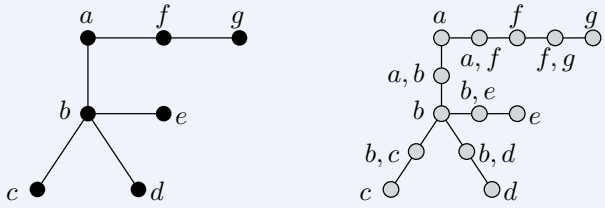
Figure 5.2: The tree decomposition of G with its "bag" plotted explicitly.

Then, we can define the treewidth formally.

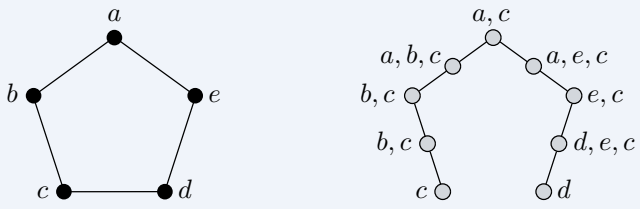
Definition 5.3.3 (Treewidth). Given a graph G, the *treewidth* $\text{tw}(G)$ is defined as $\min_T \text{width}(T) - 1$, where the minimum is taking over all possible tree decompositions of G.

The minus one is because the minimum maximum width of the tree decomposition for a tree is 2:

Example (Tree). For any tree T, there exists a tree decomposition with width 2, i.e., $\text{tw}(T) = 1$.



Example (Cycle). For any cycle C_n, there exists a tree decomposition with width 3, i.e., $\text{tw}(C_n) = 2$.



Example (Series-Parallel). We have that $\mathrm{tw}(G) \leq 2$ if and only if G is series-parallel.

Example (Clique). For any clique K_n, we have $\mathrm{tw}(K_n) = n - 1$ with the trivial tree decomposition, i.e., $T = (V_T, E_T)$ with $V_T = \{t\}$, $E_T = \varnothing$ such that $B(t) = V$.

5.3.2 Balanced Separator and Treewidth

Essentially, Definition 5.3.3 captures the desired balanced separator properties we mentioned above:

Intuition. Any graph G with $\mathrm{tw}(G) \leq k$ implies that G can be recursively decomposed via balanced separators of size k. Approximate converse also holds, i.e., if there is a subgraph H of G with no balanced separator of size k, then $\mathrm{tw}(G) \geq k/c$ for some $c > 0$.

Formally, we have the following:

Lemma 5.3.1. Suppose $\mathrm{tw}(G) \leq k$. Then for any $X \subseteq V$, there exists a balanced separator S for X of size $|S| \leq k$, i.e., $G - S$ has no component with more than $2|X|/3$ vertices from X.

The idea is that for any edge $(t, t') \in E_T$ of the tree decomposition T that achieves the optimal width (i.e., $\mathrm{tw}(G) + 1$), $B(t) \cap B(t')$ is a separator of G. Hence, $\mathrm{tw}(G) \leq k$ implies G has a balance separator S of size k. The recursive property follows from that fact that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ for any subgraph H of G. Let us rephrase Theorem 2.4.3 as follows:

Lemma 5.3.2. For any graph G, $\mathrm{tw}(G) \geq k$ if and only if there exists X such that $|X| \geq k/4$ such that X is node-well-linked.

With these characterizations, we have the following examples.

Example (Grid). For $\sqrt{n} \times \sqrt{n}$ grid G, $\mathrm{tw}(G) = \sqrt{n} - 1$.

Indeed, grid the worst case for planar graphs.

Example (Planar separator theorem). For any planar graph G, $\mathrm{tw}(G) = O(\sqrt{n})$ [LT79].

Example (Random graph). For random d-regular graph G, $\mathrm{tw}(G) = \Theta(n)$ with high probability.

Proof. Since random d-regular graph is an expander with high probability, and balanced separator is of size $\Omega(n)$ for expander. ⊛

5.3.3 Cut-Matching Game to Certify Treewidth

Now, we try to verify the treewidth. Verifying a small treewidth is easy, we just ask for a tree decomposition as a certification. In general, given G and k, checking if $\mathrm{tw}(G) \leq k$ is NP-complete [ACP87]. However, it is fixed-parameter-tractable, i.e., for every fixed k, there is a linear-time algorithm (specifically, $O(k^{k^3} n)$), that, given G, decides whether $\mathrm{tw}(G) \leq k$ [Bod93]. As for an approximation algorithm for computing treewidth, it takes $O(2^k n)$ time for a 5-approximation [Bod+16], while for the best known polynomial-time approximation algorithm is $O(k\sqrt{\log k})$-approximate [FHL05].

Now, it turns out that instead of directly certifying/computing the treewidth, from the understanding of the connection between the size of balanced separator and treewidth, with the fact that expander has large balanced separator, one natural idea is the following:

Intuition. We try to "embed" an expander using cut-matching game.

Utilizing this idea, the following is known:

Theorem 5.3.1. Let $G = (V, E)$ be a graph with treewidth $\geq k$. Then three is a randomized algorithm that computes a subgraph $H = (V, E_H)$ such that with high probability,

(a) $\mathrm{tw}(H) = \Omega(k/\mathsf{poly}\log k)$, where $\mathsf{poly}\log k$ is something like $\log^2 k$;

(b) $\deg_H(v) = O(\log^2 k)$.

Theorem 5.3.1 is further improved as follows:

Theorem 5.3.2 ([CE13; CC14])**.** Let $G = (V, E)$ be a graph with treewidth $\geq k$. Then three is a randomized algorithm that computes a subgraph $H = (V, E_H)$ such that with high probability,

(a) $\mathrm{tw}(H) = \Omega(k/\mathsf{poly}\log k)$, where $\mathsf{poly}\log k$ is something like $\log^{100} k$;

(b) $\deg_H(v) = 3$.

Remark. The full generality of Theorem 5.3.1 and Theorem 5.3.2 are stated as topological minor. We omit the exact details.

5.3.4 Algorithm with Treewidth

The reason why we care about treewidth is that there are several "templates" for applications:

- if G has *small* (constant) treewidth, we can then solve the problem via dynamic programming on the tree decomposition;
- if G has *large* treewidth, we can then use structure, in particular, obstructions such as grids:
  - answer is clear from obstruction, or
  - "reduce" the problem in some fashion and recurse.

Let's just look at the case that when the treewidth is small. In this case, dynamic programming based algorithms for trees extends naturally to bounded treewidth graphs.

Example (Problems on planar graph)**.** From the planar separator theorem, we can solve many problems in $2^{\sqrt{n}}$ using dynamic programming for planar graphs.

One such problem is the maximum (weighted) independent set:

Problem 5.3.1 (Maximum weighted independent set)**.** Given a graph $G = (V, E)$ with vertex capacity $w \colon V \to \mathbb{R}$, the *maximum weighted independent set* (MWIS) problem asks for computing $\max_{S \subseteq V} w(S)$ such that $S \subseteq V$ is an independent set.

MWIS is NP-hard, even for planar graphs, and more generally, is also very hard even to approximate in general graphs. However, we can still solve it efficiently (in polynomial time) in bounded treewidth graphs. To get some intuition, let's first see how to solve it for tree using dynamic programming.

Example (MWIS on tree)**.** Given a tree T with vertex capacity $w \colon V \to \mathbb{R}$, r be an arbitrary root and T_v be the sub-tree of T rooted at node $v \in V_T$. Define $\mathsf{OPT}(v)$ to be the optimum value of MWIS on T_v. Furthermore, let $\mathsf{OPT}(v, 1)$ be the optimum value of MWIS on T_v that includes v, and $\mathsf{OPT}(v, 0)$ is defined similarly such that it does not include v. Then, we have $\mathsf{OPT}(v) = \max(\mathsf{OPT}(v, 1), \mathsf{OPT}(v, 0))$. After some observation, we see that

$$\begin{cases} \mathsf{OPT}(v, 1) = w(v) + \displaystyle\sum_{u \text{ child of } v} \mathsf{OPT}(u, 0); \\ \mathsf{OPT}(v, 0) = \displaystyle\sum_{u \text{ child of } v} \mathsf{OPT}(u). \end{cases}$$

Using this, we can solve the corresponding dynamic programming problem efficiently.

Let's now see how to use the above idea to solve MWIS for graphs with low treewidth.

Example (MWIS on tree decomposition)**.** Given a tree decomposition of width k for a graph G, MWIS can be computed in $O(2^k \mathsf{poly}(n))$ time.

Proof. Let $T = (V_T, E_T)$ be the corresponding tree decomposition with an arbitrary root $r \in V_T$. For $t \in V_T$, let $V_t \coloneqq \bigcup_{t' \in V(T_t)} B(t') \subseteq V$ be the vertices in all bags of T_t.

Now, let $\mathsf{OPT}(t, S)$ be the optimum value of MWIS in G among independent sets $I \subseteq V_t$ such that $I \cap B(t) = S$. It's possible to define recurrence via $\mathsf{OPT}(t, S)$, hence dynamic programming.

The number of values to compute at each node $t \in V_T$ is at most 2^{k+1}, and each takes $O(k)$. Hence, one can compute all values from leaves to the root in $O(k2^{k+1}|V_T|)$ time. ⊛

The above yield a $2^{O(\sqrt{n})}$ time algorithm for planar graphs. However, it's worth noting that it's possible to get a fast $(1-\epsilon)$-approximation algorithm in planar graphs. For that, we need the following.

Theorem 5.3.3. For any planar graph $G = (V, E)$ and integer $h \geq 3$, V can be colored with h colors such that for any $i \in [h]$, $\mathrm{tr}(G - V_i) \leq 3h$ where $V_i \subseteq V$ is the set of vertices colored with color i.

Example (MWIS on planar graph)**.** Consider MWIS on planar graphs. Let $\epsilon \in (0, 1)$ and $h = 1/\epsilon$. From Theorem 5.3.3, consider removing V_i from G and compute the corresponding MWIS to get OPT_i in $G - V_i$ using the previous example's approach, which takes $O(2^{1/\epsilon} n/\epsilon)$ since $\mathrm{tr}(G - V_i) \leq 3h = O(1/\epsilon)$. In total, there are $h = 1/\epsilon$ of V_i's, hence the total running time is $O(2^{1/\epsilon} n/\epsilon^2)$.

Then, we simply output the final answer as $\max_{i \in [h]} \mathsf{OPT}_i$, say it's OPT_{i^*}. From the pigeonhole principle, the output OPT_{i^*} for the graph $G - V_{i^*}$ is $(1-\epsilon)$-approximation compared to the original optimum value OPT for the whole graph G.

Chapter 6

Blocking Flow and Push-Relabel

Lecture 22: Blocking Flow and Link-Cut Tree

19 Nov. 11:00

We have mentioned the max-flow problem multiple times throughout the course, which is used as a primitive for several problems. For example, from Theorem 1.3.6, we know that solving Steiner min-cut reduces to polynomial-logarithmically many max-flot computations via isolating cut. Hence, in this chapter, we present several classical max-flow algorithms which are not usually taught.

Notably, all the algorithms we will see are based on the augmenting path approach, first introduced by Ford and Fulkerson [FF56].

6.1 Augmenting Path Framework

In this section, we first introduce the *augmenting path framework* formally, and briefly talk about a naive algorithm for solving max-flow in $O(m^2n)$ time [Din70; EK72].

6.1.1 Ford-Fulkerson Algorithm

Consider a directed graph $G = (V, E)$ with capacities $c\colon E \to \mathbb{R}_+$. In the augmenting path framework, algorithms repeatedly find an augmenting path (or a collection of them) in the residual graph:

Definition 6.1.1 (Augmenting path)**.** An *augmenting path* is a simple path through the directed graph $G = (V, E)$ using only edges with positive capacity from the source to the sink.

Since in this chapter, we will talk about directed edge (u, v) and the reversed edge $(v,)$ quite a lot, consider the following for convenience.

Notation (Reversed edge)**.** For any directed edge $e = (u, v) \in E$, let $e^{-1} \coloneqq (v, u)$.

Definition 6.1.2 (Residual graph)**.** Let $G = (V, E)$ be a directed graph with capacities $c\colon E \to \mathbb{R}_+$ and a flow f. The *residual graph* G_f is defined as (V, E_f) with capacities $c_f\colon E_f \to \mathbb{R}_+$ where $c_f(e) = c(e) - f(e) + f(e^{-1})$ for all $e \in E$ such that $E_f \coloneqq \{e \in V \times V \mid c_f(e) > 0\}$.[a]

[a] Without loss of generality, we can assume $E = V \times V$ throughout by adding 0 capacity edge.

Figure 6.1: Residual capacity c_f induced by f on e and e^{-1}.

Remark. Definition 6.1.2 might not be the most common and natural way to define residual graph, or in particular, residual capacity. The reason is that the definition of flow we use is not the one for *net flow*. This make some formulations a bit messier but essentially equivalent.

The basic framework of the Ford-Fulkerson algorithm is to repeatedly augment the flow along an augmenting path in the current residual graph with a value equal to the bottleneck capacity, until we cannot find any augmenting path.

Algorithm 6.1: Ford-Fulkerson Algorithm for s-t Max-Flow [FF56]

Data: A connected directed graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, source s, sink t
Result: An s-t max-flow f

1 $f \leftarrow 0$ // Initialize a flow
2 $c_f \leftarrow c$ // Initialize G_f
3 **while** $\exists s$-t *augmenting path* $P \in G_f$ **do**
4 $\quad b \leftarrow \min_{e \in P} c_f(e)$ // Compute bottleneck capacity
5 $\quad$ **for** $e \in P$ **do**
6 $\quad\quad f(e) \leftarrow f(e) + b$
7 $\quad\quad c_f(e) \leftarrow c(e) - f(e) + f(e^{-1})$
8 **return** f

6.1.2 Naive Algorithm

There are several variants we can do when finding the augmenting paths in line 3. To be naive, one can literally run Algorithm 6.1 without any care. This will work already, by recalling the classical proof of max-flow min-cut theorem:

Theorem 6.1.1 (Max-flow min-cut)**.** If there is no augmenting path in the residual graph G_f for some s-t flow f, then f is the s-t max-flow.

Proof. Suppose there is no augmenting path in G_f, i.e., there is no valid simple s-t path. Then, it's clear that there is an S-T cut in G_f where $s \in S$ and $t \in T = V \setminus S$, such that all edges from S to T are saturated, i.e., $c_f(u, v) = 0$ for $u \in S$ and $v \in T$. Hence, we have $f(u, v) = c(u, v)$.

Furthermore, all edges from T to S have zero flow, i.e., $f(v, u) = 0$ for $u \in S$ and $v \in T$. To see this, suppose there is some (v, u) such that it carries some non-zero flow $f(v, u)$. Then, there exists a backward edge from u to v in G_f, and hence there is a path from s to u and to v in G_f, a contradiction.

Combining these facts, the capacity of the cut (S, T) is equal to $|f|$, and from the fact that any flow has value less than the capacity of every possible cut, (S, T) is the s-t min-cut that witness the s-t max-flow as well. ■

With Theorem 6.1.1, we can prove the following.

Theorem 6.1.2. Algorithm 6.1 computes the s-t max-flow on a directed graph with integral edge capacities in $O(m|f^*|)$ time, where $|f^*|$ is the s-t max-flow value.

Proof. The correctness is from Theorem 6.1.1. For the runtime, since each augmenting path computation can be easily done in $O(m)$ time with BFS or DFS, and for each augmentation, the flow value increased by at least 1. ■

Corollary 6.1.1. There exists an integral max-flow when the graph is with integral capacities.

Proof. It is obvious from the proof of Theorem 6.1.2. ■

Corollary 6.1.2. Let $W = \max_{e \in E} c(e)$, then Algorithm 6.1 actually runs in $O(mnW)$.

Proof. Since $|f^*|$ is equal to the s-t min-cut cost from the max-flow min-cut theorem, which is less than $c(\delta^+(s))$ for example. Then since $|f^*| \le c(\delta^+(s)) \le nC$, the algorithm runs in $O(mnW)$. ■

We see that Corollary 6.1.2 proves that the naive implementation of Algorithm 6.1 does not lead to a truly polynomial time algorithm since the running time depend linearly on W.

6.1.3 Maximum Bottleneck Capacity Augmenting Path

To be a bit more careful, the next idea is to always choose an augmenting path P that has the maximum bottleneck capacity $\max_{e \in P} c_f(e)$ in line 3. Such an augmenting path can be found in $O(m \log n)$ time via binary search over the capacity of the bottleneck edge, and, for each guess, check for the s-t path in G_f after all the edges with capacity less than the guess were removed.

> **Note.** By being (much) more sophisticated, one can get $O(\min(m + n \log n, m \log^* n))$ time.

To see this leads to an actual improvement, we want to know how much progress each such augmenting path guarantees to make.

> **As previously seen.** The flow decomposition bound tells us that any flow can be decomposed into at most m flow paths.

> **Theorem 6.1.3.** By choosing the maximum bottleneck capacity augmenting path in Algorithm 6.1, the algorithm computes the s-t max-flow on a directed graph $G = (V, E)$ with integral edge capacities in $O(m^2 \log n \log nW)$ time, where $W = \max_{e \in E} c(e)$.
>
> **Proof.** If we apply the above observation to the maximum "remaining" flow, i.e., the maximum flow in the residual graph G_f together with an averaging argument, we can conclude that at least one of these paths carries at least $1/m$-fraction of the remaining value of the flow. Hence, each flow augmentation reduces the remaining value of the flow by a factor of $1 - 1/m$.
>
> After $O(m \log |f^*|)$ augmentations, the remaining flow would be less than 1. Since we have integral capacities, the obtained solution has to be optimal. We conclude that the overall running time is $O(m \log n \cdot m \log |f^*|) = O(m^2 \log n \log nW)$. ■

Theorem 6.1.3 proves that Algorithm 6.1 with choosing maximum bottleneck capacity augmenting path leads to a weakly polynomial time algorithm since the running time now only depend on $\log W$, which is the same as the capacity representation. However, we can ask for more.

6.1.4 Shortest Augmenting Path

Unlike the maximum bottleneck approach, which depend on the value representation; instead, the next idea leads to an algorithm that runs in $O(m^2 n)$ [Din70; EK72] via a naive implementation. The key is to implement line 3 by choosing the shortest augmenting path (shortest w.r.t. the number of edges). From now on, unless specified, the capacities are not necessarily integral.

> **As previously seen.** Maximum bottleneck capacity augmenting path is "primal greedy," i.e., it applies a greedy strategy in which we improve our current solution in each step and then lower bounded the progress made by each such step in terms of the value of the s-t max-flow.

Therefore, as the value of the s-t max-flow directly depends on the value of capacities, this kind of "primal greedy" approach is inherently incapable of providing a strongly polynomial bound. Hence, we need a different way to measure our progress towards optimality. To see how can we improve upon this observation, consider the following.

> **As previously seen.** The way we certified the optimality of our current flow f was by looking whether s and t are connected in G_f. If not, f was already a s-t max-flow; otherwise, it was not optimal (and the s-t path enables us to improve it).

The question then is, can we make this yes/no statement quantitative to also be able to differentiate between the flows that are "almost maximum" and the ones that might be "far" from being maximum? To this end, consider the s-t distance $d_f(s, t)$ in G_f, where each edge with positive residual capacity has length one, and every edge with zero residual capacity has length $+\infty$. The intuition of considering $d_f(s, t)$ is the following.

Intuition. Any flow in G_f has a fixed total volume ($\leq m$ for unit capacity graph). Hence, if s and t are far apart, then not much flow can fit in G_f, as each flow path has to utilize a lot of volume.

Specifically, observe that if $d_f(s,t) \geq n$, then s and t are disconnected in G_f, meaning that f is already a s-t max-flow. Hence, naturally, our new goal is to design an augmenting path based algorithm that aims to increase the s-t distance $d_f(s,t)$ in G_f.

Problem. What are the best augmenting paths to use if we are interested in increasing $d_f(s,t)$?

Answer. Shortest augmenting paths are the obstacles to $d_f(s,t)$ being large. So, we simply try to destroy them by augmenting the flow with it. ⊛

We note that finding the shortest augmenting path correspond to running BFS in G_f. Hence, it only takes $O(m)$ time, hence the only challenge is how to ensure that this augmentation does not introduce new shortest paths in G_f.

Lemma 6.1.1. Let $d_f(s,v)$ and $d'_f(s,v)$ to be the distance from s to $v \in V$ in G_f before and after augmenting the flow along some shortest augmenting path P, respectively, then $d_f(s,v) \leq d'_f(s,v)$.

Proof. Suppose the opposite, and let $A \neq \varnothing$ to be the set of vertices v such that $d_f(s,v) > d'_f(s,v)$. Let $v \in A$ be the one with the minimum $d'_f(s,v)$. Let P' be the shortest s-v path in G'_f *after* the augmentation, and let u be the vertex preceding v on this path (since $v \neq s$, such path and u exist). Hence, we have $d'_f(s,v) = d'_f(s,u) + 1$. Moreover, we must have that $d_f(s,u) \leq d'_f(s,u)$ since otherwise, $u \in A$ and $d'_f(s,u) < d'_f(s,v)$, contradicting the minimality of v.

Claim. The last edge of P', i.e., (u,v), has zero residual capacity before augmented by P.

Proof. Otherwise, we can reach v from u before augmented by P, hence

$$d_f(s,v) \leq d_f(s,u) + 1 \leq d'_f(s,u) + 1 = d'_f(s,v),$$

contradicting to the fact that $v \in A$. ⊛

Hence, the only way for (u,v) to have non-zero residual capacity after augmented by P would be if the edge (v,u) belongs to P. Since P is the shortest path before the augmentation, i.e., $d_f(s,u) = d_f(s,v) + 1$. This means

$$d_f(s,v) = d_f(s,u) - 1 \leq d'_f(s,u) - 1 = d'_f(s,v) - 2 \leq d'_f(s,v),$$

which again contradicts to the assumption that $v \in A$. ■

Note. Lemma 6.1.1 states that the distance from s doesn't decrease, not only for t, but for every v.

By symmetry, we can argue that the distance from any v to t is also non-decreasing. Hence, augmenting the flow using shortest paths indeed does not make things worse. We can further show that we're indeed making progress.

Lemma 6.1.2. At most $mn/2$ shortest path augmentations can be made before $d_f(s,t) \geq n$.

Proof. We first note that each augmentation saturates at least one bottlenecking edge (u,v). For an already saturated edge, before it can be saturated again in some subsequent augmentation, we must have pushed some flow via an augmenting path that contained the opposite edge (v,u).

Let $d(w)$ be the distance from s to $w \in V$ in the residual graph just before the first saturation of (u,v), and let $d'(w)$ be the corresponding distance just before the flow is pushed along (v,u) in some subsequent augmentation that makes (u,v) has non-zero residual capacity again, as mentioned above. Since we always augment the shortest paths, $d(v) = d(u) + 1$ and $d'(u) = d'(v) + 1$.

From Lemma 6.1.1, we know that $d(w) \leq d'(w)$ for any $w \in V$. Hence, we have

$$d'(u) = d'(v) + 1 \geq d(v) + 1 = d(u) + 2.$$

Therefore, the distance from s to u has to increase by at least 2 by the time the edge (u, v) can again be saturated by some augmenting path. We conclude that each edge (u, v) can be saturated at most $n/2$ times before $d_f(s, u) \geq n$, so, there is at most $mn/2$ saturations and thus augmentations possible before $d_f(s, t) \geq n$. ■

Note. We have no way of lower bounding how much flow a particular flow augmentation pushed. We can only argue that overall the $mn/2$ augmentations we managed to push the whole max-flow value. This is an important feature of the so-called primal-dual algorithms.

Hence, we have the following.

Theorem 6.1.4 ([Din70; EK72]). By choosing the shortest augmenting path (w.r.t. $d_f(s, t)$) in Algorithm 6.1, the algorithm computes the s-t max-flow on a directed graph with general capacities in $O(m^2n)$ time.

Proof. From Lemma 6.1.2, we know that at most $O(mn/2)$ augmentations is needed. Since finding each shortest augmenting path requires $O(m)$, the total running time takes $O(m^2n)$. ■

6.2 Blocking Flow

Dinitz then consider augmenting the so-called blocking flow in line 3, based on the analysis of the shortest augmenting path approach in Theorem 6.1.4. It turns out that this leads to a speedup of the running time from $O(m^2n)$ to $O(mn^2)$ [Kar73; Din70; GR98]. The idea is not to be wasteful:

As previously seen. Previously, we recompute the shortest augmenting path using $O(m)$ time for each augmentation. Each search for a path via BFS gives us the whole shortest path tree, but we're only using one path from it.

The reason why this is wasteful is that augmenting along a single path can take m iterations before the current shortest s-t path distance in G_f increases by one. Instead of augmenting along a single path, we try to do this for a collection of paths.

Intuition. If we find a flow g in G_f such that g *blocks* all s-t shortest paths of length at most d, then after adding g to f in the residual graph, the shortest path distance increases by one. This can happen at most n times, since the shortest path distance between s and t is at most n.

This intuition leads to the so-called blocking flow, defined as follows.

Definition 6.2.1 (Blocking flow). An s-t flow f is a *blocking flow* if for every s-t path p, at least one edge in p is saturated by f.

Note. In other words, a blocking flow is a maximal flow in that one cannot add greedily add flow to f in G, which is different from a max-flow.

To find a blocking flow in a general graph G, we can do a simple greedy search, where in each iteration we find an s-t path and add flow along the path to saturate one bottlenecking edge. The capacities along the path are reduced and those that saturated are not considered in future path selections.

Intuition. We are not augmenting, since all we need is a maximal set that "blocks" all s-t paths.

It's clear that the greedy algorithm will terminate in $O(m)$ iterations, and a naive implementation (e.g., DFS) takes $O(m^2)$ time. We claim the following while refer the proof to Lemma 6.2.1.

Claim. Suppose f is a blocking flow in G, then $d_{G_f}(s,t) > d_G(s,t)$.

The above basically just a restatement of the intuition we have, which leads to an $O(m^2n)$ algorithm, without a runtime improvement compared to Theorem 6.1.4. We summarize this first attempt in Theorem 6.2.1.

Theorem 6.2.1. By choosing a blocking flow in Algorithm 6.1 using naive DFS search, the algorithm computes the s-t max-flow on a directed graph with general capacities in $O(m^2n)$ time.

Proof. We know that we only need to augment n times, and in each iteration, with $O(m^2)$ time, we can find a blocking flow g in G_f and (finally) augment along g. ■

6.2.1 Blocking Flow in Layer Graph

The key observation that leads to an improvement from $O(m^2n)$ to $O(mn^2)$ is that instead of finding a blocking flow in the entire graph G_f, it suffices to find it only in the "layer graph" consisting of the "forward edges" in the BFS computation. We first formalize the notion of "forward edge" as follows.

Definition 6.2.2 (Admissible). Given a directed graph $G = (V, E)$ and a source $s \in V$, an edge $e \in E$ is *admissible* if it points away from s, i.e., e is a part of some s-v shortest path for some $v \in V$.

We can also talk about the admissible graph, i.e., the graph (V, E') where E' is the set of all admissible edges.[1] Meanwhile, the admissible path is a path in the admissible graph.

Note. The admissible edges "hop"[a] exactly one BFS layer, and all other edges either stay in the same layer or go back.

[a] This is different from the previous notion of "hop" for negative edges.

The whole reason why augmenting blocking flows on the admissible graph works is because we can still guarantee that we're making progress:

Lemma 6.2.1. Suppose g is a blocking flow in the admissible graph of G_f, then $d_{f'}(s,t) > d_f(s,t)$, where $f' = f + g$.

Proof. Let f and f' be the flow before and after augmentation via the blocking flow g. Since every edge on a shortest s-t path in G_f has to "hop" exactly one BFS layer, hence it has to be admissible. Furthermore, augmentation does not create in $G_{f'}$ any edges that would be admissible in G_f, i.e., no new arcs in $G_{f'}$ would "hop" a BFS layer in G_f.

After the augmentation, there is some s-t cut in G_f such that all its admissible edges are saturated. Hence, every s-t path in $G_{f'}$ now has to use at least one edge that would not be admissible in G_f, so that edge would not hop a BFS layer in G_f. But, every edge in $G_{f'}$ as well as G_f can hop at most one BFS layer in G_f, so there is no way for an s-t path in $G_{f'}$ to make up for not hopping a G_f BFS layer at least once. All such paths had to have length larger than the s-t distance in G_f, hence the s-t distance in the new graph had to increase by at least one. ■

Hence, we conclude that after at most n blocking flow computations, we will have an s-t max-flow.

Note. The blocking flow computations only need to be done on a smaller admissible graph.

This is faster since computing blocking flow in a layer DAG (e.g., the admissible graph) can be done faster than the naive greedy algorithm in a general graph by "adaptive DFS search".

Intuition. Augmenting the flow using an admissible path does not create new admissible edges in the residual graph. Hence, when we saturate an edge on an admissible path, we can just discard it, as the reverse arc is never admissible.

Next, we discuss how to do this type of adaptive DFS search formally.

[1] Implicitly, we will discard edges (and nodes) with distance beyond $d(s,t)$.

6.2.2 Unit Capacity Graph

Let's first consider the case of unit capacity graph. In this case, observe that a blocking flow in the admissible graph corresponds to a maximal collection of edge-disjoint admissible path. Again, since we're finding a maximal set, greedy is usually the best strategy. The natural greedy strategy is to consider running a DFS, starting at s:

1. In each step, *advance* along an unexplored outgoing admissible edge.
2. If reach t, backtrack the path back to s and *block* it by adding it to the blocking flow and discarding all edges on this path.
3. If reach a vertex $v \neq t$ with no outgoing admissible edges, *retreat* back along the edge we came from to v and discard it.
4. Once there is no admissible edges leaving s, return the blocking flow.

This seems much like just search for an augmenting path, but actually this is not.

Intuition. The key difference is that we can save information about which part of the graph we already explored, hence this is "adaptive."

Crucially, as we never add new admissible edges, once a vertex is a dead end, it stays that way. This is why it is fine for us to discard an edge upon retreat.

Theorem 6.2.2. By choosing the blocking flow in Algorithm 6.1 using adaptive DFS search, the algorithm computes the s-t max-flow on a directed graph with unit capacities in $O(mn)$ time.

Proof. With Lemma 6.2.1, we only need to prove that the above adaptive DFS search takes $O(m)$ for finding a blocking flow in the admissible graph. There are three types of operations:

- *Retreat*: There is at most m retreats, as we always discard the edge when we do that. Hence, the total cost is $O(m)$.
- *Advance*: Advances can be charged to the total cost of retreats and blocks corresponding to traversing back the edge.
- *Block*: Each blocking of a path P takes $O(|P|)$ time, but also discards $|P|$ edges. Hence, with amortization, it is $O(1)$ per edge. The total time is then $O(m)$.

Hence, the total cost is $O(m)$, which proves that the total runtime is $O(mn)$. ■

However, this is just like our naive algorithm from Corollary 6.1.2, which takes $O(mnW) = O(mn)$ as $W = 1$ in this case. The upshot is that our analysis can be improved for *simple graph*:

Corollary 6.2.1 ([ET75]). Theorem 6.2.2 can be improved to $O(m \cdot \min(\sqrt{m}, n^{2/3}))$ for simple graph.

Proof. We prove the result in two cases. We first prove the $O(m^{3/2})$ bound.

Claim. We can improve the bound to be $O(m^{3/2})$.

Proof. Suppose we already did k blocking flows. Consider the max-flow in the residual graph: if we decompose it into paths, then the number of these paths will be exactly the value of the remaining flow. Since we have done k blocking flows, the s-t distance in the residual graph is greater than k, hence, each of these flow paths has length greater than k.

Since these flow paths are edge-disjoint, with their total volume being m, the number of these flow paths is less than m/k. Each blocking flow increases the value of the flow by at least one, as there always is at least one admissible path. Hence, m/k additional blocking flows[a] suffices. Combing the above, the total running time is then $O(m(k+m/k))$ from Theorem 6.2.2, which is minimized when $k = \sqrt{m}$, giving $O(m^{3/2})$ as desired. ⊛

[a] Or even just augmenting paths can do the job.

For the second case, the idea is essentially the same.

Claim. We can improve the bound to be $O(mn^{2/3})$.

Proof. Suppose we already did k blocking flows. It's clear that there are at most $k/2$ different layers in the BFS admissible graph with vertices greater than $2n/k$ from a counting argument. Then by the pigeonhole principle, there are two consecutive layers such that both have less than $2n/k$ vertices. There can be at most $O(n^2/k^2)$ edges between these two layers. Observe that these edges form an s-t cut, hence the s-t max-flow value is $O(n^2/k^2)$.

From a similar argument, we only need $O(n^2/k^2)$ additional blocking flows, hence the total running time is $O(m(k + n^2/k^2))$ from Theorem 6.2.2, which is minimized when $k = n^{2/3}$, giving $O(mn^{2/3})$ as desired. ⊛

By combining the above two cases, we're done. ■

Remark. The $O(m\sqrt{m})$ bound extends to multi-graphs, while $O(mn^{2/3})$ bound does not.

Proof. For $O(mn^{2/3})$, we assume for two consecutive layers with each less than $2n/k$ vertices, there are only $O(n^2/k^2)$ edges between them. This is not the case for multi-graphs. ⊛

6.2.3 General Capacity Graph

For graph with general capacities, as we promised at the beginning of this section, we can achieve $O(mn^2)$ running time [Kar73; Din70; GR98]. To prove this, we look into what breaks for the adaptive DFS search for unit capacity case when we try the same analysis as in Theorem 6.2.2 for general graphs.

From the previous adaptive DFS search algorithm, the basic idea of advance, retreat, and block are still valid. The key problem is that when augmenting, i.e., blocking:

Problem. We might saturate only one bottlenecking edge on the s-t path we found!

Answer. Simply backtrack to the tail of the farthest saturated edge and repeat. ⊛

Specifically, consider the following modified version of the adaptive DFS search, starting at s:

1. In each step, *advance* along an unexplored outgoing admissible edge.
2. If reach t, *block* the s-t path by adding it to the blocking flow (with value to be the bottleneck) and discarding edges with 0 residual capacity on this path. Finally, backtrack to the head of the farthest saturated edge on this path.
3. If reach a vertex $v \neq t$ with no outgoing admissible edges, *retreat* back along the edge we came from to v and discard it.
4. Once we backtrack to s and there is no admissible edges leaving s, return the blocking flow.

Theorem 6.2.3. By choosing the blocking flow in Algorithm 6.1 using adaptive DFS search, the algorithm computes the s-t max-flow on a directed graph with general capacities in $O(mn^2)$ time.

Proof. In the new adaptive DFS search, every advance is still paid for by costs of corresponding retreat or block. In this case, all retreats still cost $O(m)$ in total, and each block now costs $O(n)$ as only one bottlenecking edge can be discarded per block. Therefore, the blocking flow computation now takes $O(mn)$ time, proving the result. ■

Intuition. Using blocking flows enables us to charge m into n because of the resulting more efficient search for augmenting paths.

In fact, we can make the above intuition stronger via an even more careful analysis.

Corollary 6.2.2. By choosing the blocking flow in Algorithm 6.1 using adaptive DFS search, the algorithm computes the s-t max-flow on a directed graph with general capacities in $O(mn + |f^*|n)$ time, where $|f^*|$ is the s-t max-flow value.

Proof. Since each block corresponds to finding a new augmenting path, hence, each block increases the value of the flow by at least 1. This means that the total cost of block is only $O(n|f^*|)$. Thus, the running time is actually $O(mn + n|f^*|) = O((m + |f^*|)n)$. ■

In other words, the blocking flow approach takes $O(n)$ time "per unit of flow routed," while the basic/naive augmenting path approach took $O(m)$ time (this is true even if capacities are not unit) as shown in Theorem 6.1.4.

6.2.4 Scaling Technique

If we're dealing with integral capacities, we can utilize the idea of **scaling**, just like what we have seen in Algorithm 3.3 for the SSSP. The general idea is still the same: apply the "unit case" algorithm to obtain a "general numerical case" algorithm. This time, consider the bit representation of the capacities. We observe that bits can correspond to unit case by looking at only the i^{th} bit at a time for every $c(e)$'s! To exploit this idea, consider the following:

Intuition. Starting from an optimal solution to the rounded down instance (i.e., dropping some least significant bits for the capacities). One can repeatedly put back the next dropped bit, i.e., shifting/adding one bit to the right, and fix the solution along the way.

Such a shifting corresponds to a new capacity that is either doubled or doubled and plus 1. Hence, intuitively, the problem boils down to applying an algorithm that works well only for unit capacity case as every iteration we only need to deal with one unit of deviation in some sense.

Note. One can think of scaling as a reverse process of rounding.

For convenience, we use the following notation to refer to i^{th} bit.

Notation. Given $a \in \mathbb{N}$, let a_i be the i^{th} most significant bit in a's bit representation.

Formally, the idea of scaling leads to a weakly polynomial algorithm.

Theorem 6.2.4. The scaling-based Ford-Fulkerson algorithm that computes the s-t max-flow on a directed graph $G = (V, E)$ with integral capacities in $O(mn \log W)$ time, where $W = \max_{e \in E} c(e)$.

Proof. We first describe the algorithm. In the first iteration, look at the most significant bit $c^{(1)}(e) = c_1(e) \in \{0, 1\}$, which corresponds to a unit capacity graph. We then obtain the s-t max-flow f on the graph with capacity $c^{(1)}$ via the unit capacity algorithm in Theorem 6.2.2, which takes $O(mn)$ time. In the next iteration, we consider the new capacity $c^{(2)}(e) \in \{0, 1, \ldots, 2^2\}$ to be the integer represented by the first two most significant bits of $c(e)$, i.e., $c^{(2)}(e) = 2c^{(1)}(e) + c_2(e)$ for some $c_2(e) \in \{0, 1\}$. We also double f as $f'(e) = 2f(e)$, which is clearly feasible for $c^{(2)}$.

Now, consider the corresponding residual graph $G_{f'}$. The crucial observation is the following.

Claim. There is a min-cut in $G_{f'}$ with residual capacity at most m.

Proof. Since our flow was optimal before, there must exist an s-t cut with its residual capacity (w.r.t. f) be 0. Hence, after shifting a new bit and doubling the flow to be f', this cut has residual capacity being at most m. ⊛

Hence, we see that after the bit shifting and doubling the current flow, finding the new optimum boils down to solving the s-t max-flow in a graph with $|f^*| \leq m$. From Corollary 6.2.2, we know that this takes $O(mn)$ time.[a] As there are only $O(\log W)$ many shiftings required, the total running time of this algorithm is $O(mn \log W)$. ■

[a] Observe that after the shifting, $G_{f'}$ does not need to be unit capacity: we only have control on $|f^*|$.

Remark. The same idea with Theorem 6.1.2 improves Theorem 6.1.3 to $O(m^2 \log W)$.

6.2.5 Data Structure for Blocking Flow

So far, our best strongly polynomial time algorithm runs in $O(mn^2)$ (Theorem 6.2.3), and weakly polynomial time algorithm runs in $O(mn \log W)$ (Theorem 6.2.4). A natural question is that whether we need to pay this extra factor of n for being strongly polynomial, or maybe there is a better way to implement blocking flows in general graphs?

As previously seen. In the adaptive DFS search, we repeatedly find s-t paths while discarding parts of the graph that we have finished searching. Suppose we find an s-t path P. Then, we decrease the capacities along P as to saturate the bottleneck edge e along P, removing e from the search. Then, we start the searching over from s.

The key bottleneck is that blocking in the adaptive DFS search takes $\Theta(n)$ time but might only discard a single edge from the whole admissible path we found. Then in the next iteration, we start the searching all over again from s.

Intuition. This is a waste of the two partial paths of $p - e$, one from s and one to t, respectively.

Hence, the question is the following.

Problem. As the pieces of this path that were not discarded are still useful, can we somehow take advantage of this fact and maintain these pieces for efficient access later?

Answer. This is a data structure question, and fortunately, there is the so-called link/cut tree [ST85] that supports all operations we need for implementing the adaptive DFS search! ⊛

Before we formally introduce the link/cut tree, we first need to define what an arborescence is.

Definition 6.2.3 (Arborescence)**.** An *arborescence* is a rooted directed tree with edges always directed towards the root.

The Link/cut tree maintains a collection of arborescences with several supported operations:

Definition 6.2.4 (Link/cut tree)**.** The *link/cut tree* data structure maintains a disjoint collection of arborescences, which is modified via the following operations:

- `maketree`(v): make a new arborescence with only a root v.
- `link`(v, w): v is the root of an arborescence, and w is a vertex in a different arborescence from v, add a directed edge (v, w), i.e., making v a child of w.
- `cut`(v): detach v from its arborescence by deleting the edge toward v's root, and make the sub-arborescence rooted at v a new arborescence in the collection.

For each vertex v, the following queries are supported:

- `root`(v): return the root of v's arborescence.
- `min-capacity`(v): return the minimum capacity edge in the path from v to its root.
- `add-capacity`(v, c): add c to the capacity of every edge in the path from v to its root.

Note. In general, the link/cut trees allows for values along edges and vertices and efficient aggregate queries and updates along node-to-root paths in the arborescence. In Definition 6.2.4, we have defined two such operations (`min-capacity` and `decrease-capacity`) specifically for computing blocking flows in residual graphs.

It's know that link/cut trees can be implemented efficiently [Shi78; GN80; Sle81; ST81; ST83].

Theorem 6.2.5 ([ST83]). Link/cut tree can be maintained in amortized $O(\log n)$ for each operation.

Proof intuition. To get a high level idea of the implementation, let us just consider the representation of an arborescence when it is just a path.

> **Intuition.** A path can be represented as a dynamic binary search tree (BST) of height $O(\log n)$. Then all operations of a link/cut tree are trivial. For a general arborescence, the heavy light decomposition [ST83] can essentially break it into a heavy path and some small things.

Specifically, consider maintaining an ordered list of vertices on path in a splay tree (but we could use any balanced BST). The key trick is that instead of storing the current residual capacity of each edge, store only the "deltas" Δ, i.e., their differences. This way, the true value of the residual capacity of an edge is obtained by summing all the Δ's on the path from that edge node to the root of the tree. Note that this representation is easy to maintain under rotations.

To add a capacity of c on a path from some vertex v to the root of the path, splay successor of v to root, add c from the root of the left subtree. Similarly, one can maintain at each node the minimum capacity of its subtree (to help with min-capacity(v)). ■

We now formally describe how to use link/cut tree to speed up the adaptive DFS search for a general capacity graph, as described just above Theorem 6.2.3.

Theorem 6.2.6. By choosing the blocking flow with link/cut tree in Algorithm 6.1, the algorithm computes the s-t max-flow on a directed graph with general capacities in $O(mn \log n)$ time.

Proof. The high-level idea is to maintain a forest of arborescences of "block-able" edges, i.e., admissible edges that are non-saturated. This means, we will discard parts of the graph that we have finished searching. Specifically, these arborescences will always be subtrees of the residual graph such that s is always a leaf, and t is always a root (of some, potentially different, arborescences). Moreover, the path from s to s's root (in s's arborescence) represents a partial shortest path from s in the residual graph. The goal is to find a new s-t path to block by keep extending this path (at s's root).

- Initially, all vertices are isolated by calling `maketree`(v) for all $v \in V$.
- In each iteration, start from the current vertex, i.e., s's root $r =$ `root`(s).
- Unless $r = t$, scan the unexamined outgoing admissible edges from r in the residual graph:
  - If there is no edge at all, then there is no s-t path via r. We retreat from r:
    - Delete r via calling `cut`(e) for all e incident to r. Discard r after this (which is doable since now r is just a singleton arborescence).
    - Or if $r = s$, we finish the search.
  - Otherwise, we find a directed admissible edge (r, u) to advance:
    - Merge s's arborescence with u's arborescence via `link`(r, u). This makes s's arborescence a sub-arborescence of u, and s's new root becomes `root`(u).
    - Repeat.
- If $r = t$, then the s-r path P in s's arborescence is the desired s-t path. We block P:
  - Find the bottleneck capacity b on P via $b = c(e)$ for some $e =$ `min-capacity`(s).
  - Decrease all edge capacities on P by b via calling `add-capacity`$(s, -b)$.
  - Cut all edges with capacity 0 now by calling `cut`(e) for all e with $c(e) = b$. These edges can be identified via calling `min-capacity`(s) repeatedly and checking their capacity.
  - Repeat.

We continue the search until $r = s$ and there are no edges leaving s left to explore. We claim that constructing a blocking flow takes only constant number of link/cut tree operations.

Claim. Computing a blocking flow takes a constant number of link/cut tree operations.

Proof. Observe that the search does a constant number of operations per edge in the residual graph and per augmenting path obtained by the search. For each edge e, we `link`(e) as it is searched, and `cut`(e) as we either backtrack from e or saturate e.[a] For each augmenting path, we execute a constant number of operations (querying `min-capacity`(s) or `add-capacity`$(s, -b)$) to update the residual graph. Each augmenting path saturates an edge that is then deleted from the residual graph, so there are at most m augmenting path. ⊛

[a] When e is `cut`(e) by saturation, it is also preceded by a `min-capacity`(s) query that identifies it.

Hence, one can construct a blocking flow in only $O(m \log n)$ time, instead of $O(mn)$ from Theorem 6.2.3. We conclude that the whole algorithm now takes only $O(mn \log n)$ time. ■

Corollary 6.2.3. There is an algorithm that computes the s-t max-flow on a directed graph with general capacities in $O(mn \log n^2/m)$ time.

Proof. With more care, the $\log n$ factor in Theorem 6.2.6 can be improved to $\log n^2/m$ [GT90]. ■

Finally, for completeness, we summarized the algorithm described in Theorem 6.2.6 as follows.

Algorithm 6.2: Blocking Flow with Link/Cut Tree

Data: A connected directed graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, source s, sink t
Result: A blocking flow f

```
1  c' ← c                                              // Record the original capacity
2  E' ←Admissible(G, c, s, t)                          // Set of admissible edges via DFS from s to t
3  T ←Link/Cut Tree()                                  // Initialize an empty link/cut tree
4
5  for v ∈ V do
6      T.maketree(v)
7
8  while 1 = 1 do
9      r ← T.root()
10     if r = t then                                   // Find an s-t path to block
11         e ← T.min-capacity(s)
12         b ← c(e)
13         T.add-capacity(s, −b)                       // Assuming T and G shares c
14         for e ← T.min-capacity(s) do
15             if c(e) = 0 then                        // Cut and delete saturated edges
16                 T.cut(e)
17                 E' \ {e}
18     else if ∃e = (r, v) ∈ E' then                   // Find an admissible edge to advance
19         T.link(r, v)
20     else                                            // Retreat from r
21         if r = s then                               // No more s-t path
22             f ← c − c'
23             return f
24         for e = (v, r) ∈ E' do
25             T.cut(e)
26         T ← T \ {r}
```

Remark. In line 22, as described in Theorem 6.2.6, we're not actually augmenting the s-t path we found, we only block (i.e., add) them to the blocking flow without introducing the residual edge, hence this step is necessary. One can also directly augment during the algorithm run.

Lecture 23: Push-Relabel Algorithm

6.3 Push-Relabel

21 Nov. 11:00

While the augmenting path approach developed by Ford and Fulkerson is powerful, but there exists a very different and (in a sense) dual approach called the *push-relabel* algorithm. This approach was first introduced by Goldberg [Gol85], and further developed by Goldberg and Tarjan [GT88]. As mentioned, this approach departs from the Ford-Fulkerson framework:

Note. It does not augment along paths and may not obtain a feasible flow until the very end.

The two main components of the algorithm are a preflow and a set of vertex labels, which are relaxations of flows and distance layers of the DFS search, respectively. These relaxations (defined below) are still similar enough to flows and distance labels from DFS layer graph to allow us to define residual graphs and forwards/backwards/neutral (i.e., of same distance) edges, and similar to blocking flows, one can explore many, and often very different, interesting ideas within this framework.

6.3.1 Preflow

An s-t preflow is just a relaxation of an s-t flow where non-terminals are also allowed to have "excess":

Definition 6.3.1 (Preflow). A *preflow* is a vector $f\colon E \to \mathbb{R}_+$ such that

(a) $0 \le f(e) \le c(e)$ for all $e \in E$;

(b) $\sum_{e\in\delta^-(v)} f(e) \ge \sum_{e\in\delta^+(v)} f(e)$ for all $v \in V \setminus \{s,t\}$.

So instead of flow conservation, we only have one-sided conservation for a preflow.

Definition 6.3.2 (Excess). The *excess* $\hat{f}(v)$ at $v \neq s,t$ of a preflow f is defined as

$$\hat{f}(v) \coloneqq \sum_{e\in\delta^-(v)} f(e) - \sum_{e\in\delta^+(v)} f(e).$$

Example. If $\hat{f}(v) = 0$ for all $v \neq s,t$, then f is a flow.

The residual graph G_f w.r.t. a preflow f is the same as before, i.e., decreasing capacities by the amount of preflow on the edge; increasing capacities by the amount of preflow on the opposite edge. In particular, we still have $c_f(e) = c(e) - f(e) + f(e^{-1})$. We note the following useful and obvious lemma.

Lemma 6.3.1. Let f be a preflow. Then f decomposes to a path packing that contains $\hat{f}(v)$ unit of fraction s-v paths for every $v \in V \setminus \{s\}$; in particular, f contains an s-t flow of size $\hat{f}(t)$.

Proof. Create a new auxiliary vertex t' and connect every vertex v (including t) with $\hat{f}(v) > 0$ to t' with an edge with infinite capacity. Consider the s-t' flow f' where we send all $\hat{f}(v)$ to t' via those new edges. This flow decomposes to a fractional s-t' path packing that includes $\hat{f}(v)$ paths that go through the auxiliary (v,t') edge. Removing this last edge gives the desired path decomposition. ■

6.3.2 Label

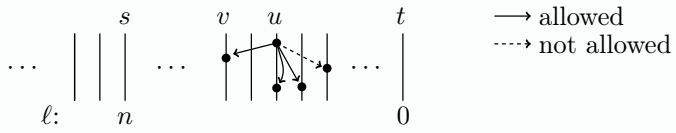
The next ingredient of the *push-relabel* framework is the so-called vertex labels:

Definition 6.3.3 (Label). Given a preflow f, a source s, and a sink t, a *label* $\ell\colon V \to \mathbb{Z}_+$ satisfies

(a) $\ell(s) = n$ and $\ell(t) = 0$;

(b) if $(u,v) \in E_f$, then $\ell(v) \ge \ell(u) - 1$.

Intuition. Definition 6.3.3 (b) means that residual edges go down towards t by at most one level.



The *push-relabel* framework maintains a preflow and a label such that the initial label ℓ is defined as

$$\ell(v) = \begin{cases} n, & \text{if } v = s; \\ 0, & \text{otherwise,} \end{cases} \tag{6.1}$$

and the initial preflow f is defined as

$$f(e) = \begin{cases} c(e), & \text{if } e \in \delta^+(s); \\ 0, & \text{otherwise,} \end{cases} \tag{6.2}$$

i.e., empty flow and saturate all edges leaving s. It's easy to verify the following.

Claim. With the initial flow f, Definition 6.3.3 (b) is satisfied.

Proof. Since the definition of f will remove all edges $(s, v) \in E$ leaving s from the residual graph, and add the reverse edges (v, s) to E_f. ⊛

The invariant (b) can be interpreted as a relaxation of distances in the following sense.

Lemma 6.3.2. For all $u, v \in V$, $\ell(u) - \ell(v) \leq d_{G_f}(u, v)$.[a]

[a] Again, $d_{G_f}(\cdot, \cdot)$ is the unweighted distance in G_f.

Proof. This is because we restrict that any forward edge can only have label more than 1. ■

From Lemma 6.3.2 and the fact that $\ell(t) = 0$, $\ell(v)$ is a lower bound on the distance from v to the sink t. In particular, we have $d_{G_f}(s, t) \geq \ell(s) = n$, i.e., there is no s-t path in the residual graph, so s can't reach t in G_f. In this sense, the *push-relabel* algorithm is dual to augmenting path-based algorithms:

Intuition. The preflow f always induces an s-t cut (via its residual graph). But since f is not an actual flow, we cannot certify this cut as an s-t min-cut.

That is, we always try to maintain the invariant that some cut is saturated, i.e., a preflow with value equal to some cut, hence more than the max-flow. This means, if f is actually a flow (with no non-terminals with positive excess), then it is immediately an s-t max-flow. Additionally, the label ℓ will induce an s-t min-cut. These are proved in Lemma 6.3.3.

Lemma 6.3.3. Suppose f is a preflow and ℓ is a valid label. Suppose f is also a flow, i.e., $\hat{f}(v) = 0$ for all $v \neq s, t$. Then, f is an s-t max-flow, and there exists an index $i \in [n-1]$ such that $U = \{v \in V \mid \ell(v) > i\}$ induces an s-t min-cut.

Proof. Since ℓ is valid w.r.t. the preflow f, we know that there is no s-t path in G_f, as discussed above. This holds true even when f is actually a flow, meaning that we cannot find any augmenting path, hence f will be an s-t max-flow.

Secondly, from the pigeonhole principle, there exists some $i \in [n-1]$ such that no vertices are labeled as i, meaning that there are no edges between $V \setminus U$ and U from the invariant (b). By the flow-conservation, the cut value $c(\delta^+(U))$ is equal to the flow value, we're done. ■

Remark. Lemma 6.3.3 is just like the classical proof of the max-flow min-cut theorem.

While $\ell \colon V \to \mathbb{Z}_+$ is not actually a distance function, invariants (a) and (b) installs just enough structure to serve the same role.

Notation. A residual edge $(u, v) \in E_f$ is *forward* (previously admissible) if $\ell(v) = \ell(u) - 1$, *neutral* if $\ell(u) = \ell(v)$, *backward* if $\ell(u) < \ell(v)$. Moreover, we say that a vertex u *has a forward edge* if there is a forward edge starting from u.

6.3.3 Generic Push-Relabel Algorithm and Analysis

There are many different *push-relabel* algorithms, but they generally follow the same framework consisting of two types of operations, *push* and *relabel*. Within this framework that are different strategies for employing these operations, some of which we discuss in greater detail below.

As previously seen. The only difference between a flow and a preflow is that a preflow allows intermediate vertices to carry positive excess.

In fact, by Lemma 6.3.3, this is the only difference between an s-t preflow and an s-t max-flow. Denote the set of "active" vertices as $A := \{v \neq s, t \mid \hat{f}(v) > 0\}$. Loosely speaking, we try to route the excess from active vertices, one vertex at a time, towards the sink t or the source s.

Intuition. The latter implies that there was too much flow out for all of it to be routed to t.

Given an active vertex v, the algorithm executes one of the following two **local** operations at v:

(a) We *push* along a forward edge e w.r.t. the label of $v \in A$. Usually,[2] we push as much flow as possible (i.e., *full push*), subject to the preflow constraints ((a) and (b)). There are two cases:

  (i) *Saturating push*: when $\hat{f}(v) > c_f(e)$, then we push $c_f(e)$, saturating the residual edge.

  (ii) *Non-saturating push*: when $\hat{f}(v) \leq c(e)$, then we push $\hat{f}(v)$, using up all the excess.

(b) If $v \in A$ and has no forward edges, then *relabel* v's label to the next level, i.e., $\ell(v) \leftarrow \ell(v) + 1$.

The push-relabel algorithm simply repeats the above two operations until there are no more active vertices, i.e., $A = \varnothing$. At that point, by Lemma 6.3.3, we know that f is an s-t max-flow.

Algorithm 6.3: Push-Relabel Algorithm for s-t Max-Flow

Data: A connected directed graph $G = (V, E)$ with edge capacity $c\colon E \to \mathbb{R}_+$, source s, sink t
Result: An s-t max-flow f

```
1  ℓ ← Initialize-Label(V, s)                    // Initialize label (Equation 6.1)
2  f ← Initialize-Preflow(E, c, s)               // Initialize preflow (Equation 6.2)
3  for e ∈ E do                                  // Initialize G_f
4      c_f(e) ← c(e) − f(e) + f(e^{-1})
5
6  while A = {v ≠ s,t | f̂(v) > 0} ≠ ∅ do         // Assuming A is updated implicitly
7      v ← A[0]                                  // Get an arbitrary active vertex
8      E' ← Forward-Edge(v, E, ℓ)                // Compute forward edges from v
9      if E' = ∅ then                            // Relabel
10         ℓ(v) ← ℓ(v) + 1
11     else                                      // Push
12         e = (v, u) ← E'[0]                    // Get an arbitrary forward edge from v
13         Δ ← min(f̂(v), c_f(e) − f(e))          // Compute a full push
14         f(e) ← f(e) + Δ
15         c_f(e) ← c(e) − f(e) + f(e^{-1})
16 return f
```

Remark. We assume that whenever we update f in Algorithm 6.3 (i.e., line 14), $\hat{f}$ is also automatically updated. In this was, line 6 will also be maintained.

[2] Some scaling algorithms do not necessarily push the maximum amount.

However, it is not at all obvious that Algorithm 6.3 should terminate. At a high level, we want to extinguish all the active vertices, but pushing flow out of one active vertex v, and as to deactivate v, may activate many more vertices one level below! Hence, it is not clear that we're making much progress.

Theorem 6.3.1. The push-relabel algorithm computes the s-t max-flow on a directed graph with general capacities in $O(mn^2)$ time.

Proof. We first analyze the push-relabel algorithm by bounding each of the local operations, i.e., saturating pushes, non-saturating pushes, and relabelings, separately. Firstly, consider relabelings.

Claim. For each v, $\ell(v)$ is non-decreasing, non-negative, and is at most $2n-1$. Hence, the total cost of relabelings is at most $O(n^2)$.

Proof. Indeed, since if $v \in A$, then there must be a v-s path in G_f. But since $\ell(s) = n$, so every active vertex has level $\ell(v) \le \ell(s) + n - 1 = 2n - 1$ from Lemma 6.3.2. Finally, since only active vertices will get relabeled, the total cost is $O(n^2)$. ⊛

Next, we consider saturating pushes.

Claim. There are at most $n-1$ saturating pushes to any fixed edge. Hence, the total cost of saturating pushes is $O(mn)$.

Proof. For any residual edge $(u,v) \in E_f$, when doing a saturating push along (u,v), we have $\ell(u) = \ell(v) + 1$. To do it along (u,v) again, (u,v) need to reappear, which requires pushing along (v,u) first, i.e., $\ell(v) = \ell(u) + 1$. Hence, v needs to be relabeled from below $\ell(u)$ to above $\ell(u)$, with the fact that ℓ is monotonically increasing, $\ell(v)$ needs to go up by at least 2. But $\ell(v)$ can only go up by at most $2n-1$ from the previous claim, so at most $n-1$ saturating pushes can be done on (u,v). There are $O(m)$ edges in the residual graph, so we're done. ⊛

Finally, we consider the trickiest case, the non-saturating pushes.

Claim. There are at most $O(mn^2)$ non-saturating pushes.

Proof. Consider a potential $\Phi = \sum_{v \in A} \ell(v)$. We observe that each

- non-saturating push decreases Φ by 1;
- saturating push increases Φ by at most $2n-2$;[a]
- relabeling increases Φ by 1.

Since there are at most $O(mn)$ saturating pushes and $O(n^2)$ relabelings, Φ can go up to at most $O(mn^2)$. Since Φ is initially zero, and always non-negative, by charging each non-saturating push to decrease in Φ, there can be at most $O(mn^2)$ non-saturating pushes. ⊛

[a] This is via activating some v with $\ell(v) \le 2n-2$ that is not in A initially.

Putting everything together, with Lemma 6.3.3, we know that after $O(mn^2)$ push/relabel operations, Algorithm 6.3 terminates (i.e., $A = \varnothing$) with an s-t max-flow. Besides these local operations, one needs to account for the running time overhead for selecting a tight vertex, and identifying a forward edge from a given vertex, and so forth. However, it is easy to see how to use some simple data structures to facilitate the operations (e.g., tracking the set of active vertices A in a list), it is easy to implement Algorithm 6.3 in $O(mn^2)$ time. ■

We see that just via simple local operations, push-relabel algorithms achieve the same running time as the blocking flow algorithm with adaptive DFS search (Theorem 6.2.3).

Remark. The bottleneck is the non-saturating pushes. The relabelings and saturating pushes only costs $O(mn)$ in total are very fast.

We now explore different variations of the push-relabel algorithm that obtain better running times;

each of these strategies are designed to limit the number of non-saturating pushes.

6.3.4 Top-Down Push-Relabel Algorithm

Algorithm 6.3 gives a very general description and Theorem 6.3.1 gives a general analysis of the push-relabel algorithm. In particular:

> **As previously seen.** We show that *any* algorithm following either push or relabel operations makes $O(mn^2)$ total operations. The bottleneck is the non-saturating pushes, which in requires $O(mn^2)$.

There are many possible strategies within this framework, and perhaps different approaches can lead to different performance in practice, or better upper bounds (especially for non-saturating pushes). The main decision is which active vertex to select at a given moment in time. Previously in Algorithm 6.3 line 7, we simply select an arbitrary one. Here are a few possible strategies:

(a) Select $v \in A$ with the highest label $\ell(v)$.

(b) Select $v \in A$ with the lowest label $\ell(v)$.

(c) Select $v \in A$ in FIFO order (e.g., add vertices to a queue as they become active).

(d) Select $v \in A$ in LIFO order (e.g., add vertices to a stack as they become active).

(e) Select $v \in A$ with the greatest excess $\hat{f}(v)$.

(f) Select $v \in A$ with the lowest excess $\hat{f}(v)$.

(g) Select $v \in A$ uniformly at random.

We consider the first strategy, i.e., always addressing the active vertex of the highest level. We refer to this strategy as *top-down*.

Lemma 6.3.4. With the top-down push-relabel algorithm, there are at most $O(n^2)$ non-saturating pushes from any fixed vertex v. Therefore, there are at most $O(n^3)$ non-saturating pushes.

Proof. Suppose we make a non-saturating push from a vertex v, making v inactive. Since v was the highest level active vertex, there is no way for v to receive flow and become active without some vertex being increased to a higher label. Each vertex's label gets increased at most $O(n)$ times as we have seen in Theorem 6.3.1, hence in total only $O(n^2)$ many times v can be overtaken and potentially become active again. ■

Lemma 6.3.4 implies the following.

Theorem 6.3.2. The top-down push-relabel algorithm computes the s-t max-flow on a directed graph with general capacities in $O(n^3)$ time.

We see that Theorem 6.3.2 already beats the best strongly polynomial algorithm without using sophisticated data structures we have before (Theorem 6.2.3).

> **Note.** The FIFO strategy also yields an $O(n^3)$ running time.

It turns out that Lemma 6.3.4 is not tight. For top-down push-relabel, one can obtain a better upper bound of $O(n^2\sqrt{m})$ for non-saturating pushes [CM89; Tun94].

Theorem 6.3.3 ([CM89; Tun94])**.** The top-down push-relabel algorithm computes the s-t max-flow on a directed graph with general capacities in $O(n^2\sqrt{m})$ time.

Proof [CM99]. We focus on showing that the top-down push-relabel makes at most $O(n^2\sqrt{m})$ non-saturating pushes. Consider a potential $\Phi := \sum_{v \in A} |\{u \colon \ell(u) \leq \ell(v)\}|$, which counts, for every active vertex v, the total number of vertices "below" or at the same level as v.

Initially, $\Phi = 0$. A relabeling or a saturating push can increase Φ by at most n. Hence, the total increase to Φ is at most $O(mn^2)$. On the other hand, we have the following.

Claim. A non-saturating push at v deactivates v and decreases Φ, by at least the number of u at v's level, $|\{u\colon \ell(u) = \ell(v)\}|$.

Proof. In the worst case, the non-saturating push from v activates a new vertex u. Since $\ell(u) = \ell(v)-1$, the difference in contribution between v and u, $|\{w\colon \ell(w) = \ell(v)\}|-|\{w\colon \ell(w) = \ell(u) = \ell(v) - 1\}|$, is precisely the number of vertices at v's level. ⊛

We now focus only on active vertices at the highest level. Let a "phase" be a consecutive sequence of basic operations made by the algorithm on active vertices from the same level. Observe the following:

Note. Phases are terminated by:

- increasing the highest active level: by relabeling an active vertex
- decreasing the highest active level: by deactivating the last active vertex with a push.

Since there are $O(n^2)$ relabelings, so we only increase the highest level $O(n^2)$ times. Each decrease can be attributed to a previous increase, like an elevator. So there are at most $O(n^2)$ phases.

Let a $k \in \mathbb{N}$ be a parameter to be determined. Call a non-saturating push "small" if the highest active level has $\leq k$ active vertices, and "big" otherwise. There are at most k small non-saturating pushes per phase, hence $O(n^2k)$ small non-saturating pushes overall. Meanwhile, each big non-saturating push decreases Φ by at least $O(k)$, so there are at most $O(mn^2/k)$ big pushes. Balancing terms at $k = \sqrt{m}$, we conclude that there are at most $O(n^2\sqrt{m})$ non-saturating pushes. ■

It is possible to further accelerate the push-relabel (with FIFO strategy[3]). The key idea is to use data structures drastically cut the computational work generated by non-saturating pushes. As you might have guessed, it is implemented with a link/cut tree! Here, we only state the result.

Theorem 6.3.4 ([GT88]). With link/cut tree, the FIFO push-relabel algorithm computes the s-t max-flow on a directed graph with general capacities in $O(mn \log(n^2/m))$ time.

Note. In fact, the proof of Theorem 6.3.4 does not actually decrease the total number of non-saturating pushes; rather, most of the non-saturating pushes are simulated efficiently by the data structures, similar to blocking flows.

As a final note, it turns out when dealing with integral capacities, scaling can also accelerate the push-relabel algorithm. We again state the result only.

Theorem 6.3.5 ([AO89]). The scaling-based push-relabel algorithm that computes the s-t max-flow on a directed graph $G = (V, E)$ with integral capacities in $O(mn + n^2 \log W)$ time, where $W = \max_{e\in E} c(e)$.

[3] It appears that most strategies would work.

Appendix

Bibliography

[ABN08] Ittai Abraham, Yair Bartal, and Ofer Neiman. "Nearly tight low stretch spanning trees". In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE. 2008, pp. 781–790.

[ACP87] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. "Complexity of finding embeddings in ak-tree". In: *SIAM Journal on Algebraic Discrete Methods* 8.2 (1987), pp. 277–284.

[AHK12] Sanjeev Arora, Elad Hazan, and Satyen Kale. "The multiplicative weights update method: a meta-algorithm and applications". In: *Theory of computing* 8.1 (2012), pp. 121–164.

[AK07] Sanjeev Arora and Satyen Kale. "A combinatorial, primal-dual approach to semidefinite programs". In: *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. 2007, pp. 227–236.

[AKT21] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. "Subcubic algorithms for Gomory–Hu tree in unweighted graphs". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 2021, pp. 1725–1737.

[Ale+17] Vedat Levi Alev et al. *Graph Clustering using Effective Resistance*. 2017.

[ALN05] Sanjeev Arora, James R Lee, and Assaf Naor. "Euclidean distortion and the sparsest cut". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 553–562.

[ALN07] Sanjeev Arora, James R Lee, and Assaf Naor. "Fréchet embeddings of negative type metrics". In: *Discrete & Computational Geometry* 38.4 (2007), pp. 726–739.

[Alo+95] Noga Alon et al. "A graph-theoretic game and its application to the k-server problem". In: *SIAM Journal on Computing* 24.1 (1995), pp. 78–100.

[AO15] Zeyuan Allen-Zhu and Lorenzo Orecchia. "Nearly-linear time positive LP solver with faster convergence rate". In: *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*. 2015, pp. 229–236.

[AO89] Ravindra K Ahuja and James B Orlin. "A fast and simple algorithm for the maximum flow problem". In: *Operations Research* 37.5 (1989), pp. 748–759.

[AR98] Yonatan Aumann and Yuval Rabani. "An $O(logk)$ approximate min-cut max-flow theorem and approximation algorithm". In: *SIAM Journal on Computing* 27.1 (1998), pp. 291–301.

[ARV09] Sanjeev Arora, Satish Rao, and Umesh Vazirani. "Expander flows, geometric embeddings and graph partitioning". In: *Journal of the ACM (JACM)* 56.2 (2009), pp. 1–37.

[Aza+03] Yossi Azar et al. "Optimal oblivious routing in polynomial time". In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 383–388.

[Bar96] Yair Bartal. "Probabilistic approximation of metric spaces and its algorithmic applications". In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 184–193.

[Bar98] Yair Bartal. "On approximating arbitrary metrices by tree metrics". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 161–168.

[BCF23] Karl Bringmann, Alejandro Cassis, and Nick Fischer. "Negative-Weight Single-Source Shortest Paths in Near-Linear Time: Now Faster!" In: *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2023, pp. 515–538.

[BG17] Nikhil Bansal and Anupam Gupta. "Potential-function proofs for first-order methods". In: *arXiv preprint arXiv:1712.04581* (2017).

[BGM14] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. "Efficient primal-dual graph algorithms for mapreduce". In: *International Workshop on Algorithms and Models for the Web-Graph*. Springer. 2014, pp. 59–78.

[BGS20] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. "Deterministic decremental reachability, scc, and shortest paths via directed expanders and congestion balancing". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 1123–1134.

[BKR03] Marcin Bienkowski, Miroslaw Korzeniowski, and Harald Räcke. "A practical algorithm for constructing oblivious routing schemes". In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. 2003, pp. 24–33.

[BNW22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. "Negative-weight single-source shortest paths in near-linear time". In: *2022 IEEE 63rd annual symposium on foundations of computer science (FOCS)*. IEEE. 2022, pp. 600–611.

[Bod+16] Hans L Bodlaender et al. "A $c^k n$ 5-approximation algorithm for treewidth". In: *SIAM Journal on Computing* 45.2 (2016), pp. 317–378.

[Bod93] Hans L Bodlaender. "A linear time algorithm for finding tree-decompositions of small treewidth". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 226–234.

[Bou85] Jean Bourgain. "On Lipschitz embedding of finite metric spaces in Hilbert space". In: *Israel Journal of Mathematics* 52 (1985), pp. 46–52.

[CC14] Chandra Chekuri and Julia Chuzhoy. "Degree-3 treewidth sparsifiers". In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2014, pp. 242–255.

[CE13] Chandra Chekuri and Alina Ene. "Poly-logarithmic approximation for maximum node disjoint paths with constant congestion". In: *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2013, pp. 326–341.

[CE15] Chandra Chekuri and Alina Ene. "The all-or-nothing flow problem in directed graphs with symmetric demand pairs". In: *Mathematical Programming* 154 (2015), pp. 249–272.

[Cha00] Bernard Chazelle. "A minimum spanning tree algorithm with inverse-Ackermann type complexity". In: *Journal of the ACM (JACM)* 47.6 (2000), pp. 1028–1047.

[Che+22] Li Chen et al. "Maximum flow and minimum-cost flow in almost-linear time". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2022, pp. 612–623.

[Chu+20] Julia Chuzhoy et al. "A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 1158–1167.

[CJV15] Chandra Chekuri, TS Jayram, and Jan Vondrák. "On multiplicative weight updates for concave and submodular function maximization". In: *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*. 2015, pp. 201–210.

[CKR05] Gruia Calinescu, Howard Karloff, and Yuval Rabani. "Approximation algorithms for the 0-extension problem". In: *SIAM Journal on Computing* 34.2 (2005), pp. 358–372.

[CKS05] Chandra Chekuri, Sanjeev Khanna, and F Bruce Shepherd. "Multicommodity flow, well-linked terminals, and routing problems". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 183–192.

[CM89] Joseph Cheriyan and SN Maheshwari. "Analysis of preflow push algorithms for maximum network flow". In: *SIAM Journal on Computing* 18.6 (1989), pp. 1057–1086.

[CM99] Joseph Cheriyan and Kurt Mehlhorn. "An analysis of the highest-level selection rule in the preflow-push max-flow algorithm". In: *Information Processing Letters* 69.5 (1999), pp. 239–242.

[CQ17] Chandra Chekuri and Kent Quanrud. "Near-linear time approximation schemes for some implicit fractional packing problems". In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 801–820.

[CQ19] Chandra Chekuri and Kent Quanrud. "On approximating (sparse) covering integer programs". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2019, pp. 1596–1615.

[CQ21] Chandra Chekuri and Kent Quanrud. "Isolating Cuts, (Bi-)Submodularity, and Faster Algorithms for Global Connectivity Problems". In: *CoRR* abs/2103.12908 (2021). arXiv: 2103.12908. URL: https://arxiv.org/abs/2103.12908.

[CQX20] Chandra Chekuri, Kent Quanrud, and Chao Xu. "LP relaxation and tree packing for minimum k-cut". In: *SIAM Journal on Discrete Mathematics* 34.2 (2020), pp. 1334–1353.

[Din70] Efim A Dinic. "Algorithm for solution of a problem of maximum flow in networks with power estimation". In: *Soviet Math. Doklady*. Vol. 11. 1970, pp. 1277–1280.

[DKL76] Efim A Dinitz, Alexander V Karzanov, and Michael V Lomonosov. "On the structure of the system of minimum edge cuts of a graph". In: *Issledovaniya po Diskretnoi Optimizatsii* (1976), pp. 290–306.

[DRT92] Brandon Dixon, Monika Rauch, and Robert E Tarjan. "Verification and sensitivity analysis of minimum spanning trees in linear time". In: *SIAM Journal on Computing* 21.6 (1992), pp. 1184–1192.

[Eis97] Jason Eisner. "State-of-the-art algorithms for minimum spanning trees". In: *Unpublished survey* (1997).

[EK72] Jack Edmonds and Richard M Karp. "Theoretical improvements in algorithmic efficiency for network flow problems". In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264.

[Elk+05] Michael Elkin et al. "Lower-stretch spanning trees". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 494–503.

[Ene+16] Alina Ene et al. "Routing under balance". In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 2016, pp. 598–611.

[ET75] Shimon Even and R Endre Tarjan. "Network flow and testing graph connectivity". In: *SIAM journal on computing* 4.4 (1975), pp. 507–518.

[Eve+00] Guy Even et al. "Divide-and-conquer approximation algorithms via spreading metrics". In: *Journal of the ACM (JACM)* 47.4 (2000), pp. 585–616.

[FF56] Lester Randolph Ford and Delbert R Fulkerson. "Maximal flow through a network". In: *Canadian journal of Mathematics* 8 (1956), pp. 399–404.

[FHL05] Uriel Feige, MohammadTaghi Hajiaghayi, and James R Lee. "Improved approximation algorithms for minimum-weight vertex separators". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 563–572.

[Fil24] Arnold Filtser. "On sparse covers of minor free graphs, low dimensional metric embeddings, and other applications". In: *arXiv preprint arXiv:2401.14060* (2024).

[Fin24] Jeremy T Fineman. "Single-Source Shortest Paths with Negative Real Weights in $\widetilde{O}(mn^{8/9})$ Time". In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. 2024, pp. 3–14.

[FRT03] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. "A tight bound on approximating arbitrary metrics by tree metrics". In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 448–455.

[FT87] Michael L Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[Gab+86] Harold N Gabow et al. "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs". In: *Combinatorica* 6.2 (1986), pp. 109–122.

[GK07] Naveen Garg and Jochen Könemann. "Faster and simpler algorithms for multicommodity flow and other fractional packing problems". In: *SIAM Journal on Computing* 37.2 (2007), pp. 630–652.

[GK94] Michael D Grigoriadis and Leonid G Khachiyan. "Fast approximation schemes for convex programs with many blocks and coupling constraints". In: *SIAM Journal on Optimization* 4.1 (1994), pp. 86–107.

[GN80] Zvi Galil and Amnon Naamad. "An O (EVlog2V) algorithm for the maximal flow problem". In: *Journal of Computer and System Sciences* 21.2 (1980), pp. 203–217.

[Gol85] Andrew V Goldberg. "A new max-flow algorithm". In: (1985).

[Gol95] Andrew V Goldberg. "Scaling algorithms for the shortest paths problem". In: *SIAM Journal on Computing* 24.3 (1995), pp. 494–504.

[Gor+21] Gramoz Goranci et al. "The expander hierarchy and its applications to dynamic graph algorithms". In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 2212–2228.

[GR98] Andrew V Goldberg and Satish Rao. "Beyond the flow decomposition barrier". In: *Journal of the ACM (JACM)* 45.5 (1998), pp. 783–797.

[GT88] Andrew V Goldberg and Robert E Tarjan. "A new approach to the maximum-flow problem". In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940.

[GT90] Andrew V Goldberg and Robert E Tarjan. "Finding minimum-cost circulations by successive approximation". In: *Mathematics of Operations Research* 15.3 (1990), pp. 430–466.

[Gup+04] Anupam Gupta et al. "Cuts, trees and ℓ_1-embeddings of graphs". In: *Combinatorica* 24.2 (2004), pp. 233–269.

[GVY93] Naveen Garg, Vijay V Vazirani, and Mihalis Yannakakis. "Approximate max-flow min-(multi) cut theorems and their applications". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 698–707.

[Haj+07] Mohammad Taghi Hajiaghayi et al. "Oblivious routing on node-capacitated and directed graphs". In: *ACM Transactions on Algorithms (TALG)* 3.4 (2007), 51–es.

[HDT01] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 723–760.

[HHR03] Chris Harrelson, Kirsten Hildrum, and Satish Rao. "A polynomial-time tree decomposition to minimize congestion". In: *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*. 2003, pp. 34–43.

[HJQ24] Yufan Huang, Peter Jin, and Kent Quanrud. *Faster single-source shortest paths with negative real weights via proper hop distance*. 2024. arXiv: 2407.04872 [cs.DS]. URL: https://arxiv.org/abs/2407.04872.

[HLW06] Shlomo Hoory, Nathan Linial, and Avi Wigderson. "Expander graphs and their applications". In: *Bulletin of the American Mathematical Society* 43.4 (2006), pp. 439–561.

[HO94] JX Hao and James B Orlin. "A faster algorithm for finding the minimum cut in a directed graph". In: *Journal of Algorithms* 17.3 (1994), pp. 424–446.

[Kar00] David R Karger. "Minimum cuts in near-linear time". In: *Journal of the ACM (JACM)* 47.1 (2000), pp. 46–76.

[Kar73] Alexander V Karzanov. "On finding maximum flows in networks with special structure and some applications". In: *Matematicheskie Voprosy Upravleniya Proizvodstvom* 5.81-94 (1973), p. 1.

[Kar95] David Ron Karger. *Random sampling in graph optimization problems*. stanford university, 1995.

[Kar98] David R Karger. "Random sampling and greedy sparsification for matroid optimization problems". In: *Mathematical Programming* 82.1 (1998), pp. 41–81.

[Kin97] Valerie King. "A simpler minimum spanning tree verification algorithm". In: *Algorithmica* 18 (1997), pp. 263–270.

[KKT95] David R Karger, Philip N Klein, and Robert E Tarjan. "A randomized linear-time algorithm to find minimum spanning trees". In: *Journal of the ACM (JACM)* 42.2 (1995), pp. 321–328.

[Kle+97] Philip N Klein et al. "Approximation algorithms for Steiner and directed multicuts". In: *Journal of Algorithms* 22.2 (1997), pp. 241–269.

[Kom85] János Komlós. "Linear verification for spanning trees". In: *Combinatorica* 5.1 (1985), pp. 57–65.

[KPR93] Philip Klein, Serge A Plotkin, and Satish Rao. "Excluded minors, network decomposition, and multicommodity flow". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 682–690.

[KRV09] Rohit Khandekar, Satish Rao, and Umesh Vazirani. "Graph partitioning using single commodity flows". In: *Journal of the ACM (JACM)* 56.4 (2009), pp. 1–15.

[KS96] David R Karger and Clifford Stein. "A new approach to the minimum cut problem". In: *Journal of the ACM (JACM)* 43.4 (1996), pp. 601–640.

[KY14] Christos Koufogiannakis and Neal E Young. "A nearly linear-time PTAS for explicit fractional packing and covering linear programs". In: *Algorithmica* 70 (2014), pp. 648–674.

[LLR95] Nathan Linial, Eran London, and Yuri Rabinovich. "The geometry of graphs and some of its algorithmic applications". In: *Combinatorica* 15 (1995), pp. 215–245.

[LN93] Michael Luby and Noam Nisan. "A parallel approximation algorithm for positive linear programming". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. 1993, pp. 448–457.

[LP20] Jason Li and Debmalya Panigrahi. "Deterministic min-cut in poly-logarithmic max-flows". In: *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2020, pp. 85–92.

[LR99] Tom Leighton and Satish Rao. "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms". In: *Journal of the ACM (JACM)* 46.6 (1999), pp. 787–832.

[LT79] Richard J Lipton and Robert Endre Tarjan. "A separator theorem for planar graphs". In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189.

[Mar08] Martin Mareš. "The saga of minimum spanning trees". In: *Computer Science Review* 2.3 (2008), pp. 165–221.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. "Computing edge-connectivity in multigraphs and capacitated graphs". In: *SIAM Journal on Discrete Mathematics* 5.1 (1992), pp. 54–66.

[Ore+08] Lorenzo Orecchia et al. "On partitioning graphs via single commodity flows". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 461–470.

[Pen16] Richard Peng. "Approximate undirected maximum flows in $o(m\mathsf{polylog}(n))$ time". In: *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2016, pp. 1862–1867.

[PR02] Seth Pettie and Vijaya Ramachandran. "An optimal minimum spanning tree algorithm". In: *Journal of the ACM (JACM)* 49.1 (2002), pp. 16–34.

[PST95] Serge A Plotkin, David B Shmoys, and Éva Tardos. "Fast approximation algorithms for fractional packing and covering problems". In: *Mathematics of Operations Research* 20.2 (1995), pp. 257–301.

[Qua19] Kent Quanrud. "Fast approximations for combinatorial optimization via multiplicative weight updates". PhD thesis. University of Illinois at Urbana-Champaign, 2019.

[Räc02] Harald Räcke. "Minimizing congestion in general networks". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings*. IEEE. 2002, pp. 43–52.

[Räc08] Harald Räcke. "Optimal hierarchical decompositions for congestion minimization in networks". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 255–264.

[Ree97] Bruce Reed. "Tree width and tangles: a new connectivity measure and some applications". In: *Surveys in combinatorics* 241 (1997), pp. 87–162.

[RS14] Harald Räcke and Chintan Shah. "Improved guarantees for tree cut sparsifiers". In: *Algorithms-ESA 2014: 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings 21*. Springer. 2014, pp. 774–785.

[Sch+03] Alexander Schrijver et al. *Combinatorial optimization: polyhedra and efficiency*. Vol. 24. 2. Springer, 2003.

[Sey95] Paul D. Seymour. "Packing directed circuits fractionally". In: *Combinatorica* 15.2 (1995), pp. 281–288.

[Shi78] Yossi Shiloach. *An $O(nI \log^2 I)$ maximum-flow algorithm*. Stanford University, 1978.

[SL21] Thatchaphol Saranurak and Jason Li. "Deterministic Weighted Expander Decomposition in Almost-linear Time". In: *CoRR* abs/2106.01567 (2021).

[Sle81] Daniel Dominic Kaplan Sleator. *An O (nm log n) algorithm for maximum network flow*. Stanford University, 1981.

[SM90] Farhad Shahrokhi and David W Matula. "The maximum concurrent flow problem". In: *Journal of the ACM (JACM)* 37.2 (1990), pp. 318–334.

[ST81] Daniel D Sleator and Robert Endre Tarjan. "A data structure for dynamic trees". In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 1981, pp. 114–122.

[ST83] Daniel D Sleator and Robert Endre Tarjan. "A data structure for dynamic trees". In: *J. Comput. Syst. Sci* 26.3 (1983), pp. 362–391.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-adjusting binary search trees". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

[SW19] Thatchaphol Saranurak and Di Wang. "Expander decomposition and pruning: Faster, stronger, and simpler". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2019, pp. 2616–2635.

[Tun94] Levent Tunçel. "On the complexity of preflow-push algorithms for maximum-flow problems". In: *Algorithmica* 11.4 (1994), pp. 353–359.

[Vaz01] Vijay V Vazirani. *Approximation Algorithms*. 2001.

[VB81] Leslie G Valiant and Gordon J Brebner. "Universal schemes for parallel communication". In: *Proceedings of the thirteenth annual ACM symposium on Theory of computing*. 1981, pp. 263–277.

[Wan17] Di Wang. *Fast approximation algorithms for positive linear programs*. University of California, Berkeley, 2017.

[WRM15] Di Wang, Satish Rao, and Michael W Mahoney. "Unified acceleration method for packing and covering problems via diameter reduction". In: *arXiv preprint arXiv:1508.02439* (2015).

[WS11] David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[Yao75] Andrew Chi-Chih Yao. "An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees". In: *Information Processing Letters* 4 (1975), pp. 21–23.

[You14] Neal E Young. "Nearly linear-work algorithms for mixed packing/covering and facility-location linear programs". In: *arXiv preprint arXiv:1407.3015* (2014).