Massively Parallel Algorithms

Computer Science, ETH Zurich

Mohsen Ghaffari

Lecture Notes by
Davin Choo

Version: 19th June, 2019

Contents

Notation and useful inequalities

Commonly used notation

- WLOG: without loss of generality
- ind.: independent / independently
- w.p.: with probability
- w.h.p: with high probability
  We say event X holds *with high probability* (w.h.p.) if

$$\Pr[X] \geqslant 1 - \frac{1}{\mathrm{poly}(n)}$$

  say, $\Pr[X] \geqslant 1 - \frac{1}{n^c}$ for some constant $c \geqslant 2$.
- Integer range $[n] = \{1, \ldots, n\}$

Useful inequalities

- For any x, $(1 - x) \leqslant e^{-x}$
- For $x \in (0, \frac{1}{2})$, $(1 - x) \geqslant e^{-x-x^2}$
- For $x \in [0, \frac{1}{2}]$, $4^{-x} \leqslant 1 - x \leqslant e^{-x}$
- $(\frac{n}{k})^k \leqslant \binom{n}{k} \leqslant (\frac{en}{k})^k$
- $\binom{n}{k} \leqslant n^k$
- $\lim_{n\to\infty}(1 - \frac{1}{n})^n = e^{-1}$
- $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi}{6}$

- $\frac{1}{1-x} \leqslant 1 + 2x$ for $x \leqslant \frac{1}{2}$

Theorem (Chernoff bound). *For independent Bernoulli variables* $X_1, \ldots, X_n$, *let* $X = \sum_{i=1}^{n} X_i$. *Then,*

$$\Pr[X \geqslant (1+\epsilon) \cdot \mathbb{E}(X)] \leqslant \exp(\frac{-\epsilon^2 \mathbb{E}(X)}{3}) \qquad \textit{for } 0 < \epsilon$$
$$\Pr[X \leqslant (1-\epsilon) \cdot \mathbb{E}(X)] \leqslant \exp(\frac{-\epsilon^2 \mathbb{E}(X)}{2}) \qquad \textit{for } 0 < \epsilon < 1$$

By union bound, for $0 < \epsilon < 1$, *we have*

$$\Pr[|X - \mathbb{E}(X)| \geqslant \epsilon \cdot \mathbb{E}(X)] \leqslant 2\exp(\frac{-\epsilon^2 \mathbb{E}(X)}{3})$$

Remark 1 There is actually a tighter form of Chernoff bounds:

$$\forall \epsilon > 0, \Pr[X \geqslant (1+\epsilon)\mathbb{E}(X)] \leqslant (\frac{e^{\epsilon}}{(1+\epsilon)^{1+\epsilon}})^{\mathbb{E}(X)}$$

Remark 2 We usually apply Chernoff bound to show that the probability of bad approximation is low by picking parameters such that $2\exp(\frac{-\epsilon^2 \mathbb{E}(X)}{3}) \leqslant \delta$, then negate to get $\Pr[|X - \mathbb{E}(X)| \leqslant \epsilon \cdot \mathbb{E}(X)] \geqslant 1 - \delta$.

Administrative matters

Purpose

Due to physical limitations, Moore's law has been breaking down over the past few years. The growth of memory and speed of *individual* computers are being outpaced by the surge in amount of data that needs to be processed. This calls for a new paradigm for computation.

Over the years, frameworks such as MapReduce [DG08], Hadoop [Whi12], Dryad [IBY+07], and Spark [ZCF+10] have emerged as a way to perform large scale computations across various machines. In this course, we discuss the expanding body of work on the *theoretical foundations of modern parallel computation*, and particularly the design of algorithms that can be parallelized. The emphasis will be on the *algorithmic tools and techniques with provable guarantees*, and not whether they can be implemented with current technologies. The theoretical framework of interest is called Massively Parallel Computation (MPC), first introduced in [KSV10] and later refined in [ANOY14, BKS13, GSZ11].

Prerequisites

No prior knowledge in parallel algorithms/computing is assumed. The only prerequisite is that one should be comfortable with randomized algorithms. Having taken a course in Algorithms, Probability, and Computing[1] or Randomized Algorithms and Probabilistic Methods[2] would suffice. To check your understanding, please attempt Problem Set 0. If you're unsure whether you're ready for this class, please consult the instructor.

[1] https://www.ti.inf.ethz.ch/ew/courses/APC18/index.html
[2] https://www.cadmo.ethz.ch/education/lectures/HS18/RandAlg/index.html

Assessment format

The course takes a more research slant. There will be no examinations but instead assessment will be a semester-long project with teams of 2.

	Deadline	Weightage
Proposal	~ end March	20%
Presentation	~ end May	20%
Report	~ end May	60%

Content

As the field is relatively new, there are no existing courses and most materials will be from research papers over the past decade. The tentative list of topics is as follows: sorting, maximum matching approximations, maximal independent set, connected components, minimum spanning tree, minimum cut, graph coloring, geometric problems, dynamic programming, clustering, triangle counting, densest subgraph, and coreness decomposition, composable coresets, impossibility results and conditional lower bounds. Depending on the feedback and questions, there may be a class on model discussion — PRAM, NC, BSP, and so on.

Chapter 1

The MPC Model

In this chapter, we introduce the Massively Parallel Computation (MPC) model, discuss how data is initially distributed, and establish some commonly used subroutines in MPC algorithms.

1.1 Computation and Communication Model

Under the setting that the size of data outstrips the memory and communication availability, it is natural to employ multiple machines for computation. As such, a MPC model is often defined by three parameters — the input data size N, the number of machines M available for computation, and the memory size per machine of S words[1].

In practice, S is *not* chosen by us but dictated by the problem instances and available computational resources / state-of-the-art hardware. Problems are easier to solve with larger S. In the extreme when $S \geqslant N$, one can just put the entire input on a single machine and solve it locally. Typically, S is polynomially smaller than N, e.g. $S = N^c$ for some constant $c < 1$.

Computation is performed in *synchronous rounds*. In each round, every machine locally performs some computation on the data that resides locally, then send/receive messages to *any* other machine. As an abstraction of the communication model, one may think of each machine creating message packages to load onto a routing network in each round. Hence, the memory size S also implicitly captures the communication bottleneck in the MPC model. Local computations frequently run in linear or near-linear time, and are ignored in the analysis of MPC algorithms because communication is the bottleneck.

[1] A word is represented with $\mathcal{O}(\log N)$ bits.

For graph problems with number of vertices $n = |V|$ and number of edges $m = |E|$, the input size N is the number of edges m. As the edges are distributed across the machines initially, we require the number of machines M to be in the order of $\widetilde{O}(\frac{N}{S})$ so that there are enough machines to store the whole input[2]. Graph algorithms in MPC can be typically classified based on three different regimes of memory sizes, each facing very different technical difficulties faced in algorithm design:

Strongly superlinear memory Memory $S = n^{1+\epsilon}$, for some constant $\epsilon > 0$.

Near linear memory Memory $S \in \widetilde{\Theta}(n)$.

Strongly sublinear memory Memory $S = n^{\alpha}$, for some constant $\alpha \in (0, 1)$.

Unsurprisingly, problems are usually easiest to solve in setting with strongly superlinear memory.

Remark on communication Communication is the largest bottleneck in the MPC model as compared to other parallel computation models such as PRAMs. Notice that the model described above assumes a pairwise communication network that ignores asynchronous communication and fault-tolerance. While these are practical and important concerns, we ignore[3] them to yield a cleaner algorithmic framework.

1.2 Initial data distribution

The whole input data is split across the M machines *arbitrarily*. For example, each edge of a graph are stored arbitrarily on some machine. Using universal hashing [CW79, WC81], one can "load balance" the initial data across all machines in $O(1)$ rounds.

A concern[4] was raised in class about duplicates in initial data and how to handle them. Consider the following: Suppose each of the N unique data items has a $\Theta(\log n)$-bit (1 word) identifier in the range $\{1, \ldots, P\}$, for some constant $P = n^{\Theta(1)}$. For a fixed S, suppose we have $M = \Theta(\frac{N}{S})$ machines and every machine knows a $O(\log n)$-wise independent hash $h : \{1, \ldots, P\} \to \{1, \ldots, M\}$.

[2] $\widetilde{O}$ hides logarithmic factors. For example, $O(n \log^{1000} n) \subseteq \widetilde{O}(n)$.
[3] There exists black-boxes to handle these issues in practice.
[4] Credit: Yuyi Wang

1. For each item identifier x on machine i, machine i sends x to machine $h(x)$. Observe that each machine sends out at most S identifiers. Since h is $\mathcal{O}(\log n)$-wise independent, each machine receives $\mathcal{O}(\frac{N}{M})$ identifiers with high probability, fitting the memory constraint of a single machine.

2. For each item identifier x, reply "Keep x" to *exactly one* machine that sent x, and "Discard x" to every other machine that sent x.

At the end of these 2 rounds, there is only one copy of every identifier in the whole system. To generate the $\mathcal{O}(\log n)$-wise independent hash, $\Theta(\log^2 n)$ bits of randomness needs to be fixed and given to each machine [WC81]. Then, [SSS95, Theorem 5] tells us that Chernoff bounds apply even in limited independence — in our case $k = \mathcal{O}(\log n)$.

1.3 Commonly used subroutines

In this section, we will introduce some commonly used subroutines in the MPC literature. In subsequent chapters, these will be used implicitly without much emphasis so that focus can be placed on the ideas discussed.

1.3.1 Independent sampling and basic facts about it

Suppose there are n items and we sample each item independently with probability $p \in o(n)$. Let X_i be the indicator for the i^{th} item to be sampled. By construction, $\mathbb{E}[X_i] = \Pr[X_i = 1] = p$, for all $i \in \{1, \ldots, n\}$. Then, $X = \sum_{i=1}^{n} X_i$ is the number of items that are sampled. By linearity of expectation,

$$\mathbb{E}[X] = \mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} p = np$$

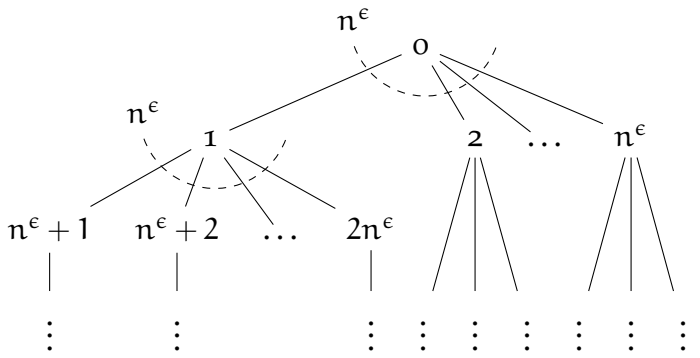
Since each item is sampled independently, the X_i's are independent. By Chernoff bounds,

$$\Pr\left[|X - \mathbb{E}[X]| \geqslant \frac{1}{2}\mathbb{E}[X]\right] \leqslant 2\exp(-\frac{np}{12})$$

This implies that with high probability, the number of sampled items X is *not* more than a factor of 2 away from its expectation $\mathbb{E}[X]$.

1.3.2 Broadcast / Converge-cast trees

Recall that S restricts amount of communication between machines in a single round. Consider the scenario where $S = n^{1+\epsilon}$ and a single machine (say machine 0) needs to convey n words to all M machines. When $n^\epsilon < M$, sending n words to all machines in a single round will exceed the communication constraint of S. Under the assumption that $S = n \cdot n^\epsilon$, one can build a communication tree with branching factor n^ϵ among all machines:



Observe that the height of the tree is $O(\frac{1}{\epsilon})$, which is a constant for a constant ϵ. Using this tree, the root can *broadcast* a message of n words to all other machines in constant rounds. Furthermore, all machines can send the root (*converge-cast*) a union/intersection of n words using the same tree. For instance, all machines can know the number of edges in the graph in constant rounds.

Actual machine identifiers involved in the tree construction can be worked out explicitly with known M, S, and n. In general, one builds such a tree by setting the branching factor to be S divided by the size of the message sent.

1.3.3 Computation output

Computational output are stored, possibly in a distributed fashion, amongst the set of M machines. Here are some examples of problem output:

Sorting The machine holding item x knows the rank/position of x.

Matching The machine holding vertex v knows whether v is an endpoint of a matched edge (and if so, what vertex is the other endpoint), or v is not involved in the matching.

Connectivity The machine holding vertex v knows the identifier of v's connected component.

In certain cases when memory S is sufficiently large, the entire output may be stored on a single machine. For example, in the near-linear memory regime, any matching can fit into a machine as there are at most $\frac{n}{2}$ matched edges. In any case, one may assume that the output can be queried in constant time after computation.

Chapter 2

Matching

Given a graph $G = (V, E)$, a matching $M \subseteq E$ is a subset of edges such that edges in M do not share an endpoint. An edge in M is called a *matched edge*, and endpoints of any matched edge are *matched vertices*. Without loss of generality, one may assume that there is no isolated vertex in G.

The problem of *maximum* matching is to find the matching with the largest cardinality. A related concept is *maximal* matching — a matching M is said to be maximal if every edge in $E \setminus M$ has an endpoint in M. It is known that any maximal matching is a 2-approximation of the maximum matching. That is, the size of any maximal matching is at least half the size of the maximum matching.

In this chapter, we first look at an MPC algorithm for computing a maximal matching. Then, we discuss how one can transform an $O(1)$-approximation of maximum matching to $(1+\epsilon)$-approximation, for any arbitrarily small constant $\epsilon > 0$.

2.1 Matching using strongly superlinear memory

We now describe a Las Vegas randomized algorithm[1] due to [LMSV11] that solves maximal matching in constant rounds with $S = n^{1+\epsilon}$, for any constant $\epsilon > 0$. The technique is also called *filtering*.

[1] A Las Vegas algorithm is *always correct* and the randomness is over the runtime.

2.1.1 Overview

At each round, we put a subset of edges $E' \subseteq E$ into a single machine. After finding a maximal matching, matched vertices are removed from the graph (by dropping edges involving matched vertices across all M machines). This iterative process continues until the number of remaining edges can all fit into a single machine.

The subset E' is randomly selected such that $|E'| \leqslant S$ with high probability. By arguing that the number of edges decrease roughly by a factor of n^{ϵ} in each round, one can show that there will only be $\mathcal{O}(\frac{1}{\epsilon})$ rounds, which is constant for a constant $\epsilon > 0$. Correctness of the approach follows from the fact that we only consider edges whose endpoints are not in the current matching, and terminate when there are no more edges.

2.1.2 Algorithm

Suppose there is a machine 0 that is free, and all edges are distributed amongst machines labelled 1 to M. Let $G_r = (V, E_r)$ be the graph at round r, for $r \in \{0, \ldots, R\}$, where G_0 is the input graph G and G_R is the empty graph at the end of the algorithm. We say an edge is *local*, with respect to a machine, if the edge resides on the machine. At round $r < R$, let m denote the number of edges present in the *current* graph $|E_r|$. Consider a round of the algorithm:

1. For $i \in \{1, \ldots, M\}$, machine i marks each local edge independently with probability $p = \frac{n^{1+\epsilon}}{2m}$.
2. For $i \in \{1, \ldots, M\}$, machine i sends the marked edges to machine 0.
3. Machine 0 computes a maximal matching M_r on the marked edges, and announces marked vertices to machines 1 to M.
4. For $i \in \{1, \ldots, M\}$, machine i discards any local edge that has marked vertices as endpoints.

We will show that there will be at most $S = n^{1+\epsilon}$ edges left after R rounds with high probability. In one final round, these remaining edges will be put on a single machine to compute a maximal matching.

2.1.3 Analysis

Theorem 2.1. *There exists an algorithm that computes a maximal matching in the strongly superlinear memory regime in constant rounds.*

The theorem follows from the lemmata below. For analysis, fix a round of the algorithm and suppose there are m edges at the start of the round.

Lemma 2.2. *With high probability, the number of marked edges fit into a single machine.*

Proof. Since each edge is sampled *independently* with probability $p = \frac{n^{1+\epsilon}}{2m}$, the expected number of marked edges is $mp = \frac{n^{1+\epsilon}}{2}$. By Chernoff bounds, there are at most $S = n^{1+\epsilon}$ marked edges with high probability. □

Lemma 2.3. *With high probability, the number of remaining edges is at most* $\frac{10m}{n^\epsilon}$.

Proof. Denote I as the set of unmarked vertices at end of the round. Observe that there is no marked edge between any pair of vertices in I — If there is a marked edge $\{u, v\}$, then $\{u, v\}$ will be sent to machine o, thus at least one of u or v will be a matched vertex and not appear in I.

Consider an arbitrary set of vertex J with $\geqslant \frac{10m}{n^\epsilon}$ induced edges, where an induced edge is an edge with both endpoints in J. Then, the probability that there are no marked induced edges in J is

$$\Pr[\text{All edges unmarked}] \leqslant (1-p)^{\frac{10m}{n^\epsilon}} \leqslant \exp(-p \cdot \frac{10m}{n^\epsilon}) = e^{-5n}$$

Union bounding over all 2^n potential sets of vertices[2], the probability that *any* set with $\geqslant \frac{10m}{n^\epsilon}$ induced edges has no marked edges is $\leqslant 2^n \cdot e^{-5n}$, which is exponentially small. Together with the observation above, there are at most $\frac{10m}{n^\epsilon}$ remaining edges with high probability. □

Lemma 2.4. *The algorithm terminates in* $R = \mathcal{O}(\frac{1}{\epsilon})$ *rounds.*

Proof. There are at most n^2 edges initially. After $R \leqslant \log_{n^\epsilon}(n^2) \in \mathcal{O}(\frac{1}{\epsilon})$ applications of Lemma 2.3, there are at most $S = n^{1+\epsilon}$ edges left. □

2.1.4 Small details

We now address some details that one may be concerned about, many of which involve a broadcast tree[3]:

[2] 2^n is a gross over-estimation but it suffices.

[3] See Section 1.3.2 for a description of broadcast trees.

Computing sampling probability Recall the sampling probability $p = \frac{n^{1+\epsilon}}{2m}$. Since ϵ is a known constant to all machines, it suffices for each machine to learn the current number of edges $m = |E_r|$. This can be done in constant rounds using a broadcast tree.

What if more than S edges are marked? Using a broadcast tree, we can count the total number of marked edges. If there are more than S marked edges, resample. As proven in Lemma 2.2, this happens with low probability and is precisely the part of the algorithm which affects the Las Vegas runtime.

Announcing marked vertices This can be done in constant rounds using a broadcast tree.

Storing the maximal matching output See Section 1.3.3 for a discussion on MPC output.

Exercises

Exercise 2.1. Sorting with strongly sublinear memory in constant time *Consider the problem of sorting N items. The desired output is to know the ranking of each item at the end of the computation. For example, given elements $\{1,3,4,7\}$, the rank of 3 is 2.*

(a) Devise an algorithm that sorts in constant time with $M \in \widetilde{\mathcal{O}}(N^{0.4})$ machines, each with memory $S = N^{0.6}$.

(b) Devise an algorithm that sorts in constant time with $M \in \widetilde{\mathcal{O}}(\frac{N}{S})$ machines, each with memory $S = N^{\alpha}$, for a given constant $\alpha \in (0,1)$.

Hint *Emulate multi-pivot quicksort by selecting pivots independently with appropriate probability. Using the broadcast tree, figure out how to partition the problem and recurse.*

2.2 Matching using near linear memory

In this section, we look at an algorithm due to Ghaffari et al. [GGK+18] that yields a constant approximation to maximum matching using $\widetilde{\mathcal{O}}(n)$

memory in $\mathcal{O}(\log\log n)$ rounds. Using round compression, a technique first introduced in Czumaj et al. [CŁM+18], Ghaffari et al. [GGK+18] emulates dual ascent (also known as water filling) in the MPC setting then perform probabilistic rounding on the resultant fractional matching.

We first review fractional matching and describe a *vertex-centric* dual ascent algorithm for obtaining an $\mathcal{O}(1)$-approximation to maximum matching. Then, we describe the round compression technique and how to compute in near linear time with $\mathcal{O}(\log\log n)$ rounds.

Remark When we reference Ghaffari et al. [GGK+18] in this section, we mean the arVix version[4] of the paper.

2.2.1 Fractional matching and randomized rounding

Relaxing an ILP to yield fractional matching

We write $e \ni v$ if vertex v is an endpoint of edge e. Maximum matching can be formulated as an integer linear program (ILP) as follows:

$$\begin{aligned} \max \sum_{e \in E} y_e & \\ \text{s.t.} \sum_{e \ni v} y_e \leqslant 1 & \qquad \forall v \in V \\ y_e \in \{0,1\} & \qquad \forall e \in E \end{aligned}$$

Each binary variable y_e indicates whether edge e is in the matching. One can *relax* the above ILP into a corresponding linear program (LP) by replacing each binary variable $y_e \in \{0,1\}$ by a real-valued variable $x_e \in [0,1]$. Solving the corresponding LP yields a *fractional* assignment x^*.

Randomized rounding to yield constant approximation

To obtain a matching from the fractional x^*_e assignments, one can independently set y_e to 1 with probability x^*_e. By linearity of expectation, $\mathbb{E}(\sum_{e\in E} y_e) = \sum_{e\in E} x^*_e$, where $\sum_{e\in E} x^*_e$ is the optimal objective value of the LP[5]. We say that edge e is *conflicting* if $y_e = 1$ and e has an adjacent edge e' with $y_{e'} = 1$. To ensure that $\sum_{e\ni v} y_e \leqslant 1$ for all vertices v, we set $y_e = 0$ for any conflicting edge e.

[4] Available at: https://arxiv.org/abs/1802.08237

[5] That is, x^* is an optimal *fractional* maximum matching.

Observe that there is a problem of the size of the rounded matching dropping as we ignore vertices with conflicting incident edges. To remedy this, let us first divide each x_e^* by 10 before rounding. While this reduces the expected number of edges being rounded to 1, it is okay since we only want an $\mathcal{O}(1)$-approximation. By rounding $\frac{x_e^*}{10}$, a vertex will not have any incident edge with $y_e = 1$ with probability $\geqslant \frac{9}{10}$. Hence, we do not expect many conflicting edges and there is a constant probability that we obtain a rounded matching whose size is a constant factor of the maximum matching. Running $\mathcal{O}(\log n)$ independent runs in parallel and taking the largest rounded matching, we will get maximal matching whose size is a constant factor of the maximum matching with high probability. See Section 5 of [GGK+18] for a sharper, dependent rounding scheme.

2.2.2 Dual Ascent

Let $\epsilon > 0$ be a small constant and $R \in \mathcal{O}(\log \Delta)$, where Δ is the maximum degree of graph G. Consider a vertex-centric algorithm `Vertex-Centric` that runs for R iterations:

1. Initialize all x_e to $\frac{1}{\Delta}$
2. For iteration $t \in \{1, \ldots, R\}$
   (a) Freeze each vertex v, and all incident edges, if $\sum_{e \ni v} x_e \geqslant 1 - 2\epsilon$
   (b) For each unfrozen edge e, set x_e to $x_e \cdot (1 + \epsilon)$

We say $(1 - 2\epsilon)$ is the *freezing threshold* of `Vertex-Centric`. Observe that freezing is *permanent*: When an edge e is frozen, it stays frozen, and the value of x_e remains unchanged for the rest of the algorithm.

Why is it called "dual ascent"? Maximum matching is the dual of the minimum vertex cover problem and `Vertex-Centric` "ascends" the value of each x_e.

Claim 2.5. *The variables x_e are always a valid fractional matching. That is, $\sum_{e \ni v} x_e \leqslant 1$ for every vertex $v \in V$.*

Proof. For unfrozen vertices v, $\sum_{e \ni v} x_e \cdot (1 + \epsilon) \leqslant (1 - 2\epsilon)(1 + \epsilon) \leqslant 1$. □

Claim 2.6. All *edges are frozen at the end of $R \in \mathcal{O}(\log \Delta)$ rounds.*

Proof. After $\log_{1+\epsilon} \Delta$ rounds, then $x_e = \frac{1}{\Delta} \cdot (1 + \epsilon)^{\log_{1+\epsilon} \Delta} \geqslant 1 - 2\epsilon$. □

Claim 2.7. *For* $\epsilon \leqslant \frac{1}{10}$,

$$\sum_{e \in E} x_e \geqslant \frac{1}{2+10\epsilon}|M^*|$$

That is, the fractional matching of x_e *is a constant approximation of the maximum matching size* $|M^*|$, *where* M^* *is a* maximum *matching.*

Proof. We perform a *charging argument* at the end of the algorithm. Fix a maximum matching M^*. Let us assign 1\$ to each edge $e \in M^*$. By Claim 2.6, we know that every edge is frozen. Since edges are frozen only if at least one of the endpoints are frozen, we can pick a frozen endpoint v for each edge $e \in M^*$ and give v the 1\$ to redistribute. In the case where both endpoints are frozen, we pick one arbitrarily.

Since M^* is a matching, every vertex only receives at most 1\$ to distribute. Each vertex v with 1\$ splits the 1\$ to incident edges proportional to their x values: Each edge e' incident to v receives $\frac{x_{e'}}{\sum_{e' \ni v} x_{e'}} \leqslant \frac{1}{1-2\epsilon} x_{e'}$ from v. The inequality holds because v being frozen implies that $\sum_{e' \ni v} x_{e'} \geqslant 1-2\epsilon$. Therefore, each x_e receives at most $\frac{2}{1-2\epsilon} x_e$ from both endpoints.

Summing up over all edges, with $\epsilon \leqslant \frac{1}{10}$, we have

$$1 \cdot |M^*| \leqslant \frac{2}{1-2\epsilon} \sum_{e \in E} x_e \leqslant (2+10\epsilon) \sum_{e \in E} x_e$$

□

Remark By assigning fractional dollars to edges in M^*, the proof of Claim 2.7 also works if M^* is a *fractional* maximum matching.

2.2.3 Round compression

The goal of round compression is to simulate multiple rounds of an iterative algorithm within a single MPC round[6]. To do so, "sufficient information" needs to fit into single machine. For example, suppose algorithm $\mathcal{A}$ requires the k-hop neighbourhood[7] of a vertex v in the k-th iteration. Then, if one can fit the entire k-hop neighbourhood of v into a single machine, k iterations of $\mathcal{A}$ can be compressed to a single MPC round. In the same spirit, we hope to do the following for `Vertex-Centric`:

[6] To keep notation consistent, we reserve the term "round" to refer to MPC rounds and use "iteration" to describe a step of the iterative algorithm.

[7] The k-hop neighbourhood of vertex v is the set of vertices that have a path from v involving at most k edges.

1. Pick appropriate step size k and send subgraphs to machines
2. Each machine simulates k steps of `Vertex-Centric` locally on subgraphs
3. Each machine broadcasts[8] which vertices are frozen, and the value of x_e's. Each edge e updates x_e to minimum $\frac{1}{\Delta}(1+\epsilon)^i$ from all machines.
4. Repeat (in few rounds) until R iterations occur in total

Ideally, we hope that our MPC execution properly emulates `Vertex-Centric` running on a single machine. However, there is a serious risk of machines freezing vertices at different iterations because not all incident edges of every vertex may be in the same machine. This can potentially destroy the property of $\sum_{e \ni v} x_e \geqslant 1 - 2\epsilon$ within a single MPC round. Moreover, we may get a small $\sum_{e \ni v} x_e$ if a vertex is frozen earlier than it should.

Problem with naive round compression

Consider the graph of unfrozen vertices and unfrozen edges G' before the start of an MPC round. Suppose $x_e \geqslant \frac{1}{d}$ for every edge in G', then every vertex in G' has at most d incident edges. Randomly partition vertices of G' into $\sqrt{d}$ machines and send the corresponding induced graphs $G'_1, \ldots, G'_{\sqrt{d}}$. That is, edge $e = \{u, v\}$ will appear in machine i if both u and v are in induced graph G'_i. Therefore, the probability of an edge appearing in *any* machine is $\sqrt{d} \cdot \frac{1}{\sqrt{d}} \cdot \frac{1}{\sqrt{d}} = \frac{1}{\sqrt{d}}$. For a vertex v on machine i, denote

$$\tau_v = \sum_{e \ni v} x_e \qquad \text{and} \qquad \widetilde{\tau}_v = \sqrt{d} \cdot \sum_{e \ni v; e \in G'_i} x_e + \sum_{e \ni v; e \in G \setminus G'} x_e$$

$\widetilde{\tau}_v$ is a local estimate of τ_v, where $\sqrt{d}$ normalizes the partitioning process and $G \setminus G'$ captures the x_e values of edges frozen in earlier iterations. Locally, each machine performs $\mathcal{O}(\log d)$ iterations of `Vertex-Centric`, freezing vertex v if $\widetilde{\tau}_v$ exceeds $(1-2\epsilon)$, the freezing threshold of `Vertex-Centric`.

Initially, $\widetilde{\tau}_v$ is a good estimate to τ_v because of the random partitioning process. However, their behaviours diverge as we run iterations within a MPC round. In an ideal situation where τ_v and $\widetilde{\tau}_v$ *always* fall on the "same side" of the freezing threshold, then both `Vertex-Centric` and the MPC simulation would treat v the same — either both freeze v, or not. In such

[8] For x_e values, only the number i needs to be communicated since $(1+\epsilon)$ is known.

a situation, $\widetilde{\tau}_v$ remains a good estimate to τ_v and the MPC simulation will be a faithful reproduction of `Vertex-Centric`.

Let $\sigma = |\widetilde{\tau}_v - \tau_v|$ denote the estimation difference. If τ_v is near the threshold $(1-2\epsilon)$, then the MPC simulation and `Vertex-Centric` could still freeze v differently even for *very* small σ (See Fig. 2.1). However, observe that if σ is small, then there is only a "small range" of values in which MPC will freeze vertex v at a different iteration as `Vertex-Centric`. This observation motivates the use of a *randomized freezing threshold* in `Vertex-Centric` to reduce the probability of the MPC simulation and `Vertex-Centric` making different decisions on a vertex v, assuming σ is small (See Fig. 2.2).

Figure 2.1: WLOG, $\widetilde{\tau}_v > \tau_v$. Even for small σ, the MPC simulation and `Vertex-Centric` may still decide differently on the freezing of vertex v.

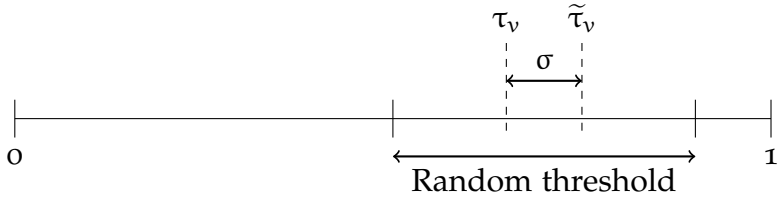


Figure 2.2: WLOG, $\widetilde{\tau}_v > \tau_v$. With a random threshold, the probability of the MPC simulation and `Vertex-Centric` deciding differently on vertex v is σ divided by the range of possible threshold values.

`Centric-Rand`: Randomizing the freezing threshold

Consider the following algorithm `Centric-Rand`, a variant of `Vertex-Centric` where the threshold for freezing a vertex is randomly chosen between $[1-3\epsilon, 1-\epsilon]$:

1. Initialize all x_e to $\frac{1}{\Delta}$
2. For iteration $t \in \{1, \ldots, R\}$

(a) For each vertex v, pick a random threshold $T_{v,t} \in [1-3\epsilon, 1-\epsilon]$

(b) Freeze each vertex v, and all incident edges, if $\sum_{e \ni v} x_e \geqslant T_{v,t}$

(c) For each unfrozen edge e, set x_e to $x_e \cdot (1+\epsilon)$

One can verify that Claim 2.5 and Claim 2.6 also hold for `Centric-Rand`, and Claim 2.7 holds for $\epsilon \leqslant \frac{1}{15}$.

A proper round compression

Consider an MPC simulation that runs for $O(\log\log n)$ rounds. We initialize all $x_e = \frac{1}{\Delta}$. Suppose the current graph $G^{(t)}$ has maximum degree $d = \Delta^t$ at the start of round t, where $G^{(1)}$ is the input graph G and $d = \Delta$ initially. Round t of the MPC simulation is as follows:

1. Randomly split the current vertex set into $\sqrt{d}$ partitions $V_1, \ldots, V_{\sqrt{d}}$.

2. Send machine i the induced graph $G^{(t)}[V_i]$, for $i \in \{1, \ldots, \sqrt{d}\}$.

3. Each machine i *locally* performs the following:

   - Pick random threshold $T_{v,t} \in [1-3\epsilon, 1-\epsilon]$ for each vertex v.
   - Locally simulate $O(\log d)$ steps of `Centric-Rand` by using $\widetilde{\tau}_v = \sqrt{d} \cdot \sum_{e \ni v; e \in G_i^{(t)}} x_e + \sum_{e \ni v; e \in G \setminus G^{(t)}} x_e$ as an estimate of τ_v. If $\widetilde{\tau}_v > T_{v,t}$, freeze v and incident edges (say, at iteration k).

4. Machines broadcast the statuses of the vertices they hold — whether they are frozen, and the latest iteration k in which they were last updated. If a vertex is *not* frozen, then k is the current simulated iteration of `Vertex-Centric`. Otherwise, if a vertex is frozen within a round, k refers to the iteration which it was frozen. An edge e then updates x_e to minimum of $\frac{1}{\Delta}(1+\epsilon)^k$ of its endpoints.

5. Update $\tau_v = \sum_{e \ni v} x_e$ for all vertices v.

6. If $\tau_v > 1$, remove v from the graph $G^{(t)}$.

7. If $\tau_v > 1-2\epsilon$, freeze v and incident edges. Remove them from $G^{(t)}$.

Analysis

Recall from the earlier discussions that if τ_v and $\widetilde{\tau}_v$ *always* fall on the "same side" of the freezing threshold, then both `Vertex-Centric` and the MPC simulation would treat v the same. A vertex v is said to be *bad* if `Centric-Rand` and the MPC simulation freeze v at different iterations. The main goal of the analysis below is to argue that $|\widetilde{\tau}_v - \tau_v|$ remains small for most vertices and we have few bad vertices (See Claim 2.11). To build up to the main claim, we first prove a few other claims.

Claim 2.8. *If the maximum degree of graph $G^{(t)}$ is d, the induced graph $G^{(t)}[V_i]$ fits into a single machine with high probability.*

Proof. (Sketch) We expect $\mathcal{O}(\frac{n}{\sqrt{d}})$ vertices in partition V_i. The probability of an edge appearing in *any* partition is $\frac{1}{\sqrt{d}}$. So, we expect any induced graph to fit in memory. Apply Chernoff bounds. □

Claim 2.9. *If the maximum degree of graph $G^{(t)}$ is d, the maximum degree drops to $d^{0.9}$ after $\mathcal{O}(\log d)$ iterations of `Centric-Rand`.*

Proof. (Sketch) After $\mathcal{O}(\log d)$ steps, weight of unfrozen edges x_e increase by $(1+\epsilon)^{\mathcal{O}(\log d)}$. Observe that frozen edges are removed from the graph at the end of each round. □

Claim 2.10. *After $\mathcal{O}(\log\log n)$ rounds, the $G^{(t)}$ fits into a single machine.*

Proof. (Sketch) By previous claim, after $\mathcal{O}(\log\log n)$ rounds, maximum degree is $\Delta^{0.9^{\mathcal{O}(\log\log n)}} \leqslant n^{\mathcal{O}(\log n)} \in \widetilde{\mathcal{O}}(n)$ since $\Delta \leqslant n$. □

Claim 2.11 (Main claim)**.** *Throughout the simulation, the estimation difference $|\widetilde{\tau}_v - \tau_v|$ remains small for most vertices and the total number of bad vertices is at most a constant fraction of the size of a maximum matching.*

Proof. See Sections 4.4.3 and 4.4.4 of Ghaffari et al. [GGK+18]. □

Claim 2.12. *The number of vertices are removed in step 6 is at most a constant fraction of the size of a maximum matching.*

Proof. See Page 20 of Ghaffari et al. [GGK+18]. □

The above discussion briefly sketches the round compression approach of Ghaffari et al. [GGK+18]. Readers are encouraged to refer to Section 4.3 of Ghaffari et al. [GGK+18] for details. Note that Ghaffari et al. [GGK+18] use different constants and terminology. For example, they use random threshold range of $[1-4\epsilon, 1-2\epsilon]$, and use terms "phase" and "round" wherever we use "round" and "iteration".

Exercises

Exercise 2.2. Maximal Independent Set (MIS) in MPC
For a graph $G = (V, E)$, a Maximal Independent Set (MIS) is a subset $I \subseteq V$ of vertices such that (i) no two vertices in I share an endpoint, and (ii) $I \cup \{v\}$ for any $v \in V \setminus I$ violates the first property.

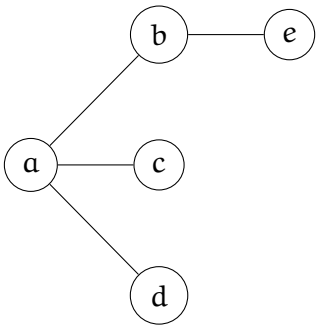


Figure 2.3: Possible MISs include $\{a, e\}$ and $\{b, c, d\}$.

(a) (Strongly superlinear regime, looser analysis) Using $M \in \widetilde{O}(\frac{N}{S})$ machines, each with memory $S = n^{1+\epsilon}$, design an algorithm that computes an MIS to an input graph in $O(\frac{1}{\epsilon})$ rounds.

(b) (Strongly superlinear regime, tighter analysis) Using $M \in \widetilde{O}(\frac{N}{S})$ machines, each with memory $S = n^{1+\epsilon}$, design an algorithm that computes an MIS to an input graph in $O(\log(\frac{1}{\epsilon}))$ rounds.

Remark *Setting $\epsilon = \frac{1}{\log n}$ in (b) yields an $O(\log \log n)$ round algorithm for the near linear memory regime with $S = n \cdot poly(\log n)$.*

Hint *Consider the following randomized greedy algorithm $\mathcal{A}$ that solves MIS on a single machine: Pick a random permutation π, and greedily try to add vertex $\pi(i)$ to the MIS. To simulate $\mathcal{A}$ in MPC, consider first putting a chunk of k vertices $\pi(1), \pi(2), \ldots, \pi(k)$ onto a single machine and running $\mathcal{A}$ on them (in a single MPC round). Argue that*

- *The induced graph involving the k vertices fit in memory, for appropriately chosen k*
- *The maximum degree of any remaining vertex is sufficiently low*

- *Recursing with the appropriate chunk sizes solves the problem in the required number of rounds*

Part (a) uses a loose bound of $n \cdot k$ *for the number of edges in the induced graph. Part (b) tightens the analysis by only considering edges whose* both *endpoints are in the same partition.*

2.3 Matching using strongly sublinear memory

In this section, we wish to compute a constant approximation to maximum matching using n^α memory, for some constant $\alpha \in (0, 1)$. We will present a method from Ghaffari and Uitto [GU19, Section 2] which computes a constant approximation to maximum matching in $\widetilde{O}(\sqrt{\log \Delta})$ rounds, where Δ is the maximum degree of the input graph G.

Outline

Let $R = O(\log \Delta)$. An ideal outline would be to pick a suitable algorithm in a LOCAL model that computes a constant approximation to maximum matching in R LOCAL rounds, use *graph exponentiation* to learn about the R-hop neighbourhood in $O(\log R)$ MPC rounds, and locally simulate the entire algorithm in one final MPC round. Unfortunately, the current technique can only compress up to $\sqrt{R}$ rounds before memory requirements exceed S. The resultant outline is as follows:

- Consider a suitable R-round algorithm in the LOCAL model
- *Sparsify* the graph (by subsampling edges) such that $\sqrt{R}$-hop neighbourhood of each vertex fits into memory $S = n^\alpha$, for $\alpha \in (0, 1)$
- Use *graph exponentiation* so that every vertex learns the $\sqrt{R}$-hop neighbourhood in $O(\log \sqrt{R})$ MPC rounds
- Simulate $O(\sqrt{R})$ LOCAL rounds in one MPC round using knowledge of the $\sqrt{R}$-hop neighbourhood

Putting together, the entire R-round LOCAL algorithm can be simulated in $\sqrt{R} \cdot O(\log \sqrt{R}) = \widetilde{O}(\sqrt{R})$ MPC rounds. We discuss the LOCAL model in Section 2.3.1, graph exponentiation in Section 2.3.2, a suitable R-round LOCAL algorithm in Section 2.3.3, and how to use sparsification in Section 2.3.4.

Simplifying assumptions

To simplify exposition, we make the following assumptions:

1. We can store a vertex and its incident edges on one machine: $\Delta \leqslant n^{\alpha}$
2. We can assign a machine for each vertex: $M \in \mathcal{O}(n)$

The first assumption can be removed by "load balancing" appropriate across machines and using broadcast trees. The second assumption may result in using a lot more total global memory than the input size N. To remove the second assumption, one can modify the described algorithm to work through $\log\log(\Delta)$ successive iterations of polynomially decreasing degree classes $[\Delta^{1/2}, \Delta]$, $[\Delta^{1/4}, \Delta^{1/2}]$, $[\Delta^{1/8}, \Delta^{1/4}]$, and so on. Doing so, the total memory requirement will not exceed $\mathcal{O}(N)$ at each step. For details, refer to Ghaffari and Uitto [GU19].

2.3.1 The LOCAL model

The LOCAL model was first formalized by Linial [Lin87, Lin92], introducing the notion of "thinking like a vertex". Suppose each vertex $v \in V$ has a (very powerful) computer, and vertices communicate across edges in *synchronous rounds* by sending one (potentially large) message to each neighbour. The vertices do not know the structure of the graph G a priori. However, the vertices may be aware of global parameters such as upper bounds on the number of vertices or the maximum degree of the graph. `Vertex-Centric` in Section 2.2.2 is an example of a LOCAL algorithm.

Lemma 2.13. Any R*-round LOCAL algorithm can be simulated locally by each vertex* v *in* a single round *if* v *knows the topology of its* R*-hop neighbourhood. If the LOCAL algorithm is a randomized algorithm, also gather the random bits used by each vertex in the* R*-hop neighbourhood.*

Proof. Since each vertex can only communicate with their neighbours in the LOCAL model, apply induction over the number of rounds R. If random bits are known, the algorithm can be simulated deterministically. □

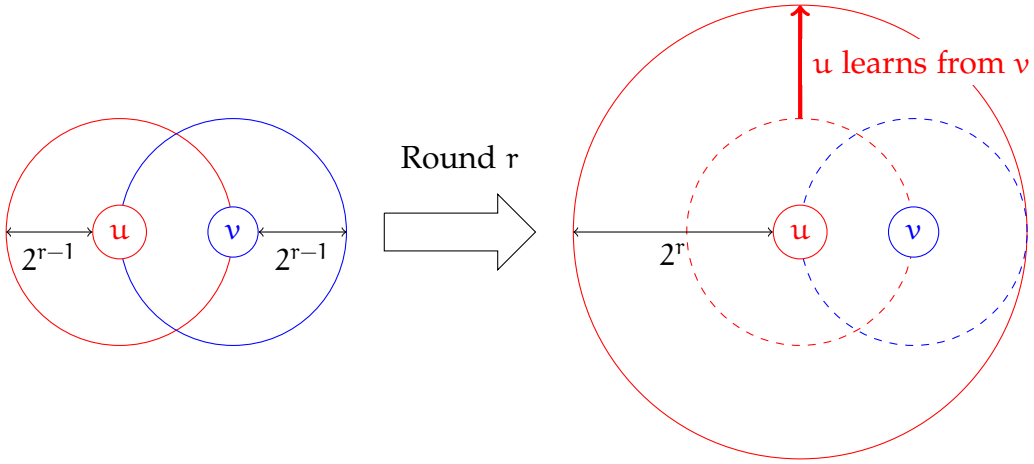
The above lemma implies that any problem in input graph G can be solved in $\mathcal{O}(\text{diameter}(G))$ rounds under the LOCAL model.

2.3.2 Graph exponentiation

The idea of graph exponentiation was first mentioned under the CONGESTED-CLIQUE model by Lenzen and Wattenhofer [LW10]. In the CONGESTED-CLIQUE model, a vertex can communicate to all other vertices under the constraint that messages sent across an edge in a single round can have size at most $\mathcal{O}(\log n)$ bits. Recently, Ghaffari [Gha17, Lemma 2.14] used graph exponentiation to obtain an improved MIS algorithm on congested clique.

Lemma 2.14. *With $\mathcal{O}(n)$ machines, each with memory to hold the k-hop neighbourhood of each vertex, all vertices in graph $G = (V, E)$ can learn their k-hop neighbourhood in at most $\log(k)$ MPC rounds.*

Proof. For $v \in V$, assign a single machine M_v to be responsible it. Denote $N^r(v)$ as the r-hop neighbourhood of v at the beginning of the r^{th} MPC round, where edges between vertices in $N^r(v)$ are stored in the M_v. Initially, the machine holding v knows only edges incident to v and can compute the 1-hop neighbourhood $N^1(v)$ implicitly. For $r = \{2, \ldots, \lceil \log k \rceil\}$, M_v sends all edges it holds to $M_{v'}$ for $v' \in N^r(v)$. By induction over r, each machine M_v knows the 2^r-hop neighbourhood of v at the end of the r^{th} MPC round.



□

The lemma assumes that the number of edges in $N^r(v)$ of *any* vertex $v \in V$ fits into the memory S of a single machine. Furthermore, it assumes that we have at least $|V|$ machines. In the case where a graph G is "sparse enough", we have the following corollary.

Corollary 2.15. *Consider a given graph* $G = (V, E)$ *with* $n = |V|$ *under the sublinear memory regime with memory* $S = n^{\alpha}$*, for some constant* $\alpha \in (0, 1)$*. If there are* $\mathcal{O}(n)$ *machines and* $|N^R(v)| \leqslant n^{\frac{\alpha}{4}}$ *for all vertices* $v \in V$*, then one can transform any* R*-round LOCAL algorithm into an* $\mathcal{O}(\log R)$*-round MPC algorithm.*

Proof. Since $|N^R(v)| \leqslant n^{\frac{\alpha}{4}}$ for all $v \in V$, there are at most $n^{\frac{\alpha}{2}} \leqslant S$ edges in the R-hop neighbourhood of *any* vertex v. By Lemma 2.14, one can assign a machine for each vertex and learn the R-hop neighbourhoods in $\mathcal{O}(\log R)$ MPC rounds. Then, Lemma 2.13 tells us that we can simulate the R-round LOCAL algorithm in one MPC round. □

Remark The technique is called graph exponentiation because one implicitly builds the 2^k-th power G^{2^k} of the input graph G in the k^{th} iteration. Here, we define the power graph[9] G^x to be the graph obtained from G by connecting every two nodes within distance x from each other, ignoring edge weights.

2.3.3 A $\Theta(\log \Delta)$-round LOCAL algorithm

Algorithm

Consider the following $\mathcal{O}(\log \Delta)$ algorithm `Peeling-Matching-LOCAL`:

1. For iteration $i \in \{0, 1, \ldots, \log \Delta\}$:
   (a) Mark each edge independently with probability $p_i = \frac{2^i}{4\Delta}$.
   (b) Let M_i be the set of *isolated marked edges* at iteration i. An edge is in M_i if it is marked and has no adjacent marked edge.
   (c) Denote V_i as the set of matched vertices with an endpoint in M_i, and vertices with degree more than $d_i = \frac{\Delta}{2^{i+1}}$.
   (d) Add M_i to the matching and remove V_i from the graph.

Analysis

`Peeling-Matching-LOCAL` "peels" off high-degree vertices iteratively by removing vertices with degree larger than $\frac{\Delta}{2^{i+1}}$ at the end of each iteration i. The peeling process ensures an upper bound on the maximum degree

[9] See https://en.wikipedia.org/wiki/Graph_power on graph power.

of the graph, allowing the sampling probability $p_i = \frac{2^i}{4\Delta}$ to extract a suitable number of isolated marked edges such that $\frac{|V_i|}{|M_i|} \in \Theta(1)$ with high probability. Hence, `Peeling-Matching-LOCAL` returns a matching which is a constant approximation to maximum matching[10].

Lemma 2.16. *At the start of each iteration i, all vertices have degree at most $\frac{\Delta}{2^i}$.*

Proof. When $i = 0$, all vertices have degree at most Δ. Subsequently, this property is ensured by the removal of V_i in each step. □

Lemma 2.17. *Suppose there are k high-degree vertices at iteration i, then*

$$\Pr[\frac{|V_i|}{|M_i|} \in \Theta(1)] \geqslant 1 - e^{-\Theta(k)}$$

Proof. (Sketch) Removed vertices V_i consists of matched vertices and high-degree vertices. We will show that the number of matched high-degree vertices is $\Theta(k)$ with high probability, which then implies the result.

Consider an arbitrary high-degree vertex v with degree more than $d_i = \frac{\Delta}{2^{i+1}}$. Since each edge is marked independently with probability $p_i = \frac{2^i}{4\Delta}$, the probability that v is *not* incident to any marked edge is less than $(1-p_i)^{d_i} \leqslant e^{-\frac{1}{8}} \leqslant 0.9$. That is, v is incident to some marked edge with probability > 0.1. For any edge x incident to v, x has no marked neighbours with probability larger than $(1-p_i)^{2d_i-2} \geqslant 4^{-\frac{1}{4}-2p_i} > 0.04$. Hence, v is matched with probability $\Omega(1)$. As this holds for any high-degree vertex, one expects $\Omega(k)$ high-degree vertices to be matched.

By Lemma 2.16, all vertices have degree at most $\frac{\Delta}{2^i}$. So, changing the marking outcome of a single edge only affects the matching status at most $2 \cdot \frac{\Delta}{2^i}$ vertices. One can then show appropriate concentration bounds. □

2.3.4 An $\widetilde{\mathcal{O}}(\sqrt{\log \Delta})$-round MPC algorithm

Using graph exponentiation, one hopes to compress multiple LOCAL rounds of `Peeling-Matching-LOCAL` using fewer MPC rounds . However, it is currently unknown how to compress all $\log(\Delta)$ LOCAL rounds. Here, we present a method to compress $R = \frac{1}{2}\sqrt{\log \Delta}$ LOCAL rounds into one MPC phase, consisting of $\log(R)$ MPC rounds. Hence, the $\Theta(\log \Delta)$-round `Peeling-Matching-LOCAL` can be simulated in $\widetilde{\mathcal{O}}(\sqrt{\log \Delta})$ MPC rounds.

[10] Recall that the maximum matching size is at most $|V|/2$ and $V = V_0 \cup V_1 \cup \cdots \cup V_{\log \Delta}$.

Algorithm

To ensure that Corollary 2.15 applies, we first sparsify the graph so that the number of edges in the R-hop neighbourhood fits into memory, then run a modified version of `Peeling-Matching-LOCAL`. The entire process `Peeling-Matching-MPC` is as follows:

1. Define parameters $K = c \log \Delta$ and $R = \frac{1}{2}\sqrt{\log \Delta}$, for some constant c
2. Denote variables $d_{j,i} = \frac{\Delta}{2^{i+1}} \cdot \frac{1}{2^{Rj}}$, $p_{j,i} = \frac{2^i}{4\Delta} \cdot \frac{1}{2^{Rj}}$ and $p'_{j,i} = \min\{1, K \cdot p_i\}$
3. For $j \in \{0, \ldots, 2\sqrt{\log \Delta} - 1\}$ phases:
   (a) For $i \in \{1, \ldots, R\}$, form subgraph H_i by sampling each edge of G independently with probability p'_i
   (b) Let $H = \cup_{i=1}^{R} H_i$ be the sparsified graph of G.
   (c) Use graph exponentiation on H, learn the R-hop neighbourhood for each vertex in $O(\log R)$ MPC rounds.
   (d) Locally, in one MPC round, simulate iterations $i \in \{1, \ldots, R\}$:
      i. Mark each edge of H_i independently w.p. $p_{j,i}/p'_{j,i} \leqslant \frac{1}{K}$.
      ii. Let M_i be the set of *isolated marked edges* at iteration i.
      iii. Denote V_i as the set of matched vertices with an endpoint in M_i, and vertices with degree more than $d_i \cdot p'_{j,i} \geqslant \frac{K}{8}$ in H_i.
      iv. Add M_i to the matching and remove V_i from $H_{i+1}, \ldots, H_R$.
   (e) Remove from G any vertex with degree more than $d_{j,R}$ in G.

Since edges are sampled independently, the degree of each vertex in each H_i is an unbiased estimate of their actual degree in G. That is, a vertex with degree d in G is *expected* to have $d \cdot p'_i$ edges in H_i. The factor K is a sampling overhead for concentration bounds purposes. Observe that all vertices have degree $\leqslant \frac{\Delta}{2^{Rj}}$ in G at the start of phase j as vertices with degree $> d_{j,R}$ in G are removed at the end of each phase.

Analysis

We analyze the first phase of `Peeling-Matching-MPC` where $j = 0$. For notational cleanliness, we drop the subscript j in the analysis to mean $j = 0$ and write $p'_{0,i} = p'_i = K \cdot p_i$. The proofs also work for the subsequent phases since we scale the upper bound on the maximum degree by 2^R via the factor $\frac{1}{2^{Rj}}$. Using concentration bounds, we will upper bound the

probability of "failure" by e^{-K}. For easier concentration bounds, one may treat $K = \Theta(\log n)$, instead of the actual $K = \Theta(\log \Delta)$, so that $e^{-K} = \frac{1}{\text{poly}(n)}$ in the proof sketches below. For details on analyzing $K = \Theta(\log \Delta)$ where $e^{-K} = \frac{1}{\text{poly}(\Delta)}$, refer to Ghaffari and Uitto [GU19].

Lemma 2.18. *W.h.p., sparsified graph H has maximum degree $\leqslant K \cdot 2^{\frac{1}{2}\sqrt{\log \Delta}}$.*

Proof. Since $p'_i = K \cdot \frac{2^i}{4\Delta}$, a vertex v with degree Δ in G is expected to have a degree $K \cdot \frac{2^i}{4}$ in H_i. Taking the union over all H_i, v is expected to have a degree of $\leqslant \sum_{i=1}^{R} K \cdot \frac{2^i}{4} \leqslant \frac{1}{2} \cdot (K \cdot 2^{\frac{1}{2}\sqrt{\log \Delta}})$. Apply Chernoff bounds. □

Lemma 2.19. *W.h.p., R-hop neighbourhoods in H have $\leqslant \Delta^{0.5}$ edges.*

Proof. By Lemma 2.18, H has maximum degree at most $K \cdot 2^{\frac{1}{2}\sqrt{\log \Delta}}$ w.h.p. So, the number of edges in the R-hop neighbourhood of any vertex is at most $(K \cdot 2^{\frac{\sqrt{\log \Delta}}{2}})^R \leqslant \Delta^{0.5}$, for appropriate constant factor of K. □

Lemma 2.20. *After iteration i, $\Pr[\deg_G(v) > 2d_i] \leqslant e^{-K}$, for all $v \in H$.*

Proof. For a vertex with degree $> 2d_i$ in G, we expect to see $> 2d_i p'_i$ edges in H. Recall that Step 3(d)(iii) removes all vertices in H with degree $> d_i p'_i$ in H. Apply Chernoff bounds. □

Lemma 2.21. *Suppose there are k high-degree vertices at iteration i, then*

$$\Pr[\frac{|V_i|}{|M_i|} \in \Theta(1)] \geqslant 1 - e^{-\Theta(k)}$$

Proof. (Sketch) Similar to Lemma 2.17 except one has to account for sparsification of G via p'_i. See Ghaffari and Uitto [GU19] for details. □

Exercises

Exercise 2.3. Orienting edges with strongly sublinear memory *For a graph $G = (V, E)$, the* arboricity λ *of G is defined as $\lambda = \max_{S \subseteq V} \frac{E(S)}{|S|-1}$. For example, trees have arboricity 1. For constants $\alpha \in (0, 1)$ and $\epsilon > 0$, devise an algorithm that uses $M \in \widetilde{\mathcal{O}}(\frac{N}{S})$ machines, each with memory $S = n^\alpha$, to orient the edges of a graph with arboricity λ in $\widetilde{\mathcal{O}}(\sqrt{\log n})$ MPC rounds such that each vertex has out-degree of at most $(2 + \epsilon) \cdot \lambda$.*

Remark *Nash-Williams [NW64] tells us that arborcity λ can also be defined as the minimum number of forests which we can partition graph edges into.*

Hint *Since G has arborcity λ, the average degree is at most 2λ. One can show that there are at most $(1 - \frac{\epsilon}{2+\epsilon}) \cdot n$ vertices with degree larger than $(2+\epsilon)\lambda$. Consider the following $O(\log n)$-round LOCAL algorithm: At each iteration, all vertices with degrees at most $(2+\epsilon)\lambda$ orient incident edges outwards.*

- *Argue that by removing oriented vertices and edges from the graph, the process ends in $O(\log n)$ LOCAL rounds.*
- *Compress $R = O(\sqrt{\log n})$ LOCAL rounds into one phase of $O(\log \log n)$ MPC rounds via graph exponentiation, while not peeling vertices with degree larger than $2^{\sqrt{\log n}} \cdot 2\lambda$ in G at the end of each phase. Let L_i be the set of vertices with degree larger than $2^{\sqrt{\log n}} \cdot 2\lambda$ in G that are not peeled at phase i. Argue that $|L_i|$ is "small". Sparsify the graph G by sampling edges with probability $\Theta(\frac{\log n}{\lambda \epsilon^2})$ so that R-hop neighbourhoods fit in memory.*

Exercise 2.4. Constant tree coloring with strongly sublinear memory
Devise an algorithm that computes an $O(1)$ coloring of any given n-node tree in $\widetilde{O}(\sqrt{\log n})$ rounds under the strongly sublinear memory regime.

Hint *Use Exercise 2.3 with a LOCAL tree coloring algorithm.*

2.4 Approximation improvement via augmenting paths

In the earlier sections, we showed how to obtain constant approximations to maximum matching under the three different memory regimes. The goal of this section is to show that one can transform any algorithm for $O(1)$-approximation of maximum matching to an algorithm for $(1+\epsilon)$-approximation, for any constant $\epsilon > 0$.

We first introduce augmenting paths and some related facts. Then, we describe an $O(m\sqrt{n})$ centralized algorithm for maximum matching on bipartite graphs due to Hopcroft and Karp [HK73]. Building on similar ideas, McGregor [McG05] gives an $(1+\epsilon)$-approximation algorithm in the streaming model. Finally, we explain how to use McGregor's idea in MPC.

Remark In the following exposition, we consider unweighted and undirected graphs. For $(1+\epsilon)$-approximations in weighted graphs, see Duan and Pettie [DP10] for the sequential setting and Gamlath et al. [GKMS18] for the MPC setting.

2.4.1 Augmenting paths and related facts

Before we begin, let us state some definitions following notation of Hopcroft and Karp [HK73]. A subset of edges $M \subseteq E$ is a matching if no two edges in M share an endpoint. With respect to a matching M, a vertex $v \in V$ is *free* if it is not incident to any edge in M, and is *matched* otherwise. A path, without repeated vertices, $P = \{v_1, v_2\}\{v_2, v_3\} \dots \{v_{2k-1}, v_{2k}\}$ is called an *augmenting path* relative to M if its endpoints v_1 and v_{2k} are free, and its edges are alternatively in $E \setminus M$ and M. That is, $\{v_2, v_3\}$, $\{v_4, v_5\}$, $\dots$, $\{v_{2k-2}, v_{2k-1}\} \in M$ while the other edges in P are in $E \setminus M$. When no ambiguity is possible, we use P to refer to both the set and sequence of edges in the augmenting path. If S and T are sets, then $|S|$ refers to the cardinality of S, and $S \oplus T$ refers to their symmetric difference of S and T.

Lemma 2.22. *If M is a matching and P is an augmenting path relative to M, then $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$.*

Proof. By definition of augmenting paths. □

Theorem 2.23. *Let M^* be a maximum matching and M be a matching. If $|M^*| = s$ and $|M| = r < s$, then there is an augmenting path relative to M of length at most $2\lfloor \frac{r}{s-r} \rfloor + 1$.*

Proof. Consider the graph $G' = (V, M \oplus M^*)$. Since M and M^* are matchings, each vertex v in G' has degree at most 2. That means every vertex is either isolated or lies in a cycle/path with edges alternating between M and M^*. If v is isolated or lies in a cycle, the number involved edges from M and M^* are equal. If v lies on a path, then there is one more edge from M^* involved than M because M^* is a maximum matching. We see that all such paths are augmenting paths relative to M and there are $|M^*| - |M| = s - r$ augmenting paths relative to M.

If there is an augmenting path that does not involve any edge from M, we have found an augmenting path of length 1. Otherwise, by averaging argument, there is an augmenting path P with $k \leqslant \frac{r}{s-r}$ edges from M. Then, since every augmenting path has one more edge from M^* than M, we see that $|P| = 2k + 1$ as desired. □

Corollary 2.24. *If there is no augmenting path of length k relative to a matching M, then M is a $(1 + \mathcal{O}(\frac{1}{k}))$-approximate maximum matching.*

Corollary 2.24 will be useful later when we wish to compute a $(1+\epsilon)$-approximation using constant approximations to maximum matching.

Corollary 2.25 (Berge [Ber57])**.** *A matching M is a maximum matching if and only if there are no augmenting paths relative to M.*

Theorem 2.26. *Let P be the shortest augmenting path relative to M. Let P' be* any *augmenting path relative to $M \oplus P$. Then, $|P'| \geqslant |P| + |P \cap P'|$.*

Proof. If $P \cap P' = \emptyset$, then $|P'| \geqslant |P| = |P| + |P \cap P'|$ since P be the shortest augmenting path relative to M.

Denote $N = (M \oplus P) \oplus P'$. Then, N is a matching of size $|M| + 2$ and $N \oplus M$ has has two vertex-disjoint augmenting paths P_1 and P_2 relative to M. See Fig. 2.4 for an illustration.

Since $N \oplus M = P \oplus P'$, $|P \oplus P'| = |N \oplus M| \geqslant |P_1| + |P_2|$. Then, since P was the shortest augmenting path relative to M, $|P| \leqslant |P_1|$ and $|P| \leqslant |P_2|$. Thus,

$$\begin{aligned} 2 \cdot |P| &\leqslant |P_1| + |P_2| \\ &\leqslant |P \oplus P'| \\ &= |P| + |P'| - |P \cap P'| \end{aligned}$$

Rearranging, we have $|P'| \geqslant |P| + |P \cap P'|$. □

Consider the following scheme `Iterative-Augmentation` for computing a maximum matching:

1. Set initial matching M_0 as $\emptyset$
2. For $i \in \{0, 1, 2, \dots\}$
   (a) If there is no augmenting paths relative to M_i, return M_i as the maximum matching
   (b) Otherwise, let P_i be the shortest augmenting path relative to M_i and define $M_{i+1} = M_i \oplus P_i$.

Corollary 2.25 tells us that `Iterative-Augmentation` produces a maximum matching while Theorem 2.26 tells us that $|P_i| \leqslant |P_{i+1}|$ for any step i and the following corollary.

Corollary 2.27. *In `Iterative-Augmentation`, if $i \neq j$ and $|P_i| = |P_j|$, then P_i and P_j are vertex-disjoint.*

Proof. Suppose, for a contradiction, that there are two augmenting paths P_i and P_j such that $|P_i| = |P_j|$ and are *not* vertex-disjoint. Then, application of Theorem 2.26 tells us that either $|P_i| \neq |P_j|$ or $|P_i \cap P_j| = 0$. See Hopcroft and Karp [HK73, Corollary 4] for details. □

Theorem 2.28. *Excluding the last $\sqrt{n}$ augmenting paths, all other augmenting paths have length at most $3\sqrt{n}$ in `Iterative-Augmentation`.*

Proof. By Theorem 2.23, any augmenting path relative to a matching M_i has length at most $2\lfloor \frac{r}{s-r} \rfloor + 1$, where s is the size of the maximum matching and $|M_i| = r \leqslant s \leqslant n$. Excluding for the last $\sqrt{n}$ steps, $r \leqslant s - \sqrt{n}$. Thus,

$$2\lfloor \frac{r}{s-r} \rfloor + 1 \leqslant 2\lfloor \frac{n}{\sqrt{n}} \rfloor + 1 \leqslant 3\sqrt{n}$$

□

Remark Theorem 2.23 is a combination of Theorem 1 and Corollary 2 from Hopcroft and Karp [HK73]. In the paper, Theorem 1 was a more general statement about any two matchings of sizes s and r. Theorem 2.28 is a variant of Theorem 3 from Hopcroft and Karp [HK73].

2.4.2 An $O(m\sqrt{n})$ centralized algorithm for maximum matching on bipartite graphs

We now present an $O(m\sqrt{n})$ centralized algorithm for maximum matching on *bipartite graphs* due to Hopcroft and Karp [HK73, Section 3]. Micali and Vazirani [MV80] have an $O(m\sqrt{n})$ algorithm on *general graphs* but it is beyond the scope of this section.

Algorithm

The idea is to perform multiple steps of `Iterative-Augmentation` in a single *phase*. In each phase, we extract a maximal set of vertex-disjoint shortest augmenting paths relative to the current matching, and augment all these paths simultaneously. For example, suppose $|P_0| = |P_1| = |P_2| < |P_3|$ in a run of `Iterative-Augmentation`. By Corollary 2.27, paths P_0, P_1 and P_2 are all vertex-disjoint, thus we can augment all three paths in a single phase. Theorem 2.28 tells us that there will be $O(\sqrt{n})$ such phases.

Consider the following algorithm `Phase-Augmentation`. For a graph $G = (V, E)$ with vertex bipartitions A and B, set initial matching M_0 as $\emptyset$ and run for $O(\sqrt{n})$ phases, for $i \in \{0, 1, 2, \ldots, 4\sqrt{n}\}$:

1. Consider a directed graph $G' = (V, E')$ derived from G and M_i. For edge $\{u, v\}$ with $u \in A$ and $v \in B$, direct u to v if $\{u, v\} \in M_i$, and direct v to u if $\{u, v\} \notin M_i$. Create auxiliary target vertex t and auxiliary edges $\{u, t\}$ for every free vertex $u \in A$. Starting from t, run Breadth-First Search (BFS) on G' using the *reversed* edge directions of E' and alternate usage of edges in M_i and $E \setminus M_i$.

2. If BFS completes before a free vertex $v \in B$ is found, return M_i as the maximum matching.

3. Otherwise, let d be the earliest BFS depth with a free vertex $v \in B$. Create auxiliary source vertex s and auxiliary edges $\{s, v\}$ for every free $v \in B$ at depth d. Extract a s-t path P_1 by running Depth-First Search (DFS). Remove P_1 from G' and iterate the process of extracting s-t paths, backtracking if needed whenever the DFS reaches a "dead-end" due to earlier removed paths. Suppose the iterative process extracted *vertex-disjoint* augmenting paths $P_1, \ldots, P_j$ relative to M_i. Set $M_{i+1} = M_i \oplus P_1 \oplus \cdots \oplus P_j$ by augmenting extracted paths.

By stopped the BFS when we first encounter a free vertex from B, any augmenting path extracted from G' will be of shortest length relative to M. The iterative DFS process ensures we extract *vertex-disjoint* paths while visiting each edge in G' once — a visited edge becomes part of an augmenting path, or is backtracked and ignored in subsequent iterations.

Example

Consider the bipartite graph G in Fig. 2.5a with 13 vertices with a matching M. Vertices $\{a, b, c, d, e, f, g\}$ are in partition A and vertices $\{h, i, j, k, l, m\}$ are in partition B. For phase $i = 3$, Fig. 2.5b shows the BFS output of Step 1 of `Phase-Augmentation`. Two possible length 3 augmenting paths that can be extracted in Step 3 of `Phase-Augmentation` are highlighted in red.

Analysis

Claim 2.29. *There are at most $4\sqrt{n}$ phases.*

Proof. By Theorem 2.28, there are at most $3\sqrt{n}$ different augmenting path lengths before the last $\sqrt{n}$ iterations of `Iterative-Augmentation`. The last $\sqrt{n}$ iterations can have at most $\sqrt{n}$ different augmenting path lengths. □

Claim 2.30. *Step 1 of `Phase-Augmentation` takes $O(m)$ time.*

Proof. Ignoring isolated vertices in G, BFS takes $\mathcal{O}(m)$ time. □

Claim 2.31. *Step 3 of* `Phase-Augmentation` *takes* $\mathcal{O}(m)$ *time.*

Proof. Ignoring isolated vertices in G, DFS takes $\mathcal{O}(m)$ time. □

Claim 2.32. `Phase-Augmentation` *takes* $\mathcal{O}(m\sqrt{n})$ *time and returns a maximum matching.*

Proof. There are at most $4\sqrt{n}$ phases, each taking $\mathcal{O}(m)$ time. Correctness of output follows from Corollary 2.25. □

2.4.3 Approximation improvement in MPC

In earlier sections (Section 2.1, Section 2.2, and Section 2.3), we saw how to compute constant approximations of maximum matching in MPC under various memory regimes. Here, we describe a streaming algorithm due to McGregor [McG05] that computes a $(1+\epsilon)$ approximation to maximum matching in *general graphs*. Then, we will show how to adapt his ideas to use our constant approximations in MPC to compute $(1+\epsilon)$-approximations to maximum matching in $\mathcal{O}_\epsilon(1)$ MPC rounds, where the constant factor $\mathcal{O}_\epsilon(1)$ depends on ϵ.

Augmenting paths in general graphs are not as "well behaved" as those in bipartite graphs. This motivates the construction of a *layer graph* (which we define later) so that any extracted path from the layer graph is a valid augmenting path, and subpaths between layers can be extracted independently of other layers. Below, we give an overview of McGregor's approach, then describe the following two key ideas: (1) Construction of layer graphs, and (2) how to extract augmenting paths from layer graphs. An example will be given to illustrate the ideas and we wrap up by explaining how to adapt McGregor's approach to the MPC setting.

Overview of approach

In similar spirit to Corollary 2.24, McGregor [McG05, Lemma 1] informally states: If there are few augmenting paths relative to M, then M is a good approximation to the maximum matching. This tells us that finding sufficiently many augmenting paths suffice to compute an approximate maximum matching.

The algorithm of McGregor [McG05] works in the streaming model and has a similar flavor to `Phase-Augmentation`. Instead of a BFS, for

$i = \{1, 2, \dots\}$, McGregor constructs a layer graph with $i + 1$ layers, then searches in the layer graph for $\text{poly}(\epsilon) \cdot |M|$ augmenting paths of length $2i + 1$ to augment the current matching M. For the full algorithm and analysis, see McGregor [McG05].

Idea 1: Random projection to layer graphs

For a constant i, matched edges $e \in M$ are treated as nodes in a layer graph G' and distributed across i layers. Edges exist between nodes in the layer graph G' if there are corresponding edges in G. Any extracted path from G' will be an augmenting path of length $2i + 1$ relative to M in G. Furthermore, edges between layers can be chosen independently.

Definition Given a matching M in phase i, a layer graph is made up of layers $L_0, L_1, \dots, L_{i+1}$ of nodes. Layers L_0 and L_{i+1} consist of free vertices while internal layers represent matched edges in M. For each node, we use a superscript to denote the layer level, and a subscript to denote its label. For example, $x^i_{u,v}$ is a projected node at level i for the edge $\{u, v\} \in E$. For path building purposes, it is important to distinguish $x^i_{u,v}$ from $x^i_{v,u}$. We denote the projection of a matched edge $e \in M$ in the layer graph by $\text{proj}(e) = x^i_{u,v}$, and write $\text{level}(x^i_{u,v}) = i$ to indicate the level of the node.

Construction The layer graph is built as follows:

- For a free node $u \in V$, project to $x^0_{u,u}$ or $x^{i+1}_{u,u}$ uniformly at random.
- For an edge $\{u, v\} \in M$, pick a random level uniformly at random from $j \in \{1, \dots, L\}$, and project to $x^j_{u,v}$ or $x^j_{v,u}$ uniformly at random.

For nodes $x^i_{a,b}$ and $x^j_{c,d}$ in the constructed graph, add an edge between them if the nodes are in adjacent levels (i.e. $|i - j| = 1$) and if the corresponding edge exists in G (i.e. $b = c$ and $\{b, c\} \in E$).

Observations Consider an augmenting path $P = e_1, e_2, \dots, e_{2i+1}$ of length $2i + 1$ with $e_2, e_4, \dots, e_{2i} \in M$. Augmenting path P appears in the layer graph if endpoints are in L_0 and L_{i+1}, and sequence of edges appear in order, including the order of $x_{u,v}$ or $x_{v,u}$. Since each free vertices and

matched edges are projected independently,

$$\begin{aligned}\Pr[P \text{ appears}] &= 2 \cdot (\text{level}(\text{proj}(e_1)) = 0 \wedge \text{level}(\text{proj}(e_2)) = 1 \wedge \\ &\qquad \text{level}(\text{proj}(e_4)) = 2 \wedge \cdots \wedge \\ &\qquad \text{level}(\text{proj}(e_{2i})) = i \wedge \text{level}(\text{proj}(e_{2i+1})) = i+1) \\ &= 2 \cdot (\frac{1}{2} \cdot \frac{1}{2i} \cdot \cdots \cdot \frac{1}{2i} \cdot \frac{1}{2}) \\ &= \frac{1}{2(2i)^i}\end{aligned}$$

Since we randomly project edges and appearance of paths are independent given edge appearances, every augmenting path of length $2i+1$ appears independently in G'. As $\frac{1}{2(2i)^i}$ is a constant, we expect a constant fraction of augmenting paths of length $2i+1$ to appear in G'.

Idea 2: Approximate layer matching via layered DFS

Recall that `Phase-Augmentation` uses DFS to extract vertex-disjoint paths. DFS is undesirable in the streaming model as it would necessitate too many passes of the data in the streaming model. Furthermore, it will take too many rounds in the MPC model. Using the layer graph from Idea 1, McGregor [McG05, See `Find-Layer-Paths`] extracts vertex-disjoint paths by simulating DFS in a layered fashion.

In the forward pass, we compute and propose a $O(1)$-approximation of maximum matching between each pairs of layers. In the backward pass, approved matchings are *fixed*[11] and we try to find another maximal matching (ignoring fixed matches in that layer) via another forward pass.

Forward pass attempts are stopped if the latest matching size found is smaller than a *threshold* fraction of the number of nodes in the next layer. The threshold is a small constant chosen based on ϵ and the approximation guarantee of the algorithm. A high threshold would result in insufficient number of extracted paths (because we stop too prematurely) while a low threshold is needed to ensure good runtime.

Remark McGregor [McG05] computes maximal matchings between layers but one can show that it suffices to use constant approximations.

[11] `Find-Layer-Paths` assigns tags to nodes in G').

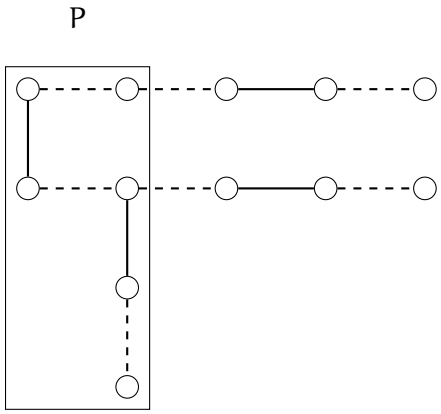
Example

Fig. 2.6 shows a graph with 26 vertices with a matching M. For $i = 2$, Fig. 2.7 shows a possible layer graph projection (Idea 1) and Fig. 2.8 highlights a possible approximate layer matching (Idea 2). See figure captions for explanation.

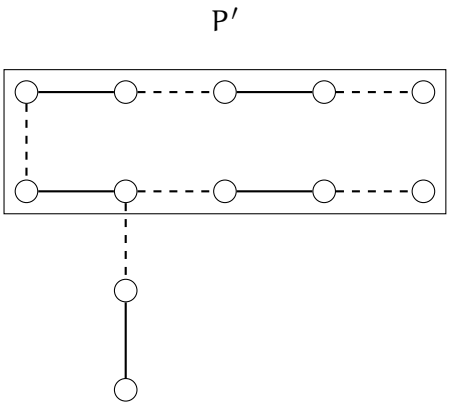
Adapting to MPC

Projection to the layer graph is done independently on the edges so it can be determined in one round of local randomness and communicated using broadcast trees.

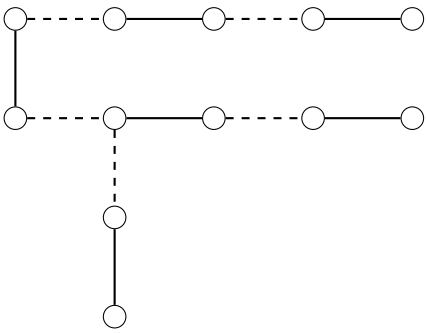
At each step of the layered DFS, we run the approximate matching algorithm described in earlier sections depending on the memory regime.



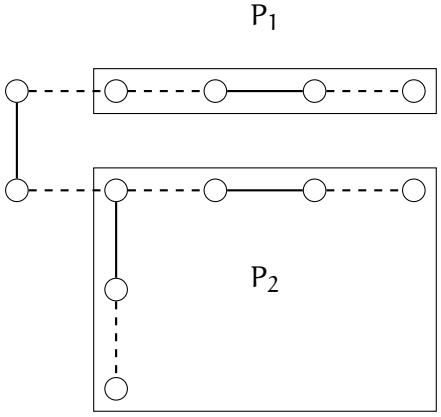
(a) Matching M with a shortest augmenting path P



(b) Matching $M \oplus P$ with an augmenting path P'

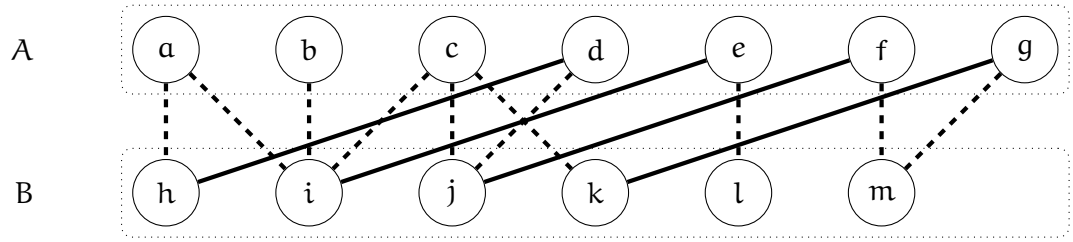


(c) Matching $N = M \oplus P \oplus P'$

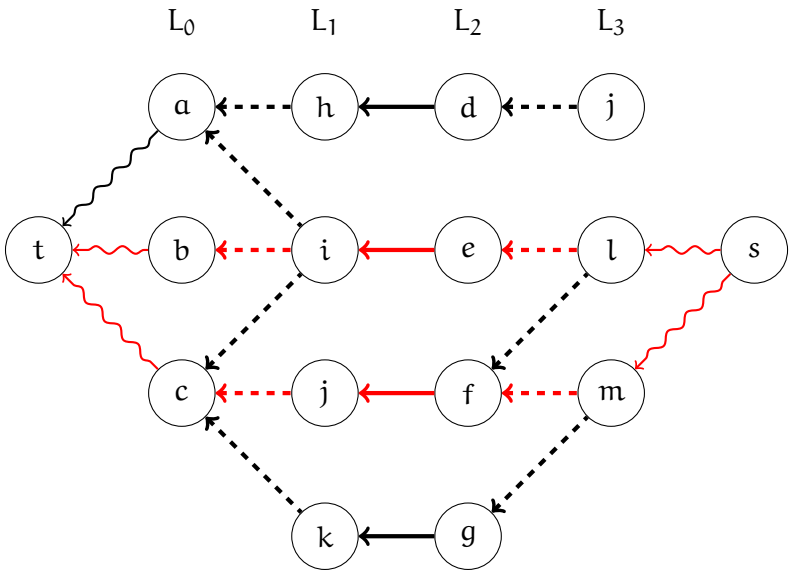


(d) $N \oplus M = P \oplus P'$ has two vertex-disjoint paths P_1 and P_2

Figure 2.4: Graph G with 12 vertices. All lines (both solid and dashed) are the edges of G. Solid lines represent matchings in (a), (b) and (c).



(a) Matching M with free vertices $\{a, b, c, l, m\}$. Vertices $\{a, b, c, d, e, f, g\}$ are in partition A and vertices $\{h, i, j, k, l, m\}$ are in partition B.



(b) Directed graph G' constructed using BFS from $\{a, b, c\}$ and ending at L_3 when $\{l, m\}$ are reached. Two augmenting paths of length 3 are highlighted in red.

Figure 2.5: A bipartite graph G with 13 vertices. All lines (both solid and dashed) are the edges of G. Solid lines represent edges in matching M.

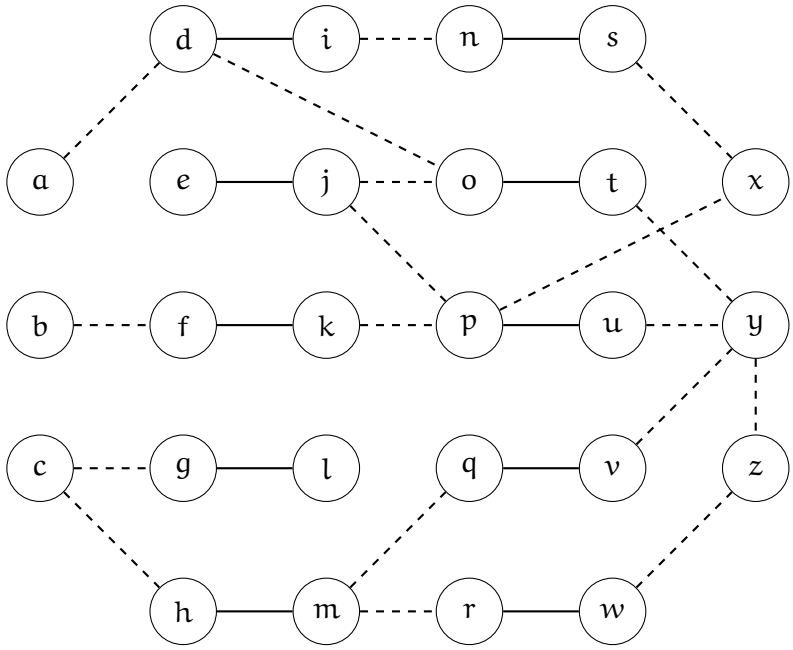


Figure 2.6: Graph G with 26 vertices. Solid lines are edges in M. The only free vertices are $\{a, b, c, x, y, z\}$, all other vertices are matched. There are three augmenting paths "adinsx", "bfkpuy" and "chmrwz" of length 5.

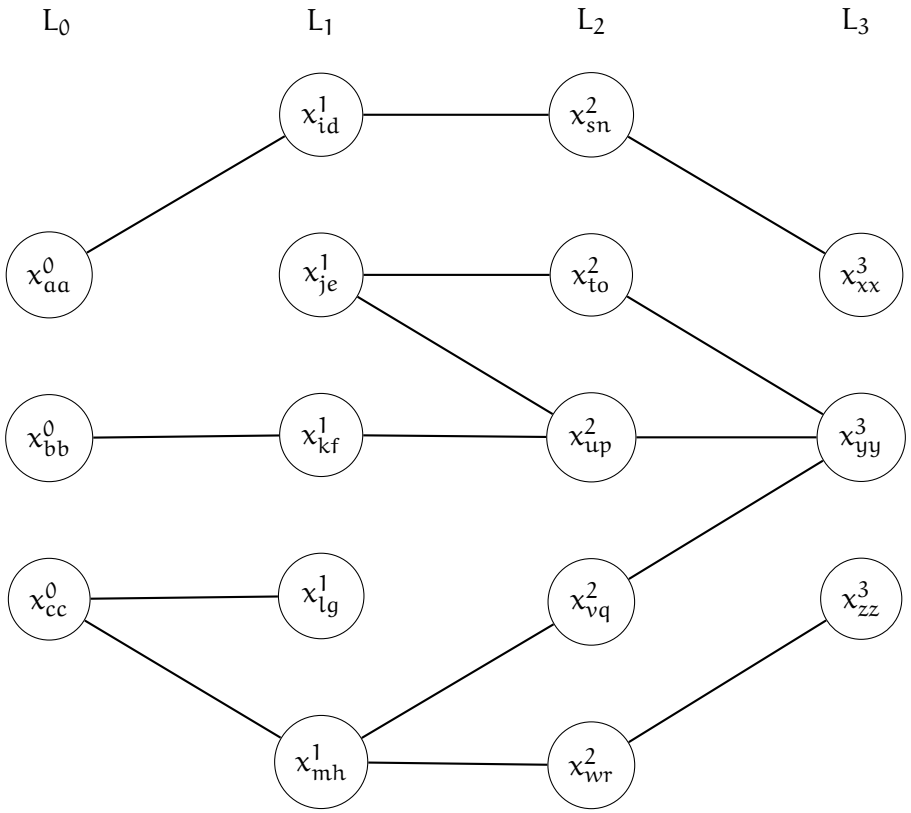


Figure 2.7: A possible layer graph G' of G Fig. 2.6 with $i = 2$. Free vertices are randomly assigned to layers L_0 or L_3. Matched edges are randomly assigned to internal layers L_1 or L_2 with a random subscript ordering of endpoints. Edges in G' correspond to edges in $E \setminus M$ in G. An edge is in G' only if an "applicable" edge is in G. For example, $\{x^0_{aa}, x^1_{id}\} \in G'$ because $\{a, d\} \in G$. Edges $\{d, o\}$ and $\{p, x\}$ are not represented in G' because the subscript orderings. Edge $\{y, z\}$ is not represented in G' because it is not adjacent to a matched edge in G.

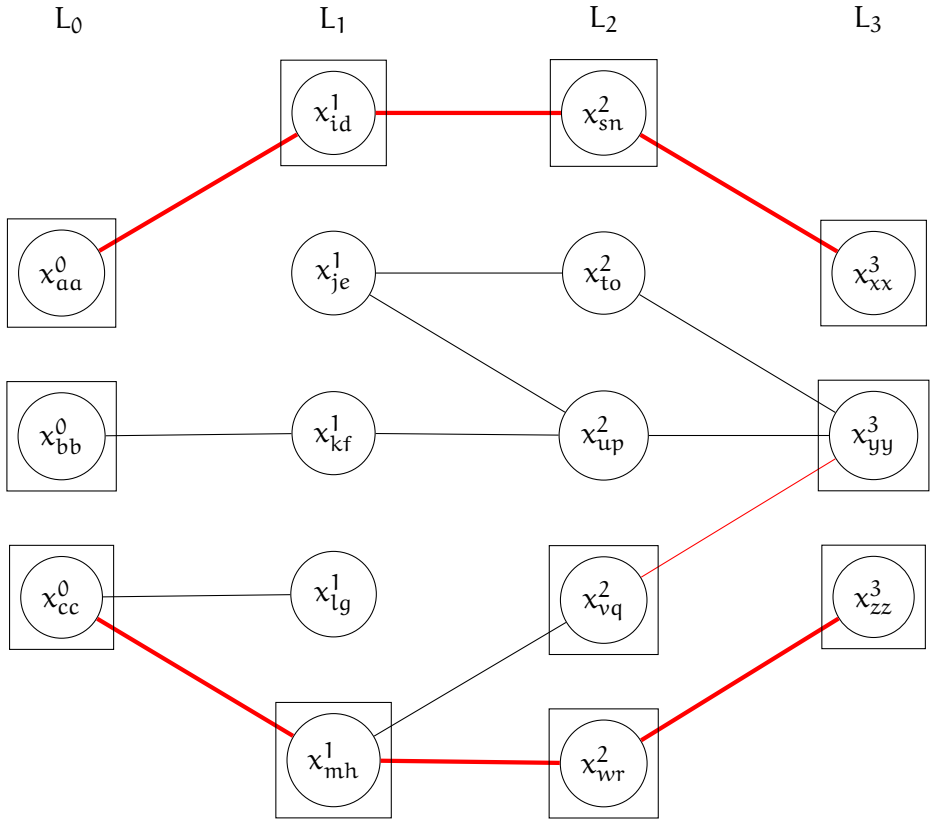


Figure 2.8: First pass from L_3 to L_0. Maximal matching between each pair of layers are in red, involved vertices are boxed up, and "accepted" matchings are bolded. For example, while finding the maximal matching between layers L_1 and L_2, only x^1_{id}, x^1_{mh}, x^2_{sn}, x^2_{vq} and x^2_{wr} are involved and the edges $\{x^1_{id}, x^2_{sn}\}$ and $\{x^1_{mh}, x^2_{wr}\}$ were selected. Since x^2_{vq} was unable to be matched between layers L_1 and L_2, it is marked "Dead". We backtrack and try to match towards L_0 with the remaining "unaccepted" vertices until the size of the subsequent matching found between layers L_i and L_{i-1} is smaller $\delta \cdot |L_{i-1}|$ for a threshold $\delta \in \text{poly}(\epsilon)$. In McGregor's notation [McG05], $S = \{x^2_{sn}, x^2_{vq}, x^2_{wr}\}$ and $S' = \{x^1_{id}, x^1_{mh}\}$ in the recursive call of `Find-Layer-Paths`$(G', S, \delta, 2)$.

Chapter 3

Connected Components and Minimum Spanning Tree

In this chapter, we look at the problem of computing connected components and minimum spanning trees. It is relatively easy to find solve them in strongly superlinear and strongly sublinear memory regimes. In the first two sections of the chapter, we focus our efforts on solving MST in near linear memory regime. Next, we discuss a connectivity algorithm due to Andoni et al. [ASS+18] that yields a round complexity depending on the diameter of the graph, under the strongly sublinear memory regime. Finally, we discuss a constant round algorithm due to Andoni et al. [ANOY14] to compute geometric minimum spanning trees under the strongly sublinear memory regime.

Problem definitions Given an undirected graph $G = (V, E)$, a connected component is a subgraph in which any two vertices are connected by a path. There are two typical ways to represent connected components on a graph:

1. Every vertex v stores a label $l(v)$ such that $l(u) = l(v)$ if and only if vertices u and v are in the same connected component.

2. Explicitly build a tree for each connected component, yielding a maximal forest of the graph.

A related graph problem is computing the minimum spanning tree (MST) on connected weighted graphs. The goal of MST is to find a subset T of edges such that the sum of edge weights in T are minimized. In general, where the graph may not be connected, the MST problem is also

called the minimum spanning forest (MSF) problem. On graphs with identical edge weights, we see that computing a minimum spanning forest is equivalent to computing connected components.

The three memory regimes One can compute MST in $O(1)$ rounds under the strongly superlinear memory regime using the *filtering* technique of Lattanzi et al. [LMSV11]. As the idea is similar to Section 2.1, we leave the details as an exercise.

Exercise 3.1. MST with strongly superlinear memory in $O(1)$ rounds *Devise an algorithm that computes the minimum spanning tree of a given graph* $G = (V, E)$ *with* $|V| = n$ *in* $O(\frac{1}{\epsilon})$ *rounds using* $O(n^{1+\epsilon})$ *memory per machine, for some constant* $\epsilon > 0$.

On the other extreme, MST can be computed in $O(\log n)$ rounds with strongly sublinear memory by carefully simulating Borůvka[1]. Note that a long chain of proposed edges may occur while simulating Borůvka, where merging them could take many rounds. This issue can be resolved by using randomness to drop a constant fraction of proposed edges, in a manner that remaining edges do not form long chains. We leave the details as an exercise.

Exercise 3.2. MST with strongly sublinear memory in $O(\log n)$ rounds *Devise an algorithm that computes the minimum spanning tree of a given graph* $G = (V, E)$ *with* $|V| = n$ *in* $O(\frac{1}{\epsilon})$ *rounds using* $O(n^{\alpha})$ *memory per machine, for some constant* $\alpha \in (0, 1)$. *You may assume that edge weights are unique.*

In Section 3.1, we see an algorithm for computing minimum spanning trees in $O(1)$ rounds in the near linear memory regime, where each machine has $S = \widetilde{O}(n)$ memory. It assumes the existence of an algorithm `connected-components` which computes connected components in $O(1)$ rounds in the near linear memory regime. In Section 3.2, we construct such an algorithm `connected-components` using *graph sketching*.

Known lower bounds It remains a major open question whether there exist connectivity MPC algorithms with sub-logarithmic time. There are strong indications that such algorithms do not exist: Beame et al. [BKS13] show logarithmic lower bounds for restricted algorithms, and no known algorithm can distinguish a n-node cycle from two $\frac{n}{2}$-node cycles

[1] See https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

in $o(\log n)$ rounds. Furthermore, Roughgarden et al. [RVW18, Theorem 6.1] showed that an unconditional lower bound would imply stronger circuit lower bounds.

Recently, there has been some work using parameters of the input graph such as diameter D of the graph: Andoni et al. [ASS+18] gave an algorithm in the strongly sublinear memory regime that uses $\Theta(m)$ total memory to compute connected components in $\mathcal{O}(\log D \cdot \log\log_{\frac{m}{n}} n)$ rounds.

3.1 MST using near linear memory

Recall from Exercise 2.1 that sorting can be done in $\mathcal{O}(1)$ rounds with strongly sublinear memory. Hence, we can sort edges in ascending weight ordering in $\mathcal{O}(1)$ rounds and label the edges $e_1, e_2, \ldots, e_m$ such that $w(e_1) \leqslant w(e_2) \leqslant \cdots \leqslant w(e_m)$. In the sequential setting, Kruskal's algorithm[2] iterates through such a sorted edge list to decide whether an edge is in the MST. This gives the following observation, which we shall exploit.

Observation Edge e_i is in the MST if and only if its endpoints are not in the same connected components in the graph with edge set $\{e_1, \ldots, e_{i-1}\}$.

Using the observation directly to determine whether all edges are in the MST would require $\mathcal{O}(m)$ calls to `connected-components`. To speed things up in MPC, we group edges into chunks of n edges. In the near linear memory regime, each chunk can fit into a single machine and all chunks can be processed simultaneously. For $i = \{1, \ldots, \frac{m}{n}\}$, denote

- $E_i = \{e_{(i-1)n+1}, \ldots, e_{in}\}$ as the i^{th} chunk of n edges
- $E'_i = \cup_{j=1}^{i} E_j$ as the union of edge sets E_1 to E_i
- F'_i as the maximal forest computed from each set of edges E'_i

Denote F'_0 as the graph without edges (i.e. all vertices are isolated). By the above observation, we know that any edge in $\{u, v\} \in E_i$ is in the MST only if the components of u and v in F'_{i-1} differ. Since both $|E_i| \in \mathcal{O}(n)$ and $|F'_{i-1}| \in \mathcal{O}(n)$, a single machine can simultaneously hold E_i and F'_{i-1}.

Assuming that there exists an algorithm `connected-components` which computes connected components in $\mathcal{O}(1)$ rounds in near linear memory

[2] See https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

regime, then all maximal forests F'_i can be computed in $O(1)$ rounds in parallel. Then, by putting E_i and F'_{i-1} into machine i, all MST edges can be determined in a one further MPC round. In Section 3.2, we construct such an algorithm `connected-components` using *graph sketching*.

3.2 Connectivity using near linear memory

We first describe the technique of *graph sketching* then explain how to use it in MPC to compute connected components in $O(1)$ rounds with near linear memory.

3.2.1 Graph sketching

Graph sketching is a technique first developed in the context of streaming algorithms. In streaming problems, updates appear one-by-one, and one has maintain/compute certain properties or data structures under limited memory. That is, one cannot store entire stream and compute offline. The length of the stream may also be unknown. For the connected components problem, edges can be added or deleted in the stream. Below, we show a randomized algorithm for computing a maximal forest with high success probability. For the streaming setting, it uses a data structure with $O(n \log^4 n)$ memory. In the near linear memory regime, this will fit into a single machine.

Coordinator model For a change in perspective[3], consider the following computation model where each vertex acts independently from each other. Then, upon request of connected components, each vertex sends some information to a centralized coordinator to perform computation and outputs the maximal forest.

The coordinator model will be helpful in our analysis of the algorithm later as each vertex will send $O(\log^4 n)$ amount of data (a local sketch of the graph) to the coordinator, totalling $O(n \log^4 n)$ memory as required. This conceptual model also suggests how to perform graph sketch in MPC.

Two warm ups Before we give the full construction, we first look at two warm up problems. Fix a subset $A \subseteq V$ and look at the cut C between A

[3]In reality, the algorithm simulates all the vertices' actions so it is not a real multi-party computation setup.

and $V \setminus A$. In the first warm up, we assume there is only one cut edge across C, and wish to find it using $\mathcal{O}(\log n)$ bits of memory. Building upon the previous warm up, the second warm up wishes to find *any* cut edge across the cut C when there are $k > 1$ cut edges.

Warm up 1: Finding the single cut edge

Definition 3.1 (The single cut problem)**.** *Fix an arbitrary subset $A \subseteq V$. Suppose there is exactly 1 cut edge $\{u, v\}$ between A and $V \setminus A$. How do we output the cut edge $\{u, v\}$ using $\mathcal{O}(\log n)$ bits of memory?*

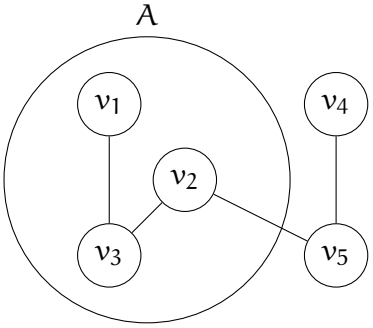
Without loss of generality, assume $u \in A$ and $v \in V \setminus A$. Note that this is not a trivial problem on first glance since it already takes $\mathcal{O}(n)$ bits for any vertex to enumerate all adjacent edges. To solve the problem, we use a *bit trick* which exploits the fact that any edge $\{a, b\} \in A$ will be considered twice by vertices in A. Since one can uniquely identify each vertex with $\mathcal{O}(\log n)$ bits, consider the following:

- Identify an edge by the concatenation the identifiers of its endpoints. Say, $\mathrm{id}(\{u, v\}) = u \circ v$ if $\mathrm{id}(u) < \mathrm{id}(v)$.
- Locally, each vertex u computes $\mathrm{XOR}_u = \oplus_{e \in E, e \ni u} \mathrm{id}(e)$.
- Vertices send the coordinator their XOR sum.
- Coordinator computes $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$.

Example Suppose $V = \{v_1, v_2, v_3, v_4, v_5\}$ where $\mathrm{id}(v_1) = 000$, $\mathrm{id}(v_2) = 001$, $\mathrm{id}(v_3) = 010$, $\mathrm{id}(v_4) = 011$, and $\mathrm{id}(v_5) = 100$. Then,

$$\mathrm{id}(\{v_1, v_3\}) = \mathrm{id}(v_1) \circ \mathrm{id}(v_3) = 000010$$

and so on. Suppose we query for the cut edge $\{v_2, v_5\}$ with $A = \{v_1, v_2, v_3\}$.



Then,

$$\begin{aligned}
\mathrm{XOR}_1 &= 000010 = \mathrm{id}(\{v_1, v_3\}) && = 000010\\
\mathrm{XOR}_2 &= 000110 = \mathrm{id}(\{v_2, v_3\}) \oplus \mathrm{id}(\{v_2, v_5\}) && = 001010 \oplus 001100\\
\mathrm{XOR}_3 &= 001000 = \mathrm{id}(\{v_2, v_3\}) \oplus \mathrm{id}(\{v_1, v_3\}) && = 001010 \oplus 000010\\
\mathrm{XOR}_4 &= 011100 = \mathrm{id}(\{v_4, v_5\}) && = 011100\\
\mathrm{XOR}_5 &= 010000 = \mathrm{id}(\{v_4, v_5\}) \oplus \mathrm{id}(\{v_2, v_5\}) && = 011100 \oplus 001100
\end{aligned}$$

Thus, $\mathrm{XOR}_A = \mathrm{XOR}_1 \oplus \mathrm{XOR}_2 \oplus \mathrm{XOR}_3 = 000010 \oplus 000110 \oplus 001000 = 001100 = \mathrm{id}(\{v_2, v_5\})$ as expected. Notice that every edge in A contributes an even number of times to the coordinator's XOR sum.

Claim 3.2. $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$ *is the identifier of the cut edge.*

Proof. For any edge $\{a, b\}$ such that $a, b \in A$, $\mathrm{id}(\{a, b\})$ is in both XOR_a and XOR_b. So, $\mathrm{XOR}_a \oplus \mathrm{XOR}_b$ will cancel out the contribution of $\mathrm{id}(\{a, b\})$. Hence, the only remaining value in $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$ will be the cut edge since only one endpoint lies in A. □

Remark Similar ideas are often used in the random linear network coding literature (e.g. Ho et al. [HMK+06]).

Warm up 2: Finding one out of $k > 1$ cut edges

Definition 3.3 (The k-cut problem)**.** *Fix an arbitrary subset $A \subseteq V$. Suppose there are exactly k cut edges (u, v) between A and $V \setminus A$, and we are given an estimate $\widehat{k}$ such that $\frac{\widehat{k}}{2} \leqslant k \leqslant \widehat{k}$. How do we output a cut edge $\{u, v\}$ using $\mathcal{O}(\log n)$ bits of memory, with high probability?*

A straight-forward idea is to independently mark each edge, each with probability $1/\widehat{k}$. In expectation, we expect one edge to be marked. Since edges are marked independently with probability $1/\widehat{k}$, the probability that a fixed cut edge $\{u, v\}$ is marked while no other edges are marked is $(1/\widehat{k})(1 - (1/\widehat{k}))^{k-1}$. Denote the marked cut edges by E', then

$$\begin{aligned}
\Pr[|E'| = 1] &= k \cdot (1/\widehat{k})(1 - (1/\widehat{k}))^{k-1} && \text{There are } k \text{ cut edges}\\
&\geqslant (\widehat{k}/2)(1/\widehat{k})(1 - (1/\widehat{k}))^{\widehat{k}} && \text{Since } \frac{\widehat{k}}{2} \leqslant k \leqslant \widehat{k}\\
&\geqslant (1/2) \cdot 4^{-1} && \text{Since } 1 - x \geqslant 4^{-x} \text{ for } x \leqslant 1/2\\
&\geqslant \frac{1}{10}
\end{aligned}$$

Remark The above analysis assumes that vertices can locally mark the edges in a consistent manner (i.e. both endpoints of any edge make the same decision whether to mark the edge or not). This can be done with a sufficiently large string of *shared randomness*. We discuss this later.

From above, we know that $\Pr[|E'| = 1] \geqslant 1/10$. If $|E'| = 1$, we can re-use the idea from the case when $k = 1$. However, if $|E'| \neq 1$, then XOR_A may correspond erroneously to another edge in the graph. In the above example, $\text{id}(\{v_1, v_2\}) \oplus \text{id}(\{v_2, v_4\}) = 000001 \oplus 001011 = 001010 = \text{id}(\{v_2, v_3\})$.

To fix this, we use *random bits as edge IDs* instead of simply concatenating vertex IDs: Randomly assign (in a consistent manner) each edge with a random ID of $k = 20 \log n$ bits. Since the XOR of random bits is random, for any edge e, $\Pr[\text{XOR}_A = \text{id}(e) \mid |E'| \neq 1] = (\frac{1}{2})^k = (\frac{1}{2})^{20 \log n}$. Hence,

$$
\begin{aligned}
&\Pr[\text{XOR}_A = \text{id}(e) \text{ for some edge } e \mid |E'| \neq 1] \\
&\leqslant \sum_{e \in \binom{V}{2}} \Pr[\text{XOR}_A = \text{id}(e) \mid |E'| \neq 1] && \text{Union bound over all edges} \\
&= \binom{n}{2} \cdot (\frac{1}{2})^{20 \log n} && \text{There are } \binom{n}{2} \text{ possible edges} \\
&= 2^{-18 \log n} && \text{Since } \binom{n}{2} \leqslant n^2 = 2^{2 \log n} \\
&= \frac{1}{n^{18}} && \text{Rewriting}
\end{aligned}
$$

Thus, with high probability, we can correctly distinguish $|E'| = 1$ from $|E'| \neq 1$. Furthermore, $\Pr[|E'| = 1] \geqslant \frac{1}{10}$. For any given $\epsilon > 0$, there exists a constant $C(\epsilon)$ such that if we repeat $t = C(\epsilon) \log n$ times, the probability that *all* t tries fail to extract a single cut is $(1 - \frac{1}{10})^t \leqslant \frac{1}{\text{poly}(n)}$.

Maximal forest with $\mathcal{O}(n \log^4 n)$ memory

Recall that Borůvka's algorithm builds a minimum spanning tree by iteratively finding the cheapest edge leaving connected components and adding them into the MST. The number of connected components decreases by at least half per iteration, so it converges in $\mathcal{O}(\log n)$ iterations.

Any arbitrary cut has cut size $k \in [0, n]$. Using $\mathcal{O}(\log n)$ guesses for $\widehat{k} = 2^0, 2^1, \ldots, 2^{\lceil \log n \rceil}$, we can use the approach to the k-cut problem to find a single cut edge:

- If $\widehat{k} \ll k$, the marking probability will select nothing (in expectation).
- If $\widehat{k} \approx k$, then we expect to find a valid edge ID.

- If $\widehat{k} \gg k$, more than one edge will get marked, which we will then detect (and ignore) since XOR_A will likely not be a valid edge ID.

Using shared randomness, vertices compute $\mathcal{O}(\log n)$ sketches. Each sketch has size $\mathcal{O}(\log^3 n)$ and is computed using random (but consistent) edge IDs and marking probabilities:

- For each vertex v, XOR_v takes $\mathcal{O}(\log n)$ bits to represent
- Make $\mathcal{O}(\log n)$ guesses of cut size k
- Repeat $\mathcal{O}(\log n)$ times independently to amplify success probability

One round of Borůvka can be simulated by using a unused sketch:

- Find an out-going edge from each connected component using the approach to the k-cut problem
- Join connected components by adding edges to graph

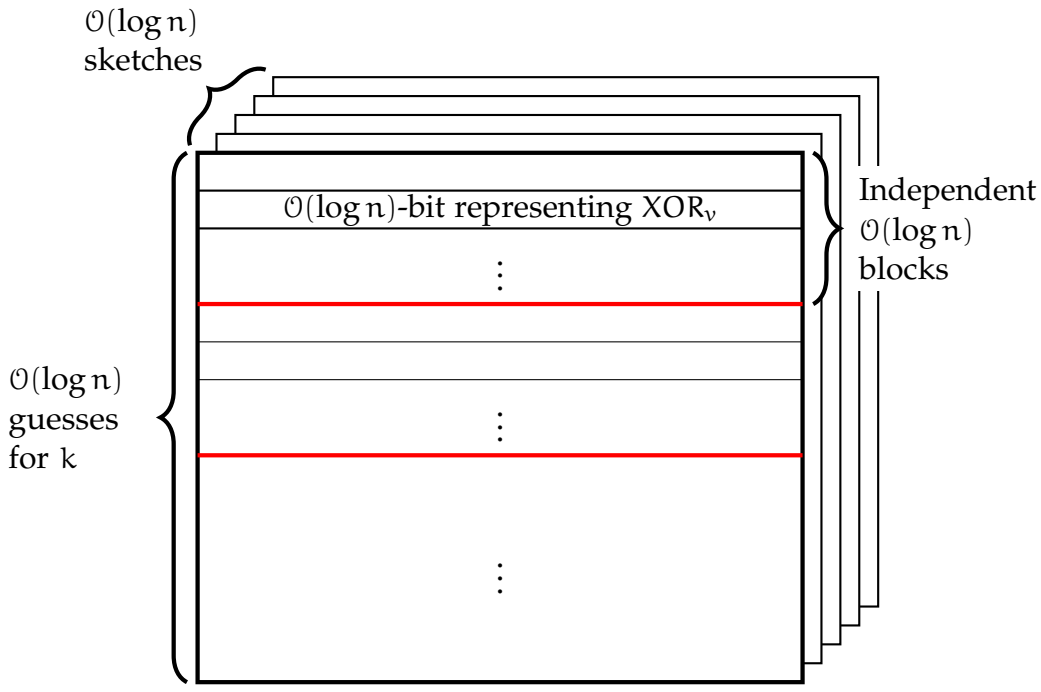


Figure 3.1: A pictorial representation of the $\mathcal{O}(\log n)$ sketches for vertex v that is sent to the coordinator. A sketch takes $\mathcal{O}(\log^3 n)$ bits. Each row is a XOR of adjacent edge IDs, subject to edge sampling probability $\widehat{k}$.

Each sketch uses $\mathcal{O}(\log^3 n)$ memory and $\mathcal{O}(\log n)$ sketches can simulate Borůvka, so a total of $\mathcal{O}(n \log^4 n)$ memory suffices. With $t = C(\epsilon) \log n$ tries at each step, we fail to find one cut edge leaving a connected component with probability $\leqslant (1 - \frac{1}{10})^t$. For a sufficiently large constant C, an application of union bound tells us that we succeed with high probability.

Remark One can drop the memory requirement per vertex from $\mathcal{O}(\log^4 n)$ to $\mathcal{O}(\log^3 n)$ by using a constant t instead of $t \in \mathcal{O}(\log n)$ such that the success probability is a constant larger than $1/2$. Then, simulate Borůvka for $\lceil 2 \log n \rceil$ steps. See Ahn, Guha and McGregor [AGM12] for details. Note that they use a slightly different sketch.

Theorem 3.4. *Any randomized distributed sketching protocol for computing spanning forest with success probability $\epsilon > 0$ must have expected average sketch size $\Omega(\log^3 n)$, for any constant $\epsilon > 0$.*

Proof. See Nelson and Yu [NY18] for details. □

Constructing random edge IDs using ϵ-bias sample spaces

We can use ϵ-bias sample spaces to obtain the random edge IDs discussed earlier. Since each bit of ID's operates independently, we focus on constructing 1-bit ID's first. Let $N = \binom{n}{2}$.

Claim 3.5. *There exists a collection B of functions $b : [N] \to \{0, 1\}$ such that:*

1. *For any $E' \subseteq [N]$ such that $|E'| \geqslant 1$,*

$$\Pr_{b \in B}[(\bigoplus_{e \in E'} b(e)) = 0] \in [\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon]$$

2. $|B| = \mathcal{O}(\frac{N}{\epsilon^2})$

Proof. Fix any arbitrary subset E' of $[N]$. Observe that for $b \in_{u.a.r.} \{0,1\}^N$, we have $\Pr[\bigoplus_{e \in E'} b(e) = 0] = \frac{1}{2}$. Fix $k = \frac{cN}{\epsilon^2}$ for some large constant c. Pick B of size k uniformly from $\{0,1\}^N$. In expectation, half of the $b \in B$ satisfy the condition $\bigoplus_{e \in E'} b(e) = 0$. By Chernoff bound, the probability that number of such b is *not* in $[(\frac{1}{2} - \epsilon)k, (\frac{1}{2} + \epsilon)k]$ is at most $2e^{-\frac{\epsilon^2 k}{2}} \ll 2^{-N}$. By taking union bound over all 2^N choices of E', the probabilistic method tells us that such a collection exists. □

To store edge IDs, it suffices to store which functions from B are used. Suppose we use IDs of length $l = C \cdot \log n$ for some constant C. Then, only $l \cdot \log |B| \in \mathcal{O}(\log^2 n)$ bits of information need to be stored. Additionally, since each bit of ID is selected independently, the probability of a collision from XOR's is upper bounded by $(\frac{1}{2} + \epsilon)^l$. By selecting C to be large enough constant, the collision probabilities can be made to be exponentially small. Fast constructions of such B's exist but are out of the scope of the course.

3.2.2 Simulating in MPC

In the near linear memory regime, each machine has $S = \widetilde{\mathcal{O}}(n)$ memory. To compute a sketch for a vertex, we first need to put all edges incident to a vertex onto a single machine. This can be done in $\mathcal{O}(1)$ rounds (See Exercise 3.3). After distribution edges, sketches can be computed independently in a single MPC round. Since each vertex produces sketches of size $\mathcal{O}(\log^4 n)$, the sketches for all n vertices fit into the memory of a single machine. Thus, the connected component can be successfully computed in one further MPC round, with high probability. As we can see, this entire process takes $\mathcal{O}(1)$ rounds in near linear memory regime.

Recall in Section 3.1 that we use connected components as a subroutine to solve MST. To be precise, we solve $\frac{m}{n}$ copies of connected component problem in parallel, in a single MPC round. This is doable because the required sketches fit into a single machine and we have $\mathcal{O}(\frac{m}{n})$ machines.

Exercises

Exercise 3.3. Edge distribution

Given an arbitrary initial distribution of edges, devise an algorithm that puts edges incident to the same node into a single machine in $\mathcal{O}(1)$ rounds with $\widetilde{\mathcal{O}}(n)$ memory per machine.

For example, suppose $E = \{\{a, b\}, \{a, c\}, \{b, d\}\}$. We wish to put $\{a, b\}$ and $\{a, c\}$ into the same machine so that all endpoints of a are in the same machine. Similarly, we wish to put $\{a, b\}$ and $\{b, d\}$ onto the same machine, possibly in a different machine as before. Note that the edge $\{a, b\}$ is duplicated and stored twice, once for a and once for b.

Hint *Sort the edges.*

Exercise 3.4. Approximate minimum cut in near linear memory
Given a graph $G = (V, E)$*, denote* λ *as the minimum cut size of* G*.*

1. *Assume that the minimum cut size* $\lambda \leqslant \log n$*. Devise an algorithm that computes a cut — i.e., identifies the vertices on the two sides of the cut — of size* λ *in* $\mathcal{O}(1)$ *rounds with* $S = \widetilde{\mathcal{O}}(n)$ *memory per machine.*

2. *Devise an algorithm that computes a cut with size* λ' *such that* $\lambda' \in (1 \pm \epsilon)\lambda$ *in* $\mathcal{O}(1)$ *rounds with* $S = \widetilde{\mathcal{O}}(n)$ *memory per machine.*

Hint *Use multiple sketches to find maximal spanning forests.*

Exercise 3.5. $(1+\epsilon)\cdot\Delta$ edge-coloring with near linear memory in constant rounds
Given a graph $G = (V, E)$*, an edge-coloring with* k *colors is a function* $c : E \to \{1, \ldots, k\}$ *such that adjacent edges do not share the same color. That is, for* $e_i, e_j \in E$ *sharing an endpoint, we must have* $c(e_i) \neq c(e_j)$*. By Vizing's theorem, we know that any graph can be edge colored by* $\Delta + 1$ *colors, where* Δ *is the maximum degree of the graph. Devise an algorithm that computes a* $(1+\epsilon)\cdot\Delta$ *edge-coloring with* $S = \widetilde{\mathcal{O}}(n)$ *memory per machine in* $\mathcal{O}(1)$ *rounds.*

Hint *A single round suffices.*

3.3 Log diameter time connectivity using sublinear memory

As MST solves the problem of connectivity, we know that $\mathcal{O}(\log n)$ rounds suffice under the strongly sublinear memory regime (See Exercise 3.2). In this section, we discuss a connectivity algorithm due to Andoni et al. [ASS+18] whose round complexity is a function of the diameter of the input graph G. Their algorithm uses $\Theta(m)$ total memory to compute connected components in $\mathcal{O}(\log D \cdot \log\log_{\frac{m}{n}} n)$ rounds under the strongly sublinear memory regime, where each machine has memory $S = n^{\alpha}$ for some constant $\alpha \in (0, 1)$.

For ease of exposition, we first discuss the ideas assuming each vertex has access to memory of size n^{α}. Note that this violates the global memory constraint of $\Theta(m)$, which we resolve later. As we shall see, connected components will be treated in a disjoint fashion in the algorithm. So, it is easier to understand and analyze the behavior of the algorithm by

mentally assuming that the input graph is a single connected component with n distinct labels, and we wish to reduce the number of labels to 1, i.e. realize that the graph is connected.

To solve connectivity, Andoni et al. [ASS+18] use *graph exponentiation* and *label contraction*[4]. We discuss these techniques in Section 3.3.1 and Section 3.3.2 respectively. In Section 3.3.3, we show how to combine the two to yield an algorithm that runs in $O(\log D)$ MPC rounds. To resolve the concern of global memory, Andoni et al. [ASS+18] use a technique they call "double exponential speed problem size reduction" (See Section 3.3.4). This introduces an $O(\log\log_{\frac{m}{n}} n)$ factor overhead in the MPC runtime, hence the total algorithm runs in $O(\log D \cdot \log\log_{\frac{m}{n}} n)$ rounds under the strongly sublinear memory regime.

3.3.1 Handling sparse graphs: Graph exponentiation

Recall the technique of graph exponentiation from Section 2.3.2 [LW10, Gha17]. Consider a graph which consists of $\frac{n}{D}$ paths of length D where $D \leqslant n^{\alpha}$. Within $O(\log D)$ MPC rounds, every vertex can gather the entire neighborhood using graph exponentiation, then compute connectivity of the collected subgraph in a single MPC round. Note that this will require a global memory of $O(nD)$.

If the D-hop neighborhood of a vertex fits into memory, then one can gather the entire neighborhood in $O(\log D)$ MPC rounds. However, for general graphs, one cannot hope to collect the entire D-hop neighborhood of a vertex into a single machine.

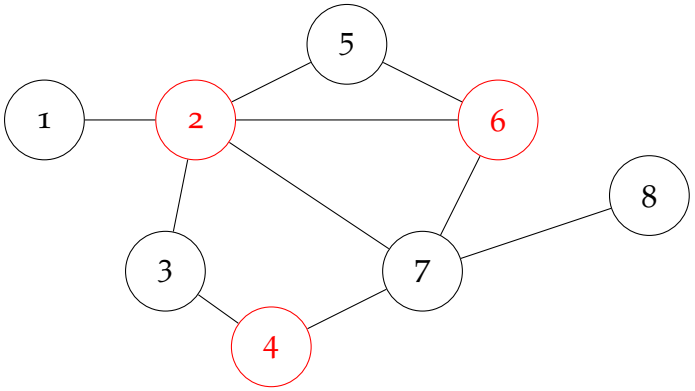
3.3.2 Handling dense graphs: Label contraction

Consider a connected graph with unique label IDs assigned to each vertex initially. The basic idea of label contraction (See Fig. 3.2) is as follows:

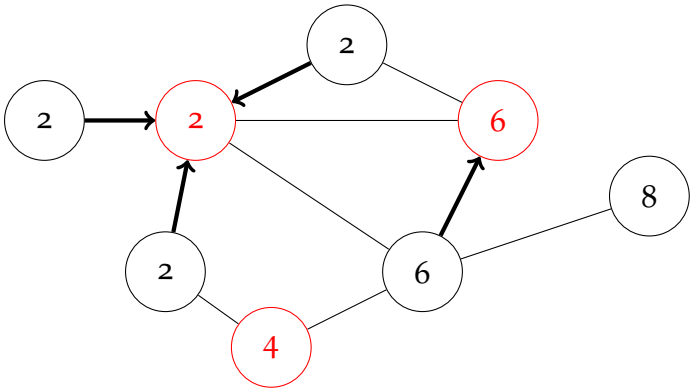
- Mark each vertex independently with probability $p = \frac{1}{2}$.
- For each unmarked vertex with a marked neighbor, relabel itself to *one of* the marked neighbor's label

By treating vertices with the same label as a supernode, we see that label contraction essentially contracts vertices in each round via relabelling.

[4] In the paper, graph exponentiation is called "neighbor increment operation" while label contraction is called "random leader selection".



(a) Vertices marked independently. Marked vertices are in red.



(b) Marked vertices are in red. Unmarked vertices with adjacent marked neighbors relabel themselves. Arrow indicates choice of marked neighbor.

Figure 3.2: The label contraction process with numbers as vertex labels.

Claim 3.6. *In expectation, at most $\frac{3}{4}$ fraction of existing labels remain after an iteration of label contraction.*

Proof. After each iteration of label contraction, the only remaining labels are those of marked vertices, and unmarked vertices without marked neighbors. By construction, each vertex is marked independently with probability $p = \frac{1}{2}$. For an unmarked vertex v, the probability that it does not have a marked neighbor is $(\frac{1}{2})^{\deg(v)} \leqslant \frac{1}{2}$. So, the probability that a

vertex is unmarked *and* has no marked neighbor is at most $\frac{1}{4}$. Thus,

$$\begin{aligned}
&\Pr[\text{Vertex } v\text{'s label remains}] \\
&= \Pr[v \text{ is marked, or } v \text{ is unmarked without a marked neighbor}] \\
&\leqslant \Pr[v \text{ is marked}] + \Pr[v \text{ is unmarked without a marked neighbor}] \\
&\leqslant \frac{1}{2} + \frac{1}{4} = \frac{3}{4}
\end{aligned}$$

Summing over all vertices, we have the result. □

By Claim 3.6, with high probability, label contraction runs for at most $O(\log n)$ rounds before all endpoints of any given edge share the same label. Observe that if we knew all vertices have high degree, we could mark vertices with a lower marking probability p, and hence gain a more significant reduction in number of labels per iteration of label contraction.

3.3.3 Putting them together

In general graphs, one cannot hope to collect the entire D-hop neighborhood in a single machine. As such, in the "neighbor increment operation" of Andoni et al. [ASS+18], they stop the graph exponentiation process when the neighborhood is sufficiently large. On the other hand, label contraction works well when vertices have high degrees. Consider the following scheme which combines graph exponentiation and label contraction in an iterative manner:

- In a phase, do the following:
  1. From each node, perform graph exponentiation until the number of edges leaving the collected set of vertices exceeds $n^{\alpha/2}$. Collapse the collected vertices into a supernode.
  2. Perform label contraction by marking vertices independently with probability $p = \frac{\log n}{n^{\alpha/2}}$.

Observe that vertices from different connected components do not affect each other in the algorithm above. The parameter $n^{\alpha/2}$ is defined with respect to the number of vertices in the original graph. To be precise, it was chosen to be $\sqrt{S}$, where S is the amount of memory per machine.

Claim 3.7. *In every phase, step 1 takes $O(\log D)$ rounds.*

Proof. Since we assume that the graph has diameter D, graph exponentiation runs for at most $\mathcal{O}(\log D)$ rounds before the entire graph resides in a single machine. Furthermore, diameter of a graph can only decrease when we contract vertices, hence the bound holds for every phase. □

Claim 3.8. *There are $\mathcal{O}(\frac{1}{\alpha})$ phases.*

Proof. We generalize the proof of Claim 3.6. When step 1 ends, all vertices have degrees at least $n^{\alpha/2}$. By marking vertices independently with probability $p = \frac{\log n}{n^{\alpha/2}}$, a vertex label is *not* removed with probability at most $p + (1-p) \cdot (1-p)^{n^{\alpha/2}} \leqslant p + (1-p) \cdot \frac{1}{n} \in \mathcal{O}(p)$. So, with high probability, the number of labels drop by a factor of $\mathcal{O}(n^{\alpha/2})$ after one phase. Thus, the process terminates in $\mathcal{O}(\frac{1}{\alpha})$ phases. □

Since α is a constant, by the above two claims, the algorithm solves the connectivity problem in $\mathcal{O}(\log D)$ rounds with sublinear memory. Note that we assumed a total memory of $n \cdot n^{\alpha}$ in the above description. This could be more than input size of the problem — the number of edges m.

3.3.4 Fitting into global memory

The algorithm in Section 3.3.3 assumes a total memory of $n \cdot n^{\alpha}$, which could be more than the number of edges m of the original graph. In phase i, suppose there are n_i vertices and we perform graph exponentiation as long as the degree is at most d_i. Consider a stricter termination condition that restricts $d_i = \min(n^{\alpha/2}, \sqrt{\frac{m}{n_i}})$. As there are at most d_i^2 edges amongst d_i vertices, this ensures that the total memory needed is $n_i \cdot d_i^2 \leqslant \Theta(m)$. We also modify the marking probability of label contraction appropriately to $p = \frac{\log n}{d_i}$. Under these modifications, a phase does the following:

1. From each node, perform graph exponentiation until the number of edges leaving the collected set of vertices exceeds $d_i = \min(n^{\alpha/2}, \sqrt{\frac{m}{n_i}})$. Collapse the collected vertices into a supernode.
2. Perform label contraction by marking vertices independently with probability $p = \frac{\log n}{d_i}$.

Claim 3.9. *There are $\mathcal{O}(\log \log_{\frac{m}{n}} n)$ phases.*

Proof. We generalize the proof of Claim 3.8. A vertex label remains with probability at most $\mathcal{O}(p)$. If we start phase i with $n_i \leqslant n$ labels, then there

will be $\widetilde{O}(n_i \cdot (\frac{n_i}{m})^{\frac{1}{2}})$ labels after phase i, with high probability. Starting with n labels, there will be at most $\widetilde{O}(n \cdot (\frac{n}{m}))$ after phase 0. By recursion, we can show that after phase r, there are at most $\widetilde{O}(n \cdot (\frac{n}{m})^{(1.5)^r})$ labels left. After $r = \mathcal{O}(\log \log_{\frac{m}{n}} n)$ phases, there will be a unique label left. □

Since each phase still takes $\mathcal{O}(\log D)$ rounds, the whole algorithm runs in $\mathcal{O}(\log D \cdot \log \log_{\frac{m}{n}} n)$ rounds using total memory of $\Theta(m)$.

3.3.5 Guessing the diameter D of the graph

We see that the algorithm of Section 3.3.4 can operate *without* explicitly knowing D. The diameter D only shows up in the analysis. If one wishes to know the diameter, one could do the following: For $i = \{0, 1, 2, \dots\}$,

- Use $\widehat{D} = 2^{2^i}$ as a guess for D.
- Run the algorithm discussed in Section 3.3.4 with $\mathcal{O}(\log \widehat{D})$ rounds of graph exponentiation per phase.
- If there exists an edge $e = \{u, v\}$ such that u and v have different labels, we have underestimated D. Try again with a larger guess.
- Otherwise, endpoints of *all* edges have the same label. Thus, the connectivity algorithm has completed and we can terminate.

One can check that the total number of rounds is dominated by the final guess of D, which results in the same number of rounds asymptotically.

Roughly speaking, the above method can be generalized to guess parameters of the input graph for MPC algorithms which are "safe" and "checkable". By "safe", we mean that running with an underestimate will not violate MPC constraints. Meanwhile, we could "check" whether we have solved the problem after executing the algorithm with a guess, i.e. check the correctness of the output.

3.4 Geometric MST using sublinear memory

Problem definition (Euclidean MST) Consider a Euclidean space $\mathbb{R}^d$ of d dimensions. Given a set n points, each with integer coordinates in $[0, \Delta]^d$ for some $\Delta = \text{poly}(n)$, we wish to compute an $(1+\epsilon)$-approximate minimum spanning tree T, for some $\epsilon > 0$. That is, we want the cost of the computed tree T to be no more than an $(1+\epsilon)$ factor away from the

true cost of the minimum spanning tree T^*. In the geometric setting, there are $\binom{n}{2}$ implicit edges between all pairs of points. However, the problem input only involves the n points, along with parameters such as d, Δ and ϵ. This is because the distance between points u and v can be computed "on-the-fly" using a distance function $\rho(u, v)$ such as Euclidean norm.

We know from before that $O(\log n)$ rounds suffice to compute a MST under the strongly sublinear memory regime (See Exercise 3.2). In this section, we look at a constant round algorithm by Andoni et al. [ANOY14] for computing a $(1+\epsilon)$-approximate minimum spanning tree T under the sublinear memory regime with $S = n^\alpha$ memory per machine, for some constants $\epsilon > 0$ and $\alpha \in (0, 1)$. There are four main ideas:

Idea 1 Exploit the geometric structure of the problem by building a S-ary tree that partitions the space. The partitioning is randomly shifted so that "close points are likely to be in the same partition".

Idea 2 Ideally, one wishes to solve MST locally in each partition and combine the solutions. However, this can be bad as short edges could be crossing partitions. Hence, we build a minimum spanning forest locally only using "sufficiently short edges".

Idea 3 To avoid violating memory constraints when combining solutions across partitions, we pick representatives to form a sketch.

Idea 4 To analyze the approximation overhead of our constructed T compared to the true cost of MST T^*, we define an auxiliary graph G' which overestimates the true edge costs, then compare Kruskal's edge selection on G' against our selected edges.

We present the ideas in Section 3.4.1, Section 3.4.2, Section 3.4.3, and Section 3.4.5 respectively. In the following exposition, we often discuss in 2-dimensions ($d = 2$) to better visualize and get intuition.

Remark In the paper, Andoni et al. [ANOY14] also used similar ideas to tackle the problem of Earth-Mover Distance. In the Earth-Mover Distance problem, points are split into red and blue points, and the goal is to find the minimum cost red-blue matching.

3.4.1 Idea 1: Hierarchical partitioning with random shift

In Euclidean space, an edge between close points are more likely to be an MST edge. To exploit the geometric structure of the problem, one can

partition the input space to ignore pairs of points which are far apart. To this end, consider the following hierarchical partitioning process where we denote P_i as the i^{th} level of partitioning:

- Randomly pick an offset vector $v = (v_1, \ldots, v_d) \in (-\Delta, 0]^d$ to define a bounding box $[v_1 + \Delta] \times [v_2 + \Delta] \times \cdots \times [v_d + \Delta]$ on all inputs points. This bounding box is the single cell of the highest level of the partitioning P_L.
- As long as there is a cell in P_i with more than one point, partition all cells in P_i into S smaller cells of dimension $S^{\frac{1}{d}} \times S^{\frac{1}{d}} \times \cdots \times S^{\frac{1}{d}}$ to yield partition level P_{i-1}.

The partitioning level which contains cells with at most one point is denoted as P_0. As points have integer coordinates in $[0, \Delta]^d$, the partitioning process stops after $L = \mathcal{O}(\log_S(n)) = \mathcal{O}(\frac{1}{\alpha})$ levels. Since α is a constant, we have constant levels of partitioning. Fig. 3.3 illustrates an example.

For any fixed partitioning, there are input points such that very close points are split into different partitions very early on. This will create many short edges across partitions. By picking a random shift of v, one can show a desirable property that close points remain in same cells until the cells become small. Let us be more precise. Denote Δ_i as the maximum distance between any two points in a cell at level i, and $C_i(x)$ as the cell which point x reside in at level i. In 2-dimensions, Δ_i is the diagonal distance between corners in a cell at level i. One can show the following:

Claim 3.10. $\Pr[C_i(x) \neq C_i(y)] \leqslant \mathcal{O}(\frac{\rho(x,y)}{\Delta_i})$, *where* ρ *is the distance function and the constant factor in* $\mathcal{O}$ *depends on the dimension* d.

Remark Hierarchical partitioning is similar in spirit to the technique of ball carving for probabilistic tree embeddings[5] by Fakcharoenphol, Rao, and Talwar [FRT03]. In ball carving, n vertices in a metric space are picked in random order, then the picked vertex and all vertices within a radius r of the picked vertex are removed. One can then show that the probability that vertices x and y are not removed *at the same time* is at most $\mathcal{O}(\log n) \cdot \frac{\rho(x,y)}{r}$. That is, close vertices are likely to be removed at the same time. By appealing to the Johnson-Lindenstrauss Lemma[6], hierarchical partitioning can be seen as an alternate way to perform ball carving.

[5]For details on ball carving and probabilistic tree embeddings, one can look at Section 5.1 of http://people.csail.mit.edu/ghaffari/AA18/Notes/AAscript.pdf.

[6]For details on the Johnson-Lindenstrauss Lemma, one can look at the technical report of Dasgupta and Gupta [DG99].

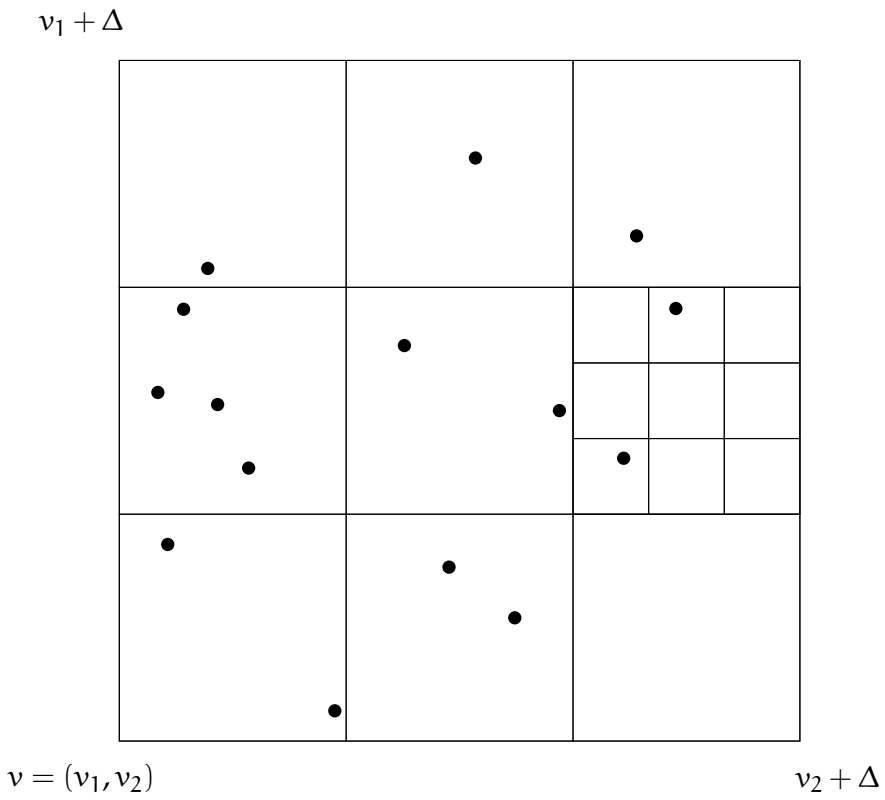


Figure 3.3: Example on 15 points in 2-dimensions: A randomly chosen $v \in (-\Delta, 0]^2$ defines a shifted bounding box. Recursive partitioning into S smaller cells stop when every cell only has at most one point.

3.4.2 Idea 2: Local Kruskal with early termination

Consider the recursive step where we want to solve a minimum spanning forest on the points in a given cell. Ideally, one wishes to solve MST locally in each partition and combine the solutions. However, this can be bad as short edges could be crossing partitions and we incur excessive overhead in the edge weights when we combine solutions. The remedy is to build a minimum spanning forest locally only using "sufficiently short edges" and pass up the partial solution to future partition levels. To be precise, at level i, we run Kruskal's algorithm as long as the minimum distance between two connected components as at most $\epsilon\Delta_i$.

While we are not particularly concerned about runtime complexity within a machine, Andoni et al. [ANOY14] showed that it is possible to efficiently find points u and v from different connected components such that $\rho(u, v) \leqslant (1 + \epsilon)\tau$, where τ is the minimum pairwise distance

between different connected components. This is done by finding an ϵ-chromatic closest pair (ϵ-CCP), where colors represent connected components. Eppstein [Epp95] gave a reduction from the ϵ-CCP problem to the ϵ-approximate nearest neighbour search (ϵ-ANNS) problem, and Arya et al. [AMN+98] and Eppstein et al. [EGS08] showed the existence of efficient solutions to the ϵ-ANNS problem.

3.4.3 Idea 3: Sketching to reduce memory footprint

After computing an approximate minimum spanning forest, we want to avoid violating memory constraints when combining solutions across partitions. To do so, we pick representatives to form a sketch. We first define δ-nets, then explain how it gives us an appropriate sketch.

A δ-net is a subset of points such that any two chosen points have distance at least δ while any unchosen point is at most distance δ from some chosen point. An appropriate sketch at level i is an $\epsilon^2\Delta_i$-net of points from the current cell, along with the partitioning (i.e. which points belong to which connected component). An $\epsilon^2\Delta_i$-net can be obtained by partitioning the cell into ϵ^{-d} grids, each of width $\epsilon^2\Delta_i \cdot \frac{1}{\sqrt{d}}$, then selecting an arbitrary point as a representative from each grid. There are three desirable properties of such a construction:

1. There are ϵ^{-d} grids, so there will be at most ϵ^{-d} representatives. Thus, each cell produces a sketch of a constant number of points.
2. As Kruskal terminates only when the distances exceed $\epsilon\Delta_i$, all points in the same grid belong to the same connected component.
3. By triangle inequality, connecting a point outside of a grid to the representative incurs an additional cost of at most $\epsilon^2\Delta_i$. See Fig. 3.4.

3.4.4 Algorithm

Putting together, we get a recursive `Solve-and-Sketch` algorithm for computing an approximate MST given n points in Euclidean space:

1. Build an S-ary tree with height $\mathcal{O}(\frac{1}{\alpha})$ using hierarchical partitioning.
2. Recursively compute minimum spanning forests from smaller cells
   (a) At level i, run Kruskal's until the weight of the minimum cost edge (without forming a cycle) in the cell exceeds $\epsilon\Delta_i$.

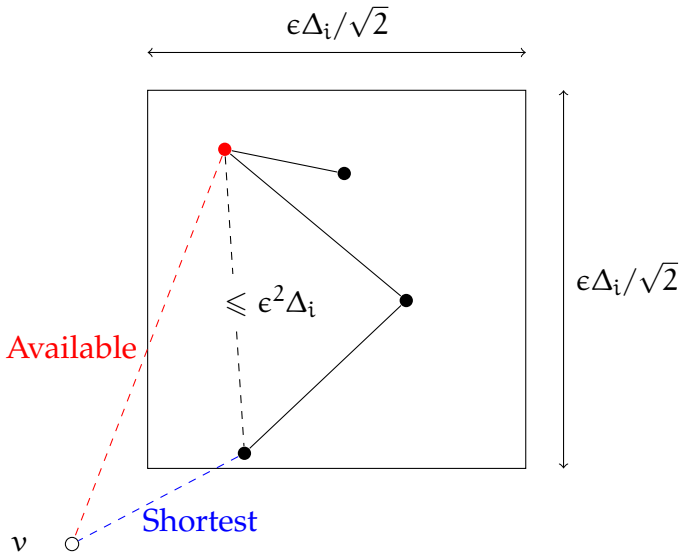


Figure 3.4: Overhead in distance due to the sketch illustrated in 2 dimensions. Red point is the chosen representative and the white point v lies outside the $\epsilon^2\Delta_i$-net. The red line indicates the available edge between v to any point in the sketch while the blue line is the shortest possible edge.

(b) Compute a sketch involving at most ϵ^{-d} representative points.

(c) Send representative points to larger cell containing current cell.

Each level of the partitioning can run simultaneously as each cell contains at most $\mathcal{O}(S)$ points. Thus, `Solve-and-Sketch` computes a spanning tree T in constant rounds under the sublinear memory regime where each machine has space $S = n^\alpha$. It remains to argue that the cost of T is at most $(1+\epsilon)$ factor away from the cost of the true minimum spanning tree T^*.

3.4.5 Idea 4: Bounding the approximation factor

Since we run a modified Kruskal's algorithm in each of our recursive step, it is natural to attempt to bound the approximation factor by comparing the produced tree against Kruskal's output. Suppose `Solve-and-Sketch` chooses edges $e_1, \ldots, e_{n-1}$ chronologically and Kruskal chooses $e_1^*, \ldots, e_{n-1}^*$ when building the MST on the n input points. If one can show that $w(e_i) \leqslant (1+\epsilon) \cdot w(e_i^*)$ for $i \in \{1, 2, \ldots, n-1\}$, then the produced MST is an $(1+\epsilon)$-approximation MST (See Exercise 3.6).

Unfortunately we cannot directly apply the above analysis because it is

hard to account for the fact that the recursive step ignores edges crossing between cells. In order to proceed, we first define an auxiliary weighted graph G' which *overestimates* the true edge costs. Let $T = (e_1, e_2, \ldots, e_{n-1})$ be the MST computed by `Solve-and-Sketch` on the n input points, and $T' = (e'_1, e'_2, \ldots, e'_{n-1})$ be the tree computed by running Kruskal on G'. If one shows that $w(e_i) \leqslant w(e'_i)$ for $i \in \{1, 2, \ldots, n-1\}$, then T is an an $(1+\epsilon)$-approximation MST of the n input points by the above argument.

Consider the following weighted graph $G' = (V, E)$ where V is the set of n input points and E is the set of possible edges between any two points $x, y \in V$. In `Solve-and-Sketch`, we continuously partition cells until every cell has at most one point. Hence, for any pair of points x and y, there is a *crossing level* l_c such that x and y are not in the same cell *for the first time*. To be precise, crossing level $l_c = \min\{i \in \{1, 2, \ldots, \log_S(n)\} : C_i(x) \neq C_i(y)\}$. Intuitively, the crossing level l_c will be small if x and y are far apart. Define the weight of edge $e = \{x, y\}$ by $w(e) = \rho(x, y) + \epsilon\Delta_{l_c}(x, y)$, where $\rho(x, y)$ is the distance between points x and y in the Euclidean space, l_c is the crossing level of points x and y, and Δ_i is the maximum distance between any two points in a cell at level i. By construction, $w(\{x, y\})$ is an overestimate of the true distance $\rho(x, y)$. However, it is not by too much: using Claim 3.10, one can show [ANOY14, Lemma 3.10] that $\mathbb{E}[w(\{x, y\})] = (1 + \mathcal{O}(\epsilon)) \cdot \rho(x, y)$.

Define *intercluster edges* as edges between connected components while constructing T. Let e_i^+ be the minimum cost intercluster edge with respect to the weight function w, at the point where e_i was chosen to be included into T. One can show [ANOY14, Proposition 3.20] that $w(e_i^+) \leqslant w(e'_i)$, then use case analysis [ANOY14, Page 13-14] to argue that for every i, $\rho(e_i) \leqslant (1 + \mathcal{O}(\epsilon)) \cdot w(e_i^+)$. Thus, $\rho(e_i) \leqslant (1 + \mathcal{O}(\epsilon)) \cdot w(e'_i)$, completing the argument.

Exercise 3.6. Approximate MST with approximate Kruskal's
Consider the modified process of Kruskal's algorithm. At each iteration, let E_i be the set of remaining edges that do not form cycles. Add edge e to the MST, where $w(e) \leqslant (1+\epsilon) \cdot \min_{e' \in E'} w(e')$. Argue that the tree produced by this modified process is an $(1+\epsilon)$-approximation to the MST of the original graph.

Hint *For each iteration, compare the weight of the chosen edge against Kruskal's.*

Chapter 4

Lower bounds and conditional hardness results

In this chapter, we introduce lower bound results and a general conditional hardness reduction. We first discuss a lower bound result of Roughgarden et al. [RVW18] in Section 4.1 that undirected graph connectivity requires $\Omega(\log_S n)$ rounds in MPC with machines memory S. In Section 4.2, we introduce Conjecture 4.5 and the notion of conditional lower bounds. Then, we briefly discuss how to construct a conditional lower bound for constant approximation maximum matching in MPC. The construction described can be generalized to transform lower bounds in the LOCAL model to lower bounds in MPC for other problem classes. Interested readers are invited to check out the manuscript of Ghaffari et al. [GKU19].

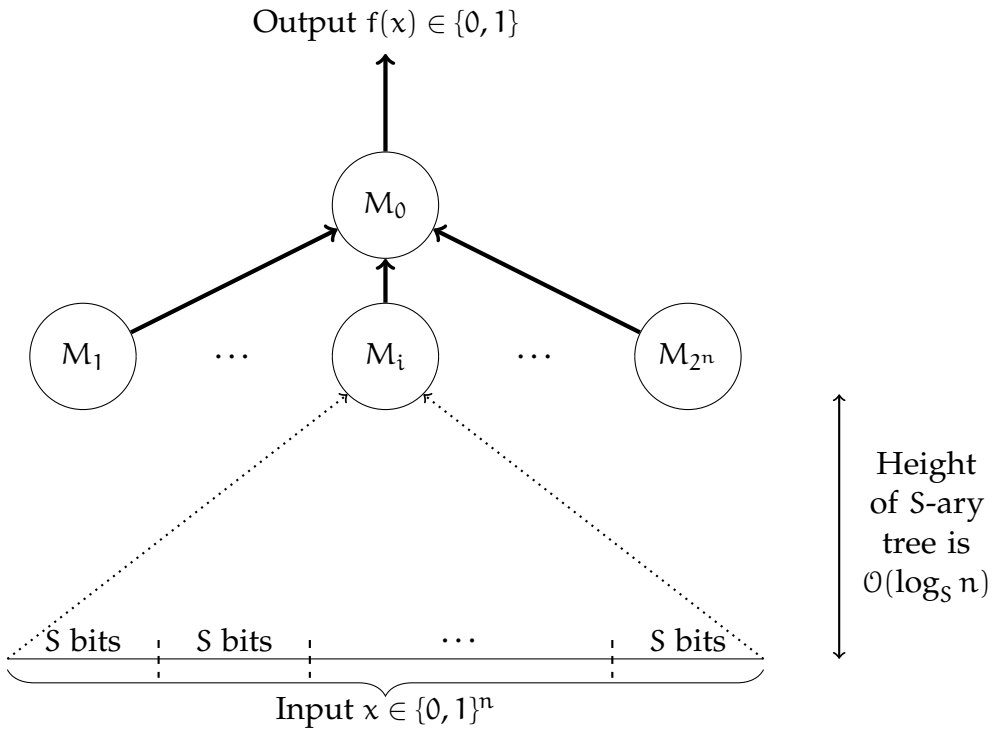
4.1 Lower bounds for connectivity and other problems

Roughgarden et al. [RVW18] showed that connectivity requires $\Omega(\log_S n)$ rounds in MPC with machines memory S. Before diving into the construction, Section 4.1.1 first motivates why showing $\Omega(\log_S n)$ number of MPC rounds is a reasonable target lower bound. Then, Section 4.1.2 gives an outline of the construction of Roughgarden et al. [RVW18] with Section 4.1.3, Section 4.1.4 and Section 4.1.5 providing the necessary details.

4.1.1 Motivation

Claim 4.1. *With "enough" machines, we can compute function* f *in* $\mathcal{O}(\log_S n)$ *MPC rounds, for any arbitrary function* $f : \{0,1\}^n \to \{0,1\}$.

Proof. A function $f : \{0,1\}^n \to \{0,1\}$ has 2^n possible inputs.



We dedicate a single machine M_0 for output, and 2^n machines $M_1, \ldots, M_{2^n}$ for handling each possible input x. That is, machine M_i will send $f(i)$ to machine M_0 if it knows that input string is i. For each machine M_i, we can construct a S-ary tree that reads the input string x and lets M_i know whether $x = i$. By construction, exactly one of the machines M_i from $\{M_1, \ldots, M_{2^n}\}$ will detect that the input $x = i$ and send the value $f(i)$ to M_0 in one round. Then, M_0 can output the value $f(x) = f(i)$ as the output of the whole computation. In total, this takes $\mathcal{O}(\log_S n)$ rounds since the S-ary trees have depth at most $\mathcal{O}(\log_S n)$. □

While we have shown that *any* n-bit function can be computed in $\mathcal{O}(\log_S n)$ rounds in MPC, the construction uses exponentially many machines. A reasonable question is to wonder whether one can show an $\Omega(\log_S n)$ round lower bound in MPC if we restrict ourselves to poly(n)

number of machines. Unfortunately, it is probably not possible (for now). To understand why, we look to the study of circuit complexity.

The circuit complexity of a function f is the minimum depth of a circuit that can compute f using logical gates with fan-in 2, such as AND and OR gates. One can show that complexity analyses assuming fan-in 2 generalize to larger fan-in numbers. Recall that $\mathcal{P}$ is the class of decision problems that can be solved on a deterministic sequential machine in polynomial time with respect to input size. One long standing question in circuit complexity is the following:

Does every problem in $\mathcal{P}$ admit an $O(\log n)$ depth circuit?

The following claim shows that if we are able to show an $\Omega(\log_S n)$-round lower bound in MPC with poly(n) number of machines, then we have answered the long standing question in the negative.

Claim 4.2. *For an arbitrary machine memory S, an $\Omega(\log_S n)$-round lower bound in MPC for a problem in $\mathcal{P}$ implies that this problem has no $O(\log n)$ depth fan-in 2 Boolean circuit.*

Proof. To prove this, we show that any depth d fan-in 2 Boolean circuit can be simulated in $O(\frac{d}{\log S})$ rounds in MPC. Denote the inputs as depth 0 and output as depth d. Since we are considering gates of only fan-in 2, a gate at depth i depends on at most 2^i inputs. Consider the set of gates G of depth $d' \in \{0, \ldots, \log S\}$. Since gates in G depend on at most $2^{\log S} = S$ inputs, we can use a single machine to simulate each gate and its connection. Thus, in one MPC round, we can effectively reduce the depth of the circuit by $O(\log S)$. Therefore, we can simulate the entire circuit in MPC by repeating this process $O(\frac{d}{\log S})$ times. □

4.1.2 Outline

There are three key steps in the lower bound construction of Roughgarden et al. [RVW18]:

1. Define the s-SHUFFLE model, and show that any MPC computation can be formulated as a s-SHUFFLE computation.

2. s-SHUFFLE can only compute polynomials of degree at most s^R in R rounds.

3. Undirected connectivity requires a sufficiently high degree polynomial to represent. To be precise, the degree of the Boolean function f_n for undirected connectivity on a graph with n vertices is $\binom{n}{2}$.

By setting machine memory to s in the s-SHUFFLE definition, steps 2 and 3 together show that graph connectivity require at least $\Omega(\log_s n)$ rounds in MPC. We explain each of the three steps in Section 4.1.3, Section 4.1.4 and Section 4.1.5 respectively.

4.1.3 Step 1: Modelling MPC computation with s-SHUFFLEs

Claim 4.2 suggests that machines in MPC are more powerful than an arbitrary logical gate (even if it is a lookup table) in circuit complexity. This is because of two key differences between machines in MPC and gates in Boolean circuits:

1. *Adaptivity of communication topology*
   In Boolean circuits, the connections between gates are *fixed*. In MPC, machines can communicate with each other in arbitrary manner, as long as the total size of received and sent messages for a machine is S at every round.

2. *Possibility of silence (i.e. not send anything)*
   In Boolean circuits, all "earlier" circuits send either a 0 or 1 to "later" circuits. In MPC, machines can choose *not* to send anything to another machine. This "silence" can actually carry information and affect computation.

To model computation in MPC, Roughgarden et al. [RVW18] used $\perp$ to denote silence and defined a R-round s-SHUFFLE computation as follows:

- Let input be $x = (x_1, x_2, \ldots, x_n)$ and output be $y = (y_1, y_2, \ldots, y_k)$.
- Let V be the set of machines in the system, including one machine for each input bit x_i and output bit y_j
- Assign a level $r(v)$ for each machine $v \in V$. Machines handling input x are on level 0 while machines handling output y are on level $R+1$. All other machines have levels in the range $\{1, 2, \ldots, R\}$.
- For each pair of machines u and v, where $r(u) < r(v)$, we have a function $\alpha_{uv} : \{0, 1, \perp\}^s \to \{0, 1, \perp\}^s$ denoting what u sends to v.

One may think of an s-SHUFFLE computation as a "super circuit" where each machine is a "gate" with s ports. Each port accepts at most 1 bit of input from *any* lower level machine. Between machines u and v, where $r(u) < r(v)$, the message $(\alpha_{uv})_i = \perp$ indicates that machine u did not send any bit to the i^{th} port of v. On input $x = (x_1, \ldots, x_n)$, the i^{th} input machine at level 0 outputs $\{x_i, \perp, \ldots, \perp\}$ and the j^{th} output machine at level $R+1$ outputs $\{y_j, \perp, \ldots, \perp\}$. For an s-SHUFFLE computation to be valid, every machine should receive a non-$\perp$ bit input for each port from *at most one* machine. That is, every other lower level machine sends $\perp$ to that port.

One can extend s-SHUFFLEs to handle unordered input messages by using Σ to denote all possible multi-sets of bit strings with length at most w. Then, one can show the following claim:

Claim 4.3 (Proposition 2.7, [RVW18])**.** *Every r-round MapReduce computation with m machines and space s per machine can be simulated by an $(r+1)$-round s-SHUFFLE(Σ) computation with $(r+1)m$ machines and word size s.*

4.1.4 Step 2: s-SHUFFLEs are low-degree polynomials

Theorem 4.4. *Suppose that an s-SHUFFLE computes a function $f : \{0,1\}^n \to \{0,1\}$ in R rounds. Then, there is a polynomial $p(x_1, \ldots, x_n)$ of degree at most s^R such that $p(x) = f(x)$ for any $x \in \{0,1\}^n$.*

Proof. The proof is by induction over the round numbers. For each machine v with $r(v) = r \in \{0, \ldots, R+1\}$ and value $z \in \{0,1,\perp\}^s$, we show that there is a polynomial $p_{v,z}(x)$ of degree at most $s^{r(v)}$ such that $p_{v,z}(x) = 1$ if the s-SHUFFLE computation evaluates input x to value z in machine v, and 0 otherwise.

In the base case of input machines v with $r(v) = 0$, the corresponding polynomials for values $z_0 = (0, \perp, \ldots, \perp)$ and $z_1 = (1, \perp, \ldots, \perp)$ are $p_{v,z_0}(x_1, \ldots, x_n) = 1 - x_i$ and $p_{v,z_1}(x_1, \ldots, x_n) = x_i$ respectively. All other values are impossible and hence $p_{v,z}(x_1, \ldots, x_n) = \perp$ for all other values z.

In the inductive case of round number $r \in \{1, \ldots, R+1\}$, consider machine v with $r(v) = r$. We first show that all inputs to machine v can be represented by polynomials of degree at most s^{r-1}. Consider the i^{th} input port of machine v. We know there exists a set of polynomials

$$P_{u,z,1} = \{p_{u,z} : \text{Machine } u \text{ with } r(u) < r(v) \text{ and } (\alpha_{uv})_i(z) = 1\}$$

that indicates machine v receiving '1' on the i^{th} port. By induction, polynomials in $P_{u,z,1}$ have degrees at most s^{r-1}. Thus, the summation

$p_{v,i,1} = \sum_{p \in P_{u,z,1}} p$ is a polynomial of degree at most s^{r-1} that represents all possible ways for machine v to receive a '1' in the i^{th} port. Similarly, we can obtain polynomials $p_{v,i,0}$ and $p_{v,i,\perp}$ of degree at most s^{r-1}. Note that we only consider valid s-SHUFFLES, so at most one machine actually sends a non-$\perp$ value to each port of v. Now, for an output value z of machine v, we look at the set of possible inputs to v such that v outputs z. Every such input is represented by a product of the individual input polynomials, yielding a polynomial of degree at most s^r. Thus[1], $p_{v,z}$ is a polynomial with degree at most s^r. □

Theorem 4.4 can be generalized to outputs of k bits by repeating the argument for each output (See Theorem 3.1 of [RVW18]). Consider the following example to better understand the inductive step of Theorem 4.4.

Example Suppose $s = 3$. Fix a value z and machine v at round $r = r(v)$. By induction, we know that $p_{v,i,0}$, $p_{v,i,1}$ and $p_{v,i\perp}$ are polynomials of degree at most s^{r-1}, for $i \in \{1, 2, 3\}$. Suppose machine v outputs value z when given inputs $(1, 0, \perp)$ or $(0, 0, 1)$. Then, $p_{v,z} = p_{v,1,1}p_{v,2,0}p_{v,3,\perp} + p_{v,1,0}p_{v,2,0}p_{v,3,1}$. Since $p_{v,1,1}p_{v,2,0}p_{v,3,\perp}$ and $p_{v,1,0}p_{v,2,0}p_{v,3,1}$ are polynomials of degree at most s^r, the polynomial $p_{v,z}$ has degree at most s^r.

4.1.5 Step 3: Connectivity as a high-degree polynomial

Theorem 4.4 tells us that an R-round s-SHUFFLE computation can be represented as a polynomial of degree at most s^R. In this step, we show that modelling undirected graph connectivity as a Boolean function requires a polynomial of degree at least $\binom{n}{2}$. Therefore, this implies that *any* valid s-SHUFFLE computation, and hence any MPC algorithm, that solves undirected graph connectivity would require at least $R = \Omega(\log_s n)$ rounds.

A Boolean function that models undirected graph connectivity on n-node graphs is a function from $\binom{n}{2}$ bits to 1 bit. Rivest and Vuillemin [RV75] proved the Aanderaa-Rosenberg conjecture [Ros73] that every non-trivial monotone graph property has the decision-tree complexity is $\Omega(n^2)$. One can verify that connectivity is a non-trivial monotone graph property — The output is not the same for *all* graphs, and adding more edges does not disconnect a graph. Using the notion of *parity difference*, one can use induction on n to show that the degree of a Boolean function representing undirected graph connectivity is $\binom{n}{2}$. We refer readers to Definition 5.1,

[1] See the example below this proof for a concrete illustration.

Lemma 5.2 and Theorem 5.3 of Roughgarden et al. [RVW18] for details on parity difference and how it is used. Thus, *any* valid s-SHUFFLE computation that solves undirected graph connectivity would require at least $R = \Omega(\log_S n)$ rounds.

4.2 Conditional hardness results for problems such as matching and vertex cover

In this section, we introduce the notion of conditional lower bounds and give an example. Using the conditional lower bound of Claim 4.7, we introduce a general framework that allows us to adapt lower bounds in the LOCAL model to conditional lower bounds in the MPC setting.

Conjecture 4.5 (One cycle versus two cycles)**.** *Suppose machines have n^α memory for $\alpha \in (0, 1)$. Even with poly(n) number of machines, distinguishing a cycle of n nodes and two cycles of $\frac{n}{2}$ nodes require $\Omega(\log n)$ MPC rounds.*

Definition 4.6 (The D-diameter s-t connectivity problem)**.** *Given vertices s and t, and a parameter D,*

- *Output YES, w.h.p., if s and t are in the same connected component that is a path of length at most D, with s and t as its two endpoints.*
- *Output NO, w.h.p., if s and t are in different connected components.*

In other cases, one may output arbitrarily. Notice that, with high probability, the output can only be YES whenever s and t are in the same connected component.

Claim 4.7. *Given Conjecture 4.5, solving the D-diameter s-t connectivity problem requires $\Omega(\log D)$ MPC rounds.*

We leave the proof of Claim 4.7 as an exercise (See Exercise 4.1). We now turn our attention towards using Claim 4.7 in a framework that lifts lower bounds in the LOCAL model to conditional lower bounds in the MPC model. The following claim shows such a result:

Claim 4.8. *Under Conjecture 4.5, computing a constant approximation to maximum matching requires $\Omega(\log \log n)$ MPC rounds in the sublinear memory regime.*

We describe the two key steps for proving Claim 4.8:

1. Kuhn et al. [KMW04] showed computing a constant approximation to maximum matching in the LOCAL model requires $\widetilde{\Omega}(\sqrt{\log n})$ rounds. Using this lower bound result, one can show that for *any* $o(\log\log n)$-round MPC algorithm $\mathcal{A}$ that computes a constant approximation to maximum matching with success probability $\geqslant \frac{2}{3}$, there exists two graphs G and G' for which $\mathcal{A}$ will fail with non-negligible probability ϵ.

2. Next, we obtain a contradiction using Claim 4.7 for $D \in \widetilde{\Omega}(\sqrt{\log n})$. Assume, for a contradiction, that there is an $o(\log\log n)$-round MPC algorithm $\mathcal{B}$ that computes a constant approximation to maximum matching. Using $\mathcal{B}$ as an oracle, one can construct an $o(\log\log n)$-round MPC algorithm $\mathcal{A}$ to solve the D-diameter s-t connectivity problem. However, running $\mathcal{A}$ on graphs G and G' from the previous step will fail with a non-negligible probability. Therefore, no such $o(\log\log n)$-round MPC algorithm $\mathcal{B}$ can exist.

Notice that the first step can be generalized to other problems. In particular, *any* R-round lower bound in the LOCAL model with shared randomness can be converted into a $(\log R)$-round MPC lower bound using the argument above. Figure Section 4.2 illustrates the generalized conditional lower bound reduction workflow. We forward interested readers to the manuscript of Ghaffari et al. [GKU19] for details.

Remark Step 1 is a special case on the problem of finding constant approximations to maximum matching. In the manuscript of Ghaffari et al. [GKU19], graphs G and G' are called r-hop-isomorphic centered graphs, and $\mathcal{A}$ is said to be (r, ϵ)-sensitive with respect to G and G'.

Exercise 4.1. D-diameter s-t connectivity *Suppose we have poly(n) machines, each with n^{α} memory for $\alpha \in (0, 1)$. If there is no $o(\log n)$ MPC algorithm for distinguishing a cycle of n nodes and two cycles of $\frac{n}{2}$ nodes, then there is no $o(\log D)$ round MPC algorithm for solving the D-diameter s-t connectivity problem.*

Hint *Assume, for a contradiction, that exists an oracle that solves the D-diameter s-t connectivity problem in $o(\log D)$ rounds. Use the oracle $O(\log_D n)$ times to distinguishing a cycle of n nodes and two cycles of $\frac{n}{2}$ nodes, hence obtaining a contradiction.*

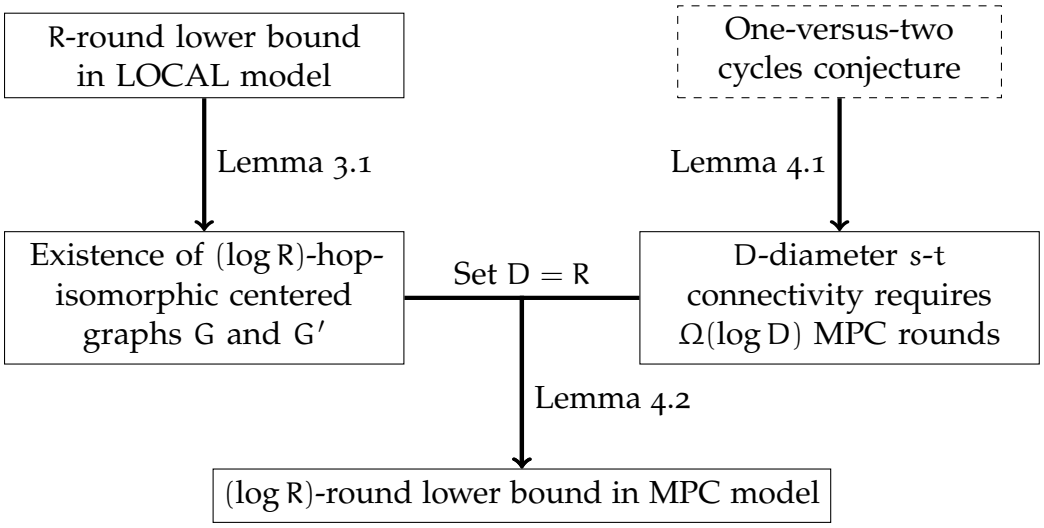


Figure 4.1: Graphical illustration of conditional lower bound reduction technique of Ghaffari et al. [GKU19] which relies on Conjecture 4.5.

Chapter 5

Dynamic Programming

In this chapter, we look to the class of problems which admit dynamic programming (DP) solutions. In particular, we study the *weighted interval selection* problem and the $(1-\epsilon)$-approximation MPC algorithms proposed by Im et al. [IMS17]. In their work, Im et al. also designed $(1+\epsilon)$-approximation algorithms for the *longest increasing subsequence* and *optimal binary search tree* problems but they are beyond the scope of this chapter.

5.1 Weighted Interval Selection

Let us first define the weighted interval selection problem.

Definition 5.1 (The weighted interval selection problem)**.** *Given n input intervals $I_i = [a_i, b_i]$ with weights $w_i \geqslant 0$, output a selection of non-overlapping intervals with the maximum total weight.*

An exact sequential DP algorithm In the sequential setting where all n intervals fit into a single machine, we know the following dynamic programming solution: First sort all intervals by their starting points a_i's, then let $A(i) = \mathsf{OPT}(\{I_i, I_{i+1}, \ldots, I_n\})$ be the maximum attainable weight between picking the i^{th} interval I_i, or ignoring it. That is,

$$A(i) = \max \begin{cases} w_i + A(j), \text{ where } j = \min\{j' : j' > i \wedge b_i \leqslant a_{j'}\} & \text{(Pick } I_i\text{)} \\ A(i+1) & \text{(Ignore } I_i\text{)} \end{cases}$$

MPC setting We are interested in computing a $(1-\epsilon)$-approximation solution to the weighted interval selection problem with m machines, each

having memory $S = \widetilde{\mathcal{O}}(n^\alpha)$ for some constant $\alpha \in (0,1)$, such that the total global memory is $\widetilde{\mathcal{O}}(n)$. Since sorting can be done in $\mathcal{O}(1)$ rounds with strongly sublinear memory[1], we assume without loss of generality that the intervals are sorted by the starting points a_i's. We describe $\mathcal{O}(\log n)$ round and constant round MPC algorithms due to Im et al. [IMS17] in Section 5.1.1 and Section 5.1.2 respectively.

Remarks on computing an approximate solution We can rescale all weights w_i to $\frac{w_i}{\max_i w_i}$ such that all weights are now in the interval $[0,1]$ and the maximum total weight is at most n. Consider the sequence $(1+\epsilon)^0, (1+\epsilon)^1, \ldots, (1+\epsilon)^{\log_{(1+\epsilon)} n} = n$. We know that one of these $\mathcal{O}(\log n)$ values will be an $(1-\epsilon)$-approximation to the optimum maximum total weight. Furthermore, we can ignore all intervals with weight less than $\frac{\epsilon}{n}$ as their contribution will not affect the approximation.

5.1.1 An $\mathcal{O}(\log n)$ round MPC algorithm

The sequential DP formulation $A(i)$ is inherently sequential and is ill-suited to be adapted into the MPC setting. We shall describe a "smooth" reformulation of the dynamic programming recursion, and show how this reformulation allows for an $\mathcal{O}(\log n)$ round MPC algorithm.

Reformulation of the dynamic programming recursion

One way to remove the sequentiality of $A(i)$ is to consider pairs of indices as follows: Let $B(i,j)$ be the maximum weight selection among intervals $\{I_i, I_{i+1}, \ldots, I_{j-1}\}$ such that the selected intervals end before a_j. This implies that for $i < j < k$, the selections from $B(i,j)$ and $B(j,k)$ can be concatenated to form a larger non-overlapping selection of intervals. Observe that for any $i' \leqslant i$ and $j' \geqslant j$, $B(i',j') \geqslant B(i,j)$. This property of $B(\cdot,\cdot)$ is called "monotonicity" by Im et al. [IMS17]. Unfortunately, while this reformulation removes the sequentiality of $A(i)$, computing and storing all possible $B(\cdot,\cdot)$ values will require $\Omega(n^2)$ space, greatly exceeding our global memory constraint of $\widetilde{\mathcal{O}}(n)$.

Instead, we consider the following reformulation: Let $C(i,w)$ be the minimum index j such that there is a selection with total weight at least w among intervals $\{I_i, I_{i+1}, \ldots, I_{j-1}\}$ such that the selected intervals end

[1] See Exercise 2.1.

before a_j. If no such index j exists, we denote $C(i, w) = \infty$. In other words,

$$C(i, w) = \min_{j' : B(i,j') \geqslant w} j'$$

While $C(\cdot, \cdot)$ still has 2 dimensions, the weight parameter w is "smooth" and we only need to consider $\mathcal{O}(\log n)$ possible w values if we only aim for an $(1 - \epsilon)$-approximation. Furthermore, since $B(\cdot, \cdot)$ is monotonic, a feasible solution (without overlapping intervals) that includes the selection from $C(i, w)$ remains feasible when we replace the selection from $C(i, w)$ by the selection of $C(i, w')$, for any $w' < w$.

To compute an $(1 - \epsilon)$-approximation, we first compute $C(i, w)$ for all $i \in [n]$ and $\mathcal{O}(\log n)$ possible w values, then output the selection that maximizes $\{w : C(1, w) \neq \infty\}$.

Algorithm

We initialize the m machines by placing the k^{th} consecutive chunk of $\frac{n}{m}$ intervals into the k^{th} machine. That is, the k^{th} machine receives intervals $I_{\frac{n}{m} \cdot (k-1)+1}$ to $I_{\frac{n}{m} \cdot k}$. See Fig. 5.1 for an illustration[2].

Let η be a constant to be defined. We use $M(i)$ to denote the machine holding interval I_i, and $C'(i, w)$ to denote the DP entries of $C(i, w)$. The values $C'(\cdot, \cdot)$ are initialized to ∞ and we iteratively updating them over $\mathcal{O}(\log n)$ rounds. At round r, $C'(\cdot, \cdot)$ is the optimal solution by considering intervals residing on 2^r machines.

1. In round 0, we compute $C'(i, w)$ locally by considering only the intervals residing on machine $M(i)$.

2. For $r \in \{1, \ldots, \log_2(n) - 1\}$

   - Consider an arbitrary $i \in [n]$ and an arbitrary weight value w out of the $\mathcal{O}(\log n)$ possible values.
   - For $\mathcal{O}(\log n)$ possible weight values of w_1
     (a) Suppose that $C'(i, w_1) = j_1$ at the start of round r.
     (b) Machine $M(i)$ sends machine $M(j)$ the pair (j_1, w_1).
     (c) Machine $M(j)$ will reply machine $M(i)$ with

$$j_2 = \min\{C'(j_1, w_2) : w_1 + w_2 \geqslant (1 - \eta)w\}$$

[2]Ignore the annotations t_1, t_2, t_3 and the phrase "local" — they will be explained later.

- Machine $M(i)$ updates $C'(i, w)$ to the min. of all j_2's received.

Notice that in round $r \in \{1, \ldots, \log_2(n) - 1\}$, we lose a factor of $(1-\eta)$ as we combine partial information across 2^r machines. After $\mathcal{O}(\log n)$ rounds, $C'(i, w)$ is computed with information from all machines, and the weight of the selected interval of $C'(i, w)$ is at least $\left((1-\eta)^{\mathcal{O}(\log n)}\right)$ times the weight of the optimal interval of $C(i, w)$. We then set η appropriately such that $(1-\eta)^{\mathcal{O}(\log n)} = (1-\epsilon)$.

Exercise 5.1. Handling potentially many requests to a single machine *In Step 2(b) of the above $\mathcal{O}(\log n)$ round MPC algorithm, a machine M^* may receive $\gg \widetilde{\mathcal{O}}(n^\alpha)$ pairs of (j, w) requests. This means machine M^* cannot handle all requests in a single MPC round as it will exceed the communication restriction of MPC. Design a constant round protocol to such that Steps 2(b) and 2(c) can be implemented in MPC with sublinear memory.*

Hint *Consider first an arbitrary weight w. Since there are a total of n requests, there cannot be too many intervals receiving $> \frac{n^\alpha}{\log n}$ requests. Build broadcast trees after randomly distributing the information held by these intervals.*

5.1.2 A constant round MPC algorithm

Before we describe the constant round algorithm, we need to introduce a couple of definitions. These definitions allow us to prove Lemma 5.2 which then suggests a constant round algorithm to approximate the optimal weighted interval selection.

Setup and definitions

We assume that the starting and ending points are unique[3]. As per Section 5.1.1, we initialize the m machines by placing the k^{th} consecutive chunk of $\frac{n}{m}$ intervals into the k^{th} machine and denote $M(i)$ as the machine that holds interval I_i.

We define machine time $t_k = \min\{a_i : \text{Interval } I_i \text{ resides in machine } k\}$ as the earliest starting time of any interval assigned to machine k. For an interval $I_i = (a, b)$, we say that I_i starts in machine k if $t_k < a < t_{k+1}$, and that I_i ends in machine k if $t_k < b < t_{k+1}$. An interval $I_i = [a, b]$ is called *local* if it starts and ends in the same machine, and *crossing* otherwise. Fig. 5.1 illustrates an instance with 9 intervals on 3 machines.

[3]We can perturb the a_i's and b_i's without affecting the feasibility of the solution.

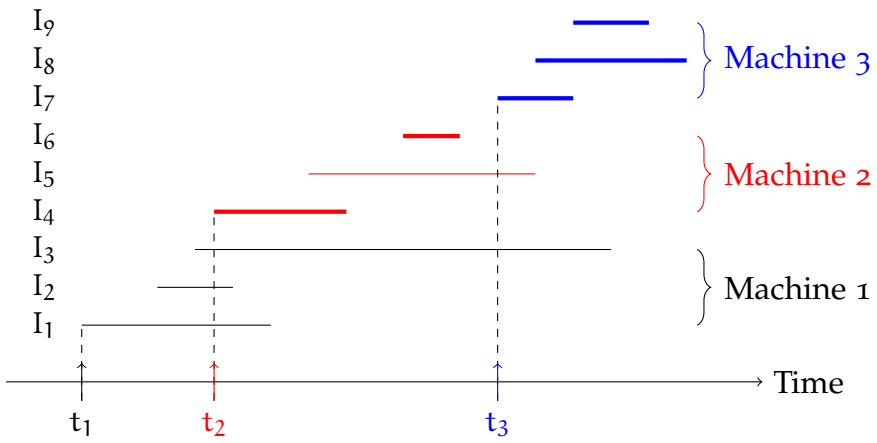


Figure 5.1: Interval assignment on 3 machines. Local intervals are in bold.

We call a subset of disjoint intervals, D, a *block*. A block D has weight $\sum_{I_i \in D} w_i$, with starting time a_D and ending time b_D defined by the earliest starting time and latest ending time of the intervals in D respectively. If block D has $\leqslant L$ crossing intervals, it is called an L-block. We say that block D *spans* machines $k, k+1, \ldots, k'$ if D starts in machine k and ends in machine k' (i.e. $t_k \leqslant a_D < t_{k+1}$ and $t_{k'} \leqslant b_D < t_{k'+1}$). Two blocks D and D' are said to be *independent* if their span are disjoint. Consider Fig. 5.1. Suppose we form blocks $D_1 = \{I_1, I_5\}$, $D_2 = \{I_1, I_6\}$ and $D_3 = \{I_7, I_9\}$, where each of them are made up of disjoint intervals. One can check that $span(D_1) = \{1, 2, 3\}$, $span(D_2) = \{1, 2\}$, and $span(D_3) = \{3\}$. Thus, D_1 and D_2 are not pairwise independent, but D_2 and D_3 are pairwise independent.

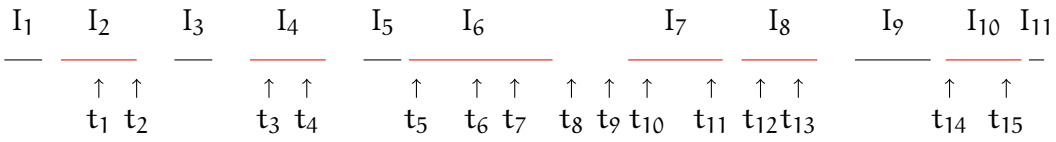
Remark Observe that a local interval only spans one machine while a crossing interval spans at least two machines.

Lemma 5.2. *For any even integer $L \geqslant 2$, there is a $(1 - \frac{2}{L})$-approximate weighted interval selection problem consisting of pairwise independent L-blocks.*

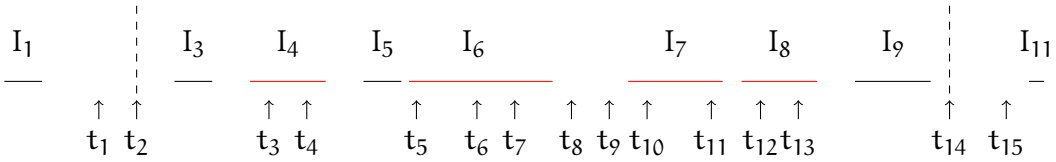
Proof. The proof is constructive: We will construct independent blocks out of any arbitrary optimal solution. Fix an optimal solution and order the chosen intervals in ascending starting time. Consider L consecutive crossing intervals, then remove the minimum weight interval I_{min} amongst each of them. Since we remove the minimum cost out of L intervals, we lose at most a factor of L in the total weight. Partitioning the chosen intervals at a machine spanned by the removed I_{min}'s, we have a collection of pairwise independent $(2L)$-blocks. We have shown how to transform

any arbitrary optimal solution into a selection of pairwise independent $(2L)$-blocks. By halving L, we obtain the lemma. □

Example for Lemma 5.2 Let $L = 3$ and consider an optimal selection in the diagram below where crossing intervals are in red. We remove the minimum weight interval from each of $\{I_2, I_4, I_6\}$ and $\{I_7, I_8, I_{10}\}$.



Say $w_2 = \min\{w_2, w_4, w_6\}$ and $w_{10} = \min\{w_7, w_8, w_{10}\}$. After removing intervals I_2 and I_{10}, we can form $(2L-2)$-blocks by using machines spanned by I_2 and I_{10} as "split points" — $\{I_1\}$, $\{I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$, $\{I_{11}\}$.



Implication of Lemma 5.2 We only need to consider a concatenation of independent $\frac{2}{\epsilon}$-blocks. In the spirit of Section 5.1.1 where we concatenate blocks recursively, Im et al. [IMS17] showed that $\frac{2}{\epsilon}$-blocks can be computed in $\mathcal{O}(\log \frac{2}{\epsilon})$ rounds. Then, as we are only dealing with independent blocks, it suffices to approximate the starting and ending points of these blocks by the machine times $t_1, t_2, \ldots, t_m$. Doing so allows us to obtain a smaller problem instance of size $\frac{n}{n^\alpha}$, as we shall see. The above discussion suggests an $\mathcal{O}(1_{\epsilon,\alpha})$ round algorithm, where $1_{\epsilon,\alpha}$ is a constant function of ϵ and α.

Let W be the set of $\mathcal{O}(\log n)$ weight options. Let $\mathrm{OPT}(i, \mu, L)$ be the smallest index j such that there is an L-block of weight $\geqslant \mu$ among $\{I_i, I_{i+1}, \ldots, I_{j-1}\}$ which ends before a_j. Similarly, we denote $D(i, \mu, L)$ as an L-block of weight $\geqslant \mu$ among $\{I_i, I_{i+1}, \ldots, I_{j-1}\}$ which ends before a_j.

A family $\mathcal{F} = \{D(i, \mu, L)\}_{i\in[n],\mu\in W}$ of blocks is said to be an $(1-\gamma)$-*approximate compact family* of L-blocks if $D(i, \mu, L)$ has weight $\geqslant (1-\gamma)\mu$ and $D(i, \mu, L)$ ends *no later than* a_j, for $j = \mathrm{OPT}(i, \mu, L)$. This implies that a selection of intervals remains feasible if an L-block defined by $\mathrm{OPT}(i, \mu, L)$ is replaced by the block $D(i, \mu, L) \in \mathcal{F}$ while not losing "too much weight". By Lemma 5.2, we can construct approximate solutions from $\mathcal{F}$.

Algorithm

Consider the following algorithm:

1. While there are $> S = \widetilde{\mathcal{O}}(n^{\alpha})$ intervals:
   (a) Suppose there are n intervals. Build a compact family $\mathcal{F}$.
   (b) Recall that we only need pairwise independent blocks from the compact family $\mathcal{F}$. For each $\mathcal{O}(\log n)$ possible weights $w \in W$, we only need the blocks corresponding to
   $$D(1, w, L), D(\frac{n}{m} + 1, w, L), D(\frac{2n}{m} + 1, w, L), \ldots$$
   (c) Suppose $D(i, \cdot, L)$ has weight w' and spans machines j to k. Treat $D(i, \cdot, L)$ as an "interval" with weight w' that starts at t_j and ends before t_{k+1}. Treat all blocks likewise.
   (d) Recurse on the new problem instance with $\leqslant \frac{n}{n^{\alpha}}$ "intervals".
2. Solve on a single machine.

Since each iteration reduces the number of intervals by a factor of n^{α}, the algorithm runs for $\mathcal{O}(\frac{1}{\alpha})$ before it fits in a single machine. The construction of compact families $\mathcal{F}$ can be done in constant time using ideas from Section 5.1.1. For details, we refer readers to the section on *Constructing the Desired Compact Family using DP* in Im et al. [IMS17].

Chapter 6

Submodular Maximization

In this chapter, we study algorithms that maximize monotone submodular functions under cardinality constraints. To be precise, given a *ground set* U of n elements, a set function $f : 2^U \to \mathbb{R}^+$, and an integer $k \geqslant 0$, we wish to find a subset $X \subset U$ of size k such that $f(X)$ is maximized. The set function f has the following properties:

Monotone $f(A) \leqslant f(B)$, for any $A \subseteq B \subseteq U$

Submodular $f(A \cup \{e\}) - f(A) \geqslant f(B \cup \{e\}) - f(B)$, for any $A \subseteq B \subseteq U$

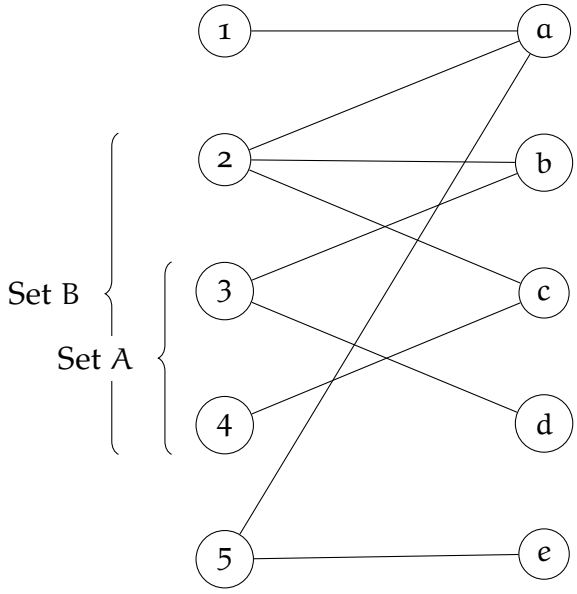
The submodularity of f can be interpreted as "diminishing returns" where the marginal gain for adding an element to a smaller set A is at least the marginal gain for adding the same element to a superset of A. Notationally, we write $f_A(e) = f(A \cup \{e\}) - f(A)$. So, the submodularity property can also be written as $f_A(e) \geqslant f_B(e)$.

Exercise 6.1. Equivalence of submodular property
Consider the following characterization of submodularity. For any $A, B \subseteq U$,

$$f(A) + f(B) \geqslant f(A \cup B) + f(A \cap B)$$

Show that the above is equivalent to $f_A(e) \geqslant f_B(e)$, *for any* $A \subseteq B \subseteq U$.

Example An instance of a submodular maximization problem with cardinality constraints is the k-coverage problem. Consider a bipartite graph $G = (L \cup R, E)$ where the ground set are vertices in L. The goal is to select a set $X \subseteq L$ of size k such that the number of covered elements from R is maximized. In the example below, $A \subseteq B$ with $A = \{3, 4\}$ and $B = \{2, 3, 4\}$. We see that $1 = f_A(\{1\}) \geqslant f_B(\{1\}) = 0$ and $2 = f_A(\{5\}) \geqslant f_B(\{5\}) = 1$.



Although submodular maximization is NP-hard, there is a natural greedy algorithm that achieves a $(1-\frac{1}{e})$-approximation. In the following, we first describe the sequential greedy algorithm `Greedy` in Section 6.1, then introduce two MPC algorithms in Section 6.2 and Section 6.3.

6.1 A greedy sequential algorithm

The sequential algorithm `Greedy` greedily adds, one by one, the element with the largest marginal gain with respect to the current set:

1. $X \leftarrow \emptyset$
2. While $|X| < k$
   (a) $u \leftarrow \operatorname{argmax}_{e \in U \setminus X} f_X(e) = \operatorname{argmax}_{e \in U \setminus X} f(X \cup \{e\}) - f(X)$
   (b) $X \leftarrow X \cup \{u\}$
3. Return X

Claim 6.1. *`Greedy` is a $(1-\frac{1}{e})$-approximation algorithm. To be precise,*

$$f(X) \geqslant \left(1 - (1 - \frac{1}{k})^k\right) \cdot f(\mathrm{OPT})$$

where $X =$ `Greedy`(U) and $\mathrm{OPT} = \mathit{argmax}_{X \subseteq U, |X|=k} f(X) = \{e_1, \ldots, e_k\}$.

Proof. Let X_i be the set computed by `Greedy` in the i^{th} iteration. It suffices to show that $f(X_{i+1}) - f(X_i) \geqslant \frac{1}{k}\Big(f(OPT) - f(X_i)\Big)$ for all $i \in [k]$.

$$
\begin{aligned}
f(OPT) &\leqslant f(X_i \cup OPT) && \text{Monotonicity of } f \\
&= f(X_i) + \Big(f(X_i \cup \{e_1\}) - f(X_i)\Big) && \text{Add items from OPT to } X_i \\
&\quad + \dots \\
&\quad + \Big(f(X_i \cup OPT) - f(X_i \cup OPT \setminus \{e_k\})\Big) \\
&\leqslant f(S_i) + \Big(f(X_i \cup \{e_1\}) - f(X_i)\Big) && \text{Submodularity of } f \\
&\quad + \dots \\
&\quad + \Big(f(X_i \cup \{e_k\}) - f(X_i)\Big) \\
&\leqslant f(X_i) + k \cdot \Big(f(X_{i+1}) - f(X_i)\Big) && \text{Greedy choice of } X_{i+1}
\end{aligned}
$$

Rearranging, we have $f(X_{i+1}) - f(X_i) \geqslant \frac{1}{k}\Big(f(OPT) - f(X_i)\Big)$. One can then rearrange the expression to $f(OPT) - f(X_{i+1}) \leqslant (1 - \frac{1}{k}) \cdot \Big(f(OPT) - f(X_i)\Big)$ and telescope to obtain the claim[1]. □

6.2 Constant approximation in 2 MPC rounds

6.2.1 Overview

Mirzasoleiman et al. [MKSK13] introduced a 2-round MPC called `GreeDI`. `GreeDI` splits the elements *arbitrarily* into machines to compute a greedy solution on the subset of elements, then combine the outputs together. While Mirzasoleiman et al. could not prove very strong bounds, their empirical results for `GreeDI` were promising. Barbosa et al. [BENW15] analyzed a randomized variant of `GreeDI` dubbed `RandGreeDI` which distributes the elements *randomly* across the machines. They were able to prove that `RandGreeDI` only loses a factor of 2 in the approximation factor compared to running `Greedy` on all the elements on a single machine under the assumption that each machine has memory $S \geqslant \max\{k \cdot m, \frac{n}{m}\}$, where m is the number of machines. The assumption is so that a single machine can hold the output of `Greedy` from m machines. We now describe `RandGreeDI` and analyze its approximation guarantees.

[1] Without loss of generality, we may assume that $f(X_0) = f(\emptyset) = 0$.

6.2.2 Algorithm

Let m be the number of machines, each with memory $S \geqslant \max\{k \cdot m, \frac{n}{m}\}$ so that a single machine can hold the output of `Greedy` from m machines. Given a ground set U of n elements, a monotone submodular set function f, and an integer $k \geqslant 0$, `RandGreeDI` works as follows:

1. *Randomly* partition U into $U_1, \ldots, U_m$ and distribute to m machines.
2. In parallel, the i^{th} machine does the following:
   - Compute the set of elements $G_i = \texttt{Greedy}(U_i)$ greedily.
   - Send the set G_i to a coordinator/leader machine M_0.
3. The leader machine machine M_0 computes $G_0 = \texttt{Greedy}(\cup_{i=1}^{m} G_i)$ and outputs the set of elements G_{i^*}, where $i = \text{argmax}_{i \in \{0,1,\ldots,m\}} f(G_i)$.

Exercise 6.2. Deterministic `GreeDI` can be bad
Observe that `RandGreeDI` randomly partitions the elements into the m machines. If one were to arbitrarily partition U across m machines as in `GreeDI`, show that there are instances where `GreeDI` does not get a constant approximation to OPT.

By definition, the submodular set function f is discrete. To analyze `RandGreeDI`, we use a continuous extension of the submodular set functions called the Lovász extension.

6.2.3 The Lovász extension

Instead of considering discrete selection vectors $x \in [0,1]^n$, let us consider continuous selection vectors $x \in [0,1]^n$. Then, the Lovász extension is defined as

$$f^- = \mathbb{E}_{\theta \sim \text{Uniform}(0,1)}(f(\{i : x_i \geqslant \theta\}))$$

Geometrically (See Fig. 6.1), consider a U-dimension ball of radius θ centered at 0 where we round all x_i's within the ball to 0 and all x_i's outside the ball to 1. The Lovász extension is then the expectation of the set function f evaluated over all the roundings of x due to $\theta \sim \text{Uniform}(0,1)$.

The Lovász extension f^- of function f has the following 3 properties:

1. $f^-(\mathbb{1}_X) = f(X)$, where $(\mathbb{1}_X)_i = 1$ if $i \in S$ and $(\mathbb{1}_X)_i = 0$ if $i \notin X$
2. f^- is convex
3. For any constant $c \in [0,1]$, $f^-(cx) \geqslant c \cdot f^-(x)$

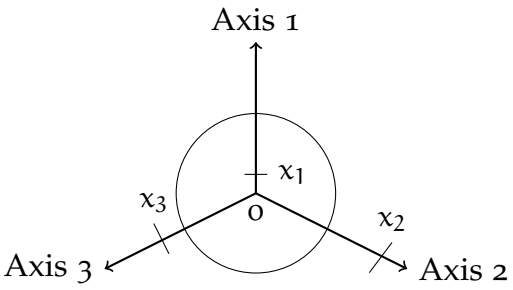
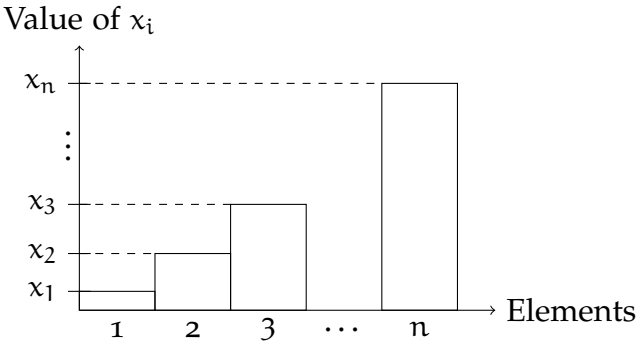


Figure 6.1: A 3-dimensional geometric interpretation of rounding x under the Lovász extension. x_1 is rounded to 0 while x_2 and x_3 are rounded to 1.

Property 1 is self-explanatory: If the probability vector x is binary, then any θ will round x to select X. To see why f^- is convex, we refer readers to Theorem 7 of this lecture notes[2] and this lecture video[3]. We explain Property 3 below. WLOG, suppose $x_1 \leqslant x_2 \leqslant \cdots \leqslant x_n$ in vector $x \in [0,1]^n$.



Since $f^- = \mathbb{E}_{\theta \sim \text{Uniform}(0,1)}(f(\{i : x_i \geqslant \theta\}))$, one can see that

$$\begin{aligned} f^-(x) = \quad & x_1 \cdot f(\{1,2,3,\ldots,n\}) \\ & + (x_2 - x_1) \cdot f(\{2,3,\ldots,n\}) \\ & + (x_3 - x_2) \cdot f(\{3,\ldots,n\}) \\ & + \ldots \\ & + (x_n - x_{n-1}) \cdot f(\{n\}) \\ & + (1 - x_n) \cdot f(\emptyset) \end{aligned}$$

The first n terms scale linearly with a multiplicative constant $c \in [0,1]$ while $(1 - c \cdot x_n) \geqslant c \cdot (1 - x_n)$, yielding the third property of f^-. Observe that if $f(\emptyset) = 0$, then f^- scales linearly and $f^-(cx) = c \cdot f^-(x)$.

[2] http://people.csail.mit.edu/moitra/docs/6854lec13.pdf
[3] https://youtu.be/SDv18sickuM?t=2153

Lemma 6.2. *Let c be a constant and $p \in [0,1]^n$ be a vector. If X is a random subset of U such that $\mathbb{E}[\mathbb{1}_X] = cp$, then $\mathbb{E}[f(X)] \geqslant c \cdot f^-(p)$.*

Proof. We use the three property of the Lovász extension f^-.

$$\begin{aligned}
\mathbb{E}[f(X)] &= \mathbb{E}[f^-(\mathbb{1}_X)] && \text{Property 1 of } f^- \\
&\geqslant f^-(\mathbb{E}[\mathbb{1}_X]) && \text{Property 2 of } f^-\text{, with Jensen's inequality} \\
&= f^-(cp) && \text{Assumption that } \mathbb{E}[\mathbb{1}_X] = cp \\
&\geqslant c \cdot f^-(p) && \text{Property 3 of } f^-
\end{aligned}$$

□

6.2.4 Analysis of `RandGreeDI`

We will prove that `RandGreeDI` is a $\frac{1}{2}(1-\frac{1}{e})$-approximation algorithm, in expectation over the random partitions of U. To do so, we do the following:

- Define $\mathcal{D}(m)$, a distribution over the random partitionings of U
- Lower bound $\mathbb{E}[f\left(\texttt{Greedy}(\cup_{i=1}^m G_i)\right)]$ in Lemma 6.3
- Lower bound $\mathbb{E}[f(G_i)]$ for each machine i in Lemma 6.4
- Combine the bounds from Lemma 6.3 and Lemma 6.4

Let $\mathcal{D}(m)$ be the distribution over random subsets of U where each element is included independently with probability $\frac{1}{m}$. Intuitively, the distribution $\mathcal{D}(m)$ represents what a machine receives under `RandGreeDI`'s random partitioning of U. Consider vector $p = (p_1, \ldots, p_n) \in [0,1]^n$ where

$$p_i = \begin{cases} \Pr_{A \sim \mathcal{D}(m)}[i \in \texttt{Greedy}(A \cup \{i\})] & \text{if element } i \text{ is in OPT} \\ 0 & \text{if element } i \text{ is not in OPT} \end{cases}$$

Lemma 6.3. *Let U_i be the set of elements given to machine i and G_i be the set returned by $\texttt{Greedy}(U_i)$. Then, $\mathbb{E}[f\left(\texttt{Greedy}(\cup_{i=1}^m G_i)\right)] \geqslant (1-\frac{1}{e}) \cdot f^-(p)$.*

Proof. For an element $e \in \text{OPT}$,

$$\begin{aligned}
\Pr[e \in \cup_{i=1}^m G_i \mid e \in U_i] &= \Pr[e \in G_i \mid e \in U_i] \\
&= \Pr_{A \sim \mathcal{D}(m)}[e \in \texttt{Greedy}(A) \mid e \in A] \\
&= \Pr_{B \sim \mathcal{D}(m)}[e \in \texttt{Greedy}(B \cup \{e\})] \\
&= p_e
\end{aligned}$$

The first equality is because $U_1, \ldots, U_n$ is a partition of U, so $e \in \cup_{i=1}^{m} G_i$ if and only if $e \in G_i = \texttt{Greedy}(U_i)$, for the partition U_i that the element e was assigned to. The second equality is because the distribution of U_i is the same as $\mathcal{D}(m)$. The third inequality is because the distribution of $A \sim \mathcal{D}(m)$ conditioned on $e \in A$ is identical to the distribution of $B \cup \{e\}$ where $B \sim \mathcal{D}(m)$. Thus, $\Pr[e \in (\cup_{i=1}^{m} G_i \bigcap \text{OPT})] = p_e$ and $\mathbb{E}[\mathbb{1}_{\cup_{i=1}^{m} G_i \bigcap \text{OPT}}] = p$.

$$\begin{aligned}
\mathbb{E}[f\Big(\texttt{Greedy}(\cup_{i=1}^{m} G_i)\Big)] &\geqslant (1-\frac{1}{e}) \cdot \mathbb{E}[f(\text{OPT})] && \text{By Claim 6.1} \\
&\geqslant (1-\frac{1}{e}) \cdot \mathbb{E}[f\Big(\cup_{i=1}^{m} G_i \bigcap \text{OPT}\Big)] && \text{Monotonicity of } f \\
&\geqslant (1-\frac{1}{e}) \cdot f^{-}(p) && \text{By Lemma 6.2}
\end{aligned}$$

□

Lemma 6.4. *Let U_i be the set of elements given to machine i and G_i be the set returned by $\texttt{Greedy}(U_i)$. Then, $\mathbb{E}[f(G_i)] \geqslant (1-\frac{1}{e}) \cdot f^{-}(\mathbb{1}_{\text{OPT}} - p)$.*

Proof. Consider machine i that receives set of elements U_i. Let O_i be the set of individual elements from OPT, each of which will not be selected by Greedy if it is given together with U_i. That is,

$$O_i = \{e \in \text{OPT} : e \notin \texttt{Greedy}(U_i \cup \{e\})\}$$

One can show that $\texttt{Greedy}(U_i \cup O_i) = \texttt{Greedy}(U_i) = G_i$ (See Exercise 6.3). So,

$$\begin{aligned}
f(G_i) &= f(\texttt{Greedy}(U_i \cup O_i)) \\
&\geqslant (1-\frac{1}{e}) \cdot f\Big((U_i \cup O_i) \bigcap \text{OPT}\Big) && \text{By Claim 6.1} \\
&\geqslant (1-\frac{1}{e}) \cdot f(O_i) && \text{Monotonicity of } f
\end{aligned}$$

Taking expectations, we have

$$\begin{aligned}
\mathbb{E}[f(G_i)] &\geqslant (1-\frac{1}{e}) \cdot \mathbb{E}[f(O_i)] && \text{From above} \\
&\geqslant (1-\frac{1}{e}) \cdot f^{-}(\mathbb{1}_{\text{OPT}} - p) && \text{By Lemma 6.2}
\end{aligned}$$

□

Exercise 6.3. Irrelevance of unselected elements
Consider machine i that receives set of elements U_i. Let O_i be the set of individual elements from OPT, each of which will not be selected by `Greedy` if it is given together with U_i. That is, $O_i = \{e \in \mathrm{OPT} : e \notin \texttt{Greedy}(U_i \cup \{e\})\}$. Show that

$$\texttt{Greedy}(U_i \cup O_i) = \texttt{Greedy}(U_i) = G_i$$

Hint *Derive a contradiction by using the greedy process of `Greedy`.*

Theorem 6.5. *In expectation (over the random partitions of U), `RandGreeDI` is an $\frac{1}{2}(1-\frac{1}{e})$-approximation algorithm.*

Proof. Let U_i be the set of elements given to machine i and G_i be the set returned by $\texttt{Greedy}(U_i)$. From Lemma 6.4, we know that

$$\max_{i\in\{1,\dots,m\}} \mathbb{E}[f(G_i)] \geqslant (1-\frac{1}{e}) \cdot f^-(\mathbb{1}_{\mathrm{OPT}} - p)$$

Furthermore, for $x, y \in [0,1]^n$, we have $\frac{1}{2}f^-(x) + \frac{1}{2}f^-(y) \geqslant f^-(\frac{1}{2}x + \frac{1}{2}y))$ because f^- is convex. Thus,

$$\begin{aligned}
& \max\left\{\mathbb{E}[f\left(\texttt{Greedy}(\cup_{i=1}^m G_i)\right)], \max_{i\in\{1,\dots,m\}} \mathbb{E}[f(G_i)]\right\} \\
\geqslant\ & \frac{1}{2}\cdot\left[\mathbb{E}[f\left(\texttt{Greedy}(\cup_{i=1}^m G_i)\right)] + \max_{i\in\{1,\dots,m\}} \mathbb{E}[f(G_i)]\right] && \text{Maximum} \geqslant \text{Average} \\
\geqslant\ & \frac{1}{2}\cdot\left[(1-\frac{1}{e})\cdot f^-(p) + (1-\frac{1}{e})\cdot f^-(\mathbb{1}_{\mathrm{OPT}} - p)\right] && \text{Lemma 6.3 and Lemma 6.4} \\
\geqslant\ & \frac{1}{2}(1-\frac{1}{e})\cdot f^-(\mathbb{1}_{\mathrm{OPT}}) && \text{Convexity of } f^- \\
=\ & \frac{1}{2}(1-\frac{1}{e})\cdot f(\mathrm{OPT}) && \text{Property 1 of } f^-
\end{aligned}$$

□

In their work, Barbosa et al. [BENW15] also modified `RandGreeDI` for non-monotone submodular functions f. This is beyond the scope of this chapter and interested readers may refer to Section 4 of the paper.

Exercise 6.4. 4-approximation to the k-center problem
Consider a set X of n points in Euclidean space. The k-center problem is to find a subset $C \subseteq X$ of size k such that the maximum distance of any point to the nearest point in C is minimized. The k-center problem is known to be NP-hard but there is a sequential 2-approximation algorithm that runs in $O(nk)$ time.

Given machine memory of $S \geqslant \max\{k\cdot m, \frac{n}{m}\}$, design a 2-round MPC algorithm that outputs a 4-approximation to the k-center problem.

Hint *Use the outline of* `RandGreeDI`*.*

6.3 Optimal approximation via `Sample-and-Prune`

Previously, we saw a 2-round MPC algorithm `RandGreeDI` which computes an $\frac{1}{2}(1-\frac{1}{e})$-approximation using machine memory $S \geqslant \max\{k \cdot m, \frac{n}{m}\}$. Kumar et al. [KMVV15] introduced a threshold-based MPC algorithm `Threshold-MPC` that computes an $\frac{1}{1+\epsilon}(1-\frac{1}{e})$-approximation with machine memory $S = \widetilde{\mathcal{O}}(kn^{\alpha})$, for any constant $\alpha \in (0,1)$. Unlike `RandGreeDI`, `Threshold-MPC` runs in $\mathcal{O}(\frac{1}{\epsilon\alpha}\log k)$ rounds.

As a warm up, we first describe a $\frac{1}{2}$-approximation sequential algorithm `Threshold-Greedy`, and how to simulate it in a constant number of MPC rounds via `SampleAndPrune`. One can then obtain an $\frac{1}{1+\epsilon}(\frac{1}{2})$-approximation MPC algorithm that runs in constant rounds. To obtain an $\frac{1}{1+\epsilon}(1-\frac{1}{e})$-approximation, we introduce a $\frac{1}{1+\epsilon}$-approximate version of `Greedy` and describe `Threshold-MPC` which simulates some sequential steps of it with the above `SampleAndPrune` technique.

6.3.1 Warm up

The main purpose of this warm up is to introduce the framework of `Sample-and-Prune`. For now, let us assume we know the value of $f(OPT)$.

The `Threshold-Greedy` algorithm

Given a set of elements U and a threshold τ, `Threshold-Greedy` does the following:

1. $X \leftarrow \emptyset$
2. While $|X| < k$, and $\exists e \in U \setminus X$ such that $f_X(e) \geqslant \tau$
   - $X \leftarrow X \cup \{e\}$
3. Return X

Claim 6.6. *If X is the set returned by running* `Threshold-Greedy` *on the set of all elements U and the threshold $\tau = \frac{f(OPT)}{2k}$, then $f(X) \geqslant \frac{f(OPT)}{2}$.*

Proof. Recall that `Threshold-Greedy` terminates if either $|X| = k$ or there are no more elements that have marginal gain at least τ. Since $|OPT| = k$ and $|X| \leqslant k$, we have $0 < |OPT \setminus X| \leqslant k$ within `Threshold-Greedy`.

Suppose $f(X) < \frac{f(OPT)}{2}$. Then, $f_X(OPT \setminus X) = f(OPT \cup X) - f(X) \geqslant f(OPT) - f(X) > \frac{f(OPT)}{2}$. Since f is submodular, we know that there is an element $e \in OPT \setminus X \subseteq U \setminus X$ with marginal gain at least $\frac{f(OPT)}{2 \cdot |OPT \setminus X|} > \frac{f(OPT)}{2k} = \tau$ by similar arguments as in the proof of Claim 6.1.

This means that if the returned X has size $< k$, then $f(X) \geqslant \frac{f(OPT)}{2}$. On the other hand, if the returned X is of size k, then the last element added to X has marginal gain at least $\frac{f(OPT)}{2 \cdot |OPT \setminus X|} = \frac{f(OPT)}{2}$. □

The `Sample-and-Prune` technique

`Threshold-Greedy` is inherently sequential and a direct adaptation to the MPC setting would require $\Omega(k)$ rounds because one has to check the marginal gains of each element after adding an element to the set X. On the other hand, `Sample-and-Prune` provides a way to execute `Threshold-Greedy` in the MPC setting in constant number of rounds. For cleanliness, we shall describe `Sample-and-Prune` assuming machine memory $S \geqslant k \cdot \sqrt{n}$. It can be shown that a smaller machine memory of $S \in \widetilde{O}(kn^{\alpha})$ also works (See Lemma 6.9).

Given a set of elements U and a threshold τ, `Sample-and-Prune` does the following:

1. $T \leftarrow$ Sample each element with probability $\frac{1}{\sqrt{n}}$.
2. $X \leftarrow$ `Threshold-Greedy`(T, τ).
3. $R \leftarrow \{e \in U \setminus X : f_X(e) \geqslant \tau\}$
4. Return X, R

Intuitively, set R is the set of remaining elements with "sufficiently large" marginal gains with respect to output X. Observe that elements in T that are not chosen by `Threshold-Greedy` will have small marginal gains.

Claim 6.7. *`Sample-and-Prune` can be implemented in 2 MPC rounds with machine memory $S \geqslant k \cdot \sqrt{n}$.*

Proof. Sampling of T can be done independently across all machines. With high probability, $|T| \leqslant S \in \widetilde{O}(\sqrt{n})$ so T can be sent to a single machine. The set X, of size $\leqslant k$, is computed locally on a single machine and can be

broadcasted to all other machines in 1 round. Each machine then locally drops elements which have marginal contributions less than τ. □

Claim 6.8. *If R is the returned set from `Sample-and-Prune` on n elements, then $|R| < 2k\sqrt{n}\log n$ with high probability.*

Proof. Let us upper bound the negation of the claim. Consider an arbitrary output X from `Threshold-Greedy` and suppose $|R| \geqslant 2k\sqrt{n}\log n$. That is, for this fixed output X, we suppose there are at least $2k\sqrt{n}\log n$ elements that have marginal gains larger than τ with respect to X.

$$\Pr[R \cap T = \emptyset] = (1 - \frac{1}{\sqrt{n}})^{|R|} \leqslant e^{-\frac{|R|}{\sqrt{n}}} \leqslant e^{-2k\log n} = n^{-2k}$$

Taking the union bound over all at most $\sum_{i=0}^{k}\binom{n}{i} \leqslant 2n^k$ possible outputs X, the probability that there is some instance where $|R| \geqslant 2k\sqrt{n}\log n$ is at most $2n^k \cdot n^{-2k} \leqslant 2n^{-k}$. In other words, $\Pr\left[|R| < 2k\sqrt{n}\log n\right] \geqslant 1 - 2n^{-k}$. □

Lemma 6.9. *Suppose we have machine memory $S = \widetilde{O}(kn^\alpha)$ for some $\alpha \in (0,1)$. Modify `Sample-and-Prune` to sample each element independently with probability $\frac{1}{n^{1-\alpha}}$ into T. Prove that if $|T| = n$, then $|R| \in O(kn^{1-\alpha}\log n)$ w.h.p.*

Suppose we have machine memory $S = \widetilde{O}(kn^\alpha)$ for some $\alpha \in (0,1)$. Consider running `Sample-and-Prune` for $O(\frac{1}{\alpha})$ repetitions as follows:

1. $X \leftarrow \emptyset$
2. $R_0 \leftarrow U$
3. For $i \in \{1, \ldots, O(\frac{1}{\alpha})\}$:
   (a) $X_i, R_i \leftarrow$ `Sample-and-Prune`(R_{i-1}, τ)
   (b) $X \leftarrow X \cup X_i$
4. Return X

Corollary 6.10. *Show that after $T \in O(\frac{1}{\alpha})$, the set X contains all elements that contribute a marginal gain of at least τ and $R_T = \emptyset$.*

Exercise 6.5. Using a smaller machine memory *Prove Lemma 6.9, then use it to prove Corollary 6.10.*

Hint *Modify the proofs of Claim 6.7 and Claim 6.8 appropriately to prove Lemma 6.9 under the modified sampling probability.*

Removing the assumption of knowing $f(\mathsf{OPT})$ Let $\Delta = \max_{e \in U} f(\{e\})$, then $f(\mathsf{OPT}) \leqslant k \cdot \Delta$. By guessing $f(\mathsf{OPT}) \in \{\frac{k\Delta}{(1+\epsilon)^0}, \frac{k\Delta}{(1+\epsilon)^1}, \ldots, \frac{k\Delta}{(1+\epsilon)^{\log_{1+\epsilon} k}}\}$, we lose a factor of at most $\frac{1}{1+\epsilon}$. That is, one can obtain an $\frac{1}{1+\epsilon}(\frac{1}{2})$-approximation MPC algorithm that runs in constant rounds by trying $\mathcal{O}(\log k)$ guesses for $f(\mathsf{OPT})$ in parallel.

Exercise 6.6. Selecting the k largest elements in constant rounds
Given machine memory $S \geqslant k \cdot n^{\alpha}$ for some constant $\alpha \in (0,1)$, design a $\mathcal{O}(\frac{1}{\alpha})$-round algorithm that returns the k largest elements out of n elements.

Hint *Use the outline of `Sample-and-Prune`.*

Remark Observe the similarities between Exercise 6.6 and Exercise 2.1 on sorting n items. `Sample-and-Prune` can also be applied to give a constant round MPC algorithms for selecting the k^{th} largest item amongst n items.

6.3.2 The `Threshold-MPC` algorithm

In the warm up, we saw that one could design a constant round MPC algorithm `Sample-and-Prune` to compute an $\frac{1}{1+\epsilon}(\frac{1}{2})$-approximation. To improve upon the approximation factor, `Threshold-MPC` uses the `SampleAndPrune` technique to simulate some sequential steps of the $\frac{1}{1+\epsilon}$-approximate version of `Greedy` we describe below.

Let $\Delta = \max_{e \in U} f(\{e\})$ be the largest possible marginal gain. Since $f(\mathsf{OPT}) \leqslant k \cdot \Delta$, all elements with marginal gains smaller than $\frac{\epsilon\Delta}{k}$ can be ignored when computing a $\frac{1}{1+\epsilon}$-approximation to OPT. Observe that marginal gains can only decrease as the set X grows. The following algorithm is a $\frac{1}{1+\epsilon}$-approximate version of `Greedy`:

1. $\Delta \leftarrow \max_{e \in U} f(\{e\})$
2. $X \leftarrow \emptyset$
3. For $i \in \{0, 1, 2, \ldots, \frac{\log_{1+\epsilon} k}{\epsilon}\}$:
   (a) $\tau \leftarrow \frac{\Delta}{(1+\epsilon)^i}$
   (b) While $|X| < k$, and $\exists e \in U \setminus X$ such that $f_X(e) \geqslant \tau$
   - $X \leftarrow X \cup \{e\}$
4. Return X

Instead of running the while loop, `Threshold-MPC` runs `SampleAndPrune` repeatedly for $\mathcal{O}(\frac{1}{\alpha})$ iterations to extract the set X' of all elements that contribute a marginal gain of at least τ (See Corollary 6.10):

1. $\Delta \leftarrow \max_{e \in U} f(\{e\})$
2. $X \leftarrow \emptyset$
3. For $i \in \{0, 1, 2, \ldots, \frac{\log_{1+\epsilon} k}{\epsilon}\}$:
   (a) $\tau \leftarrow \frac{\Delta}{(1+\epsilon)^i}$
   (b) $X' \leftarrow \mathcal{O}(\frac{1}{\alpha})$ repetitions of `Sample-and-Prune`(U, τ)
   (c) $X \leftarrow X \cup X'$
4. Return X

Claim 6.11. *`Threshold-MPC` runs in $\mathcal{O}(\frac{1}{\epsilon\alpha} \log k)$ MPC rounds with machine memory $S \geqslant k \cdot n^{\alpha}$ for any constant $\alpha \in (0, 1)$.*

Proof. Each invocation of `Sample-and-Prune` runs in 2 MPC rounds. There are $\mathcal{O}(\frac{1}{\epsilon\alpha} \log k)$ invocations of `Sample-and-Prune`. □

Claim 6.12. *`Threshold-MPC` is a $\frac{1}{1+\epsilon}(1 - \frac{1}{e})$-approximation algorithm.*

Proof. `Threshold-MPC` emulates `Greedy` except that at each step, we may be adding an element whose marginal gain is a factor $\frac{1}{1+\epsilon}$ smaller. □

6.4 Optimal approximation in constant time

In the earlier sections, we saw algorithms `RandGreeDI` from Section 6.2 and `Threshold-MPC` from Section 6.3. `RandGreeDI` computes an $\frac{1}{2}(1 - \frac{1}{e})$-approximation using machine memory $S \geqslant \max\{k \cdot m, \frac{n}{m}\}$ in 2 rounds. `Threshold-MPC` computes an $\frac{1}{1+\epsilon}(1 - \frac{1}{e})$-approximation using machine memory $S \geqslant \widetilde{\mathcal{O}}(kn^{\alpha})$ in $\mathcal{O}(\frac{1}{\epsilon\alpha} \log k)$ rounds.

In this section, we look at two pieces of work, due to Barbosa et al. [BENW16] and Liu and Vondrak [LV18], that bring down the round complexity for $(1 - \frac{1}{e} - \mathcal{O}(\epsilon))$-approximation to a constant. For cleanliness, we consider a setting with $\mathcal{O}(\sqrt{\frac{n}{k}})$ machines, each with memory of $S \in \widetilde{\Theta}(\sqrt{nk})$ in the rest of this section.

6.4.1 `ParallelAlg` by Barbosa et al. [BENW16]

Barbosa et al. [BENW16] describe a framework for converting a sequential α-approximate algorithm $\mathcal{A}$ for monotone submodular functions into an $(\alpha - \epsilon)$-approximate algorithm in MPC that runs in $\mathcal{O}(\frac{1}{\epsilon})$ rounds. In our context, $\mathcal{A} = \texttt{Greedy}$ which has an $\alpha = (1 - \frac{1}{e})$ approximation.

The algorithm `ParallelAlg` is similar in spirit to `RandGreeDI` which distributes elements from U across m machines and computes `Greedy` on those subset of elements locally. In addition to this random process, `ParallelAlg` maintains a pool $C \subseteq U$ of "good" elements. This pool is given to the machines together with the random distribution of elements, and we grow this pool over time by adding elements chosen by `Greedy` across all machines. To amplify success probability, a round involves executing multiple groups in parallel.

Algorithm

Let $g = \Theta(\frac{1}{\epsilon \cdot (1 - \frac{1}{e})})$ be the number of groups of m machines such that gm is the total number of machines, and C_r be the pool of "good" elements computed after round r. We now describe the algorithm `ParallelAlg`:

1. $C_0 \leftarrow \emptyset$
2. For round $r \in \{1, 2, \ldots, \mathcal{O}(\frac{1}{\epsilon})\}$:
   (a) Within each group, distribute the elements of U uniformly at random across the m machines. In round r, let $X^{(j)}_{i,r}$ be the set of elements sent to i^{th} machine in the j^{th} group.
   (b) $G^{(j)}_{i,r} \leftarrow \texttt{Greedy}(X^{(j)}_{i,r} \cup C_{r-1})$, for each machine i in each group j.
   (c) $C_r \leftarrow C_{r-1} \cup \left(G^{(1)}_{1,r} \cup \cdots \cup G^{(1)}_{m,r}\right) \cup \cdots \cup \left(G^{(g)}_{1,r} \cup \cdots \cup G^{(g)}_{m,r}\right)$
3. Return the best output observed throughout all executions of `Greedy`.

Since the size of C_r grows by at most $kgm \in \widetilde{\Theta}(\sqrt{nk})$ additional elements per round and there are only $\mathcal{O}(\frac{1}{\epsilon})$ rounds, the set $X^{(j)}_{i,r} \cup C_r$ can always fit in memory of a machine. To avoid duplication of the data, one can also run each iteration g times to simulate the parallel execution of g groups. Since g is a constant, the resultant algorithm will still take a constant number of rounds.

Analysis

For analysis, we fix an arbitrary round r, machine i, group j, and pool $\widehat{C}_{r-1} \subseteq U$. Let $\mathcal{D}(m)$ be the distribution over random subsets of U where each element is included independently with probability $\frac{1}{m}$. We define

$$p_r(e) = \begin{cases} \Pr_{A\sim\mathcal{D}(m)}[e \in \texttt{Greedy}(\widehat{C}_{r-1} \cup X^{(j)}_{i,r} \cup \{e\})] & \text{if } e \in \mathrm{OPT} \setminus C_{r-1} \\ 0 & \text{if } e \notin \mathrm{OPT} \setminus C_{r-1} \end{cases}$$

So, an element $e \in \mathrm{OPT} \setminus C_{r-1}$ will be added to the pool C_r with probability $1-(1-p_r(e))^g$. We first give an overview of the proof idea by assuming that $p_r(e) \in \{0,1\}$ for all elements.

The idea is to argue that in each round either `ParallelAlg` finds an $(1-\frac{1}{e}-\mathcal{O}(\epsilon))$-approximate solution, or it manages to grow the pool to gain an $\mathcal{O}(\epsilon)$ fraction of $f(\mathrm{OPT})$. Consider sets $P_r = \{e \in \mathrm{OPT} \setminus \widehat{C}_{r-1} \mid p_r(e) = 0\}$ and $Q_r = \{e \in \mathrm{OPT} \setminus \widehat{C}_{r-1} \mid p_r(e) = 1\}$. One can show these two inequalities:

$$f\Big(\texttt{Greedy}(\widehat{C}_{r-1} \cup X^{(j)}_{i,r} \cup P_r)\Big) \geqslant (1-\frac{1}{e}) \cdot f\Big((\mathrm{OPT} \cap \widehat{C}_{r-1}) \cup P_r\Big)$$

$$f\Big(Q_r \cup (\mathrm{OPT} \cap \widehat{C}_{r-1})\Big) - f\Big((\mathrm{OPT} \cap \widehat{C}_{r-1})\Big) \geqslant f(\mathrm{OPT}) - f\Big((\mathrm{OPT} \cap \widehat{C}_{r-1}) \cup P_r\Big)$$

Thus, if $f\Big((\mathrm{OPT} \cap \widehat{C}_{r-1}) \cup P_r\Big) \geqslant (1-\epsilon) \cdot f(\mathrm{OPT})$, then `ParallelAlg` finds an $(1-\frac{1}{e}-\mathcal{O}(\epsilon))$-approximate solution. Otherwise, `ParallelAlg` manages to grow the pool to gain an $\mathcal{O}(\epsilon)$ fraction of $f(\mathrm{OPT})$.

Using the above proof idea, let us handle the case where $p_r(e) \in [0,1]$. We prove a useful lemma using the Lovász extension (See Section 6.2.3) and a theorem that tells us that in expectation, `ParallelAlg` makes "good progress" in each round. An application of Markov's inequality then tells us that that `ParallelAlg` returns an $(1-\frac{1}{e}-\mathcal{O}(\epsilon))$-approximation to OPT with constant probability.

Lemma 6.13. *Consider two sets S and T. Let $R \subseteq T$ be a subset where each element of T is included into R with probability p. Then,*

$$\mathbb{E}[f(R \cup S)] \geqslant p \cdot f(T \cup S) + (1-p) \cdot f(S)$$

Proof. Fix a set $R \subseteq T$.

$$\begin{aligned}
\mathbb{E}[f(R \cup S)] &= \mathbb{E}[f^-(\mathbb{1}_{R\cup S})] && \text{Property 1 of } f^- \\
&\geqslant f^-(\mathbb{E}[\mathbb{1}_{R\cup S}]) && \text{Property 2 of } f^-\text{, with Jensen's inequality} \\
&= f^-(\mathbb{E}[\mathbb{1}_{R\setminus S}] + \mathbb{1}_S) && \text{Separating deterministic selection of } S \\
&\geqslant f^-(p \cdot \mathbb{1}_{T\setminus S} + \mathbb{1}_S) && \text{Sampling probability into } R \\
&= p \cdot f(\mathbb{1}_{T\cup S}) + (1-p) \cdot f(\mathbb{1}_S) && \text{Definition of } f^-
\end{aligned}$$

□

Theorem 6.14. *For round r and pool $\widehat{C}_{r-1} \subseteq U$, one of the following holds:*

1. *For some machine i in some group j,*
   $\mathbb{E}[f(\textit{Greedy}(X_{i,r}^{(j)} \cup C_{r-1}) \mid C_{r-1} = \widehat{C}_{r-1}] \geqslant (1-\epsilon)^2 \cdot (1-\frac{1}{e}) \cdot f(OPT)$
2. $\mathbb{E}[f(C_r \cap OPT) \mid C_{r-1} = \widehat{C}_{r-1}] - f(\widehat{C}_{r-1} \cap OPT) \geqslant \frac{\epsilon}{2} \cdot f(OPT)$

Proof. We consider the following sets:

- $P_r = \{e \in OPT \setminus \widehat{C}_{r-1} \mid p_r(e) < \epsilon\}$
- $Q_r = \{e \in OPT \setminus \widehat{C}_{r-1} \mid p_r(e) \geqslant \epsilon\}$
- $P'_r = \{e \in P_r \mid e \notin \texttt{Greedy}(\widehat{C}_{r-1} \cup X_{i,r}^{(j)} \cup \{e\})\}$
- $Q'_r = Q_r \cap \{\cup_{i=1}^{gm} \texttt{Greedy}(\widehat{C}_{r-1} \cup X_{i,r}^{(j)})\}$

By definition, the sets P_r and Q_r partition the set of elements $OPT \setminus \widehat{C}_{r-1}$. Consider the following condition $(\star)$:

$$f\Big((OPT \cap \widehat{C}_{r-1}) \cup P_r\Big) \geqslant (1-\epsilon) \cdot f(OPT)$$

Suppose condition $(\star)$ is satisfied. We now show that statement (1) of the theorem holds. Since $\Pr[e \in P'_r \mid e \in P_r] \geqslant 1-\epsilon$,

$$\begin{aligned}
& \mathbb{E}[f(\texttt{Greedy}(\widehat{C}_{r-1} \cup X_{i,r}^{(j)}))] && \\
= \; & \mathbb{E}[f(\texttt{Greedy}(\widehat{C}_{r-1} \cup X_{i,r}^{(j)} \cup P'_r))] && \text{See Exercise 6.3 with } O_i = P'_r \\
\geqslant \; & \mathbb{E}[f(\texttt{Greedy}\Big((OPT \cap \widehat{C}_{r-1}) \cup P'_r\Big))] && f \text{ is monotone} \\
\geqslant \; & (1-\frac{1}{e}) \cdot \mathbb{E}[f\Big((OPT \cap \widehat{C}_{r-1}) \cup P'_r\Big)] && (1-\frac{1}{e})\text{-approximation of } \texttt{Greedy} \\
\geqslant \; & (1-\epsilon) \cdot (1-\frac{1}{e}) \cdot f\Big((OPT \cap \widehat{C}_{r-1}) \cup P_r\Big) && \text{Lemma 6.13 with } P'_r \subseteq P_r \\
\geqslant \; & (1-\epsilon)^2 \cdot (1-\frac{1}{e}) \cdot f(OPT) && \text{If } (\star) \text{ is satisified}
\end{aligned}$$

Hence, statement (1) holds. Now, suppose condition $(\star)$ is not satisfied. We now show that statement (2) of the theorem holds.

$$\begin{aligned}
& f\Big(Q_r \cup (OPT \cap \widehat{C}_{r-1})\Big) - f\Big((OPT \cap \widehat{C}_{r-1})\Big) && \\
\geqslant \; & f\Big(P_r \cup Q_r \cup (OPT \cap \widehat{C}_{r-1})\Big) - f\Big((OPT \cap \widehat{C}_{r-1}) \cup P_r\Big) && f \text{ is submodular} \\
\geqslant \; & f(OPT) - f\Big((OPT \cap \widehat{C}_{r-1}) \cup P_r\Big) && P_r \cup Q_r = OPT \setminus \widehat{C}_{r-1}
\end{aligned}$$

Since $\Pr[e \in Q'_r \mid e \in Q_r] = 1-(1-p_r(e))^g \geqslant 1-\frac{1}{e} \geqslant \frac{1}{2}$, by applying Lemma 6.13 with $Q'_r \subseteq Q_r$, we see that

$$\mathbb{E}[f\Big(Q'_r \cup (OPT \cap \widehat{C}_{r-1})\Big)] \geqslant \frac{1}{2} \cdot f\Big(Q_r \cup (OPT \cap \widehat{C}_{r-1})\Big) + \frac{1}{2} \cdot f\Big(OPT \cap \widehat{C}_{r-1}\Big)$$

By definition, $\Big(Q'_r \cup (OPT \cap \widehat{C}_{r-1})\Big) \subseteq C_r \cap OPT$. Thus,

$$\begin{aligned}
& \mathbb{E}[f(C_r \cap OPT) \mid C_{r-1} = \widehat{C}_{r-1}] - f(\widehat{C}_{r-1} \cap OPT) \\
\geqslant\ & \mathbb{E}[f\Big(Q'_r \cup (OPT \cap \widehat{C}_{r-1})\Big)] - f(OPT \cap \widehat{C}_{r-1}) && f \text{ is monotone} \\
\geqslant\ & \frac{1}{2} \cdot \Big[f\Big(Q_r \cup (OPT \cap \widehat{C}_{r-1})\Big) - f(OPT \cap \widehat{C}_{r-1})\Big] && \text{From above} \\
\geqslant\ & \frac{1}{2} \cdot \Big[f(OPT) - f\Big((OPT \cap \widehat{C}_{r-1}) \cup P_r\Big)\Big] && \text{From above} \\
\geqslant\ & \frac{\epsilon}{2} \cdot f(OPT) && \text{If } (\star) \text{ is not satisified}
\end{aligned}$$

Hence, statement (2) holds. □

6.4.2 A new analysis by Liu and Vondrak [LV18]

Liu and Vondrak [LV18] defined a slight variation to the algorithms of Section 6.3 and gave a new analysis which showed that their algorithm achieves an $(1-\frac{1}{e}-\mathcal{O}(\epsilon))$-approximation in $\mathcal{O}(\frac{1}{\epsilon})$ rounds. We first describe their entire algorithm, then describe their analysis.

Throughout this section, we assume that $f(OPT)$ is known. As seen in earlier sections, for an $(1-\epsilon-\mathcal{O}(\epsilon))$-approximation to $f(OPT)$, one can remove this assumption by running $\mathcal{O}(\log k)$ guesses for $f(OPT)$ in parallel.

Algorithm

Given a set S, a partial greedy solution G with $|G| \leqslant k$, and a threshold τ, algorithm `Threshold-Greedy` adds to G, one by one, elements from S with marginal gain larger than τ with respect to the current G.

1. While $|G| < k$, and $\exists e \in S$ such that $f_G(e) \geqslant \tau$
   - $G \leftarrow G \cup \{e\}$
2. Return G

Given a set S, a partial greedy solution G, and a threshold τ, algorithm `Threshold-Filter` removes from set S any element with marginal gain less than τ with respect to G.

1. $X \leftarrow S$
2. For $e \in S$
   - If $f_G(e) < \tau$, $X \leftarrow X \setminus \{e\}$
3. Return X

Using `Threshold-Greedy` and `Threshold-Filter` above, given a set of elements U and an approximation ϵ, `Threshold-MPC` does the following:

1. $G \leftarrow \emptyset$
2. For round $r \in \{1, \ldots, t \in \mathcal{O}(\frac{1}{\epsilon})\}$
   (a) Set threshold for current round $\tau_r \leftarrow (1 - \frac{1}{t+1})^r \cdot \frac{f(OPT)}{k}$
   (b) $S \leftarrow$ Sample elements from U independently w.p. $p = 4\sqrt{\frac{k}{n}}$.
   (c) $U_1, \ldots, U_m \leftarrow$ Partition U uniformly at random into m sets.
   (d) On a single central machine M_0
      - $G \leftarrow$ `Threshold-Greedy`(S, G, τ_r)
      - Send U_i, G, and τ to machine i, for $i \in \{1, \ldots, m\}$.
   (e) For each machine $i \in [m]$ that receives U_i, G, and τ:
      - If $|G| = k$, set $R_i \leftarrow \emptyset$
      - If $|G| < k$, set $R_i \leftarrow$ `Threshold-Filter`(U_i, G, τ)
      - Send R_i to machine M_0
   (f) Update $G \leftarrow$ `Threshold-Greedy`$(\cup_{i=1}^{m} R_i, G, \tau)$ on machine M_0

Analysis

Exercise 6.7. *Prove that `Threshold-MPC` gives an $\frac{1}{2}$-approximation if we run for $t = 1$ rounds.*

`Threshold-MPC` with 1 round gives an $\frac{1}{2}$-approximation

Hint *See the analysis in Section 6.3.1.*

Claim 6.15. *With probability $1 - e^{-k}$, $|\cup_{i=1}^{m} R_i|$ is at most $\sqrt{nk}$.*

Proof. Recall that we sampled each element with probability $p = 4\sqrt{\frac{k}{n}}$ into S. By Chernoff bounds, S has at least $3\sqrt{nk}$ elements with high probability.

We analyze S as being provided in $3k$ phases, each with $\sqrt{\frac{n}{k}}$ elements. At any phase, if $|\cup_{i=1}^{m} R_i| \leqslant \sqrt{nk}$, the claim holds. Henceforth, suppose $|\cup_{i=1}^{m} R_i| > \sqrt{nk}$ at every phase. Consider S_j of size $\sqrt{\frac{n}{k}}$ at some phase j. Then,

$$\Pr[S_j \cap (\cup_{i=1}^{m} R_i) = \emptyset] \leqslant (1 - \frac{1}{\sqrt{\frac{n}{k}}})^{\sqrt{nk}} \leqslant e^{-k} \leqslant \frac{1}{2}$$

That is, some element from $\cup_{i=1}^{m} R_i$ will be sampled into S_j with probability at least half. Since we run $3k$ phases, we expect $\frac{3k}{2}$ elements from $\cup_{i=1}^{m} R_i$ to be sampled. Hence, with probability at least $1 - e^{-k}$, we sampled k elements with marginal gain (with respect to G) of $\geqslant \tau$ into S. In that case, $G = \texttt{Threshold-Greedy}(S, \emptyset, \tau)$ will be of size k and $\cup_{i=1}^{m} R_i = \emptyset$. □

Remark Claim 6.15 is of similar flavor as Lemma 6.9. Let R be the set of remaining elements with sufficiently high marginal gain. Lemma 6.9 argues that $|R| \in \mathcal{O}(\frac{k \log n}{p})$ while Claim 6.15 argues that $|R| \in \mathcal{O}(\frac{k}{p})$.

Lemma 6.16. *Let X_r be the set of first $\frac{r}{t}k$ elements chosen by `Threshold-MPC`. Then, $f(X_r) \geqslant (1 - (1 - \frac{1}{t+1})^r) \cdot f(\mathrm{OPT})$.*

Proof. For cleanliness, we assume that $\frac{r}{t}k$ is integral. We prove by induction on the round number r. When $r = 0$, the statement trivially holds.

Recall that $\tau_r = (1 - \frac{1}{t+1})^r \cdot \frac{f(\mathrm{OPT})}{k}$.

- Case 1: All of X_r were selected when threshold was above τ_r

  By the induction hypothesis,

  $$f(X_{r-1}) \geqslant (1 - (1 - \frac{1}{t+1})^{r-1}) \cdot f(\mathrm{OPT})$$

  In case 1, the remaining $(\frac{r}{t}k - \frac{r-1}{t}k)$ items of $X_r \setminus X_{r-1}$ each has marginal gain of at least τ_r. Thus,

  $$\begin{aligned} f(X_r) &\geqslant f(X_{r-1}) + (\frac{r}{t}k - \frac{r-1}{t}k) \cdot \tau_r \\ &\geqslant (1 - (1 - \frac{1}{t+1})^{r-1}) \cdot f(\mathrm{OPT}) + \frac{1}{t} \cdot (1 - \frac{1}{t+1})^r \cdot f(\mathrm{OPT}) \\ &= (1 - (1 - \frac{1}{t+1})^r) \cdot f(\mathrm{OPT}) \end{aligned}$$

- Case 2: Not all of X_r were selected when threshold was above τ_r

Let $S_r \subseteq X_r$ be the set of elements selected when threshold value was at least τ_r. Then, there are no elements with marginal value more than τ_r with respect to S. Since f is monotone,

$$f(OPT) - f(S_r) \leqslant f(OPT \cup S_r) - f(S_r) = f_S(OPT) \leqslant k\tau_r$$

Rearranging,

$$f(S_r) \geqslant f(OPT) - k\tau_r = (1 - (1 - \frac{1}{t+1})^r) \cdot f(OPT)$$

□

Theorem 6.17. *`Threshold-MPC` gives an $(1 - \frac{1}{e} - \mathcal{O}(\epsilon))$-approximation if we run for $t \in \mathcal{O}(\frac{1}{\epsilon})$ rounds.*

Proof. Pick $t \in \mathcal{O}(\frac{1}{\epsilon})$ in Lemma 6.16. □

Chapter 7

Data clustering

In this chapter, we look at a clustering problem that is commonly used in data science. Given a large corpus of data, clustering is often used to group closely related elements so that one can get the rough sense of a cluster by examining its representative element. Amongst the variants of clustering, we will look at the problem of k-means clustering.

Definition 7.1 (The k-means clustering problem)**.** *Given a set* $X = \{x_1, \ldots, x_n\}$ *of* n *points in* $\mathbb{R}^d$ *and an integer* $k \in [1, n]$*, output a set* $C = \{c_1, \ldots, c_k\}$ *of* k *points in* $\mathbb{R}^d$*, not necessarily from* X*, that minimizes*

$$\phi_X(C) = \sum_{x \in X} d^2(x, C) = \sum_{x \in X} \min_{i \in \{1,\ldots,k\}} \|x - c_i\|^2$$

For notational cleanliness, for sets with only one element, we write $\phi_X(c)$ and $d^2(x, c)$ to mean $\phi_X(\{c\})$ and $d^2(x, \{c\})$ respectively. In Section 7.1, we will describe the standard k-means clustering algorithm which iteratively refines an initial set C. Then, we describe a way to initialize the set C to obtain theoretical competitive guarantees in Section 7.2, and discuss a "parallelized" way to implement the initialization in Section 7.3.

Remark We saw another variant of clustering in Exercise 6.4 that aims to minimize a different loss metric: $\max_{x \in X} d^2(x, C)$.

7.1 `k-means`

Drineas et a. [DFK+04, Section 3, Theorem 1] showed that k-means clustering is $\mathbb{NP}$-hard even for $k \geqslant 2$. However, simple and efficient algorithms

with good empirical results exist. For example, Lloyd [Llo82] proposed a local search algorithm that iteratively refines an initial set of k centers by alternating the two following steps until convergence:

Assignment step Assign points in X to the nearest $c \in C$

Update step For each $c \in C$, let $X_c \subseteq X$ be the set of points assigned to c. Update point c to minimize $\phi_{X_c}(c) = \sum_{x \in X_c} \|x - c\|^2$

In the update step, the best centroid for a fixed group of points X_c is the average of the points of X_c. That is, we update $c \in C$ to be the *mean* of the data points assigned to it:

$$\operatorname{argmin} \phi_{X_c}(c) = \frac{1}{|X_c|} \sum_{x \in X_c} x$$

Figure Fig. 7.1 illustrates `k-means` with randomly initial centroids C.

Remark The k-means algorithm is closely related to the Expectation-Maximization algorithm[1] and Gaussian mixture models[2].

7.2 `k-means++`: Initializing with guarantees

Although k-means is simple and efficient in practice, it is known that running it with different initial C will often end in different local minima. In particular, there exists data distributions in which certain initializations can be much worse than the optimum achievable $\phi_X(C)$. To achieve theoretical guarantees, Arthur and Vassilvitskii [AV07] proposed `k-means++` to initialize the k centroids so that running k-means on this initialization will be $\mathcal{O}(\log k)$-competitive, regardless of data distribution:

1. Pick c_1 uniformly at random from X.
2. For $i \in \{2, \ldots, k\}$,
   (a) For $x \in X$, denote $d(x) = d^2(x, \{c_1, \ldots, c_{i-1}\})$ as the distance of x to the closest point in $\{c_1, \ldots, c_{i-1}\}$.
   (b) Pick $x \in X$ as c_i with probability $\frac{d(x)}{\sum_{x \in X} d(x)}$.

[1] https://en.wikipedia.org/wiki/Expectation-maximization_algorithm
[2] https://en.wikipedia.org/wiki/Mixture_model#Gaussian_mixture_model

Figure Fig. 7.2 illustrates k-means++. The intuition of this initialization process is to favor picking points which are currently "badly represented" by the current set of chosen centroids. We do not analyze k-means++ here, but we will analyze a closely related variant in the next section.

7.3 k-means||: Parallelizing the initialization

The initialization process of k-means++ is inherently sequential. Let Δ be the maximum distance among any pair of points from X. Bahmani et al. [BMV+12] proposed an $\mathcal{O}(\log(n\Delta))$ phase algorithm k-means|| to first reduce the data size to $\mathcal{O}(k \log n)$ such that the loss is only a constant factor in approximation. Then, a subsequent call to a weighted clustering algorithm computes a set of k points using the $\mathcal{O}(k \log n)$ representatives.

7.3.1 Algorithm

Fix $l \in \Theta(k)$. The algorithm k-means|| chooses $\mathcal{O}(l \log n)$ points from X to represent all points in X. Then, k centroids are chosen amongst these $\mathcal{O}(l \log n)$ representatives as the initial set C for subsequent clustering algorithms such as k-means++. For each representative $c \in C$, one can think of "moving" points $x \in X$ to their labels $c \in C$.

1. $C \leftarrow$ Sample a point uniformly at random from X
2. For $\mathcal{O}(\log(n\Delta))$ iterations
   (a) $C' \leftarrow$ Sample $x \in X$ independently with probability $l \cdot \frac{d^2(x,C)}{\phi_X(C)}$
   (b) $C \leftarrow C \cup C'$
3. For each $c \in C$, set w_c to be the number of points closest to c than any other point in C
4. In one machine, cluster the weighted points of C into k points.

Claim 7.2. *If an α-approximation algorithm $\mathcal{A}$ is used to recluster the points in Step 4, then k-means|| produces a set of k centroids that is $\mathcal{O}(\alpha)$-approximate to the optimum clustering of the data points X.*

Proof. Let C be the gathered set of representatives, OPT be the set of optimum centroid for X, and OPT' be the set of optimum centroids for C. In the analysis later, we will show that $\mathbb{E}[\phi_X(C)] \in \mathcal{O}(\phi_X(\mathrm{OPT}))$. So,

$$\mathbb{E}[\mathcal{A}(C)] \leqslant \mathbb{E}[\alpha \cdot \phi_X(\mathrm{OPT}')] \leqslant \mathbb{E}[\alpha \cdot \phi_X(C)] \in \mathcal{O}(\alpha \cdot \phi_X(\mathrm{OPT}))$$

If we use $\mathcal{A} = \texttt{k-means++}$, then $\texttt{k-means}\|$ produces a set of k centroids that is $\mathcal{O}(\log k)$-competitive to OPT. □

$\texttt{k-means}\|$ in MPC Suppose each machine has memory $S \in \widetilde{\Omega}(k)$, then each iteration of $\texttt{k-means}\|$ can be executed in $\mathcal{O}(1)$ MPC rounds — points from X are sampled independently on the machines and broadcasted. The set of representatives C are then clustered on a single machine.

7.3.2 Analysis

We first discuss two successively complete intuitions about $\texttt{k-means}\|$ before presenting the analysis.

First order intuition

Consider set C from some iteration of $\texttt{k-means}\|$ and the clusters of OPT. Suppose A is a cluster where $\phi_A(C) \geqslant \frac{1}{2k}\phi_X(C)$. Since we sample $l \in \Omega(k)$ points and each sample lies in A with probability at least $\frac{1}{2k}$, a point from A will be sampled into C' in the current iteration with constant probability. Fig. 7.3 gives a visual interpretation of this intuition.

Second order intuition

Fix a cluster A in OPT. Suppose $|A| = T$ and denote its centroid as $a^* = \frac{1}{|A|}\sum_{a \in A} a$. Define the points A by $a_1, \ldots, a_T$ such that a_i is closer to a^* than a_j for any $i < j$. That is, $d^2(a_i, a^*) \leqslant d^2(a_j, a^*)$ for any $i < j$.

Claim 7.3. *For $1 \leqslant i < j \leqslant T$, a_i is a better centroid for set A than a_j. That is,*

$$\phi_A(a_i) \leqslant \phi_A(a_j)$$

Proof. For a point $a \in A$,

$$\begin{aligned}\|a - a_i\|^2 &= \|(a - a^*) + (a^* - a_i)\|^2 \\ &= \|a - a^*\|^2 - 2\langle a - a^*, a^* - a_i\rangle + \|a^* - a_i\|^2\end{aligned}$$

Since $a^* = \frac{1}{|A|}\sum_{a\in A} a$,

$$\begin{aligned}
\phi_A(a_i) &= \sum_{a\in A} \|a - a_i\|^2 \\
&= \sum_{a\in A} \left(\|a - a^*\|^2 - 2\langle a - a^*, a^* - a_i\rangle + \|a^* - a_i\|^2\right) \\
&= \sum_{a\in A} \left(\|a - a^*\|^2\right) - 2\sum_{a\in A} \left(\langle a - a^*, a^* - a_i\rangle\right) + |A| \cdot \|a^* - a_i\|^2 \\
&= \sum_{a\in A} \left(\|a - a^*\|^2\right) + |A| \cdot \|a^* - a_i\|^2
\end{aligned}$$

By definition, a_i is closer to a^* than a_j. Thus,

$$\phi_A(a_j) - \phi_A(a_i) = |A| \cdot \left(\|a^* - a_j\|^2 - \|a^* - a_i\|^2\right) \geqslant 0$$

□

Consider set C from some iteration of `k-means||`. For $t \in \{1, \ldots, T\}$, let us define the following quantities:

- The probability for sampling point a_t into C' (with respect to A)

$$p_t = l \cdot \frac{d^2(a_t, C)}{\phi_A(C)}$$

- The probability that a_t is the first / minimum index point sampled

$$q_t = p_t \cdot \Pi_{i=1}^{t-1}(1 - p_i)$$

- An *upper bound* on $\phi_A(C \cup C')$, the new ϕ cost of A

$$s_t = \min\{\phi_A(C), \phi_A(a_t)\}$$

  where $\phi_A(C)$ is the old cost, and $\phi_A(a_t) = \sum_{a\in A} \|a - a_t\|^2$. Note that $\phi_A(C \cup C')$ could be smaller than $\phi_A(a_t)$ if multiple points of A are sampled into C', or if the points are closer to a centroid outside of A.

We further define $q_{T+1} = 1 - \sum_{t=1}^{T} q_t$ as the probability of *not* sampling any point from A into C', and $s_{T+1} = \phi_A(C)$ as the corresponding upper bound of $\phi_A(C \cup C')$.

Lemma 7.4. $\sum_{t=1}^{T} \phi_A(a_t) = 2 \cdot T \cdot \phi_A(a^*)$ *and* $\mathbb{E}_t[\phi_A(a_t)] = 2\phi_A(a^*)$

Proof. Recall that $\phi_A(a_i) = \sum_{a \in A} \left(\|a - a^*\|^2 \right) + T \cdot \|a^* - a_i\|^2$. Thus,

$$\begin{aligned}
\sum_{t=1}^{T} \phi_A(a_t) &= \sum_{t=1}^{T} \Big(\sum_{a \in A} \left(\|a - a^*\|^2 \right) + T \cdot \|a^* - a_t\|^2 \Big) \\
&= \sum_{t=1}^{T} \left(\phi_A(a^*) \right) + T \cdot \sum_{t=1}^{T} \left(\|a^* - a_t\|^2 \right) \\
&= 2 \cdot T \cdot \phi_A(a^*)
\end{aligned}$$

Hence, $\mathbb{E}_t[\phi_A(a_t)] = \frac{1}{T} \sum_{t=1}^{T} \phi_A(a_t) = 2\phi_A(a^*)$. □

Under the assumption that $\{q_t\}_t$ is a decreasing sequence, the following claim shows that the potential $\phi_A(C \cup C')$ of A "moves towards" a constant approximation of $\phi_A(a^*)$, in expectation. One scenario in which the assumption holds is when $p_1 = \cdots = p_T$ (consider a cluster A that is far from all the chosen centroids so far, then all p_i's are roughly the same). We will analyze the situation for general q_t's later.

Claim 7.5. *If* $p_1 = \cdots = p_T$, *then*

$$\mathbb{E}[\phi_A(C \cup C')] \leqslant (1 - q_{T+1}) \cdot \left(2\phi_A(a^*) \right) + q_{T+1} \cdot \phi_A(C)$$

Proof. Since $p_1 = \cdots = p_t$, $q_i \geqslant q_j$ for $i < j$. Furthermore, $\phi_A(a_i) \leqslant \phi_A(a_j)$ for $i < j$ by Claim 7.3. So, $\sum_{t=1}^{T} q_t \phi_A(a_t)$ is a sum of products between the increasing sequence $\{q_t\}_t$ and decreasing sequence $\{\phi_A(a_t)\}_t$. Therefore, we can conclude that $\sum_{t=1}^{T} q_t \phi_A(a_t) \leqslant \frac{1}{T} \Big(\sum_{t=1}^{T} q_t \sum_{t=1}^{T} \phi_A(a_t) \Big)$.

$$\begin{aligned}
\mathbb{E}[\phi_A(C \cup C')] &\leqslant \sum_{t=1}^{T} q_t s_t + q_{T+1} \phi_A(C) && \text{Union bound over } a_t\text{'s} \\
&\leqslant \sum_{t=1}^{T} q_t \phi_A(a_t) + q_{T+1} \phi_A(C) && s_t = \min\{\phi_A(C), \phi_A(a_t)\} \\
&\leqslant \frac{1}{T} \Big(\sum_{t=1}^{T} q_t \sum_{t=1}^{T} \phi_A(a_t) \Big) + q_{T+1} \phi_A(C) && \text{From above} \\
&= \Big(\frac{1}{T} \sum_{t=1}^{T} q_t \Big) \cdot \left(2\phi_A(a^*) \right) + q_{T+1} \phi_A(C) && \text{By Lemma 7.4} \\
&\leqslant (1 - q_{T+1}) \cdot \left(2\phi(a^*) \right) + q_{T+1} \cdot \phi_A(C)
\end{aligned}$$

□

Remark The inequality $\sum_{t=1}^{T} q_t \phi_A(a_t) \leqslant \frac{1}{T}\left(\sum_{t=1}^{T} q_t \sum_{t=1}^{T} \phi_A(a_t)\right)$ is a case of the Chebyshev's sum inequality[3].

From the first inituition, we know that if cluster A has $\phi_A(C) \geqslant \frac{1}{2k}\phi_X(C)$ at some iteration of `k-means||`, then a point from A is likely to be sampled with constant probability, so q_{T+1} will be small. The above claim assumes that the sequence $\{q_t\}_t$ is decreasing (via $p_1 = \cdots = p_T$). This may not be true in general and we will show a general upper bound of $\phi_A(C \cup C')$.

Analysis

The first intuition tells us that if there is a cluster A which has high potential $\phi_A(C)$ with respect to the current selection C, then we are likely to pick something from the cluster A with constant probability. If $\phi_A(C)$ remains high, `k-means||` will pick a point from A with high probability over its $\mathcal{O}(\log n)$ iterations.

The second intuition takes a closer look at such a cluster A and tells us that we expect to "move" $\phi_A(C)$ towards a constant approximation of $\phi_A(a^*)$ in expectation. We saw an analysis assuming that the sequence $\{q_t\}_t$ is decreasing. For a proper analysis, one has to properly upper bound $\phi_A(C \cup C')$. To do so, Bahmani et al. [BMV+12] proved a linear constraint on q_t's (See Lemma 7.6) and formulated a linear program where the objective is the upper bound $\sum_{t=1}^{T+1} q_t s_t$. While the linear program is complicated to solve, the optimum value of its dual is easy to compute.

Lemma 7.6. *[BMV+12, Lemma 5] Let* $\eta_0 = 1$, *and for any* $1 \leqslant t \leqslant T$, $\eta_t = \Pi_{j=1}^{t}\left(1 - \frac{d^2(a_j, C)}{\phi_A(C)}(1 - q_{T+1})\right)$. *Then, for any* $0 \leqslant t \leqslant T$, $\sum_{r=t+1}^{T+1} q_r \leqslant \eta_t$.

[3] https://en.wikipedia.org/wiki/Chebyshev%27s_sum_inequality

Primal LP

$$\begin{aligned} \text{maximize} \quad & \sum_{t=1}^{T+1} q_t s_t \\ \text{subject to} \quad & \sum_{r=t+1}^{T+1} q_r \leqslant \eta_t \quad \forall t \in \{0,\dots,T\} \\ & q_t \geqslant 0 \quad \forall t \in \{1,\dots,T+1\} \end{aligned}$$

Dual LP

$$\begin{aligned} \text{minimize} \quad & \sum_{t=0}^{T} \eta_t \alpha_t \\ \text{subject to} \quad & \sum_{r=0}^{t-1} \alpha_r \geqslant s_t \quad \forall t \in \{1,\dots,T+1\} \\ & \alpha_t \geqslant 0 \quad \forall t \in \{0,\dots,T\} \end{aligned}$$

The linear program formulates the upper bound on $\phi_A(C \cup C')$ as the primal objective function under the constraint $\sum_{r=t+1}^{T+1} q_r \leqslant \eta_t$ from Lemma 7.6. In the dual LP, since s_t is an increasing sequence, the optimal solution to the dual is $\alpha_t = s_{t+1} - s_t$ for $t \in \{0,\dots,T\}$ (setting $s_0 = 0$).

Lemma 7.7. *[BMV+12, Lemma 6, Corollary 7] Let $\alpha = \exp\left(-(1-e^{-\frac{l}{2k}})\right) \approx e^{-\frac{l}{2k}}$. Then, the expected potential of an optimal cluster A after a sampling step is upper bounded as*

$$\mathbb{E}[\phi_X(C \cup C')] \leqslant 8\phi_A(OPT) \cdot (1 - q_{T+1}) + \phi_A(C) \cdot e^{-(1-q_{T+1})}$$

We now prove the main theorem that an iteration of `k-means||` "moves" the potential towards a constant approximation of $\phi_X(OPT)$ in expectation.

Theorem 7.8. *Let $\alpha = \exp\left(-(1-e^{-\frac{l}{2k}})\right) \approx e^{-\frac{l}{2k}}$. If C' is the random set of points added to C in an iteration, then*

$$\mathbb{E}[\phi_X(C \cup C')] \leqslant 8\phi_X(OPT) + \frac{1+\alpha}{2}\phi_X(C)$$

Proof. Let $A_1,\dots,A_k$ be the clusters of the optimal solution OPT. We say that cluster A is "heavy" if $\frac{\phi_A(C)}{\phi_X(C)} \geqslant \frac{1}{2k}$. Otherwise, we say that A is "light".

Since $q_{T+1} = \Pi_{t=1}^{T}(1-p_t) \leqslant e^{-\sum_{t=1}^{T} p_t} = e^{-l \cdot \frac{\phi_A(C)}{\phi_X(C)}}$, by Lemma 7.7,

$$\begin{aligned} \mathbb{E}[\phi_A(C \cup C')] &\leqslant 8\phi_A(OPT) \cdot (1 - q_{T+1}) + \phi_A(C) \cdot e^{-(1-q_{T+1})} \\ &\leqslant 8\phi_A(OPT) + \alpha\phi_A(C) \end{aligned}$$

for any heavy cluster A. Meanwhile,

$$\mathbb{E}[\phi_A(C \cup C')] \leqslant \phi_A(C) \leqslant \frac{\phi_X(C)}{2k}$$

for any light cluster A. Let A_H be the set of heavy clusters and A_L be the set of light clusters. Then,

$$\mathbb{E}[\phi_{A_H}(C \cup C')] \leqslant 8\phi_{A_H}(OPT) + \alpha\phi_{A_H}(C)$$

and, because $|A_L| \leqslant k$,

$$\mathbb{E}[\phi_{A_L}(C \cup C')] \leqslant \phi_{A_L}(C) \leqslant \frac{\phi_X(C)}{2}$$

Since $\phi_X(C) = \phi_{A_H}(C) + \phi_{A_L}(C)$,

$$\begin{aligned}
\mathbb{E}[\phi_X(C \cup C')] &= \mathbb{E}[\phi_{A_H}(C \cup C')] + \mathbb{E}[\phi_{A_L}(C \cup C')] \\
&\leqslant 8\phi_{A_H}(OPT) + \alpha\phi_{A_H}(C) + \phi_{A_L}(C) \\
&= 8\phi_{A_H}(OPT) + \alpha\phi_X(C) + (1-\alpha)\phi_{A_L}(C) \\
&\leqslant 8\phi_{A_H}(OPT) + \alpha\phi_X(C) + (1-\alpha)\frac{\phi_X(C)}{2} \\
&= 8\phi_X(OPT) + \frac{1+\alpha}{2}\phi_X(C)
\end{aligned}$$

□

By induction over the $\mathcal{O}(\log n)$ iterations of `k-means||`, we see that for the representative set C produced by `k-means||`, $\mathbb{E}[\phi_X(C)] \in \mathcal{O}(\phi_X(OPT))$.

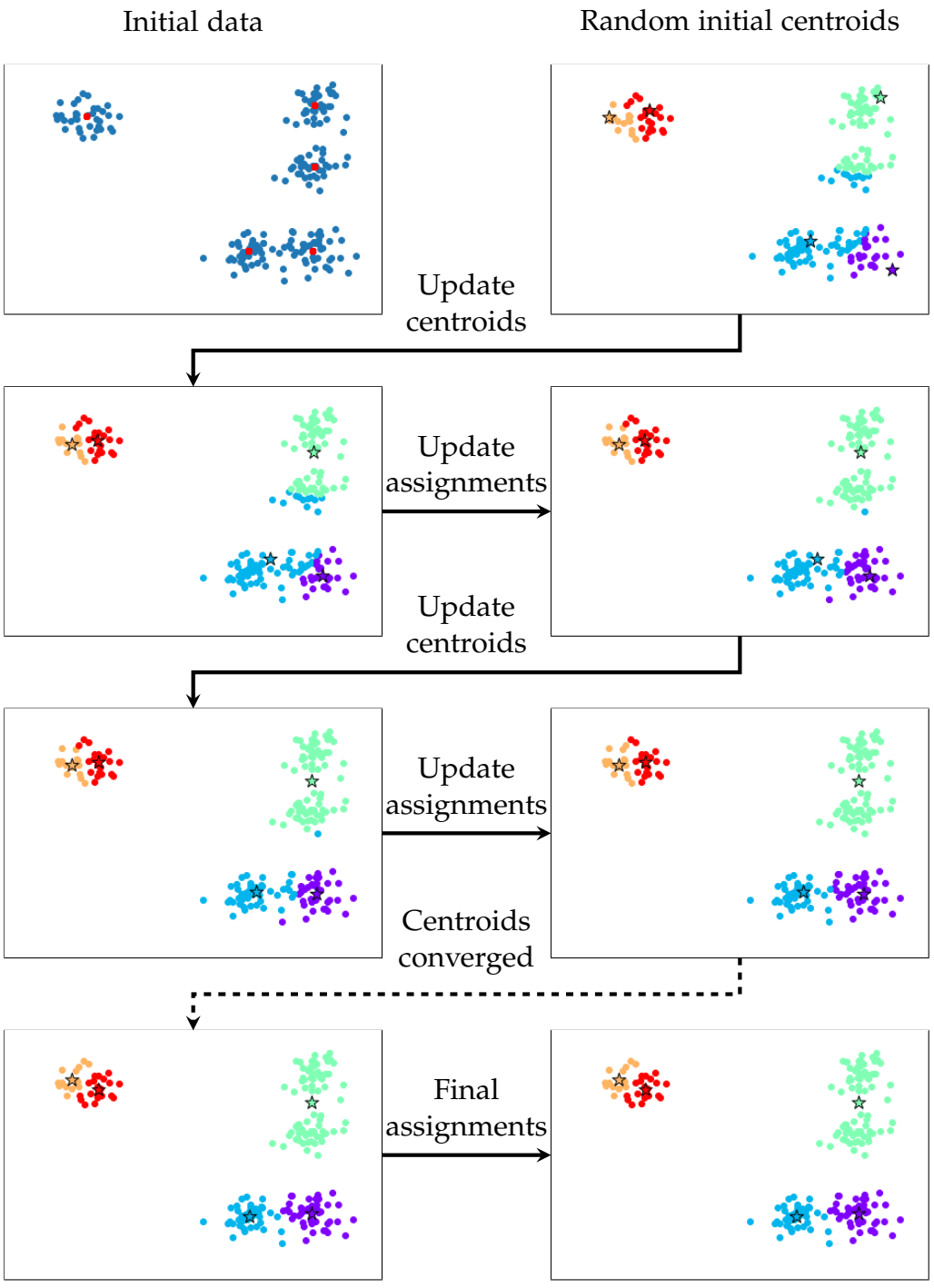


Figure 7.1: K-means with random initial C on data points X that are generated from 5 Gaussians

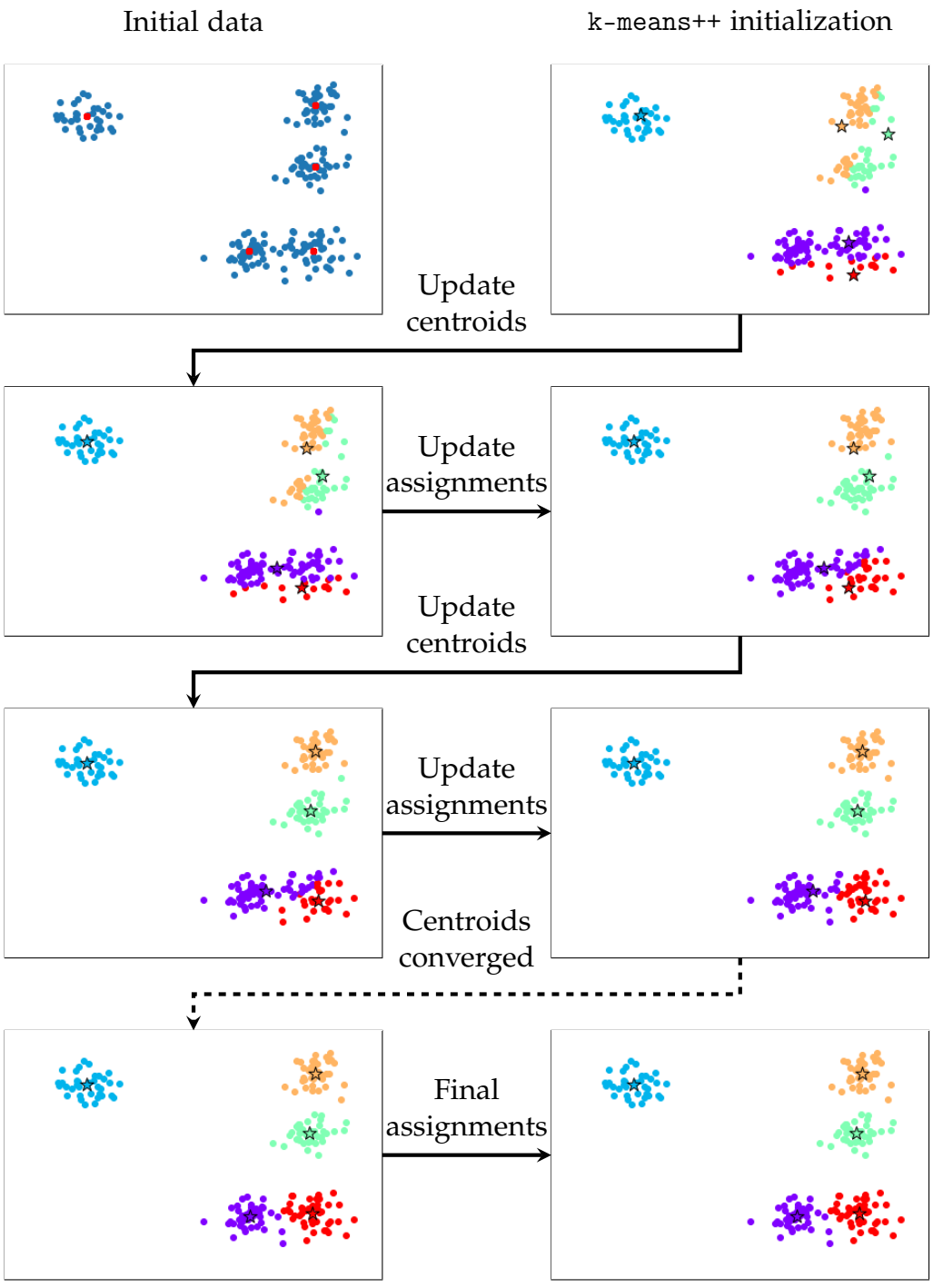


Figure 7.2: K-means with k-means++ initialization on data points X that are generated from 5 Gaussians

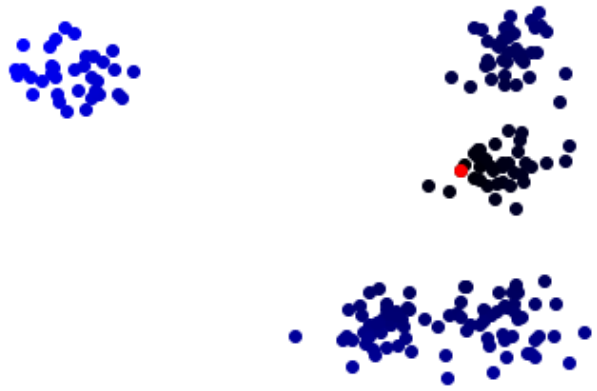


Figure 7.3: There are 5 optimal clusters and $|C| = 1$ currently. Lighter colors indicate higher probability of being sampled into C'. Visually, we see that a point from the top left cluster is likely to be sampled.

Chapter 8

Exact minimum cut in near linear memory

In Exercise 3.4, we worked towards an algorithm to compute an $(1+\epsilon)$-approximation to the minimum cut in $O(1)$ rounds with $S = \widetilde{O}(n)$ memory per machine. In this chapter, we discuss an algorithm by Ghaffari and Nowicki [GN19] that computes an *exact* minimum cut for *simple* and *unweighted* graphs under the near linear memory regime.

The key ingredient is a new contraction process called k-*out*. k-out contraction is a simple process that reduces the number of edges to $O(n)$ while preserving *any* non-trivial[1] minimum cut with constant probability. Under the near linear memory regime, the reduced graph fits in a single machine and an exact minimum cut can be found in $O(1)$ rounds. By running $O(\log n)$ copies in parallel, we succeed with high probability.

A new contraction process

The algorithm works in two phases — Phase 1 performs a k-out contraction for $k = 3$ while Phase 2 uses Karger's single edge contraction.

1. Each node proposes 3 of its edges (independently, with repetition). All proposed edges are contracted simultaneously.

2. Contract random edges, one at a time, so long as there are at least $10n$ edges, where n is the number of nodes in the input graph.

[1] A trivial cut is a cut where one partition contains only a single vertex and this can be easily checked by looking at the minimum degree of the graph.

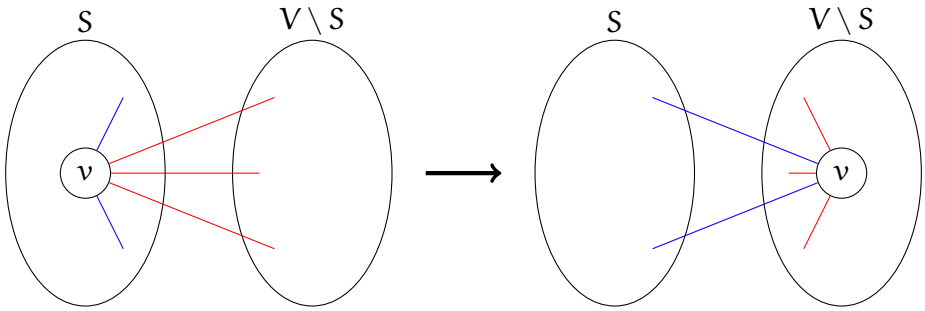
Analysis

Definitions

For analysis, we fix a minimum cut $(S, V \setminus S)$ of size λ such that $2 \leqslant |S| \leqslant n-1$. For a vertex $v \in V$, we denote $d(v)$ as its degree and $c(v)$ as the number of edges of v that lie in the cut $(S, V \setminus S)$. Let $\delta = \min_{v \in V} d(v)$ be the minimum degree of the input graph. Since we ignore trivial cuts, we can safely assume that $\lambda < \delta$.

Claim 8.1. $\frac{c(v)}{d(v)} \leqslant \frac{1}{2}$

Proof.



If $c(v) > \frac{1}{2} d(v)$, moving v to the other partition forms a smaller cut. □

Overview of analysis

To prove the main result Theorem 8.2, it suffices to show that each of the phases preserve $(S, V \setminus S)$ with constant probability (See Lemma 8.3 and Lemma 8.4 respectively). We will prove Lemma 8.3 using the inequality in Claim 8.1 while Lemma 8.4 can be shown with Lemma 8.5. Lemma 8.5 is then proven in two steps by exposing the randomness of the 3-out contraction suitably.

Theorem 8.2. *Fix a non-trivial cut $(S, V \setminus S)$ of size λ such that $2 \leqslant |S| \leqslant n-1$. With constant probability, the resulting multigraph of the algorithm preserves $(S, V \setminus S)$. That is, we did not contract* any *edge of $(S, V \setminus S)$.*

Remark Recall the inequality $(1-x)^k \geqslant 4^{-kx}$ for $x \leqslant \frac{1}{2}$.

Analysis

Lemma 8.3. *Phase 1 preserves $(S, V \setminus S)$ with constant probability.*

Proof. In Phase 1, each vertex propose 3 random edges (independently, with repetitions) to be contracted. We expose the randomness by first considering the case of all vertices picking one random edge (k-out contraction for $k = 1$). If we can show that $(S, V \setminus S)$ is preserved with at least constant probability p, then Phase 1 preserves $(S, V \setminus S)$ with probability at least p^3 because the proposal of edges are independent.

$$\begin{aligned}
\Pr[\text{1-out preserves } (S, V \setminus S)] &= \Pi_{v \in V}\left(1 - \frac{c(v)}{d(v)}\right) \\
&\geqslant \Pi_{v \in V} 4^{-\frac{c(v)}{d(v)}} && \text{Claim 8.1} \\
&= 4^{-\sum_{v \in V} \frac{c(v)}{d(v)}} \\
&\geqslant 4^{-\sum_{v \in V} \frac{c(v)}{\lambda}} && d(v) > \lambda \\
&= 4^{-\frac{2\lambda}{\lambda}} && \sum_{v \in V} c(v) = 2\lambda \\
&= \frac{1}{16}
\end{aligned}$$

□

Lemma 8.4. *Phase 2 preserves $(S, V \setminus S)$ with constant probability.*

Proof. By Lemma 8.5, the number of vertices remaining after Phase 1 is at most $\frac{20n}{\delta} \leqslant \frac{20n}{\lambda}$ with high probability. For each edge contraction until we have less than $10n$ edges, the probability of *not* picking an edge from $(S, V \setminus S)$ is at least $(1 - \frac{\lambda}{10n})$. Since each contraction reduces the number of vertices by one, the contraction can happen at most $\frac{20n}{\lambda}$ times. So,

$$\Pr[\text{Phase 2 preserves } (S, V \setminus S)] \geqslant (1 - \frac{\lambda}{10n})^{\frac{20n}{\lambda}} \geqslant \frac{1}{16}$$

□

Lemma 8.5. *With high probability, the number of connected components after 3-out contraction (i.e. the number of contracted vertices) remaining after Phase 1 is at most $\frac{20n}{\delta} \leqslant \frac{20n}{\lambda}$.*

For cleanliness of the proof[2], we assume $\delta \leqslant \frac{n}{\text{poly}\log n}$. We expose the randomness of 3-out in two stages. We first analyze the number of connected components after each vertex proposes 2 edges, then analyze how the 3rd proposed edge connects the resultant connected components.

Claim 8.6. *After 2-out contraction, there are at most $\frac{5n}{\delta}$ connected components with less than $10\log\delta$ vertices with high probability.*

Proof. Consider the process of growing a connected component:

- Let $S = \{v\}$ for some vertex v.
- We grow a connected component C by expanding each vertex $v \in S$:
  - Remove v from S. Add v to C.
  - Consider the two random edges $\{v, a\}$ and $\{v, b\}$ that v proposed.
  - If $a \notin C$, add a to S. If $b \notin C$, add b to S.

When S becomes empty, the growing process stops. Either we have created a new connected component, or the current component connects to a some previous component. Let us now upper bound the number of bad events when a *new*[3] component is created with $10\log\delta$ vertices.

During the growing process, a vertex v proposes edges $\{v, a\}$ and $\{v, b\}$. If C is a bad component with at most $10\log\delta$ vertices, then the probability that $a \in C$ or $b \in C$ is at most $\frac{10\log\delta}{\delta}$ independently. So, the probability that vertex v prevents growth of the connected component C (because *both* $a \in C$ and $b \in C$) is at most $(\frac{10\log\delta}{\delta})^2$. Taking the union bound over vertices in bad component C,

$$\Pr[C \text{ is bad}] \leqslant \sum_{v \in C} \Pr[v \text{ prevented growth}] \leqslant 10\log\delta \cdot (\frac{10\log\delta}{\delta})^2$$

Since we start the process n times, we expect at most $\frac{1000n\log^3\delta}{\delta^2} \leqslant \frac{5n}{\delta^2}$ components with less than $< 10\log\delta$ vertices. Apply Chernoff bounds. □

Claim 8.7. *Consider the connected components after 2-out contraction. After another 1-out contraction, there are at most $\frac{12n}{\delta}$ connected components with less than $\frac{\delta}{2}$ vertices with high probability.*

[2] See Ghaffari and Nowicki [GN19, Lemma 2.5] for the full proof.

[3] If the current component connects to an older one, this is not a bad event. If the old component was small, we would have accounted for it.

Proof. We analyze in a similar manner as before, except now we connect components instead of connect vertices.

- Let $S = \{C\}$ for some component C from 2-out contraction's output.
- We grow a connected component C^* by expanding each $C \in S$:
  - Remove component C from S. Add vertices of C to C^*.
  - Consider the set of edges $\left\{\{v, u\}\right\}_{v \in C}$ proposed by 1-out of $v \in C$.
  - If $u \notin C^*$, add the component containing u to S.

When S becomes empty, the connection process stops. Let us upper bound the number of bad events where the newly formed component has less than $\frac{\delta}{2}$ vertices. If the component C^* has less than $\frac{\delta}{2}$ vertices, then each edge proposed points to some $u \in C^*$ with probability at most $\frac{\delta/2}{\delta} = \frac{1}{2}$.

With high probability, there are at most $\frac{5n}{\delta}$ connected components with less than $10 \log \delta$ vertices after 2-out. These "small components" can contribute at most $\frac{5n}{\delta}$ bad events. Let us consider components of size at least $10 \log \delta$ at the end of the first 2-out process. Any such component $C \in S$ prevents growth with probability at most $(\frac{1}{2})^{10 \log \delta}$. Taking union bound over all $\frac{\delta}{2}$ possible components in a failed component,

$$\Pr[C^* \text{ is bad}] \leqslant \frac{\delta}{2} \cdot (\frac{1}{2})^{10 \log \delta} \leqslant \frac{1}{2\delta^9} \leqslant \frac{1}{\delta}$$

Since we start the process at most n times, we expect at most $\frac{n}{\delta}$ components with less than $\frac{\delta}{2}$ vertices. Including "small components", we expect at most $\frac{5n}{\delta} + \frac{n}{\delta} = \frac{6n}{\delta}$ bad events. By Chernoff bounds, the claim holds. □

Proof Lemma 8.5. By the above claims, there are $\leqslant \frac{12n}{\delta}$ connected components with less than $< \frac{\delta}{2}$ vertices with high probability. Meanwhile, there are $\leqslant \frac{n}{\delta/2}$ components with $\geqslant \frac{\delta}{2}$ vertices for an input graph of n vertices. Since each connected component contracts into a single vertex, we have at most $\frac{12n}{\delta} + \frac{2n}{\delta} \leqslant \frac{20n}{\delta}$ vertices after the 3-out contraction in Phase 1. □

Implementing the algorithm in constant rounds under the near linear memory regime

In Phase 1 of the algorithm, each vertex proposes 3 edges for contraction. The subgraph induced by these proposed edges fits in a single machine

and thus Phase 1 can be done in constant rounds. It suffices to argue that Phase 2, where we contract edges one at a time until there are at most $10n$ edges, can be done in constant rounds. Observe that Phase 2 is equivalent to the following process:

- Independently assign a random number $r(e) \in [0, 1]$ for each edge e.
- Run Kruskal's algorithm to build a MST according to these weights.
- Terminate Kruskal's when there are less than $10n$ edges remaining.

Since edge weights are randomly assigned independently, the order in which edges are chosen to merge is equivalent to picking a random edge for contraction. The proof of Lemma 8.4 requires $\Omega(n)$ edges to remain to preserve a minimum cut $(S, V \setminus S)$ with constant probability. So, despite being able to MST can be solved in constant rounds under the near linear memory regime (See Section 3.1), it remains to argue that we are able to terminate the MST algorithm before there are too little edges remaining.

Recall the idea of graph sketching from Section 3.2 where we guess $\mathcal{O}(\log n)$ values $\widehat{k} \in \{(1+\epsilon)^0, (1+\epsilon)^1, \ldots, (1+\epsilon)^{\log_{1+\epsilon} n}\}$ for the cut size between an arbitrary vertex partition $(A, V \setminus A)$. For the guess $\widehat{k}$ which is an $(1+\epsilon)$-approximation of the true cut size, we will be able to detect a cut edge with high probability. Hence, we can obtain an $(1+\epsilon)$-approximation[4] of the number of edges leaving a component during the execution of Borůvka's MST algorithm with high probability. Since the summation of the number of edges leaving each component is twice the number of edges remaining in the graph, we can obtain a constant approximation of the number of remaining edges using graph sketches, and terminate the MST construction appropriately.

[4]Note that any $\mathcal{O}(1)$-approximation suffices for our purposes.

Chapter 9

Vertex coloring

In this chapter, we look at the problem of vertex coloring of graphs with maximum degree Δ. In the sequential setting, greedily coloring a vertex by a free color yields a $(\Delta+1)$ coloring in linear time and space. Recently, Assadi et al. [ACK19] showed a MPC algorithm[1] under the near linear memory regime that runs in constant number of rounds.

At the core of their algorithm is the following process: Each node v samples a list $L(v)$ of size $\Theta(\log n)$ from $\{1,\ldots,\Delta+1\}$. One may think of picking each color independently with probability $p = \Theta(\frac{\log n}{\Delta+1})$, or picking the first few colors in a random permutation of $\{1,\ldots,\Delta+1\}$.

Theorem 9.1 (Palette-Sparsification Theorem)**.** *Let $G(V,E)$ be an n-vertex graph with maximum degree Δ. Suppose for any vertex $v \in V$, we sample $\Theta(\log n)$ colors $L(v)$ from $\{1,\ldots,\Delta+1\}$ independently and uniformly at random. Then with high probability, there exists a proper $(\Delta+1)$ coloring of G in which the color for every vertex v is chosen from $L(v)$.*

By Theorem 9.1, the above sampling process significantly reduces the size of the graph that we have to consider in order to find a coloring. Observe that there is still a potential coloring conflict between vertices u and v only if $L(u)$ and $L(v)$ share a common color. Consider a subgraph H of G where we keep edge $\{u,v\}$ if and only if $|L(u) \cap L(v)| \neq \emptyset$. Then, the probability of an edge $\{u,v\}$ being in H is at most $(\Delta+1)\cdot p^2 \in \Theta(\frac{\log^2 n}{\Delta+1})$. Since G has at most $n\Delta$ edges, we expect to see $\mathcal{O}(n\log^2 n)$ edges in H, and with high probability, there are $\widetilde{\mathcal{O}}(n)$ edges in H. Theorem 9.1 states the key claim about the above process.

[1] They also showed a single-pass semi-streaming algorithm using $\widetilde{\mathcal{O}}(n)$ space and a sublinear-time algorithm in the standard query model using $\widetilde{\mathcal{O}}(n\sqrt{n})$ time.

While solving list coloring is $\mathbb{NP}$-hard in general, Theorem 9.1 implies that one can non-adaptively sparsify a graph and apply the power of the underlying computational model to compute a $(\Delta+1)$ coloring of G "quickly". For example, sampling of color lists $L(v)$ and sending H to a single machine can be done in constant MPC rounds. As the MPC model values communication overhead, solving list coloring on the sparsified graph in a single machine is treated as a single round computation.

In the following sections, we prove Theorem 9.1. The proof relies on a structural decomposition of the input graph and a three phase analysis.

9.1 Warm up

We first look at a variant of Theorem 9.1 that uses a larger color palette of $(1+\epsilon)\Delta$ instead of just $(\Delta+1)$ colors. The proof illustrates the idea that having a "slack" of $\epsilon\Delta$ colors allows for a feasible coloring using the subsampled lists. Next, we discuss how Theorem 9.1 holds for two extreme cases when G is sparse/dense. The subsequent analysis in Section 9.3 then interpolates between these two extreme cases.

9.1.1 Using slightly more colors

As a warm up, we consider the process with a larger color palette of $(1+\epsilon)\Delta$ instead of just $(\Delta+1)$ colors, for some constant $\epsilon > 0$.

Claim 9.2. *Let $G(V,E)$ be an n-vertex graph with maximum degree Δ. Suppose for any vertex $v \in V$, we sample $\Theta(\frac{\log n}{\epsilon})$ colors $L(v)$ from $\{1,\ldots,(1+\epsilon)\Delta\}$ independently and uniformly at random. Then with high probability, there exists a proper $(1+\epsilon)\Delta$ coloring of G in which the color for every vertex v is chosen from $L(v)$.*

Proof. Consider an arbitrary vertex v and let $p = \Theta(\frac{\log n}{\epsilon(1+\epsilon)\Delta})$. Since v has at most Δ neighbors, at most Δ colors will be blocked from the possible colors for v. Of the remaining $\geqslant \epsilon\Delta$ colors, at least one will be chosen into $L(v)$ with probability at least $1-(1-p)^{\epsilon\Delta} \geqslant 1-\frac{1}{\text{poly}(n)}$. □

Remark With $(\Delta+1)$ colors, if all vertices pick a single color *at random*, the probability that a given vertex v has a color that is not "blocked" by any neighbor is $(1-\frac{1}{\Delta+1})^{\deg(v)} \geqslant (1-\frac{1}{\Delta+1})^{\Delta} \in \Theta(1)$.

9.1.2 Handling relatively sparse graphs

Consider vertex v of degree Δ with neighborhood $N(v)$. Suppose $G[N(v)]$ is relatively sparse and has at most $(1-\epsilon)\binom{\Delta}{2}$ edges, for some constant ϵ. Denote $\overline{E}$ as the set of non-edges in the subgraph $G[N(v)]$ induced by $N(v)$. By assumption on sparseness of $G[N(v)]$, $|\overline{E}| \geqslant \epsilon\binom{\Delta}{2}$.

We say that edge $\{u, v\} \in \overline{E}$ is *good* if we can color the endpoints u and v with the same first color in their selected lists. That is, both u and v sampled the same first color *and* none of their neighbors sampled it. So, an edge in $\overline{E}$ is good with probability $\geqslant (\Delta+1)(\frac{1}{\Delta+1})^2(1-\frac{1}{\Delta+1})^{2\Delta} \in \Omega(\frac{1}{\Delta})$. Thus, we expect to see $\Omega(\epsilon\Delta)$ good edges. One can show[2] that with probability $1-e^{\Omega(\epsilon\Delta)}$, there are at least $\Omega(\epsilon\Delta)$ good edges in $\overline{E}$.

Let $\overline{E}^{good} \subseteq \overline{E}$ be the good edges. By coloring both endpoints of all good edges with their good color, a "slack" of $\Omega(\epsilon\Delta)$ appears in the difference between the number of remaining free colors and number of uncolored neighbors of v. To be precise,

$$\left((\Delta+1) - \left|\overline{E}^{good}\right|\right) - \left(|N(v)| - 2\left|\overline{E}^{good}\right|\right) \geqslant \Omega(\epsilon\Delta)$$

By an argument similar to Claim 9.2, v can be colored with high probability.

9.1.3 Handling very dense graphs

Let $n = \Delta + 1$. Consider the clique $K_{\Delta+1}$ and its corresponding complete bipartite graph $H = (V_1, V_2)$ between $\Delta+1$ vertices (V_1) and $\Delta+1$ available colors (V_2). By sampling colors into $L(v)$ with probability $p = \Theta(\frac{\log n}{\Delta+1})$, we get a subgraph where an edge exists between a vertex v and a color c if $c \in L(v)$. See Fig. 9.1 for an illustration.

Observe that a perfect matching in the above bipartite graph induces a coloring on the vertices — we can color each vertex $v \in V_1$ by the color $c \in V_2$ that v is matched to. We will argue that there is a perfect matching in the sampled bipartite graph with high probability.

By Hall's theorem[3], each vertex can be matched to some color if for every subset of vertices $S \subseteq V_1$, the total number of covered colors $|N(S)| \geqslant |S|$. Fix subset $S \subseteq V_1$. We expect $|S| \cdot (\Delta+1) \cdot p \in \Theta(|S| \log n)$ edges incident to S. We say color $c \in V_2$ is *covered* if edge $\{v, c\}$ is sampled for some $v \in S$.

- Case 1: When $|S|$ is "small". Say, $|S| \leqslant k \cdot \Delta$ for some constant k.
  For any color c, we expect c to have $|S| \cdot p = |S| \cdot \Theta(\frac{\log n}{\Delta}) \in \mathcal{O}(\log n)$

[2] See Assadi et al. [ACK19, Appendix A, Lemma A.1].

[3] https://en.wikipedia.org/wiki/Hall%27s_marriage_theorem

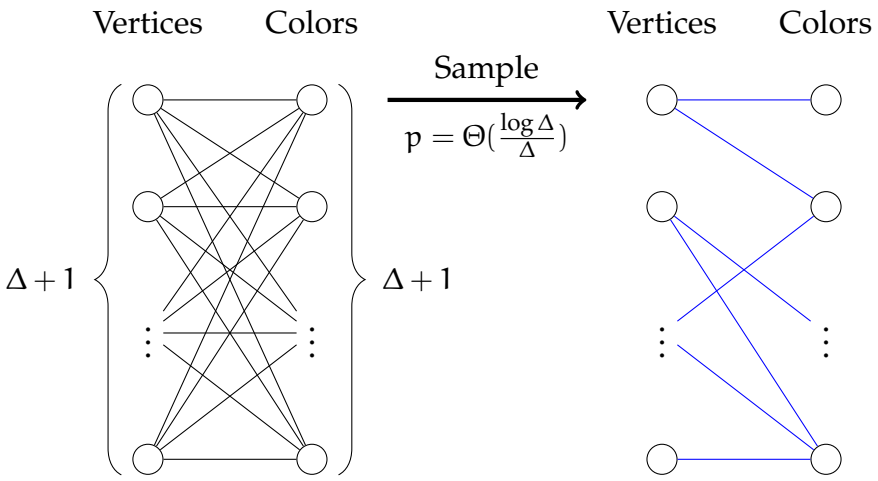


Figure 9.1: Illustration of a bipartite graph between vertices and colors. Blue edges are subset of edges by sampling each edge independently.

sampled neighbors in S. By Chernoff bounds, each color is covered by at most $\mathcal{O}(\log n)$ sampled edges with high probability. Taking union bound over all $(\Delta + 1)$ colors, the sampled edges incident to S covers at least $|S|$ different colors with high probability.

- Case 2: When $|S|$ is "large". Say, $|S| \geqslant k \cdot \Delta$ for some constant k. The probability that color c is not covered by any sampled edge is $(1-p)^{|S|} \leqslant e^{-\Theta(\frac{|S| \log n}{\Delta})} \leqslant e^{-\Theta(\log n)}$. Then, using the inequality $\binom{n}{k} \leqslant n^k$, the probability that $|S|$ colors are not covered is at most

$$\binom{\Delta+1}{|S|} \cdot \left(e^{-\Theta(\log n)}\right)^{|S|} = \binom{n}{|S|} \cdot e^{-\Theta(|S| \log n)} \leqslant \frac{1}{\text{poly}(n)}$$

Taking union bound over all $\binom{\Delta+1}{|S|}$ subsets of size $|S|$, and all $(\Delta+1)$ possible sizes, we have that $|N(S)| \geqslant |S|$ with high probability for any subset $S \subseteq V_1$.

9.2 A structural decomposition

Structural decomposition was first introduced by Reed [Ree98] in the context of the problem of total coloring. See Molloy and Reed [MB02, Chapter 15] for a detailed exposition.

The proof of Theorem 9.1 uses a variant of a network decomposition result from Harris, Schneider and Su [HSS16]. Their decomposition,

henceforth HSS-decomposition, partitions a graph $G(V, E)$ into sparse and dense regions measured with respect to a parameter $\epsilon \in [0, 1)$.

Definition 9.3 (Friends and dense/sparse vertices)**.** *Two adjacent vertices u and v are called* friends *if and only if they share at least $(1-\epsilon)\Delta$ common neighbors. That is, $|N(u) \cap N(v)| \geqslant (1-\epsilon)\Delta$. A vertex v is* ϵ-dense *if v has at least $(1-\epsilon)\Delta$ friends, and* ϵ-sparse *otherwise.*

Claim 9.4. *For a sparse vertex v, the number of edges in the graph induced by $N(v)$ is at most $(1-\epsilon^2)\binom{\Delta}{2}$.*

Proof. If $\deg(v) < \Delta$, we can add dummy vertices that are adjacent only to v. Suppose $\deg(v) = \Delta$. If v is sparse, then it has at least $\epsilon\Delta$ non-friends. Each non-friend can be adjacent to at most $(1-\epsilon)\Delta$ vertices in $N(v)$. So, there are $\geqslant \frac{(\epsilon\Delta)^2}{2}$ edges missing among all possible $\binom{\Delta}{2}$ edges within $N(v)$. □

Let V^{dense} denote the set of dense vertices, and let F denote the set of friendship edges in G. Then, subgraph $H = (V^{dense}, F)$ is the graph induced by dense vertices and their friendship edges. The HSS-decomposition partitions vertices into connected components $C_1, C_2, \ldots, C_k$ of H and a sparse region $V^{sparse} = V \setminus H$. These connected components are also called *almost-cliques* because they are dense regions of the graph. See Fig. 9.2.

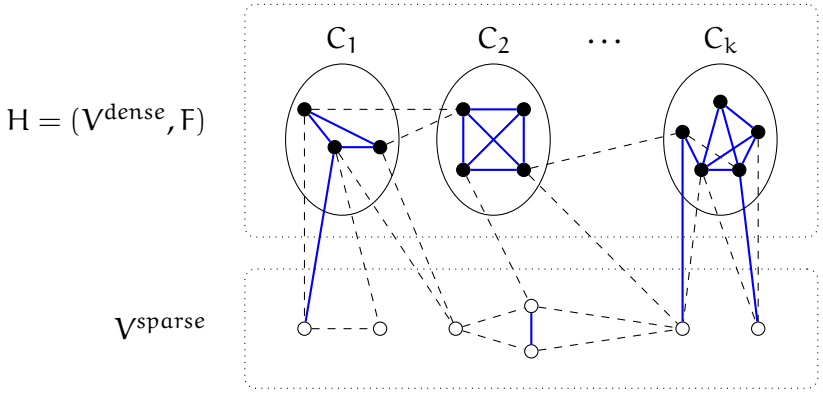


Figure 9.2: HSS-decomposition with $\Delta = 6$ and $\epsilon = \frac{2}{3}$. Vertices are friends if they share $\geqslant 2$ common neighbors. A vertex is dense if it has $\geqslant 2$ friends. Friendship edges are in blue and dense vertices are in black. $H = (V^{dense}, F)$ is induced by black vertices and blue edges. Almost-cliques $C_1, \ldots, C_k$ are connected components of H. Observe that there can be friendship edges missing in H, and non-friendship edges present in an almost-clique.

We state two properties of a HSS-decomposition that hold for vertices in an almost-clique C_i, and give a proof sketch for each of them:

- Property: $v \in C_i$ has at most $\epsilon\Delta$ dense neighbors in G but not in C_i.

  We compute almost-clique C_i by taking the connected component of $H = (V^{dense}, F)$, so any dense neighbor of v connected by a friendship edge will also be in C_i. Dense vertex v has at most $\epsilon\Delta$ non-friendship edges. So, at most $\epsilon\Delta$ dense neighbors of v is not in C_i.

- Property: For any two vertices $u, v \in C_i$, $|N(u) \cap N(v)| \geqslant (1-2\epsilon)\Delta$.

  If u and v are friends, then $|N(u) \cap N(v)| \geqslant (1-\epsilon)\Delta \geqslant (1-2\epsilon)\Delta$. Suppose u and v are not friends. Since C_i is a connected component with respect to friendship edges, there exists a path of vertices $u = z_0, z_1, \ldots, z_k = v$ in C_i where any two adjacent vertices are friends. We will argue that the shortest such $u-v$ path between has at only one intermediate vertex. Since adjacent vertices are friends in the path, $|N(z_i) \cap N(z_{i+1})| \geqslant (1-\epsilon)\Delta$ for $i \in \{0, \ldots, k-1\}$. Since each vertex has at most Δ neighbors, there are at most $\epsilon\Delta$ neighbors of z_{i+1} that are not neighbors of z_i or neighbors of z_{i+2}. So, $|N(z_i) \cap N(z_{i+2})| \geqslant (1-2\epsilon)\Delta$ for $i \in \{0, \ldots, k-2\}$. This means that z_i and z_{i+2} shares a common neighbor. Therefore, the shortest such $u-v$ path has only one intermediate vertex (their common neighbor z_1). Thus, $|N(u) \cap N(v)| = |N(z_0) \cap N(z_2)| \geqslant (1-2\epsilon)\Delta$.

Assadi et al. [ACK19, Section 2.2] defined a slightly different extension[4] of the HSS-decomposition with the following 4 properties[5]:

1. Vertex $v \in C_i$ has at most $7\epsilon\Delta$ neighbors (both dense and sparse) in G but not in C_i.

2. Vertex $v \in C_i$ has at most $6\epsilon\Delta$ non-neighbors in G that are also in C_i.

3. For any two vertices $u, v \in C_i$, we have $|N(u) \cap N(v)| \geqslant (1-2\epsilon)\Delta$.

4. $(1-\epsilon)\Delta \leqslant |C_i| \leqslant (1+6\epsilon)\Delta$

Henceforth, whenever we refer to a HSS-decomposition, we mean one with the above 4 properties. Note that the decomposition is purely for analysis and we do not explicitly build it while computing a vertex coloring.

[4] By considering a vertex sparse if it is at least (2ϵ)-sparse and at most ϵ-sparse.
[5] See Assadi et al. [ACK19, Lemma 2.3].

9.3 A three phase analysis

Recall that we sample a list of colors $L(v)$ for each vertex v. For the purpose of analysis, we consider $L(v)$ as a union of three sets $L_1(v)$, $L_2(v)$ and $L_3(v)$. Each $L_i(v)$ is created by picking each color in $\{1, \ldots, \Delta + 1\}$ independently and with probability $p = \Theta(\frac{\log n}{\Delta+1})$. So, the list $L(v) = L_1(v) \cup L_2(v) \cup L_3(v)$ has size $\Theta(\log n)$ with high probability. By the HSS-decomposition, we can partition the vertices of the input graph into $(V^{sparse}, C_1, \ldots, C_k)$.

Ideally, we would like to color vertices in V^{sparse} in a way similar to Section 9.1.2, and each almost-clique C_i in a way similar to Section 9.1.3. However, vertices in V^{sparse} may be adjacent to vertices in almost-cliques. After coloring the sparse vertices, some colors will be blocked from possible valid colors for the dense vertices. Sampling edges from the resultant bipartite graph (See Fig. 9.1) may not yield a subgraph which fulfills the conditions of Hall's theorem. Hence, an intermediate phase is introduced to "fix" the almost-cliques. The analysis proceeds in three phases:

1. Using $\{L_1(v)\}_{v \in V}$, color the sparse region V^{sparse} similar to Section 9.1.2.
2. Using $\{L_2(v)\}_{v \in V}$, "fix" each C_i via colorful matchings (to be defined).
3. Using $\{L_3(v)\}_{v \in V}$, color the rest of each C_i similar to Section 9.1.3.

Each phase partially colors the graph G while leaving colored vertices untouched such that all vertices are colored at the end of the third phase.

9.3.1 Phase 1

By Claim 9.4, the number of edges in the graph induced by the neighborhood $N(v)$ of a sparse vertex v is at most $(1 - \epsilon^2)\binom{\Delta}{2}$. By the argument from Section 9.1.2, each sparse vertex can be colored from $L_1(v)$ with high probability. In this phase, we color of vertices $v \in V^{sparse}$ while leaving the vertices $v \in V^{dense}$ uncolored.

9.3.2 Phase 2

The purpose of this phase is to "fix" each almost-clique such that phase 3 can use a similar argument to Section 9.1.3. To do so, we will attempt to remove colors and vertices at the rate of 2 vertices per color.

Fix an arbitrary almost-clique C_i and consider its complement graph $\overline{C_i}$. Observe that any two adjacent vertices in $\overline{C_i}$ can be colored by the same

color and hence creating a unit of "slack" in the number of remaining colors versus the number of remaining vertices in C_i.

Phase 2 iterates through almost-cliques $C_1, \ldots, C_k$ using $\{L_2(v)\}_{v \in V}$. For each almost-clique, say with average degree $\overline{d}$, we find a *colorful matching* of size $4\overline{d}$ and color their endpoints accordingly.

Definition 9.5 (Colorful matching)**.** *We say that a matching M_i in the complement graph $\overline{C_i}$ of an almost-clique is a coloring matching if and only if*

- *For any edge $\{u, v\} \in M_i$, there is a color $c \in L_2(u) \cap L_2(v)$ such that c has not been taken by colored neighbors of u and v.*
- *All edges in M_i are colored differently.*

We define the following potential function a_D with respect to a subset of colors $D \subseteq \{1, \ldots, \Delta + 1\}$. For an edge $e = \{u, v\}$, $a_D(e)$ is the number of possible colors in D that are still valid for both u and v. For a set of edges E', we naturally define $a_D(E') = \sum_{e \in E'} a_D(e)$. Lemma 9.6 tells us that for once we accumulated "sufficient" potential of $a_D(E(C'))$, we can extract at least one colored edge.

Lemma 9.6. *Consider any arbitrary complement graph $\overline{C}$ of an almost-clique. Fix a subgraph $C' \subseteq \overline{C}$ and a subset of colors D such that $a_D(E(C')) \geqslant 120\epsilon^2\Delta^2$. If each node samples each color in D with probability $\frac{20 \log n}{\epsilon\Delta}$, then there is some colorful edge in C' with high probability.*

Proof. Let $I(e, q)$ be the indicator variable whether edge e can be colored by color q. Recall the definition of the potential function a_D:

$$a_D(E(C')) = \sum_{e \in C'} a_D(e) = \sum_{e \in C'} \sum_{q \in D} I(e, q) = \sum_{q \in D} \sum_{e \in C'} I(e, q)$$

We now consider $\log n$ phases where every vertex samples each color in D with probability $\frac{20}{\epsilon\Delta}$ in each phase. We will show that each phase succeeds in finding a colorful edge with constant probability, hence $\log n$ phases ensure that we find one with high probability.

Fix a phase. For a color q, an edge in $\{e \in C' : I(e, q) = 1\}$ is sampled with probability $(\frac{20}{\epsilon\Delta})^2$, if both endpoints sampled color q. So, we expect $\sum_{e \in C'} I(e, q) \cdot \frac{40}{\epsilon^2\Delta^2}$ edges to be sampled for color q. Over all $q \in D$ colors,

$$\sum_{q \in D} \sum_{e \in C'} I(e, q) \cdot \frac{40}{\epsilon^2\Delta^2} = \frac{40}{\epsilon^2\Delta^2} \cdot \sum_{q \in D} \sum_{e \in C'} I(e, q) \geqslant \frac{40}{\epsilon^2\Delta^2} \cdot 120\epsilon^2\Delta^2 \in \Omega(1)$$

where the inequality is from the assumption that $a_D(E(C')) \geqslant 120\epsilon^2\Delta^2$. Then, using variance arguments, one can argue that a colorful edge is sampled with constant probability in one phase. Hence, $\log n$ phases ensure that we find one with high probability. □

Claim 9.7. *With high probability, $\overline{C_i}$ has a colorful matching of size $4\overline{d}$, where $\overline{d}$ is the average degree of $\overline{C_i}$.*

Proof. We start with $D = \emptyset$ and iteratively add colors to D as we walk over the colors $\{1, \ldots, \Delta+1\}$ one by one. When we find a colorful edge $\{u, v\}$ using color $c \in D$, we remove c from D and remove u, v from $\overline{C_i}$.

From property 2 of HSS-decomposition, a vertex $v \in C_i$ has at most $6\epsilon\Delta$ non-neighbors in G that are also in C_i. This means that removal of a vertex decreases $a_D(E(\overline{C_i}))$ by at most $6\epsilon\Delta \cdot \Delta = 6\epsilon\Delta^2$. Meanwhile, property 4 of HSS-decomposition tells us that there are at most $(1+6\epsilon)\Delta$ vertices in $\overline{C_i}$, so removal of a color from D decreases $a_D(E(\overline{C_i}))$ by at most $a_{\{c\}}(E(\overline{C_i})) = \overline{d} \cdot \frac{|\overline{C_i}|}{2} \leqslant \overline{d}\Delta \leqslant 6\epsilon\Delta^2$. Therefore, removal of c, u and v cause a decrease in $a_D(E(\overline{C_i}))$ by at most $6\epsilon\Delta^2 + 2 \cdot 6\epsilon\Delta^2 = 18\epsilon\Delta^2$.

Lemma 9.6 tells us that a colorful matching can be found with high probability after accumulating a potential of $120\epsilon^2\Delta^2$ while iterating through the colors. This means that we lose at most $138\epsilon^2\Delta^2$ potential each time we find and add an edge to the colorful matching.

From property 1 of HSS-decomposition, a vertex $v \in C_i$ has at most $7\epsilon\Delta$ neighbors in G but not in C_i. So, there are at least $(1-14\epsilon)\Delta$ available colors for an edge $e = \{u, v\} \in \overline{C_i}$ even if we have assigned colors to all neighbors of u and v that are outside of C_i. That is, $a_D(e) \geqslant (1-14\epsilon)\Delta$ for any edge $e \in \overline{C_i}$. From property 4 of HSS-decomposition, there are at least $(1-\epsilon)\Delta$ vertices in C_i. So, the total potential of $\overline{C_i}$ considering all colors is

$$\begin{aligned}
a_{\{1,\ldots,\Delta+1\}}(E(\overline{C_i})) &\geqslant |E(\overline{C_i})| \cdot (1-14\epsilon)\Delta && \text{since } a_D(e) \geqslant (1-14\epsilon)\Delta \\
&= \frac{\overline{d} \cdot |\overline{C_i}|}{2} \cdot (1-14\epsilon)\Delta \\
&\geqslant \frac{1}{2}(1-\epsilon)(1-14\epsilon)\overline{d}\Delta^2 && \text{since } |\overline{C_i}| \geqslant (1-\epsilon)\Delta \\
&\geqslant 0.45\overline{d}\Delta^2 && \text{for sufficiently small } \epsilon
\end{aligned}$$

Since $a_{\{1,\ldots,\Delta+1\}}(E(\overline{C_i})) \geqslant 0.45\overline{d}\Delta^2$, we would be able to extract at least $\frac{0.45\overline{d}\Delta^2}{138\epsilon^2\Delta^2} \geqslant \frac{\overline{d}}{320\epsilon}$ colorful edges after iterating through all $(\Delta+1)$ colors. For a sufficiently small ϵ, $\frac{\overline{d}}{320\epsilon} \geqslant 4\overline{d}$. □

9.3.3 Phase 3

In similar spirit to Section 9.1.3, we construct a bipartite graph and argue that the edges due to $L_3(v)$ allow for a matching that colors all remaining uncolored vertices in each almost-clique. Morally, we want to check for Hall's condition on the sampled graph as per Section 9.1.3.

Fix an arbitrary almost-clique C_i and consider the bipartite graph $H = (V_1, V_2)$ where V_1 is the set of uncolored vertices of C_i and V_2 is the set of colors $\{1, \ldots, \Delta + 1\}$. An edge exists between vertex $v \in V_1$ and color $c \in V_2$ if and only if the color c is not used by any of v's colored neighbors. For each vertex v, its color list $L_3(v)$ can be seen as being constructed by independently sampling each incident edge in H with probability $p \in \Theta(\frac{\log n}{\Delta+1})$. If the resultant graph has a matching of size $|V_1|$, then the remaining uncolored vertices of C_i can be colored.

Lemma 9.8 (Lemma 3.5 of Assadi et al. [ACK19]). *Suppose $N \leqslant n$ and $0 \leqslant \delta \leqslant \frac{1}{12}$. Let $H = (V_1, V_2)$ be a bipartite graph such that:*

1. *$|V_1| \leqslant (1 - 3\delta)N$ and $|V_2| \leqslant 2N$*
2. *Each vertex in V_1 has degree at least $\frac{2N}{3}$ and at most N*
3. *The average degree of vertices in V_1 is at least $(1 - \delta)N$*

Then, a subgraph of H obtained by sampling each edge with probability $p = \frac{90 \log n}{N}$ has a matching of size $|V_1|$ with high probability.

Recall that colorful edges remove 2 vertices for 1 color. By Claim 9.7, $4\overline{d}$ colorful edges can be found and removed, where $\overline{d}$ is the average degree of V_1. So, after phase 2, one can show[6] that the bipartite graph H fulfills the conditions of the above lemma. Thus, all remaining uncolored vertices in each almost-clique can be colored with $\{L_3(v)\}_{v \in V}$ with high probability.

[6] See Assadi et al. [ACK19, Lemma 3.4].

Bibliography

[ACK19] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for (δ+ 1) vertex coloring. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 767–786. Society for Industrial and Applied Mathematics, 2019.

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[AMN+98] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 574–583. ACM, 2014.

[ASS+18] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 674–685. IEEE, 2018.

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[BENW15] Rafael Barbosa, Alina Ene, Huy Nguyen, and Justin Ward. The power of randomization: Distributed submodular maximiza-

tion on massive datasets. In *International Conference on Machine Learning*, pages 1236–1244, 2015.

[BENW16] Rafael da Ponte Barbosa, Alina Ene, Huy L Nguyen, and Justin Ward. A new framework for distributed submodular maximization. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 645–654. Ieee, 2016.

[Ber57] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the United States of America*, 43(9):842, 1957.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

[BMV+12] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[CŁM+18] Artur Czumaj, Jakub Łącki, Aleksander Mądry, Slobodan Mitrović, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 471–484. ACM, 2018.

[CW79] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[DFK+04] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V Vinay. Clustering large graphs via the singular value decomposition. *Machine learning*, 56(1-3):9–33, 2004.

[DG99] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of the johnson-lindenstrauss lemma. *International Computer Science Institute, Technical Report*, 22(1):1–5, 1999.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DP10] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 673–682. IEEE, 2010.

[EGS08] David Eppstein, Michael T Goodrich, and Jonathan Z Sun. Skip quadtrees: Dynamic data structures for multidimensional point sets. *International Journal of Computational Geometry & Applications*, 18(01n02):131–160, 2008.

[Epp95] David Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete & Computational Geometry*, 13(1):111–122, 1995.

[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*, pages 448–455. ACM, 2003.

[GGK+18] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. Improved massively parallel computation algorithms for mis, matching, and vertex cover. *arXiv preprint arXiv:1802.08237*, 2018.

[Gha17] Mohsen Ghaffari. Distributed mis via all-to-all communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 141–149. ACM, 2017.

[GKMS18] Buddhima Gamlath, Sagar Kale, Slobodan Mitrović, and Ola Svensson. Weighted matchings via unweighted augmentations. *arXiv preprint arXiv:1811.02760*, 2018.

[GKU19] Mohsen Ghaffari, Fabian Kuhn, and Jara Uitto. Conditional hardness results for massively parallel computation from distributed lower bounds. 2019. Manuscript.

[GN19] Mohsen Ghaffari and Krzysztof Nowicki. Faster algorithms for edge connectivity via random out contractions. 2019. Manuscript.

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In

International Symposium on Algorithms and Computation, pages 374–383. Springer, 2011.

[GU19] Mohsen Ghaffari and Jara Uitto. Sparsifying distributed algorithms with ramifications in massively parallel computation and centralized local computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1636–1653. SIAM, 2019.

[HK73] John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[HMK+06] Tracey Ho, Muriel Médard, Ralf Koetter, David R Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, 2006.

[HSS16] David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed (âĹĘ+ 1)-coloring in sublogarithmic rounds. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 465–478. ACM, 2016.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, volume 41, pages 59–72. ACM, 2007.

[IMS17] Sungjin Im, Benjamin Moseley, and Xiaorui Sun. Efficient massively parallel methods for dynamic programming. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 798–811. ACM, 2017.

[KMVV15] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *ACM Transactions on Parallel Computing (TOPC)*, 2(3):14, 2015.

[KMW04] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. What cannot be computed locally! In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 300–309. ACM, 2004.

[KSV10] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[Lin87] Nathan Linial. Distributive graph algorithms global solutions from local data. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 331–335. IEEE, 1987.

[Lin92] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[Llo82] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[LV18] Paul Liu and Jan Vondrak. Submodular optimization in the mapreduce model. *arXiv preprint arXiv:1810.01489*, 2018.

[LW10] Christoph Lenzen and Roger Wattenhofer. Brief announcement: Exponential speed-up of local algorithms using non-local communication. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 295–296. ACM, 2010.

[MB02] Molloy Michael and Reed Bruce. Graph coloring and the probabilistic method. *New York I Springer*, 23:1329–356, 2002.

[McG05] Andrew McGregor. Finding graph matchings in data streams. In *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*, pages 170–181. Springer, 2005.

[MKSK13] Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *Advances in Neural Information Processing Systems*, pages 2049–2057, 2013.

[MV80] Silvio Micali and Vijay V Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27. IEEE, 1980.

[NW64] C St JA Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1(1):12–12, 1964.

[NY18] Jelani Nelson and Huacheng Yu. Optimal lower bounds for distributed and streaming spanning forest computation. *arXiv preprint arXiv:1807.05135*, 2018.

[Ree98] Bruce Reed. ω, δ, and χ. *Journal of Graph Theory*, 27(4):177–212, 1998.

[Ros73] Arnold L Rosenberg. On the time required to recognize properties of graphs: A problem. *ACM SIGACT News*, 5(4):15–16, 1973.

[RV75] Ronald L Rivest and Jean Vuillemin. A generalization and proof of the aanderaa-rosenberg conjecture. In *Proceedings of the seventh annual ACM symposium on Theory of computing*, pages 6–11. ACM, 1975.

[RVW18] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R Wang. Shuffles and circuits (on lower bounds for modern parallel computation). *Journal of the ACM (JACM)*, 65(6):41, 2018.

[SSS95] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[WC81] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

[Whi12] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.