COMP_SCI 496: Graduate Algorithms

Randy Truong

Spring 2024

Contents

Unit 1:
Probability + Graph Theory

Chapter 1:
A Prelude in Graph Theory and Combinatorics

This is a chapter that is devoted to Randy learning how graph theory works in finer granularity.

1 Subgraphs

Subgraph. Given a graph $G = (V, E)$, a subgraph of G, which is denoted as $G' = (V', E')$ is a graph whose vertices V' are a subset of V and whose edges E' are a subset of E.

2 Induced Subgraphs

Induced Subgraph. Given a graph $G = (V, E)$, an induced subgraph $G' = (V', E')$ is a subgraph of G that contains the following property:

- all edges $e' \in E'$ must have (both) endpoints be in V'

We define the subgraph induced by V', where V' is a set of vertices $V' \subseteq V$, as being the subgraph $G' = (V', E')$ in which all $e' \in E'$ must have both endpoints $r'_1, r'_2 \in V'$.

3 Graph Orientations

Orientation. Given an undirected graph G, an *orientation* of the graph G would be the resulting graph in which we assign each edge a direction, resulting in a directed graph.

4 Tournaments

Tournaments. Given a set of vertices V, a tournament of V, denoted as T on V, is an *orientation* of the vertices, such that the resulting graph is connected.

- Note, in a tournament, two endpoints i, j can only have a single edge between them, in which $i \to j$ or $j \to i$, but not both

5 Dominating Sets

Dominating Set. A *dominating set* of an undirected graph $G = (V, E)$ is a set $U \subseteq V$ such that every vertex $v \in V - U$ has at least one neighbor in U.

Chapter 2:
The Basic Method

6 The Probabilistic Method

- Powerful tool for tackling problems in discrete math
- Main idea of the method
  - **Objective.** We see kto prove that a structure with certain desired properties *exists*
  - We first define an appropriate probabilisty space of structures
  - then, we show that the desired properties hold in these structures with positive probability

7 Example 1

We note that the *Ramsey number* $R(k, \ell)$ is the smallest integer n such that in any wo-coloring of the edges of a complete graph on n vertices K_n by red and blue, either there is a red K_k (ie, a ocmplete subgraph on k vertices all of whose edges are colored red) or there is a blue K_ℓ.

- Ramsey showed that $R(k, \ell)$ is finite for any two integers k and l.
- We can obtain a lower bound for hte diagonal Ramsey numbers $R(k, k)$
  - **Remark (Complete Graph).**
    A *complete graph* K on n (which is denoted as K_n) is a graph in which each pair of distinct n vertices is connected together by an edge.

7.1 Prop. 1

If $\binom{n}{k} \cdot 2^{1-\binom{k}{2}} < 1$, then $R(k, k) > n$. Thus $R(k, k) > \left\lfloor 2^{\frac{k}{2}} \right\rfloor$ for all $k \geq 3$

- Note, here $R(k, k)$ represents a graph R in which the induced monochromatic graphs must *both* be of size k, which is a stronger condition than $R(k, \ell)$

Proof. In Section 7.1, we first derive the following:

- We first seek out the probability that an induced graph K_n is monochromatic, which is equiv. to

$$\Pr[A_R] = \left(\frac{1}{2}\right)^{\binom{n}{2}} \to 2^{\binom{n}{2}} \leftrightarrow 2^{1-\binom{n}{2}} \tag{1}$$

- We next seek the probability that any induced graph (aka a n-combination of the original k-graph) is monochromatic

$$\binom{k}{n} \cdot 2^{1-\binom{n}{2}} \tag{2}$$

- **Observation 1.** We understand that the probability of this event occuring must be bounded by 1, thus leading us to the conclusion that

$$\binom{k}{n} \Pr[A_R] < 1 \tag{3}$$

  which must imply that the probability that event $\Pr[A_R]$ doesn't occur must be non-zero
  - What exactly is the *negation* of event A_R? If A_R denotes the event that the induced subgraph of K_k on R is monochromatic, then $\neg(A_R)$ must be the probability that a two-coloring of the graph R does *not* produce a monochromatic induced subgraph

  - We would denote this is as the Ramsey number of induced grpah sizes n and n, or $R(n, n)$. Given that we want the negation, then we know that the size must be greater than n_0, since the Ramsey number must be n_0
- **Observation 2.** We understand that if the size of the induced graph, denoted as $n \geq 3$ and if we take the size of the graph k to be $k = \left\lfloor 2^{\frac{k}{2}} \right\rfloor$, then we know that

$$\binom{k}{n} 2^{1-\binom{n}{2}} < \left(\frac{2^{1+\frac{n}{2}}}{n!} \right) \cdot \left(\frac{k^n}{2^{\frac{n^2}{2}}} \right) < 1 \tag{4}$$

  Thus, $R(n, n) > \left\lfloor 2^{\frac{n}{2}} \right\rfloor \forall n \geq 3$

8 Optional Proof Notes

- Because there are $\binom{k}{n}$ choosings of the graph G, then it follows that at least one of these events occurring must be non-zero but strictly less than 1.
  - Thus, the inverse of this statement must be true- ($\exists$ a two-coloring of the graph G of k vertices G_k that doesn't have a monochromatic induced graph G_n)
  - This is represented as the Ramsey number $R(k, k)$, which semantically equates to smallest size of a graph that has a monochromatic edge-colored subgraph of size k
  - Given that we know that the Ramsey number refers to the smallest n for which there is a monochromatic induced subgraph of sizes k or ℓ, then it follows that in order for both monographic subgraph sto be of sizes k, then the graph must be at least the size of the graph whose Ramsey number $R(k, \ell)$ is n.

9 The Essence of the Probabilistic Method

Note the way that we evaluated this problem.

- **Objective.** Prove the existence of a good coloring K_n given a graph K.
  - We first defined what a "good" coloring was– which was a non-monochromatic graph formed from the induced graph of the two-colored graph.
  - Then, we showed that it *exists*, in a nonconstructive way
    - We defined a probability space of events, and we narrowed that probability space down to events that described structure sof particular properties
    - From there, we then just showed that the desired properties that we want will hold in this narrowed down probability space, with positive probability.

10 Why is this approach effective?

This approach is effective because the vast majority of probability spaces in combinatorial problems are *finite*

- Sure, we could use an algorithm to try and find such a structure with a particular property
- For example, if we wanted to actually find an edge two-coloring of K_n without a monochromatic induced graph, we could just iterate through all possible edge-colorings and find their induced grpahs.
  - Obviously, this is impractical (it's actually class $\mathbb{P}$ haha)
  - Although these problems could be solved using *exhaustive searches*, we want a faster way.
  - This is the difference between *constructivist* and *nonconstructivist* ideas in proofs
    - Although we don't have a deterministic way of forming the graph, we are able to define an algorithm that could potentially lead to the desired graph, which, which is more effective than just trying to deterministically create one
    - In the case of the Ramsey-number problem, it would be more effective to find a good coloring (a non-monochromatic induced graph) by just letting a fair coin toss decide on how to color the nodes

11 Second Look at the Probabilistic Method

12 Property S_k

We state that a tournament T has the property S_k if and only if, for every set of k Players, there is one that beats them all./

- Formally, this would mean that given a tournament $T = \langle V, E \rangle$ and subsets K of size k

$$\exists v \in T - K : (v, k) \forall k \in K \tag{5}$$

Claim. *Is it true that for every finite k that there exists a tournament T (on more than k vertices) with the property S_k?*

Proof. In order to prove this, let us consider a random tournament T.

Given this random tournament T, let's determine the probability that a node v in $T - K$ beats all of the nodes $j \in K$. This is a difficult probability to calculate, however, and it is this probability as the complement of its negation (that there isn't a node in $V - K$ that beats all the nodes $j \in K$).

Let us find probability that a fixed node v in $V - K$ beats all the nodes $j \in K$.

- **Remark (Tournamentk).**
  Because T is a tournament, we know that if we're considering a vertex v, it must be connected to all of the nodes within the subset K. Thus, there is a $\frac{1}{2}$ probability with which the edge with endpoints $v, j : j \in K$ is directed $v \to j$.

  - Since there are k nodes in K and that the event of v beating a vertex j is independent, then we just find the product

$$\Pr(v \text{ beats them all}) \to \prod_{1}^{k} \left(\frac{1}{2}\right) \to \left(\frac{1}{2}\right)^{k} \leftrightarrow (2)^{-k} \tag{6}$$

From this, it follows that the probability that v doesn't beat them all is given by

$$\begin{aligned}\Pr(v \text{ does not beat them all}) &= \\ &= (1 - \Pr(v \text{ beats them all})) \\ &= \left(1 - 2^{-k}\right)\end{aligned} \tag{7}$$

Now, we simply just need to find the probability that *any* fixed v doesn't beat them all.

$$\begin{aligned}\Pr(\text{no vertex beats them all}) &= \\ &= (\text{number of possible } v \in V - K) \\ &\times \Pr(v \text{ doesnot beat them all}) \\ &= \prod_{v \in V - K} \Pr(v \text{ does not beat them all}) \\ &= \prod_{v \in V - K} \left(1 - 2^{-k}\right) \\ &= \left(1 - 2^{-k}\right)^{n-k}\end{aligned} \tag{8}$$

Finally, we just need to consider this scenario for all subsets K of size k in V

$$\sum_{\substack{K \subset V \\ |K|=k}} \Pr(\text{no vertex beats them all}) =$$

$$= \binom{n}{k} \Pr(\text{no vertex beats them all}) \tag{9}$$

$$= \binom{n}{k} \left(1 - 2^{-k}\right)^{n-k}$$

Unit 2:
Randomization Algorithms

Figure 1: how it feels to learn probability for the first time while literally in a graduate class that uses probability

Chapter 3: Hashing (TODO)

Chapter 4:
Universal and Perfect Hashing (TODO)

Chapter 5:
Bloom Filters (TODO)

Chapter 6:
Balls and Bins (TODO)

Chapter 7:
Power of Random C=hoi es (TODO)

Chapter 8:
Power of 2 Random Choices (TODO)

Chapter 9:
Hypercubes + Permutation Routing (TODO)

Unit 3:
Streaming Algorithms

Figure 2:

Chapter 10:
Motivation: Finding the Most Frequent Elements (TODO)

Chapter 11:
Misra-Gries Algorithm (TODO)

Chapter 12: Count-Min Sketch (TODO)

Chapter 13:
Counting Distinct Elements (TODO)

Unit 4: Online Algorithms

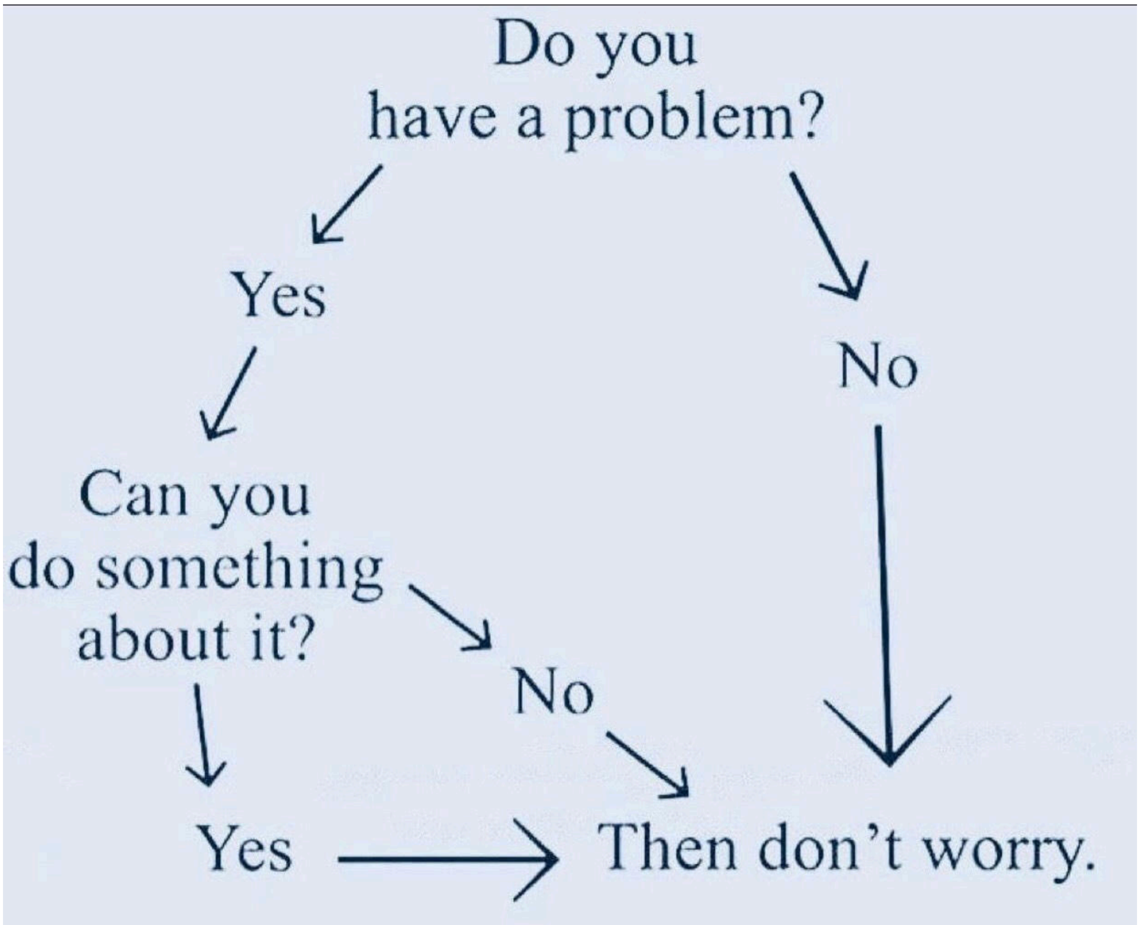


Figure 3: balling

Chapter 14:
Online Algorithms (Part 1)

13 Motivation

- In the former lectures, we utlized and proved the runtime and correctness of algorithms that approximated the solutions of computationally difficult problems
  - However, we now move on to a new problem:

...how can we optimize the solution to a problem in which the algorithm doesn't have all of the information that it needs?...

...by using ***Online Algorithms!***

Definition 1 (Online Algorithms).
Algorithms used in settings where data/inputs arrive *over time,* thus requiring us to *make decisions on the fly,* without knowing what's going to happen in the future.

Of course, the opposite type of algorithm is exactly what we're used to.

- Using bubble sort? You usually know all of the inputs beforehand
- We can think of data structures themselves as being inherently online algorithms, since they handle sequences of requests, without knowledge of the future

14 List of Classical Online Algorithm Problems

- Rent or buy? (Ski rental problem)
- The elevator problem

15 The Ski Rental Problem (Rent or Buy?)

- This is the problem statement:

"Say you are just starting to go skiing. You can either rent skis for $50 or buy them again for $500. You don't know if you're going to enjoy skiing, so you opt to rent. Then you decide to go again, and again, and after a while, you realize that you have shelled out more money renting and wish you had bought right at the start."

- The intuitive solution:
  - In order to optimally solve this problem, we just need to set a

threshold. If we plan on going to ski for more than 10 days, then just buy the skis. Otherwise, if you don't plan on skiing for more than 10 days, then just rent them.

- **Objective.** What is the optimal strategy for saving money, assuming that you didn't know how often you were going to ski?

16 What exactly makes an online algorithm solution (or any solution in this case) *good?*

- We consider the *competitive ratio* of the algorithm.

Definition 2 (Competitive Ratio).

The *competitive ratio* of an online algorithm defines the **worst- case** ratio of the *online algorithm ALG* would result in on an input sequence I to the cost of the optimal offline algorithm *OPT*.

$$\text{competitive ratio} = \frac{ALG(I)}{OPT(I)} \tag{10}$$

Example (Competitive Ratio 1).

- **Case 1 (Buy instantly).** Let us consider the algorithm *ALG* in which we just buy the skis instantly. We know that the worst case scenario for this problem occurs if we only ski once, ie $I = \{1\}$. If we utilize the competitive ratio definition, we observe

$$\frac{ALG(I)}{OPT(I)} = \frac{500}{50} = 10. \tag{11}$$

Equation 11 semantically equates to "the online algorithm *ALG* is ten times worse than the most optimal offline algorithm *OPT*."

Example (Competitive Ratio 2).

- **Case 2 (Rent forever).** Let us consider the algorithm *ALG* in which we just continuously rent. The worst case set of futures I is the same, as it is possible we just ski for one time. This results in

$$\frac{ALG(I)}{OPT(I)} = \frac{50 \times \infty}{50} = \infty. \tag{12}$$

This ratio is unbounded, so we can safely say that this particular *ALG* is not very good lol

Example (Competitive Ratio 3).

Let us now consider a decent strategy, which we call the *better-late-than-never* algorithm. This algorithm, denoted as *ALG*, simply states that we just rent the skis until we realize that we should have just bought them. At that point, we just buy the skis.

- Formally, if rental cost is r and the purchase cost is p, then the algorithm is to rent $\lceil \frac{p}{r} \rceil - 1$ times, then buy.

Theorem (Better-Late-Than-Never Performance).

The algorithm *better-late-than-never* has a competitive ratio ≤ 2. If the purchase cost p is an integer multiple of the rental cost r, then the competitive ratio is $2 - \left(\frac{r}{p}\right)$

Proof of BLTN Performance.

We can prove this theokem directly via case analysis.

- **Case 1.** If you went skiing less than $\lceil \frac{p}{r} \rceil$ times, which for $p = 500$ and $q = 50$ would be less than 10 times, then you are performing optimally
  - This is because all you can do is rent, otherwise if you buy, then you're wasting money!
- **Case 2.** If you go skiing $\geq \lceil \frac{p}{r} \rceil$ times, then the best solution otherwise would have just been to buy the skis from the start or $OPT = p$. We find that in the circumstance that we do buy, *ALG* will pay $\left(r \times \left(\lceil \frac{p}{r} \rceil - 1\right)\right) + p$ which is essentially equivalent to the rental price r multiplied by the maximum number of times we rent without exceeding p, plus the p we pay whenever we buy the skis. We know that arithmetically $r \times \left(\lceil \frac{p}{r} \rceil - 1\right) + p$ must be less than or equal to $2p$. If p is a multiple of r, then it will just be $r \times \left(\left(\frac{p}{r}\right) - 1\right) + p$, which is equal to $p - r + p \Rightarrow 2p - r$

For case 1, we demonstrated that the competitive ratio was 1. In the second case, we demonstrated that the competitive ratio was ≤ 2. Given that the worst case is 2, this must be the competitive ratio. ■

Theorem (BLTN Optimality).
Algorithm BLTN has the best possible competitive ratio for the ski-rental problem for determiinstic algorithms when p is a multiple of r.

Proof of BLTN Optimality.
Let us consider the event that the day that you purchase the skis is the last day that you even use them.

- Consider first that this is **feasible**, since
  1. *ALG* never purchases $\Rightarrow$ competitive ratio is unbounded, which is undesired
  2. *ALG* is *deterministic*, which implies that a purchase *must* occur at some point.

Now that we've established that it *is* possible that we may purchase the skis and never use them again, let us consider the cases in which we rent *more* times than BLTN as well as *less* times:

- Renting longer than BLTN $\Rightarrow r$ increases, which implies that the competitive ratio must increase, making the algorithm worse
- Renting less than BLTN $\Rightarrow$ ratio of $\frac{r}{p}$ decreases, but this must imply that both the denominator and the numerator must decrease by $k \times r$, which also increases the competitive ratio, which is still worse.

■

Note.
i definitely need to look this over bc i need to mathematically work out why this is sound sad

Chapter 15:
Online Algorithms (Part 2)

17 From last time...

In the last lecture, we were introduced to online algorithms as well as the ski rental problem.

Remark (Online Algorithms).
Online algorithms are algorithms that attempt to solve a problem *without* knowing the entire input space. Online algorithms will optimize their answer as inputs come in, ie *on the fly*.

Remark (Ski Rental Problem).
The ski rental problem is as follows:

> *Say you are just starting to go skiing.*
> *You can either rent skis for \$50 or buy them again for \$500. You don't know if you're going to enjoy skiing, so you opt to rent. Then you decide to go again, and again, and after a while, you realize that you have shelled out more money renting and wish you had bought right at the start.*

In the previous lecture, we discussed a **deterministic** strategy for evaluating this problem.

- Our strategy, the better-late-than-never strategy, was as follows:
  - If the cost of renting was about to exceed the cost of buying the skis, then we would simply just by the skis at that point.
  - We proved that in such a case, the online algorithm, denoted as ALG, would require that the customer rents the skis for $\lceil \frac{p}{r} \rceil - 1$ times before buying the skis outright in order to guarantee the minimal loss.
- Costs of rent vs buying
  - Best evaluated using *randomization*

Remark (k-competitivity).
An algorithm is α-competitive iff

$$\mathbb{E}[ALG(I)] \leq \alpha \cdot OPT(I) \tag{13}$$

18 Algorithm

(1) Pick a "random" threshold $T \in [0, B]$

Remark (Deterministic Online Algorithms).
T was $B - 1$ in deterministic variant

(2) Rent for the first T days
(3) Then, buy

- Picking distribution for T?
  - Evaluate it utilizing a differnetial equation

- In principal T must be a continuous random variable for large values of B
  - Assume that we can rent up until 4.75 days, then buying

19 Picking a distribution for T

$$\begin{aligned} \text{rental cost} &= \sum_i = 1^n \Pr(\text{still rent on day } i) \\ \text{rental cost} &= \sum_i = 1^n \Pr(T > i) \end{aligned} \tag{14}$$

where n is the number of days that we ski

$$\Pr(T \leq t) = \begin{cases} 1 \text{ , if } t \geq B \\ \frac{e^{\frac{t}{b}} - 1}{e} - 1 \text{ , for } t \in [0, B] \end{cases} \tag{15}$$

If $T = B \Rightarrow \Pr(T) = 1$, and $T = 0 \Rightarrow \Pr(T) = 0$
- Let

$$density = f_T = \frac{1}{B} \cdot \frac{e^{\frac{t}{b}}}{e - 1}, t \in [0, B] \tag{16}$$

Suppose that we ski for n days, there are two cases:
(1) Case 1 ($n \leq B$).

$$OPT = n \tag{17}$$

$$\mathbb{E}[ALG] = \mathbb{E}[\text{cost of renting}] + \mathbb{E}[\text{cost of buying}] \tag{18}$$

Let us derive $\mathbb{E}[\text{cost of buying}]$

$$\mathbb{E}[\text{cost of buying}] = \Pr(T \leq n) \cdot B \tag{19}$$

We know from cases that this equals

$$= \frac{e^{\frac{n}{B}} - 1}{e - 1} \cdot B \tag{20}$$

Now let us evaluate for the cost of renting

$$\mathbb{E}[\text{cost of renting}] \tag{21}$$

We utilize the following equality

$$\mathbb{E}[\text{cost of renting}] = \mathbb{E}[\min(n, T)] \tag{22}$$

We either rent for first n steps, or if T is sufficiently small, then buy.
- However, what is $\mathbb{E}[\min(n, T)]$

$$= \mathbb{E}_T\left[\int_0^n \mathbb{I}(t \leq T) dt\right] \tag{23}$$

Conceptually, we are integrating from $[0, n]$. It happens that T may be less tha nor equal to n. The indicator function states that the value of the function $\mathbb{I}(t \leq T) \in [0, 1]$. If $t \leq T$, then this integral is just equal to T.

- From here, what else can we do? We can utilize *linearity of expec. to swap the integral and the* $\mathbb{E}$.

$$\begin{aligned} &\Rightarrow \int_0^n \mathbb{E}_T[\mathbb{I}(t \leq T)]\ dt \\ &\Rightarrow \int_0^n \Pr(T \leq T)\ dt \quad = (*) \end{aligned} \tag{24}$$

This is just analogous utilizing a summation and finding the probability across multiple days

- Now, let us substitute the value of $\Pr(t \leq T)$ as being $(1 - \Pr(T \leq t))$.

$$\begin{aligned} (*) &= \int_0^n 1 - \frac{e^{\frac{t}{B}} - 1}{e - 1}\ dt \\ &= \int_0^n \frac{e - e^{\frac{t}{B}}}{e - 1}\ dt \\ &= \frac{e}{e-1} \cdot n - \frac{e^{\frac{n}{B}} - 1}{e - 1} \cdot B \end{aligned} \tag{25}$$

Lmao, study up how to integrate again idiot

Up until this point, we have found $\mathbb{E}[\text{renting}]$ and $\mathbb{E}[\text{buying}]$, thus, we assert that

$$ALG = \frac{e}{e-1} \cdot n \approx 1.58 \cdot n \tag{26}$$

where n is the value of OPT. This already performs a lot better than the deterministic solution of the problem.

Now let us consider the second case:

- **Case 2** $> B$. We observe that for a sufficiently large n, then we are guaranteed that both algs would have bought skis at or before time B, thus the cost won't change.
  - Thus, the costs of both algorithms, the offline and online algorithms, would have bought the skis by time B.

Thus, we surmise that the online algorithm performs better than or equal to an optimal algorithm.

But can we do better?

There are two strategies here:

- **Approach 1.** Assume there's a distribution f with which the alg picks a threshold that is better than ours. We'll examine n and thorugh computation, determine that f is the best possible function.

]

- **Approach 2.** Utilize the *minimax* principle (Yao's minimax principle for online algorithms)
  - Instead of one n, look at the distribution of n's, then do the analyss on the random distribution of n's
  - Just "do things at random for various n's"
    - If we assume that things are done at random, then it simplifies the computations

20 Caching

* Caching: All modern processors have caches L1, L2, etc.

Let us imagine that we have a cache of size k that can store k elements/pages. Whenever we store data, we store it witihn the cache. If the cache is full, however, then we just evict an element.

- What's the optimal strategy for choosing caches to read from and evict elements from?
- LRU Cache is the most famous one
  - Basic, but practical
  - Essentially, we just keep track of whenever a cache was last used, and when we have a collision, we just evict the oldest one
- **Motivation.** We want to compare LRU with the best offline algorithm

Theorem. The competitive ratio is k, thus

$$\sup_{I} \frac{ALG(I)}{OPT(I)} = k \tag{27}$$

In the context of caches, we measure perforamnce based solely on the number of cache misses.

Note. If we care about randomized algorithms, then the best competitive ratio is $O(\log k)$

Objective. We seek to prove that the number of cache misses that LRU has is bounded by $k \cdot OPT(I)$.

- Again, this could be improved utilizing ranodmization

Let us consider a sequence of instances (sequences of numbers)

$$I_1, I_2, ..., I_n \tag{28}$$

Our strat: Fix an instance I. Let us consider a block i that starts and ends with a cache miss. We find that in such a case, the optimal algorithm has 2 misses.

- How much does OPT page after the cache becomes completely full?

$$\text{numMisses}(OPT) = \text{number of blocks} \tag{29}$$

  Meanwhile

$$\text{numMisses}(LRU) = k \tag{30}$$

  - Analysis
    - How many ways does LRU page?
    - k, since if there's more zz
  - We claim that LRU has one cache miss
    - This is because LRU guarantees that no new element will be evicted
      - For any distinct number in an instance I, we iwll never evict it more than once per block
    - Thus, since OPT must evict at least one element by definition of blocks
    - and ALG will evict at least k elements, then we complete the proof ■
  - Let us reiterate the strats here:
    - LRU always misses after filling the cache
    - The optimal strategy? Always evict the element that is least likely to be evicted per block. This, of course, works only for offline algs

21 Next TIme (Beyond Worst Case Analysis)

- If we use resource segmentation, we find that the competitive ratio of this algorithm improves. We can demonstrate that the competitive ratio can be reduced to 2.

Chapter 16:
2-Competitive Algorithm for Online Paging
(05/14/24)

22 Reminders

- Homework 03 is due Friday (with extension to Monday)

23 Is 2-competitive real for LRU cache?

Remark ().
We proved that k-competitive is the best that we can do

24 Today

- Look at randomized online algorithms
  - Martingale Algorithm
  - Beyond worst-case analysis
    - Resource augmentation

25 Resource Augmentation

- Tactic for analyzing online algorithms
  - Compare this algorithm with the optimal offline algorithm *OPT*
  - In order to compensate for difference, allocate more resources for *ALG*

26 Scenario

- *OPT*
  - Has k pages
- *ALG*
  - Has $2k$ pages (an advantage!)
  - Could be $1.5k$ or $k + 1$, etc

Note.
Intuition here is that: Suppose that more resources $\Rightarrow$ comparable perforamnce with *OPT*. We can also imagine that for *OPT*, resource augmentation has similar perforamnce, thus the online must be comparable

27 Why Resource Augmentation?

- Allows us to bypass the worst-case scenario

Theorem ().
The number of LRU misses is $\leq 2 \times OPT(I)$ where LRU runs with $2k$ pages and *OPT* with k

> **Note.**
> Proof is similar to former proofs

Proof for LRU Augmentation.
Let us consider an arbitrary sequence of futures I. We partition this block.

$$\{1, 5, 11, 1, ...\} \tag{31}$$

Let us split these into $2k$ distinct pages. For any block, consider the number of misses from both *OPT* and *ALG*.

- ***OPT* analysis.** In the offline cache there are k pages. If its lucky, k pages are added to the cache. This leaves, by dfn of the block, k remaining pages that weren't added, resulting in a miss. Thus, there are $\geq k$ misses per block.
- ***ALG* analysis.** We argue that *ALG* makes $\leq 2k$ misses. By definition of LRU, and the old-new page system. We will never miss on the same page within a block, but since there are $2k$ pages, then we can potentially miss *every* time
  -

Thus, asymptotically, the ratio is 2. ■

> **Note.**
> LRU Cache is a reasonable alg because it will perform twice as worse as *OPT* in any time

Unit 5:
FPT Algorithms

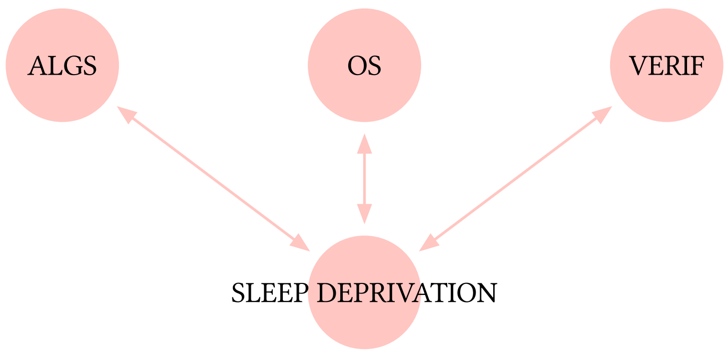


Figure 4: week 9

Chapter 17:
Preview of Second Half for Class

28 Motivation: Solving Computationally Hard Problems

- Imagine we had a very hard problem to solve, perhaps one that is NP-complete.
- ...What compromises can we make in order to "solve" the problem?
  - We can find the *exact* answer $\leftrightarrow$ compromise on time
    - Exact algorithms
  - We can *approximate* the answer $\leftrightarrow$ save on time
    - Approximation algorithms
  - We can *reduce* the problem as to focus on a particular parameter, rather than the entire input size $\leftrightarrow$ save on time
    - Fixed Parameter Tractable Algorithms

Here's kind of the itinerary of algorithm analysis for reference lol

29 Fields of Algorithm Design and Analysis

29.1 Undergraduate Algorithms

- Basically just looking at general solutions to non-computationally difficult problems

(1) **Greedy Algorithms**
(2) **Divide and Conquer**
(3) **Dynamic Programming**

29.2 First Half of 496:

- Basically just looking at new approaches to solving these problems in order to save space and time, as well as how to optimize the solution in uncertain situations

(1) **Randomized Algorithms**
(2) **Streaming Algorithms**
(3) **Online Algorithms**

29.3 Second Half of 496

- **How can we solve computationally-hard problems?**

(1) **Fixed Parameter Tractable (FPT Algorithms).**
(2) **Approximation Algorithms.**
   - **Intuition.** We can't *exactly* solve this problem in polynomial time. So let's just find an answer that is slightly worse than the optimal.
     - Similar to online algorithms, since we just care about cost
     - Emphasis on speed over correctness
(3) **Exact Algorithms.**
(4) **Beyond worst-case Analysis of Algorithms.**

Definitions will be revealed...

Chapter 18:
FPT Algorithms

30 Motivation

Remark (Techniques for Solving Computationally-Hard Problems).
One of the ways that we can solve computationally-hard problems is to utilize FPT (Fixed Parameter Tractable) algorithms. These algorithms are an alternative to **approximation algorithms**, which compromise correctness for speed. In the case of FPT, we will see that they make no compromises on correctness for speed.

- If computing the problem over an *entire* input space is too costly… what if we compute the answer utilizing smaller subproblems?

31 Motivation (cont'd): DNA Strand Length Problem

Note.
This problem isn't the *perfect* situation for utilizing FPT, since this is a polynomial time problem that we're just reducing to constant time, whereas the typical FPT problem is reducing an exponential time of n problem into an exponential time of k problem.

Let us consider the following problem:

Consider two strands of DNA. Align the two strands of DNA with the minimum possible cost (ie, with the minimum number of gaps δ).

- **Typical Solution.** We would just apply a two-dimensional dynamic programming algorithm.
  - But what if a n^2 complexity was too slow?
  - What if we knew that we just wanted to see if an alignment existed for k-cost?
- **Optimization.** Instead of calculating all alignments of all costs for DNA strands, attempt to generate the alignments to some threshold k, thus we bound the number of computations by k.

The focus of FPT algorithms is to fix some parameter k, which we can derive based on the problem's input, or as a desired result of the problem. By fixing a parameter k, we can then design an algorithm that calculates over k, thereby reducing the overall complexity, while still preserving correctness.

32 Defining FPT Algorithms

Definition 3 (Fixed Parameter Tractable (FPT) Algorithms).
FPT Algorithms are algorithms whose asymptotic running time is upper-bounded by the exponential function of a *specific parameter* k of the problem. In practice, running time is represented as this exponential function $f(k)$ multiplied by a polynomial of the input size n.

$$f(k) \times poly(n) \tag{32}$$

32.1 The Inputs in More Detail

- What is k?
  - k is some parameter of the input or the solution. k is unique to the problem
- What is n?
  - The size of the input to the problem.

33 Examples of Problems that Can Be Optimized Using FPT

(1) Hamiltonian Paths
(2) k-paths (parameterized by k)
(3) Feedback Vertex Set (parameterized by k)

34 Motivating Problem 1: Hamiltonian Paths

Problem Statement. *Suppose we have a graph G that is unweighted. Find a path that visits every vertex exactly once.*

> **Note.**
> We will get back to this one, since this one is a little difficult– first discuss the k-paths problem

35 Motivating Problem 2: k-Paths (AKA Longest Simple Path)

Problem Statement. *Given a graph G, determine if there exists a simple path of k vertices in G.*

- Evaluating the k-paths problem $\Rightarrow$ We can solve the hamiltonian path problem

35.1 Naive Solution

- **Solution 1**. Calculate all subsets of k vertices and just determine if there exists a path P between them.
- **Solution 2**. Starting from any vertex $v \in G$, just perform a breadth-first search to see if there exists a valid k-path.
- **Analysis.** Let us consider Solution (1).
  - Given a graph G such that G is Δ-regular:
    - (suppose that $\Delta \approx n$ or $\Delta \approx \sqrt{n}$)
  - **Runtime.** $O(\Delta^k)$, since we exhaustively search all paths of k vertices for each vertex
  - Clearly not FPT, since $\Delta^k \approx n^k$

Given our naive solution, can we do better?

Yes! We can improve our exponential, naive algorithm by utilizing the FPT framework. We can do this by utilizing the **color coding** technique.

35.2 FPT Solution

> **Remark (k-paths Problem).**
> The graph G in the problem is *unweighted* and *undirected.*

Intuition. If we can somehow transform G into a directed graph, then we can evaulate the problem utilizing **dynamic programming**.

- Without DP, we were essentially just enumerating all possible combinations of the vertices and verifying them, which is *slow*
- By color coding the graph, we are able to filter out some combinations and make the verification process faster

> **Definition 4 (Color Coding).**
> Technique in which, given G, pick some k colors s.t., and then color all vertices $1, \ldots, k$ at random (independently + uniformly)

Let us now consider a basic usage of the color-coding technique, and then a more difficult one that optimizes it.

36 Evaluating k-Paths via Basic Color Coding

- This algorithm is going to inherent the "randomization" aspect of our naive algorithm by randomly selecting k-element combinations, but will now utilize **dynamic programming**, due to the added coloring

- **Algorithm.** Given a graph G:
  (1) Color G using k colors (enumerated as colors $\{1, \ldots, k\}$)
  (2) Based on this coloring, determine if a good path exists by randomly selecting k-element subsets of vertices
    - ie, determine if there exists a path of colors $1 \to 2 \to \ldots \to (k-1) \to k$
    - we can do this part using dp

> **Note.**
> This algorithm only works if the ith vertex receives the ith color.

36.1 Algorithm In Detail

- **Intuition.** Split the graph into layers based on colors $1, \ldots, k$
  - 1-dimensional DP
  - Utilize dynamic programming in order to demonstrate if there exists a path from 1 to k

- For each layer:
  (1) Mark the ith layer (or dp_i) as reachable if and only if the vertex that precedes it is reachable

1							…						k

Figure 5: DP Table for the Longest Simple Path Problem

36.2 Algorithm Analysis

We derived, by definition of the algorithm, that for any combination of k-vertices, it will take k iterations to determine if its good.

- That being said, however, how many graph colorings/repetitions do we have to try in order to get the answer?

In order to evaluate this, we consider the probability that we're going find a good path.

Theorem (Probability of Good Path).

The probability that, for a path of length k, the ith vertex has the correct corresponding color i is given by

$$\Pr[i\text{-th vertex gets color } i \text{ for } \{1, ..., k\}] = \left(\frac{1}{k}\right)^k = \frac{1}{k^k} \tag{33}$$

at the lowest bound.

Note.

This probability is the **worst-case bound**, which occurs assuming that there is only one path. If there are multiple paths, then it's better for us.

We are able to use Equation 33 in order to determine approximately how many times we will have to run the algorithm (coloring and all) in order to obtain the answer.

Claim.

The maximum number of repetitions (of the algorithm) required to find the solution is $\approx k^k$ (ofc there's k^k different combinations).

Even though we have a bound on how many combinations we need to reveal the answer, we can concisely narrow down how many repetitions we need to guarantee that we find a path of k-colors

36.3 Making the Bound of Repetitions More Accurate

Let us now consider the failure bound of this algorithm (ie, the probability that we aren't able to find a good-colored path). In this case, we let δ be this failure bound.

Claim.

The total number of repetitions t of the Basic Graph Coloring Algorithm in order to evaluate the k-paths problem is $t \approx k^k \times \ln\left(\frac{1}{\delta}\right)$.

Proof to Claim.

- Let δ be the probability that we aren't able to find a path P of length k
- Let p be the probability that a path P of length k is good
- Let t be the number of repetitions

$$
\begin{aligned}
\Pr[\text{after } t \text{ attempts } \nexists \text{ path P of length } k] &= (1-p)^t \\
\Pr[\text{after } t \text{ attempts } \nexists \text{ path P of length } k] &\leq \delta \\
(1-p)^t &\leq \delta \\
\ln\left((1-p)^t\right) &\leq \ln(\delta) \\
t \times \ln(1-p) &\leq \ln(\delta) \\
\approx t \times (-p) &\leq \ln(\delta) \\
\approx t &\geq \frac{\ln\left(\frac{1}{\delta}\right)}{p} \\
\approx t &\geq \frac{\ln\left(\frac{1}{\delta}\right)}{\left(\frac{1}{k}\right)^k} \\
\approx t &\geq \frac{\ln\left(\frac{1}{\delta}\right)}{\frac{1}{k^k}} \\
\approx t &\geq k^k \times \ln\left(\frac{1}{\delta}\right)
\end{aligned}
\tag{34}
$$

Given Equation 34, we observe that the number of repetitions needed to guarantee that we find a path-P of length k, as required. ■ Now, we can make some claims about the true runtime of the FPT algorithm, based on the definition of FPT.

Claim.
The total runtime of the basic color coding FPT algorithm is

$$
\begin{aligned}
k^k \times \ln\left(\frac{1}{\delta}\right) \times O(m) = \\
= O\left(k^k\right).
\end{aligned}
\tag{35}
$$

Proof.
We already know that the number of repetitions of the algorithm required in order to find a good path P of length k is given by $k^k \times \ln\left(\frac{1}{\delta}\right)$. We simply add on the new fact:

during each repetition though, the amount of time that it takes in order to verify that the path is a good path is $poly(n)$, which we will denote as $O(m)$.

Thus, since we have to verify some $poly(n)$ paths, as well as k nodes for each path with probability $\left(\frac{1}{k}\right)$, then we know that the totla runtime must be

$$
\begin{aligned}
&= f(k) \times poly(n) \\
&= k^k \times \ln\left(\frac{1}{\delta}\right) \times O(m)
\end{aligned}
\tag{36}
$$

as required. ■

Although it may seem like we're done with this problem... as with all CS problems...

...how can we improve this algorithm?...

37 Evaluating k-Paths via Advanced Color Coding

Remark (Complexity of Naive Color Coding).

The asymptotic time complexity of the FPT k-Paths solution using basic color coding was $k^k \times \ln\left(\frac{1}{\delta}\right) \times O(m) = O\left(k^k\right)$.

...how can we do better...?

38 Algorithm Intuition (Advanced Color Coding)

Intuition. In our new solution, we hope to improve the runtime of the original algorithm by increasing the probability of success by *loosening* the constraint of a good path.

- In this case, we will be loosening the constraint of a good path by simply making a good path equivalent to a path in which all k vertices have distinct colors

39 Algorithm Design (Advanced Color Coding)

- Given an unweighted, undirected graph G:
  - Color each vertex $v \in V$ uniquely
  - Determine if $\exists$ a good path P in the resulting k-colored graph using dynamic programming
    - A k-length path P is good if and only if all $v \in P$ have distinct colors

40 Algorithm in Detail (Advanced Color Coding)

- Given a fixed color u, find $path(u, \mathcal{S})$ where $\mathcal{S} \subseteq \{1, ..., k\}$
  - We want each path to have only one vertex of each color in $\mathcal{S}$
  - We can use S in the original problem
- The space of the DP will be $2^k \times n$ (FPT) (still way better than $n^{\log(n)}$)
  - Visit all neighbors v
  - We know that $path(u, \mathcal{S})$ exists if
    - $\exists v \in nei(u$ where $path(v, \frac{\mathcal{S}}{color}(u)$ exists
    - where, of course, $color(u) \in \mathcal{S}$
- We would need to repeat this algorithm $e^k \times \log\left(\frac{1}{\delta}\right)$
- Runtime is $e^k \times \log\left(\frac{1}{\delta}\right)$ multiplid by runtime of dp

41 Algorithm Analysis (Advanced Color Coding)

Claim.

The probability that all colors in a k-length path P are distinct is $\left(\frac{1}{e}\right)^k$.

Proof.

Let us first consider the probability of the event in which all colors in P are distinct, then we bound it more precisely.

We know that for any path, there are k^k possible ways to choose each vertex distinctly for a path that is k elements long. We also know that there are $k!$ permutations of k distinct elements. Thus,

$$
\begin{aligned}
&\Pr[\text{all colors in } P \text{ are distinct}] = \\
&= (\text{num possible } k \text{ color perms}) \times \Pr[\text{all elements in } P \text{ are distinct}] \\
&= k! \times \left(\frac{1}{k}\right)^k = \frac{k!}{k^k}
\end{aligned} \tag{37}
$$

Based on Equation 37, we demonstrate that $\Pr[\text{all colors in } P \text{ are distinct}] = \frac{k!}{k^k}$. We can better bound this probability by utilizing Stirling's Inequality.

Remark (Stirling's Approximation).

$$
n! \sim \sqrt{2\pi n} \times \left(\frac{n}{e}\right)^n \tag{38}
$$

which, we can simply just regard as

$$
n! \sim \left(\frac{n}{e}\right)^n \tag{39}
$$

Using Stirling's Approximation,

$$
\begin{aligned}
\Pr[\text{all colors in } P \text{ are distinct}] &= \frac{k!}{k^k} \\
&\approxeq \frac{\left(\frac{k}{e}\right)^k}{k^k} \\
&\approxeq \left(\frac{1}{e}\right)^k
\end{aligned} \tag{40}
$$

Based on our calculations in Equation 40, we determine that the probability that all colors in P are distinct is $\left(\frac{1}{e}\right)^k$, as required. ■

From here, we are able to utilize the same calculations as in Equation 36 in order to demonstrate that the runtime of the Optimized Graph Color-Coding FPT solution is $\left(\frac{1}{e}\right)^k$.

Claim.

The runtime of the Optimized Graph Color Coding FPT solution is

$$
e^k \times \log\left(\frac{1}{\delta}\right) \times O(m) = O\left(e^k\right). \tag{41}
$$

Proof.

Utilize the same calculations as in the basic graph color coding FPT solution.■

Chapter 19: FPT Algorithms (Part 2) (05/16/24)

42 Reminders

- For HW3
  - Questions about Q1
    - Aim is to show that if you don't use a randomized routing approach, then delay suffers and will be $O(d)$ with high probaiblity and worsens exponentially
    - Demonstrate that $O(k)$ for randomized approach and $\Omega(2^k)$ for the naive/greedy approach
    - We want to find the **optimal** algorithm for Q2
    - ie, the most optimal *deterministic* and *randomized* strategies
  - For Q3
    - We want a 2-competitive algorithm (ie, the competitive ratio is $\frac{1}{2}$)
- Homework 03 is due Friday (with extension to Monday)

43 Remark

- We previewed the color-coding technique in the last lecture:

> **Remark (Color-Coding FPT Algorithm).**
> We can utilize two variants of color coding:
>
> - **Basic.** We color the graph randomly, determining that a good path was one in which all vertices $v_1, ..., v_k \in P$ were colored $1, ..., k$.
>   - Time complexity of $\left(k^k \times \ln\left(\frac{1}{\delta}\right) \times O(m)\right)$
> - **Optimized.** We color the graph randomly, determining that a good path was one in which all vertices $v_1, ..., v_k \in P$ were colored *distinctly*.
>   - Time complexity of $\left(e^k \times \ln\left(\frac{1}{\delta}\right) \times O(m)\right)$

44 Outline of Lecture

Today's lecture, we basically just finished up the discussion on FPT algorithms. We first revised the FPT algorithm for the k-paths problem as well as how to apply it to the Hamiltonian-path problem. Then we discussed a new problem that can be optimized utilizing FPT algorithms:

- Vertex Cover
  - Utilizes a technique called *kernelization* in order to solve

which we then just used to motivate approximation algorithms, which offer us a different insight on solving the problem. Finally, we concluded this lecture by discussing the **set cover problem** which we also evaluate using approximation algorithms.

45 Vertex Cover Problem

45.1 Introduction to Kernelization

Kernelization is a technique that is employed with FPT (or generally parameterized) algorithms.

The primary intuition about kernelization is that we want to be able to take a problem, which we will denote as Q, and make an argument about an equivalent problem, of which we can make the same argument about the original problem.

Note.

Honestly, sounds a little fake, but as with most combinatorics/ discrete math-related arguments, it works.

...(the "probabilistic method" is another fake argument btw)...

In kernelization, we call equivalent problems *instances.*

45.2 Defining Kernels

Not to be confused with *literally every other definition of a kernel in computer science*, kernelization is a *paradigm for solving parameterized problems.*

From *Parameterized Algorithms* by Fedor V. Fomin and Lokshtanov:

Definition 5 (Kernelization).

A *kernelization algorithm*, or simply a kernel, for a parameterized problem Q is an algorithm $\mathcal{A}$ that, given an instance (I, k) of Q, works in polynomial time and returns an equivalent instance (I', k') of Q (Fomin et. Lokshtanov).

Note.

It is required that size $(k)_{\mathcal{A}} \leq g(k)$ for some computable function $g : \mathbb{N} \to \mathbb{N}$. We can essentially think of g as some computable function on k.

Definition 6 (Reduction Steps).

A reduction rule for a parameterized problem $\mathcal{Q}$ is a function ϕ such that

$$\phi : \Sigma^* \times \mathbb{N} \Rightarrow \Sigma^* \times \mathbb{N} \tag{42}$$

maps an instance $(I, k) \in Q$ to an equivalent instance (I, k') of Q such that ϕ can be computed in $|I|$ and k time.

Theorem (Safe and Soundness Rule).

We state that two instances of a problem Q are equivalent if $(I, k) \in Q$ if and only if $(I, k') \in Q$

Essentially, we can think of kernelization algorithms as algorithms that utilize *reduction rules* in order to reduce the a problem instance it into its "computationally difficult 'core' structure".

- In other words, kernelization just algorithmically reduces a problem down into simpler instances until we reach the crux of the problem. We do this to make finding the true answer easier and more systematic.

45.3 Introduction to Vertex Covers

45.3.1 Definitions

In order to evaluate this problem, we first need a notion of what exactly a vertex cover is.

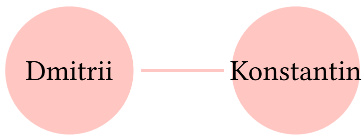
Definition 7 (Vertex Cover).
Given a graph $G = \langle V, E \rangle$, a **vertex cover** of G is a subset of vertices $\mathcal{S} \subseteq V_G$ that includes all unique endpoints of every edge in G at least once.
Mathematically, this can be denoted as

$$\operatorname{cover}(G) = [S \subseteq V_G \mid \forall (u, v) \in E_G (u \in \mathcal{S}) \vee (v \in \mathcal{S}) \vee (u, v \in \mathcal{S})] \tag{43}$$

For the purposes of the problem and its corresponding FPT algorithm, we will let $\left(G^{(i)}, k^{(i)}\right)$ represent the ith vertex cover (in the sequence of generated vertex covers) of size k.

45.3.2 Example

Example 1. Here is a 2-graph:



G's vertex cover could be

$$\begin{aligned} \mathit{cover}(G) &= \{\text{Dmitrii}\} \\ \vee\, \mathit{cover}(G) &= \{\text{Konstantin}\} \end{aligned} \tag{44}$$

since the edge that "Dmitrii" and "Konstantin" are incident to also connects their neighbor "Dmitrii" and "Konstantin," respectively.

Note.
If it wasn't clear, we just want the vertices such that the connected edges to these vertices connects all of the other edges in the graph.

46 Minimum Vertex Cover Problem Statement

Problem Statement 1 (Miminum Vertex Cover Problem).
Given a graph $G = \langle V, E \rangle$, find the minimum size vertex cover.

47 FPT Algorithm Approach

Remark (FPT Algorithms).
FPT algorithms are algorithms that are able to minimize the runtime of an NP-hard problem. They do this by reducing the exponential time $O(c^n)$ to one that is dependent on some parameter k. The runtime of such algorithms is defined as:

$$f(k) \times \mathit{poly}(n) \tag{45}$$

where k is some *fixed parameter* of the problem (which is problem/instance- dependent) and n is the size of the input/universe.

48 FPT Algorithm Intuition (Vertex Cover Problem)

In our FPT solution for the Vertex Cover problem, we employ *two* reduction steps that, informally:

(1) Eliminate all *unnecessary vertices* from the problem space and the solution
(2) Add all vertices that are *guaranteed* to be in the solution.

> **Note.**
> Given that we are restricting the input space to a constant k which is fixed, we are able to freely perform a *brute force* algorithm on the input space, since it'll always be upper bounded by an algorithm on n.

49 Algorithm Design (Vertex Cover FPT Solution)

Given the Vertex Cover problem, we define the two following reduction rules for an instance $\left(G^{(i)}, k^{(i)}\right)$, where $\left(G^{(i)}, k^{(i)}\right)$ is defined as the ith vertex cover of G that is of size k.

- **Reduction Rule 1.** Given an instance $\left(G^{(i)}, k^{(i)}\right)$, if there exists an isolated vertex v, then remove v from the graph. The resulting instance is represented as $\left(G^{(i)} - v, k^{(i)}\right)$.
- **Reduction Rule 2.** Given an instance $\left(G^{(i)}, k^{(i)}\right)$, if there exists a vertex $v \in G^{(i)}$ such that $deg(v) > k^{(i)} + 1$, then remove v and its incident edges from G.

50 Algorithm (Vertex Cover FPT Solution)

Utilizing the two FPT reduction rules, we derive the following algorithm for the Vertex Cover problem. algorithm:

```
while kernel K is a valid cover
  randomly choose k or k' vertices
  if k or k' vertices are a valid vertex color do
    continue
  end

  return
end
```

> **Remark (Kernelization).**
> *Kernelization* is a technique that is utilized in parameterized algorithms in which, given an instance (I, k) of a parameterized problem Q, returns an (easier) equivalent instance (I', k') of Q.

51 Algorithm Analysis (Vertex Cover FPT Solution)

Utilizing the two reduction rules defined in the algorithm, let us make the following

Theorem (Vertex Cover FPT Solution Feasibility).
If $\left(G^{(i)}, k^{(i)}\right)$ is a yes-instance of the Vertex Cover problem and no further reductions can be performed on it, then
(1) The number of vertices in $G^{(i)}$ must be less than or equal to $k^2 + k$. Formally,

$$|V(G^{(i)})| \leq k^2 + k \tag{46}$$

(2) The number of edges in $G^{(i)}$ must be less than or equal to k^2. Formally,

$$|E(G^{(i)})| \leq k^2 \tag{47}$$

Proof to Vertex Cover FPT Solution Feasibility.
We can prove Vertex Cover FPT Solution Feasibility by way of direct proof.

(1) If an instance $\left(G^{(i)}, k^i\right)$ is a yes-instance and is irreducible, then it must follow that for all vertices that exist outside of the cover, then there must exist an edge inside of the cover that is incident to it.
Formally, we notate this as

$$\forall v \in \left(G^{(i)} \setminus S\right), \exists e \in E\left(G^{(i)}\right) \text{ where } e = (u, v) \vee e = (v, u) \tag{48}$$

(a) This is true by definition of Reduction Rule 1.
- The instance of the graph $G^{(i)}$ must be connected.

(2) If an instance $\left(G^{(i)}, k^{(i)}\right)$ is a yes-instance and is irreducible, then the number of vertices not included in the vertex cover must be less than $k \cdot |S|$. This, in turn, implies that the number of vertices in the graph must be upper-bounded by $(k+1) \cdot k$.
Formally,

$$|V\left(G^{(i)} \setminus S\right)| \leq k \cdot |S| \Rightarrow |V\left(G^{(i)}\right)| \leq (k+1) \cdot k \tag{49}$$

(a) $\forall v \in V\left(G^{(i)}\right),\ deg(v) \leq k$ is true by definition of Reduction Rule 2.
- This is because we removed all vertices with higher degrees.

(b) $|V\left(G^{(i)} \setminus S\right)| \leq k \cdot |S|$ is true by 2(a).
- All vertices in the graph, and thus the cover can have at most k edges, so the number of vertices not in the cover is at most the number of vertices in the cover multiplied by the upper bound of edges coming out of each vertex.

(c) $|V\left(G^{(i)}\right)| \leq (k+1) \cdot |S|$ is true by Reduction Rule 2 and 2(b).
- By Reduction Rule 2, we know that the degree of any vertex is strictly less than $k + 1$. Thus, it follows that the number of vertices in the graph must be striclty less than the number of vertices in S multiplied by the upper bound $(k+1)$.

(3) If an instance $\left(G^{(i)}, k^{(i)}\right)$ is a yes-instance and is irreducible, then the number of edges in $G^{(i)}$ is bounded by $\leq k^2$.
(a) This is true by definition of Reduction Rule 2
- Any vertex v can have at most k outgoing edges, to which those vertices can have at most k outgoing edges.

(4) QED
(a) By (1) and (2) and (3)

■

Utilizing our Vertex Cover Feasibility Theorem, we can now derive Reduction Rule 3, which will prove that the FPT algorithm will always create a solution to the Vertex Cover Problem.

- **Reduction Rule 3.** Let $\left(G^{(i)}, k^{i}\right)$ be an input instance of the Vertex Cover problem in which Rule 1 and Rule 2 are not applicable. If $k < 0$ or $G^{(i)}$ has more than $k^2 + k$ vertices or $G^{(i)}$ has more than k^2 edges, then $\left(G^{(i)}, k^{(i)}\right)$ is a no-instance.
  - We know that if $k < 0$ or $G^{(i)}$ has more than $k^2 + k$ vertices or $G^{(i)}$ has more than k^2 edges, then this must be invalid by the Vertex Cover Feasibility Theorem.

Using the theorem, we finally arrive at the following lemma utilizing our former calculations:

Lemma ().

If G has $2^k k^2$, then $|E(G)| > k^2$. For any graph G that has a vertex cover of size $\leq k$, then $|E(G)| \leq k^2$ edges.

Unit 6:
Approximation Algorithms

Figure 7: $FFIE TO THE MOON

Chapter 20:
Approximation Algorithms

52 Citations

(1) *Approximation Algorithms* (Vijay V. Vazirani)
- Chapters Referenced
  (a) Chapter 1 (Introduction)
    - An approximation algorithm for cardinality vertex cover (page 3)
  (b) Chapter 2 (Set Cover) (pgs. 15-26)
  (c) Chapter 14 (Rounding Applied to Set Cover) (pgs. 119-124)

(2) UW CSE 421: Introduction to Algorithms
  (a) Lecture 23: Approximation Algorithms (Set Cover)

(3) CMU 15-451/651: Design and Analysis of Algorithms
  (a) Lecture #17: Approximation Algorithms

53 Introduction

In this new section, we consider the design and analysis of **Approximation Algorithms**, which is another field of algorithms that tackles the issue of evaluating NP-hard and NP-complete problems with "good-enough" accuracy in polynomial time.

Remark (Motivation for Approximation Algorithms).

We want to study approximation algorithms because they provide a polynomial time way of computing NP-hard problems.

Definition 8 (Approximation Algorithms).

Approximation algorithms are algorithms that both maximize a **cost ratio** as well as a **value ratio** such that

$$\begin{aligned} \text{Cost Ratio} &= ALG(I) \leq \alpha \times OPT(I) \\ \text{Value Ratio} &= ALG(I) \geq \alpha \times OPT(I) \end{aligned} \tag{50}$$

where α is the *approximation factor/ratio* such that $\alpha \leq 1$, and we strive for α to be as close to 1 as possible.

Note.

Whenever we discuss Approximation Algorithms, we will consider each algorithm based on their α factor. For an approximation algorithm ALG that has an approximation factor of α, then we consider that algorithm to be α-approximate.

Additionally, we will also consider algorithms of paritcular α-approximate ratios. For example, we might discuss 2-approximation algorithms.

(1) 2-approximation algorithms

54 (Re-)Introduction to the Minimum Vertex Cover problem

Remark (FPT Approximation Solution for Minimum Vertex Cover).

TODO

Now that we have some exposure on how we can *FPT reduce* the Minimum Vertex Cover problem in order to get an exact solution in $f(k) + poly(n)$ time, let us now analyze various *approximation algorithms* techniques for evlauating the Minimum Vertex Cover problem.

54.1 Approximation Strategies for Minimum Vertex Cover

(1) **Strategy 1.** Greedy Approximation
(2) **Strategy 2.** LP-Relaxation Approximation

55 Strategy 1: Greedy Approximation Algorithm (Minimum Vertex Cover)

This section discusses the Greedy Approximation Algorithm strategy for evaluating the minimum vertex cover problem.

As with most greedy algorithms, we can sum up the behavior of this algorithm as simply just adding vertices $v \in V(G)$ as long as there exists an uncovered edge that is incident to it.

55.1 Algorithm Intuition (minimum vertex cover)

Intuition.

If we pick an arbitrary edge $e \in E(G)$, then we know that any vertex cover $cover(G)$ must contain at least one of the endpoints of such an edge. Thus, add both endpoints to the solution, remove all edges covered by these endpoints, and repeat.

Here is the pseudocode for the Greedy Approximation Solution:

```
while exists (u,v) in E(G) such that (u,v) not covered by S do
    S.add(u)
    S.add(v)
    remove all (m, n) in E(G) covered by S
end
```

Again, it is important to note that we are essentially doing the following

(1) We iterate through all remaining edges in G
    (a) If there is an edge in which at least one of the endpoints does not exist in S, then we just add the endpoints
    (b) Then, based on the new vertex cover S, remove all edges from the graph G that are now covered by S

55.2 Analysis (Minimum Vertex Cover)

We find that based on the design of the algorithm, we arrive at the following performance claim:
Claim

Claim.

Using the greedy approximation algorithm for the MINIMUM VERTEX COVER problem, we find that the algorithm is 2-approximation. Mathematically:

$$ALG(I) \leq 2 \times OPT(I) \tag{51}$$

TODO

Proof.

ALG considers edges $(u, v), ..., (u_k, v_k)$

$$\begin{aligned} u_i &= u_j \\ OPT(S) &\geq k \\ ALG(S) &= 2k \end{aligned} \tag{52}$$

■

56 Problem 2: SET COVER Problem

In this section, we pivot to a new problem, the SET COVER PROBLEM, which is a fundamental problem that is one of the fundamental problems of approximation algorithms.

Here is the problem statement:

Problem Statement 2 (SET COVER).

Given a universe $\Omega = \{u_1, ..., u_n\}$ and a collection of subsets $\mathcal{S} = \{S_1, ..., S_2\}$ where $S_i \subseteq \Omega$ for $1 \leq i \leq n$, find a minimum-size subcollection of $\mathcal{C} \subseteq \mathcal{S}$ such that

$$\bigcup_{S_i \in \mathcal{C}} S_i = \Omega \tag{53}$$

Theorem ().

The $\ln(n)$-approximation algorithm for $n = |\Omega|$.

57 Applications (SET COVER Problem)

(1) **Application 1.** Imagine that a company wants to hire candidates such that all required skills are covered
(2) **Application 2.** “Fuzz” testing in software
(3) **Application 3.** A manufacturer wants to get all items from different suppliers at minimum cost

58 Algorithm Strategies (SET COVER PROBLEM)

(1) **Strategy 1.** Greedy Strategy
(2) **Strategy 2.** Linear/Integer Programming Strategy

TODO

59 Greedy Approach Intuition (Set Cover Problem)

Intuition.
Pick the set that maximizes the number of **new** elements covered by the set cover of G, denoted as $cover(G)$

Here is the algorithm's pseudocode:

```
while (v_t != 0)
  find S_i that covers most elements in U_t
  add it to the solution
  v_t+1 = v_t \ S_i
  t = t + 1
```

60 Greedy Approach Analysis (Set Cover Problem)

Given the greedy approach, let us make the following claim:

Claim.
The greedy approximation algorithm for the Set Cover problem gives an $O(\ln(n))$ approximation of the optimum.

Claim.
If $OPT = k$, then the greedy approximation algorithm ALG will find at most $k \times \ln(n)$ sets.

Proof for Greedy Approach Approximation.
Let $OPT = k$, where k is a number of sets (ie, the minimum number of sets in order to construct a full set cover of G). Additionally, let n represent the number of elements in the universe Ω.

(1) Since we know that $OPT = k$, then there must exist a set of vertices that covers at least $\frac{n}{k}$ elements
   (a) We know that in order for k sets to cover all n elements in the universe Ω that there must be on average $\geq \frac{n}{k}$ elements per set.
   (b) If each set did not have at least $\frac{n}{k}$ elements in Ω, then $OPT > k$, which is false by definition of OPT.
   (c) In terms of n, this would mean that each set, on average, contains $\frac{\frac{n}{k}}{n} = \frac{1}{k}$th of the elements in Ω.
   (d) If each set did not have at least $\frac{1}{k}$th of the elements in Ω, then $OPT > k$, which is false by definition of OPT.

(2) The first set that OPT chooses, denoted as S_1 will have a size of $\geq \frac{n}{k}$.
   (a) By (1)(a), (1)(b), (1)(c)

(3) When OPT picks S_1 such that $|S_1| \geq \frac{n}{k}$, then the number of remaining elements in the set must be $n_1 \leq n\left(1 - \left(\frac{1}{k}\right)\right)$
   - ...where n_1 represents the number of elements in the first state, after OPT has removed elements from Ω in order to add them to first set cover S_1.

   (a) We can prove (3) using the following computation:

$$\begin{aligned}
\text{(num. elements remaining)} &\leq [\text{(original num. elements)} \\
&\quad - \text{(size of set cover after 1 iteration)}] \\
n_1 &\leq [n - S_1] \\
n_1 &\leq \left[n - \frac{n}{k}\right] \\
n_1 &\leq n\left(1 - \frac{1}{k}\right)
\end{aligned} \tag{54}$$

(4) After *OPT* chooses its first set cover S_1 after a single iteration of the algorithm, then there must exist a set S_2 with at least $\frac{n_1}{k-1}$ elements.
   (a) We can utilize similar logic to that in (1)(a-c)
   (b) Given that $OPT = k$, then the average number of elements per set k_i is $\frac{n_1}{k-1}$.

(5) After the second iteration S_2, then the number of remaining elements that are left uncovered, denoted as n_2 must be upper-bounded by $n_1(1 - \frac{1}{k-1})$
   (a) We simply just need to perform similar computations to Equation 54

$$\begin{aligned}
n_2 &\leq \left[\text{(num. elements in 1st iter)} - \left(\begin{matrix}\text{size of set} \\ \text{cover after 2 iters.}\end{matrix}\right)\right] \\
&\leq \left[n_1 - \frac{n_1}{k-1}\right] \\
&\leq n_1\left(1 - \frac{1}{k-1}\right) \\
&\leq n\left(1 - \frac{1}{k}\right)\left(1 - \frac{1}{k-1}\right) && \triangledown \text{ by (51)} \\
&\leq n\left(1 - \frac{1}{k}\right)\left(1 - \frac{1}{k-1}\right) \leq n\left(1 - \frac{1}{k}\right)\left(1 - \frac{1}{k}\right) \\
&\leq n \times \left(1 - \frac{1}{k}\right)^2
\end{aligned} \tag{55}$$

(6) The number of elements left after the ith selection from *OPT* is given by

$$n_i \leq n \times \left(1 - \frac{1}{k}\right)^i \tag{56}$$

   (a) By (1), (2), (3), (4), (5)

(7) The number of elements left behind by *OPT* approaches $< \frac{1}{n}$
   (a) Utilize upper bounds $1 + x \leq e^x$

$$n_i \leq n\left(1 - \frac{1}{k}\right)^i \leq n \cdot e^{-\frac{i}{k}} \tag{57}$$

   (b) After $k \times \ln(n)$ iterations, the number of uncovered elements must be less than 1 (just multiply $k \cdot \ln(n)$ with $n \cdot e^{-\frac{i}{k}}$).

$$|V_i| \leq \left(1 - \frac{1}{k}\right)^i \times nr\left(1 - \frac{1}{k}\right)^{k\ln(n)} < \frac{1}{n} \qquad \triangledown \text{ what is this?} \tag{58}$$

Thus, we demonstrate that in order for *ALG* to construct a Minimum Set Cover of G, it must utilize $k \times \ln(n)$ iterations, which is $O(\log(n))$, which implies that it is a $O(\log(n))$ approximation of *OPT*, as required. ■

Chapter 21: Approximation Algorithms Pt. 2 (05/17/24 Lecture)

61 Citations

(1) *Approximation Algorithms* (Vijay V. Vazirani)

- Chapters Referenced
  - (a) Chapter 1 (Introduction)
    - An approximation algorithm for cardinality vertex cover (page 3)
  - (b) Chapter 2 (Set Cover) (pgs. 15-26)
  - (c) Chapter 14 (Rounding Applied to Set Cover) (pgs. 119-124)

62 Remarks

- Homework 4 assigned (due 05/31/24)
  - Problem 1 is about LRU Cache
  - Problem 2 is similar to the FPT problems discussed in class
    - Discusses a problem about **Contracting Edges**

> **Definition 9 (Contracting Edges).**
> Given two connected vertices, we contract the edge between them by removing that edge between them and then treat both vertices as the same vertex

  - Problem 3 is going to cover one of today's problems, the "edge-deletion to make a graph triangle free" problem

> **Note.**
> Today we are going to be discussing linear programmming (which will be utilized in problem 3 on the homework)

63 Last Time

> **Remark (Set Cover Problem).**
>
> *Given two sets within a universe Ω such that $|\Omega| = n$. Assume that if we choose all sets that we will seleect all of the elements within Ω*

64 Lecture Skeleton

(1) Weighted Set Cover problem
- Variation on Set Cover problem
- Discuss traditional approximation algorithm solution
- Discuss *linear programming* solution

(2) Linear Programming
- very informal definition in-class, formalities included post-lecture (thanks future-randy)

(3) Minimum Triangle-Free Edge-Deletion problem
- Discuss linear programming solution

65 Set Cover with Weights Problem

65.1 Problem Statement (Set Cover with Weights Problem)

Remark (Set Cover Problem).

Given a universe $\Omega = \{u_1, ..., u_n\}$ *and a collection of subsets* $\mathcal{S} = \{S_1, ..., S_2\}$ *where* $S_i \subseteq \Omega$ *for* $1 \leq i \leq n$*, find a minimum-size subcollection of* $\mathcal{C} \subseteq \mathcal{S}$ *such that*

$$\bigcup_{S \in \mathcal{C}} S = \Omega \tag{59}$$

Problem Statement 3 (Weighted Set Cover Problem).

Utilizing the Set Cover problem, suppose now that each set $S_i \in \mathcal{S}$ has a corresponding weight w_i. Find a minimum-size subcollection $\mathcal{C} \subseteq \mathcal{S}$ in order to cover the universe Ω while minimizing the total subcollection's weight.

$$\sum_i w_i : S_i \subseteq \mathcal{C} \tag{60}$$

66 Strategy 1: Layering Technique (Weighted Set Cover)

In this section, we note the **layering technique** for evaluating the Weighted Set Cover problem.

Intuition.

Our basic strategy is to find the sets that cover the most amount of elements with the minimum possible cost.

The primary basis for this algorithm is to uitlize *induction*. The premise here is to just demonstrate that after

- Determine that after t steps, *ALG does not* cover

$$e^{-\frac{w_t}{OPT}} \times n \text{ elements} \tag{61}$$

This is pretty straightforward (supposedly lol)

However, we'll learn an alternative method utilizing **linear programming**

Remark (Linear Programming).

(will discuss next time)

67 Linear Programming/Integer Programming Approach

- But why use linear programming since there's another approximation approach?
  - An LP interpretation of the problems can be solved via IP solvers, which can be solved faster than a pure combinatorial approach
    - A "good-enough" solution that can be solved in *polynomial time*

- We are able to “relax” the problem
  - We can replace constraints to make the problem easier

Note.
Our intuitoin is to think of our set as a set of *equations*

We are now going to consider some indicator varaibles such that

$$x_i = \begin{cases} 1 \text{ if } s_i \text{ is in the solution} \\ 0 \text{ otherwise} \end{cases} \tag{62}$$

We want to minimize

$$\sum_i w_i \times x_i \tag{63}$$

by the way of linear programming, we are able to add *linear constraints and equations*.

Thus, for Equation 63, we want to find all $u \in \Omega$, such that

$$\sum_{i : u \in s_i} x_i \geq 1 \tag{64}$$

The value of Equation 63 would correspond to *some* combinatorial solution, which is true since Equation 63 is only true if x_1 corresponds to 1.

Let us now relax the problem such that $x \notin \{0, 1\}$, but $x \in [0, 1]$ such that

Definition 10 (Linear Programming (Informal)).
A set of linear constriants on some variables $x_1, ..., x_n$. THis is similar to linear equations (which contains equalities) such as

$$A \times x = B \tag{65}$$

where A is a matrix and x is a set of variables

whereas for linear programming, we have

$$A \times x \geq B \tag{66}$$

with an additional constraint $\langle c, x \rangle$ such that

$$c^t \times x = \sum_i c_i x_i \text{ where } x \geq 0 \tag{67}$$

If the non-LP interpretation of the problem is feasible by the original LP problem, it stays feasible by the LP-relaxed problem.

Theorem ().
The optimal value of the LP problem, denoted as *LP*,

$$LP \leq OPT \tag{68}$$

where *OPT* represents the optimal solution to the combinatorial problem.

Remark ().
OPT in this case represents the *cost* of the problem.

We find that this is not going to be a big issue *unless LP* is significantly lesser than that of *OPT* (integrality gap).

But how can we actually compare the solution of the LP problem with that of the combinatorial problem?

- We want to *encode* the solutions of the combinatorial problem

68 LP-Based Approximation Algorithm for Set Cover

68.1 Roadmap

(1) Solve the LP-problem
  - **Obj.** Acquire $x_1, ..., x_m$ where $0 \leq x_i \leq 1$
  - But what exactly is a non-integral value of x_i?
    - Philosphically, we can think of it as a probability that i is in the set

(2) Select S_i in the solution with probability $(x_i \times \ln(n)) \vee 1$
  - Where $\ln(n)$ is some multiplicative factor (that is usually small)
  - lol where did this come from

(3) Two Possibilities
  - **Brute Force.** Be done here and repeat until a feasible solution is calculated (ie, all elements are covered)
  - **Randomized Rounding Technique.** For any element u that is uncovered, pick the cheapest set S_i containing it and just add it to the solution

68.2 Approach

(1) Set a feasible solution with probability 1

(2) Evaluate the probability that after

$$\begin{aligned} & \Pr(u \text{ is not covered (after step 2)}) = \Pr(\forall i, u \in S_i \text{ where } S_i \text{ is not chosen}) \\ = & \prod_{i : u \in S_i} \Pr(S_i \text{ is not chosen}) \\ = & \prod_{i : u \in S_i} (1 - x_i \ln(n)) \end{aligned} \tag{69}$$

Assume that any term $(1 - x_i \ln(n)) \geq 0$

$$= \prod_{i : u \in S_i} \max(1 - x_i \ln(n), 0) \tag{70}$$

Using the inequality that $e^{-t} \geq 1 - t$,

$$\leq \prod_{i : u \in S_i} e^{-x_i \ln(n)} \tag{71}$$

$$\leq e^{-\sum_{i : u \in S_i} x_i \ln(n)} \leq e^{-\ln(n)} = \frac{1}{n} \tag{72}$$

Let us now consider the cost of the LP solution

Remark ().

Cost of sets included in step 2

$$\begin{aligned}\mathbb{E}[\text{cost at Step 2}] &= \left(\sum_i w_i \times x_i\right) \times \ln(n) \\ &= \text{LP} \times \ln(n)\end{aligned} \tag{73}$$

Remark ().

Cost of sets included in step 3

$$\begin{aligned}\mathbb{E}[\text{cost at Step 3}] &\le \sum_{u \in \Omega} \Pr(u \text{ is not covered (after Step 2)}) \times \mathit{OPT} \\ &\le n \times \frac{1}{n} \times \mathit{OPT} = \mathit{OPT}\end{aligned} \tag{74}$$

where $\sum_i w_i \times x_i$ is our linear program expression.

Note.

OPT is not known by the algorithm. The algorithm doesn't need to know it in this case

69 Minimum Triangle-Free Edge-Deletion Problem

Problem Statement 4 (Minimum Triangle-Free Edge-Deletion).

Given a graph G, find the minimum number of edges needed to make G triangle-free.

To remove any confusion, if we had a graph that had multiple triangles (ie, connected components containing 3 vertices), we just want to remove the minimum number of edges to remove those connected components.

69.1 Motivation

- Triangles are found constantly in *social networks* for friend recommendation, since we would want to find vertices (people) who are adjacent to other people

70 Solution Approach (Triangle-Free Edge-Deletion Problem)

Here, we detail a linear/integer-programming based solution for the triangle-free edge-deletion problem.

Let us first consider what exactly the solution to the equivalent LP-problem would actually represent.

Claim.

The solution to the triangle-free edge-deletion problem would be **the number of edges** that need to be deleted in order to make the graph triangle-free.

70.1 Linear Programming Construction (Triangle-Free Edge-Deletion Problem)

Remark (Soln. for the Triangle-Free Edge-Deletion Problem).

The solution to the triangle-free edge-deletion problem would be **the number of edges** that need to be deleted in order to make the graph triangle-free.

- Construct an LP for the problem
  - Construct an indicator varaible that indicates whether or not we should delete an edge

For an edge $e \in E(G)$, we have the LP/objective function

$$\min \sum_{e \in E(G)} x_e : x_e \in [0, 1] \text{ by our "default constraint"} \tag{75}$$

Are there any other constraints that we can impose?

(1) For each triangle (u, v, w), at least one edge *must* be removed
  - (such that $(u, v), (u, w), (v, w) \in E(G)$)

We can represent this as

$$x_{(u,v)} + x_{(v,v)} + x_{(u,w)} \geq 1 \tag{76}$$

thus, we utilize the objective function Equation 75 with the new consraint Equation 76.

> **Note.**

70.2 Algorithm

> **Algorithm Intuition 1 (Triangle Deletion LP Algorithm).**
> (1) Solve the LP to obtain the optimal set of edges
> (2) Delete all edges (u, v) with $x_{(u,v)} \geq \frac{1}{3}$

By utilizing this algorithmic solution, we will *not* obtain a feasible solution.

> **Note.**
> For the homework, you want to modify this algorithm to achieve a better approximation

70.3 Discussing Feasibility of the Approximation Soluiton

> **Remark ().**
> there r triangles in the graph

By our LP and its constraint, we know that for each edge, its can either be part of triangle(s) or not (ie, it must be either $\geq \frac{1}{3}$ or 0)

70.4 The Cost of the LP Solution

$$\begin{aligned} m' &\triangleq \#\text{ edges whose value is} \geq \frac{1}{3} \\ &= \sum_{(u,v) \in E(G)} \end{aligned} \tag{77}$$

$$\begin{aligned} &= ALG = m' \\ &= OPT \geq ALG \geq \frac{m'}{3} \\ &= LP \geq \sum_{(u,v) \in E(G)} x_{(u,v)} \geq \sum_{(u,v) \text{ is removed}} x_{(u,v)} \geq m' \times \frac{1}{3} \end{aligned} \tag{78}$$

Thus by Equation 78, $ALG \ll 3 \times OPT$

71 Next Time

- LP-Duality

Chapter 22:
Approximation Algorithms (Part 3)

72 Remarks

(1) In the last lecture, we discussed two problems:
   (a) Weighted Set Cover Problem
      - Layering Solution (Combinatorial)
      - Linear Programming Solution
   (b) Minimum Triangle-Free Edge-Deletion Problem
      - Linear Programming Solution

73 Lecture Skeleton

In this lecture we will be discussing three problems as well as multiple ways to evaluate them utilizing various combinatorics, linear programming, and even real analysis methods.

(1) Minimum $s-t$ Cut Problem
   (a) Ellipsoid Technique
   (b) Linear Programming Technique
   (c) Metric Space Technique

(2) Multiway Cut/Multiterminal Cut Problem
   (a) Isolated Cut Heuristic (Combinatorics)
   (b) Linear Programming

(3) Max Cut Problem
   (a) Semi-definite Programming

74 Minimum $s-t$ Cut Problem

Problem Statement 5 (Minimum $s-t$ Cut Problem).

Given an directed graph $G = \langle V, E\rangle$ with edge costs $c : E \to \mathbb{R}^+$. Let $s, t \in V(G)$ be distinct vertices.

The Minimum $s-t$ Cut problem is to find the cheapest set of edges $E' \subseteq E$ such that there is no $s-t$ path in $G - E'$.

75 Background

75.1 The Ellipsoid Algorithm

In this method, we will be utilizing the Ellipsoid Algorithm in order to solve the Minimum $s-t$ Cut problem.

Definition 11 (Ellipsoid Algorithm).

The Ellipsoid Algorithm is a theoretically-polynomial algorithm for evaluating linear programs.

The Ellipsoid Algorithm works as follows:

(1) Given a linear program, denoted as $\mathcal{P}$, consider an initial ellipsoid that contains the feasible region of the answer

(2) In each iteration of the algorithm, determine whether or not the center of the ellipsoid is a feasible solution to $\mathcal{P}$ (ie, satisfies all constraints)
(3) If not feasible, identify the violated constraint. The constraint acts as a hyperplane that separates the current center from the feasible region
(4) Update the ellipsoid accordingly to the violated constraint
(5) Repeat algorithm until ellipsoid is sufficiently small, which indicates that the center must be close to the optimal solution.

Now, let us define what a *hyperplane* is as well as a *separation oracle*.

Definition 12 (Hyperplanes).
A *hyperplane* in $\mathbb{R}^n$ is any set of points in n-dimensional space that obey a *single* linear constraint within a linear programming problem.
- In 3D, a hyperplane would be a plane, and we can generalize this for any n-dimensional space.

Definition 13 (Separation Oracle).
A *separation oracle* for a convex set $S \subseteq \mathbb{R}^n$ is a function that, given a point $x \in \mathbb{R}^n$:
(1) If $x \in S$, then the separation oracle will confirm that x is feasible (ie, $x \in S$)
(2) If $x \notin S$, then the separation oracle provides a vector $a \in \mathbb{R}^n$ that defines the hyperplane that separates x from the feasible region S
    (a) tbh this is not relevant for this application, since we'll just ignore all the wrong answers…

76 Solution 1: Ellipsoid Algorithm (Minimum $s - t$ Cut)

In order to evaluate the Minimum $s - t$ Cut problem, let us generalize the problem via linear programming, utilizing a typical linear programming method, the Ellipsoid Algorithm.

Remark (Minimum $s - t$ Cut Problem).
Given a directed $G = \langle V, E \rangle$ the Minimum $s - t$ Cut Problem seeks to find the minimum cost set of edges $E' \in E(G)$ such that if E' is removed, there is no path between two vertices s and t.

By way of the Ellipsoid Algorithm, we must define and utilize a separation oracle, which will offer us a metric for determining whether or not a point $x \in \mathbb{R}^n$ is part of our solution/feasible space S.

In the context of this problem, the separation oracle will simply indicate whether or not a point x is a feasible solution.

Intuition.
A separation oracle for the Minimum $s - t$ Cut problem will indicate whether or not there exists a "good path" from s to t. In this case, we will consider a "good path" P as having a weight that is $|P| \leq 1$

- Given some x, in this case a vector of x that assigns values to each edge, we want to determine if for each path, a constraint is violated

$$\min \sum_{(u,v)\in E(G)} X_{(u,v)} \tag{79}$$

such that $\forall$ paths P from s to t:

$$\sum_{(u,v)\in P} x_{(u,v)} \geq 1 \tag{80}$$

such that $x_{(u,v)} \in [0,1]$

- We want to utilize an “oracle” as a calback function
  - This oracle needs to be given a constraint that is violated within the LP
- “I found a point x, is this point feasible?”
- Ellipsoid algorithm

$$x_{(u,v)} = \begin{cases} 1, \text{ if feasible} \\ 0 \text{ else} \end{cases} \tag{81}$$

76.1 Strategy

Let our Oracle be Dijkstra’s algorithm

- Looking at each path, if a path is not good, ie the distance from s to t is less than 1, then we just add it to the constraints

77 Alternative Stsrategy

- We utilize a new constraint

$$x_{(u,v)} = \begin{cases} 1, \text{ if } (u \in S, u \in T, u \in T, u \in S) \\ 0 \\ \text{if } (u, v \in S \vee u, v \in T) \end{cases} \tag{82}$$

Definition 14 (Metric Space).

Given a universe Ω and a funciton d where

$$d : \Omega \times \Omega \to \mathbb{R}^{+} \tag{83}$$

That upholds the following properties:

$$\begin{aligned} d(u,v) &= d(v,u) \forall u, v \in \Omega \\ d(u,u) &= 0 \\ d(u,v) &> 0 \text{ if } u \neq v \\ d(u,w) &\leq d(u,v) + d(v,w) \qquad \text{triangle inequality!} \end{aligned} \tag{84}$$

Remark (LP).

We want to minimize the sum of weights across all edges

$$\min \sum_{u,v \in E(G)} x_{(u,v)} \tag{85}$$

with the following constraints, which we derived from the definition of a metric space:

(1) $x_{(u,v)} = x_{(v,u)}$ metric space property

(2) $x_{(u,v)} \geq 0$ metric space property

(3) $\forall u, v, w x_{(u,w)} \leq x_{(u,v)} + x_{(v,w)}$ metric space property

(4) $x_{(s,t)} \geq 1$colgr("metric space property")

Now that we have an LP defined for this problem, how can we exactly deal with non-integral answers?

(1) **Randomized Rounding**

(2) **Threshold Rounding**

- Not optimal

78 New Technique: Random Threshold

Intuition.

Given two vertices s and t, draw a ball around s or radius R (the set of all points u around r such that $dist_{(s,u)} \leq r$, also denoted as $X_{((s,u) \leq R}$).

$$ball(s, R). \tag{86}$$

We observe that so long as $R \in [0, 1]$, then t will always be outside of the ball.

Given this ball, let us pick a radius $R \in (0, 1)$, uniformly at random.

78.1 Algorithm

(1) Let $S = ball(s, R)$

(2) Let $T = V - S$

Now, given this new algorithm, what exactly is the expected value? What is the approximation factor?

79 Algorithm Analysis

$$\begin{gathered} \mathbb{E}[\text{size of } (S, T)] = \mathbb{E}\left[\sum_{(u,v) \in E(G)} \mathbb{X}\right] \\ \mathbb{X} = \begin{cases} 1, & (u, v) \text{ is cut} \\ 0, & \text{otherwise} \end{cases} \end{gathered} \tag{87}$$

$$= \sum_{(u,v) \in E(G)} \Pr((u, v) \text{ is cut}) \tag{88}$$

In order to find Pr, let us first make the assumption that

$$x_{(s,u)} \leq x_{(s,v)} \tag{89}$$

by definition of the ball, we know that (u, v) must be cut if and only if u is in the ball and v is not in the ball

$$\begin{aligned}
\Pr((u,v) \text{ is cut}) &= \Pr\left(\begin{pmatrix} x_{(s,u)} \leq R \\ x_{(s,v)} > R \end{pmatrix}\right) \\
&= \Pr\left(x_{(s,u)} \leq R < x_{(s,v)}\right) \\
&= \Pr\left(R \in \left[x_{(s,u)}, x_{(s,v)}\right]\right) \\
&= \mid x_{(s,v)} - x_{(s,u)} \mid \leq x_{(u,v)}
\end{aligned} \tag{90}$$

Again, we know that (u, v) is cut if and only if the radius is larger than that of (u, v)

- What other information can we extrapolate?
  - If we form a triangle between s, u, and v, then we are able to determine the distance form (s, u) via the triangle inequality.

$$\begin{aligned}
x_{(s,v)} &\leq x_{(s,u)} + x_{(u,v)} \\
x_{(s,v)} - x_{(s,u)} &\leq x_{(u,v)}
\end{aligned} \tag{91}$$

Thus,

$$\mathbb{E}[\text{size of cut}] \leq \sum_{(u,v)\in E(G)} x_{(u,v)} = LP \leq OPT \tag{92}$$

In summary, we demonstrated that it doesn't matter what cut we're taking, its value must always be the same.

For every cut ball(s, r), the cost $= OPT$

80 Multiway Cut/ Multiterminal Cut Problem

80.1 Sources

(1) *Approximation Algorithms* (Vazirani)
  (a) Chapter 19 (Multiway Cut)
(2) UIUC 598CSC: Approximation Algorithms
  (a) Lecture 7 (Multiway Cut and k-Cut Problem) **(THIS IS A REALLY GOOD SOURCE)**

81 Background (Multiway Cut/ Multiterminal Cut)

Let us first re-define what a cut is:

81.1 Graph Cuts

Definition 15 (Cut).
Given a connected, undirected graph $G = \langle V, E\rangle$ with an assignment of weights to edges, $w : E \to \mathbb{R}^+$, a *cut* is a partition of $V(G)$ into two sets V' and $V(G) - V'$ and consists of all edges that have one endpoint in each partition.

Now, let us discuss a new problem, the Multiway Cut problem.

Problem Statement 6 (Multiway Cut Problem).
Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}^+$, and a set of terminals $S = \{s_1, ..., s_n\} \subseteq V(G)$. A *multiway cut* is a set of edges that leaves each of the terminals in a *separate component*. Find the minimum weight of such a set of edges $E' \subseteq E(G)$ where removing E' from G separates all terminals.

Let us define such a set of edges as the *isolating cut*.

82 Solutions (Multiway Cut/Multiterminal Cut)

We observe that there are two primary ways of evaluating the Multiway Cut problem:

(1) Isolating Cut Heuristic (Combinatorics/Greedy)

(2) Linear Programming

In either case, whether it be the Isolating Cut Heuristic solution or the linear programming solution, we make the following claim about the approximation ratio for both solutions:

Claim.

There exists a 2-approximate algorithm that solves the Multiway Cut problem, which we can formalize as being the following

$$\min \left[\sum_{(u,v) \in E'} x_{uv} \right] \text{ is this formulation correct?} \tag{93}$$

where x_{uv} represents the weight of the edge from vertex u to vertex v and E' represents the *isolating cut*.

This semantically, of course, simply just refers to the minimum total weight of all edges in the isolating cut E'.

83 Algorithm 1: Isolating Cut Heuristic Solution (Multiway Cut)

In this first solution, known as the Isolating Cut Heuristic, we, as with most approximation algorithms, apply a *greedy* approach.

More specificlaly, we leverage the intuition of finding all necessary *isolating cuts* to isolate each terminal s_i from each other.

After computing all *isolating cuts*, simply just union the cuts in order to derive a cut that isolates each terminal from each other.

Intuition.

For each terminal s_i, find the *minimum isolating cut* that removes it from all other terminals $s_j, i \neq j$. In practice, we will connect all other terminals $s_j, i \neq j$ to a new shared vertex t with "uncuttable"/infinite weight edges.

By repeatedly finding the minimum isolating cut for each of the terminals, we can find the union between all of these minimum isolating cuts in order to achieve a superset cut that isolates each terminal.

Note.

For the sake of lecture, Prof. Makarychev stated that we union *all* of these isolating cuts; however, the optimal solution (as Makarychev also stated) requires us to just union the first $k - 1$ cuts, where k is the number of terminals.

Note.

If there are overlapping cuts– we just disregard them.

84 Algorithm Pseudocode: Isolating Cut Heuristic (Multiway Cut)

```
For each i = 1,...,k do
-> compute the minimum weight isolating cut C(i)
end

union all isolating cuts C(i)
```

85 Algorithm Analysis: Isolating Cut Heuristic (Multiway Cut)

Now, let us analyze the algorithm and prove why the Isolating Cut Heuristic is a 2-approximation of the optimal solution.

Remark (Multiway Cut Problem).
In the Multiway Cut problem, we seek the *isolating cut* $E' \subseteq E(G)$ of minimum weight, that isolates all terminals $S = \{s_1, \ldots, s_k\}$.

Remark (Isolating Cut Heuristic Solution).
For each terminal $s_i \in S$, we calculate the *minimum weight isolating cut* that isolates s_i from $s_j \in S - \{s_i\}$. At the end of the algorithm, we find the union of all such minimum weight isolating cuts in order to form a superset cut that isolates all terminals.

Theorem (Isolating Cut Heuristic Accuracy).
The Isolating Cut Heuristic is a 2-approximate solution to the Multiway Cut problem.

Proof of Isolating Cut Heuristic Accuracy.
In order to demonstrate that the Isolating Cut Heuristic is a 2-approximate solution to the Multiway Cut problem, we must first derive some terminology and some structures for comparing the optimal solution *OPT* to the Isolating Cut Heuristic algorithm *ALG*.

Let us denote the *minimum isolating cuts* that isolate each terminal in the set of terminals $S = \{s_1, \ldots, s_k\}$ as $E_1, \ldots, E_k$. Let *OPT* represent the optimal multiway cut of G, which we will denote as E^*.

By definition, we understand that by removing the superset of cuts E^* from the edges of the original graph $E(G)$, then we will have k individual, connected components, denoted as $V_1, \ldots, V_k$ where V_i contains its corresponding terminal s_i.

Let E_i^* represent the cut that separates the component V_i containing terminal s_i from the rest of the graph.

Thus,

$$E^* = \bigcup_{i=1}^{k} E_i^* \tag{94}$$

Let $\partial(V_i)$ represent the set of all edges leaving the component V_i, which we will call the *edge boundary* of V_i. Formally, we find that

$$\partial(P_i) = \{(u, v) : u \in P_i, v \notin P_i\} \tag{95}$$

With this background in mind, let us now formulate our proof.

(1) We observe that the weight of a minimum isolating cut for a terminal s_i must be less than or equal to the weight of all edges leaving its corresponding component V_i. Mathematically,

$$|E_i| < |\partial(V_i)| \text{ for all } 1 \leq i \leq k \tag{96}$$

   (a) This is true because $\partial(V_i)$ is an isolating cut of s_i (and intuitively, the isolating cut can either consist of all of the outgoing edges of V_i or less.)

(2) The weight of the set of the optimal solution is equivalent to $\frac{1}{2}$ of the total weight of all edge boundaries for all components V_i for all $1 \leq i \leq k$.

   Formally,

$$\begin{aligned} E^* &= \frac{1}{2} \sum_{i=1}^{k} |\partial(V_i)| \\ &\geq \frac{1}{2} \sum_{i=1}^{k} |E_i^*| \text{ ...i think this is wrong} \\ 2 \times E^* &= \sum_{i=1}^{k} |\partial(V_i)| \end{aligned} \tag{97}$$

   (a) We observe that every edge in the superset of *isolated cuts* must be incident to two components. (This makes sense intuitively because in order to isolate the different components, we must remove their ougoing edges.)
   (b) Given that each edge, in order to be within the superset of *isolated cuts*, must contain an endpoint in two unique components, then we need to *normalize* the total weight by dividing the total weight of the edge boundaries by 2.

> **Note.**
> The edge boundary of V_i, denoted as $\partial(V_i)$, must isolate a terminal s_i, thus this must be a feasible solution for a multiway cut.

(3) The Isolating Cut Heuristic is a 2-approximate algorithm of *OPT*.
   (a) By 1(a), 2(a), 2(b)

In Equation 97, we simply just showed that the Isolated Cut Heuristic is a feasible solution to the Multiway Cut Problem that is upper-bounded by the optimal solution. Thus, the Isolated Cut Heuristic must be 2-approximate of the optimal solution, as required. ■

Now that we have designed and analyzed a combinatorial/greedy solution, let us now examine the linear programming solution.

86 Algorithm 2: Linear Programming Solution (Multiway Cut)

Let us define the following constraints:

(1) $x_{(u,v)} = x_{(v,u)}$

(2) $x_{(u,v)} \leq x_{(u,v)} + x_{(v,w)}$
(3) $x_{(u,v)} \geq 0$
(4) $x_{(t_i,t_j)} \geq 1$

We note that this is a **very similar** set of constraints to the single terminal variation of this problem. As it turns out, there is a better way of evaluating this problem.

TODO

87 Max Cut Problem

In this section, we detail the Max Cut problem, which is *another* graph theory problem that is NP-hard.

87.1 Citations

87.2 Background

Remark (Cuts).
Given a connected, undirected graph $G = \langle V, E \rangle$ with an assignment of weights to edges $w : E \to \mathbb{R}^+$, a *cut* is a partition of $V(G)$ into two sets V' and $V(G) - V'$ and consists of all edges that have one endpoint in each partition.

Problem Statement 7 (Max Cut).
Given a graph G, and two components L and R, we want to maximize the number of cut edges in order to cut G in L and R.

The primary idea here is that for any two components $L \subseteq G$ and $R \subseteq G$, we want to maximize the number of edges necessary to split G into L and R.

87.3 Max Cut Solutions

(1) Randomized Algorithm
(2)

88 Naive Solution/Algorithm (Max Cut)

We just split the graph into two parts randomly.

Claim.
Utilizing the random assignment solution, there exists a 0.5-approximation algorithm.

Definition 16 (Approximation Resistant Algorithms).

.
Let us first consider

$$\begin{aligned} \Pr((u, v) \text{ is cut}) &= \\ &= \frac{1}{2} \end{aligned} \tag{98}$$

Thus,

$$\mathbb{E}[\text{cut edges}] = \frac{1}{2} \times |E| \geq \frac{1}{2}\ OPT \tag{99}$$

■

Note.
It's worth noting that even a greedy algorithm can also achieve 0.5-approximation

89 Improved Solution: Linear Programming (Max Cut Problem)

Let us define the following integer program

$$\max \sum_{(u,v) \in E(G)} x_{(u,v)} \tag{100}$$

where it is x-metric and $x_{(u,v)} \in [0, 1]$

- This is a very weak LP, since essentially it allows us to just delete all edges, which is feasible, but not worthwhile

LP is useless!

Note.

We want to now try semi-definite programming, which is essentially a generalization of LP.

Intuition.
We want to fix the graphs onto a high-dimensional sphere in high-dimensional space, thus the distance from each vertex to the center is exactly 1.

$$\max \sum_{(u,v) \in E(G)} \|X_u - X_v\| \tag{101}$$

such that $\|x_u\| = 1$

Intuition.
We want to maximize the distance between the vertices within the high-dimensional space

Note.
As it turns out, no LP can solve this problem as is, we can only solve the new problem:

$$\max \sum_{(u,v) \in E(G)} \|X_u - X_v\|^2 \tag{102}$$

such that $\|x_u\|^2 = 1$.

Now, we just have a lot of X_u's and X_v's in high dimensional space.

The ideal solution here is that the distance must exist within $[-1, 1]$, in which we can delineate into the L and R components.

Here is our new objective function:

$$\max \sum_{(u,v)\in E(G)} \frac{\| X_u - X_v \|^2}{4} \quad \text{we normalize using } \frac{1}{4} \tag{103}$$

Next time. We will discuss the actual algorithm, which occurs whenever we cut a hyperplane randomly, and split it into two halves L and R.

Chapter 23: (05/28/24 Lecture)

90 Semi-Definite Programming

91 Max Cut Problem, Continued

91.1 Structure of the Solution

(1) We want to define a SDP-relaxation of the problem
(2) Evaluate the SDP-relaxation of the problem and obtain

a set of nodes $\{X_u\}u \in V$
(3) Pick a random unit vector z (in practice,

a Gaussian vector or normal vector) such that

$$\begin{aligned} L &= \{u : \langle x_u, z \rangle \leq 0\} \\ R &= \{u : \langle x_u, z \rangle > 0\} \end{aligned} \tag{104}$$

where z is normal to the hyperplane.

91.2 Solution (cont'd) Max Cut

Remark ().
We derived the objective function

$$\frac{\| X_u - X_v \|^2}{4} \tag{105}$$

as a means of *relaxing* the constraints of the problem,

We just want to determine whether or not a vertex exists in the L half or the R half.
After imposing the graph into n-dimensional space, what kind of algorithm can we utilize?

Intuition.
Randomly choose a hyperplane that cuts our imposed sphere into two halves, L and R, respectively.

- Surprisingly, this is the *best* possible algorithm for evaluating this problem (under particular constraints)

92 Algorithm Max Cut

93 Algorithm Analysis Max Cut

Theorem (Randomized Max Cut Approximation).

$$ALG \geq \alpha \cdot \text{SDP} \tag{106}$$

where α is 0.87.

Remark ().
The optimal value must be upper-bounded by the solution to the relaxation of the SDP-problem.

$$OPT \leq \text{SDP} \tag{107}$$

We seek to show that

$$\mathbb{E}[ALG] \geq \alpha \cdot \text{SDP} \geq \alpha \cdot OPT \tag{108}$$

We can achieve this by simply maximizing the following expression for α:

$$\sum_{u,v \in E(G)} \Pr(\text{sign}(\langle x_u, z \rangle) \neq \text{sign}\ (\langle x_v, z \rangle)) \geq \alpha \cdot \sum_{(u,v) \in E(G)} (*) \tag{109}$$

Let us now represent X_u and X_v using θ, versus their explicit values:

$$\begin{aligned}(*) &= \frac{\|x_u\|^2 + \|x_v\|^2 - 2\langle x_u, x_v \rangle}{4} \\ &= \frac{1 - \langle x_u, x_v \rangle}{2} \\ &= \frac{1 - \cos(\theta)}{2}\end{aligned} \tag{110}$$

As for the $\Pr(-)$, we have $\Pr(-) = \frac{\theta}{\pi}$. Now, we can perform the necessary substitutions in order to obtain

$$\begin{aligned}\Pr(-) &\geq \Pr(*) \\ \frac{\theta}{\pi} &\geq \alpha \cdot \frac{1 - \cos(\theta)}{2}\end{aligned} \tag{111}$$

From, here we can find a generalizable α
(1) $\theta = \pi \Rightarrow \alpha = 1$
(2) ...

For all possible theta values, however, we find that the best possible one is located at $\theta^{\star} = \frac{3}{4}$. Given $\theta^{*} = \frac{3}{4}$, then,

$$\begin{aligned}&= \frac{1 - \cos(\theta^{\star})}{2} \\ &= \frac{1 + \frac{\sqrt{2}}{2}}{2} \\ &\approxeq 0.87\end{aligned} \tag{112}$$

In the worst possible case, then ALG is 0.87-approximate, in the best-case.

94 Linear Programming

95 Definition (Linear Programming)
Given a set of variables

$$X = (x_1, ..., x_n), \tag{113}$$

we seek to minimize some objective function:

$$\begin{gathered} \min\langle c, x\rangle \\ \cdots \\ \min c^T x \\ \cdots \\ \min \sum_i c_i \cdot x_i \end{gathered} \tag{114}$$

In which we possess constraints, which have the form

$$Ax \geq b \tag{115}$$

96 Standard Forms of Linear Programs
In the minimization sense, we will be given a problem

$$\min(c, x) \tag{116}$$

with which we have constraints $Ax \geq b$ where $x \geq 0$.

which is also equivalent to

$$\max(b, y) \tag{117}$$

for constraints

$$A^T y \leq c \tag{118}$$

where $y \geq 0$.

96.1 Intuition of Constraints
For a minimization problem, we can think of the constraints as simply setting the floor for the problem, as well as constraints setting a ceiling in the context of a maximization problem.

Whenever we are actually visualizing this, the constraints are simply just a line, and we want to determine on which side of the line our solution should be in. If we had multiple lines, then we would form a polygon.

96.2 Intuition of Linear Programming
We can think our linear program, then we just want to find the minimum of the maximum value of that polytope.

Note.
The points that satisfy a LP are convex

97 Defining Convexity

Definition 17 (Convexity).

98 Solving Linear Programs
There are three common algorithms for solving LPs:

(1) Simplex Method
(2) Interior Point Method
(3) Ellipsoid Method

99 Simplex Method

Definition 18 (Simplex Method).
Given the constraints, examine a vertex of the polytope and examine the incident edges, following the points until we cannot possibly minimize/ maximize the value of the problem

100 Interior Point Method

Definition 19 (Interior Point Method).
Starting from *any* interior point of the polytope, walk down to the minimum value.

101 Ellipsoid Method

Definition 20 (Ellipsoid Method).
Define an ellipsoid $\mathcal{E}$ such that $\mathcal{E}$ encapsulates the polytope. Then, examine the center of such an ellipsoid, by which we split the ellipse, then we just perform divide-and-conquer.

Chapter 24:
LP Duality

102 Motivation

Let us minimize

$$\min x_1 \tag{119}$$

given the following constraints:

$$\begin{aligned} x_1 - 2x_2 &\geq 3 \\ x_1 + x_2 &\geq 9 \end{aligned} \tag{120}$$

Let us now consider two arbitrary points

$$\begin{aligned} x_1 &= 7 \\ x_2 &= 2 \end{aligned} \tag{121}$$

We know that $(7, 2)$ is a feasible solution, so we know that it is at least upper-bounded by $(7, 2)$. But how can we prove this?

Isolate x_1

$$\begin{aligned} x_1 - 2x_2 \geq 3 \\ -2(x_1 + x_2 \geq 9) \\ = 3x_1 \geq 21 \end{aligned} \tag{122}$$

We can perform this isolation for any n- dimensional linear-programming problem.

103 Ball and Cone as LP Duality

Our constraints represent “counter forces” which we can leverage

Unit 7: Dynamic Graph Algorithms

Chapter 25:

]

Chapter 26:
Graph Orienting