CS 388R: Randomized Algorithms, Fall 2021 August 26th, 2021

Lecture 1: Introduction; Min-Cut

Prof. Eric Price *Scribe: Shengsong Gao and Nathaniel Sauerberg*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In this lecture we covered some important course logistics, went over some introductory material on randomized algorithms, including giving some examples of algorithms, defining the types of randomized algorithms, giving reasons for using randomness, and giving some common applications of randomness in algorithms. We also examined the global min-cut problem, including analyzing a randomized algorithm to solve the problem and comparing it to a deterministic one.

2 Course Logistics

- Reminder that the midterm exam for this course is on October 12th.
- Course Website: https://www.cs.utexas.edu/~ecprice/courses/randomized/fa21/
- Grade Calculation:

40%	Homework
20%	Midterm Exam
20%	Final Exam
20%	Scribe Notes

- There will be a Google Sheet for signing up to scribe for specific lectures. For each lecture, two students work as scribes together, taking notes and formatting the notes in LaTeX.
- In the future, classes may begin with puzzles about randomness.

3 Introduction to Randomized Algorithms

A randomized algorithm is one that uses randomness internally. To analyze randomized algorithms, we'll need to do lots of analysis of random variables.

3.1 Examples of randomized algorithms

There are several examples of randomized algorithms you have seen in undergrad.

- **Random number generation**

- **Quicksort**
- **Random projection** (this topic will be covered in October)
- **Randomized Rounding** (may not be covered in this class)
- **K-means** (randomized initialization; afterwards deterministic)
- **Hash tables** (this topic will be covered in September)

3.2 Random Seeds

We can think of the random choices used in an algorithm as a "random seed". For any input, the algorithm should work on *most* seeds (definition of "most" will come later). This means there can exist seeds for which the algorithm doesn't work well. Indeed, if this was not true then we could just fix a seed and get a deterministic algorithm.

So if the algorithm doesn't work well for every random seed, why do we want to use randomness? What are the advantages of randomized algorithms over deterministic ones?

3.3 Reasons to use randomness in an algorithm

- *Simplicity*
- *Speed*
- *To combat adversarial inputs*

"Adversarial" here means "undesirable" and not necessarily "intentionally malicious." For example, an already sorted input might be adversarial for Quicksort, but this might arise just because somebody sorts a list twice. It might also mean input from a real adversary. For example, someone might try to crash your server by creating many accounts with usernames that cause hash collisions or something.

3.4 Uses of randomness in algorithm

- **Fingerprinting**

  Suppose we have large items that we want to compare. It would be faster to compare their hashes rather than comparing every bit of the full items. This could, for example, allow us to find duplicates quickly.

- **Random Sampling**

  An example of this is spectral specification, the approximation of arbitrary graphs by sparse graphs. Note that we might want to sample uniformly at random, or we might want to intentionally over-sample rare but important elements.

- **Load balancing**

  For example, this could be sues to help ensure that each part of a distributed system is assigned the same amount of work. Coordination might be difficult, but it might be easier to ensure an equal load for each component in expectation.

- **Symmetry breaking**

  Consider a distributed computing system or routing system. If each component computer is exactly identical, the might all try to communicate at the exact same time, use the same edges in the communication network, etc. This could overload the system, so it might be better to use randomness to decide when to communicate or which edge to use.

- **Probabilistic existence proofs**

  One can prove existence of a structure by creating a randomized algorithm that produces one. This might be easier than a deterministic construction in some cases.

  For example, in many regimes, the best existence proofs of expander graphs are randomized constructions.

  Another example of probabilistic existence proofs is the Lovász Local Lemma, which we discuss next.

3.5 The Lovász Local Lemma

Consider a setting where we have n events that each occur with relatively low probability p, and we'd like to show that there is a non-zero probability that none of them occur. For example, perhaps we'd like a non-zero probability that *none* of the computers in a network crash.

If the events are independent, then the probability that none occur is just $(1-p)^n > 0$.

What if they are dependent, but not "too dependent"? The Lovász Local Lemma lets us prove the same nonzero probability that none occur.

Suppose we're given a dependency graph for the events, where the events are nodes and edges represent dependence between events. We assume that the graph is sparse and the nodes have degree at most d. This is the "not too dependent" assumption.

The Lovász Local Lemma guarantees that if $4pd < 1$, then there is a positive probability that no event happens.

Note that if $pd \geq 1$, then there could be a clique of d events are adversarially correlated so that at one of them always occurs. Hence, if $pd \geq 1$, we cannot guarantee a probability greater than 0 that no event occurs. So in a sense, we only need a constant factor stronger of an assumption to give our guarantee!

3.6 Game Tree Evaluation problem

Sometimes, randomized algorithms can even achieve performance that is impossible to guarantee with deterministic algorithms. An example is the problem of game tree evaluation.

Let's say we have a binary tree built from using the NOR function: the leaves contain the initial input, either 0 or 1, and each non-leaf node contains the output of an NOR operation whose operands are the children of that node. (One reason this might be desired is that NOR is Turing-complete, so evaluating this tree is equivalent to doing computation.) The task is to find the value of the root node, and we'd like to minimize the number of leaves that need to be observed in order to find the value of the root. Let n be the number of leaves.

We did not go over the algorithms or details in this lecture, but we stated the following:

1. Any deterministic algorithm will needs to look at each of the n leaves to determine the root value in the worst case.
2. For a good randomized algorithm, in the best case (if we get really lucky), we need only observe $\sqrt{n}$ leaves.
3. For a good randomized algorithm, with "average" luck, it will still only need to check around $n^{.6}$ leaves.

3.7 Fun fact about Python

Older versions of Python (2.x) use a fixed seed for its internal hashtables. This gave deterministic behavior but was not resilient against adversarial inputs. For example, if a program that simply adds some elements to a dictionary and then prints it will always print the items in the same order. Python 3 uses random seeds that are actually random, so such a program is no longer guaranteed to print items in the same order every time it is run.

3.8 Types of Randomized Algorithms

There are two main classes of randomized algorithms. For an algorithm that uses randomness internally, either the runtime or the correctness should be a random variable that depends on the random seed.

Las Vegas Algorithms:

- always correct
- runtime is random (has chance of being slow)
- Example: Quicksort

Monte Carlo Algorithms:

- runtime is always bounded
- correctness is random (has chance of being wrong
- Example: Random projection

For any problem, the existence of a Las Vegas algorithm implies the existence of a Monte Carlo algorithm – we can just give up and make a potentially incorrect guess when the Las Vegas solution takes too long.

However, existence of a Monte Carlo algorithm does not always imply the existence of a Las Vegas algorithm. This reverse implication requires us to be able to verify if the Monte Carlo algorithm's output is correct. In such situations, we can convert a Monte Carlo algorithm into a Las Vegas algorithm by repeatedly running the algorithm until we get a correct output.

4 The Global Min-Cut Problem

4.1 Problem Definition

The global min-cut problem is a graph problem, defined below. Note that we are only concerned with undirected, unweighted graphs for this class.

Recall that a **cut** of a graph is a partition in the vertices into two disjoint subsets. Each subset must be nonempty (otherwise the problem would be trivial). We will usually identify a cut with a subset $S \subsetneq V$; the other part in the cut is then the complement $\overline{S}$. Also, we'll define $cut(S)$ to be the set of edges crossing the cut S.

An edge $(u, v) \in E$ is said to **cross a cut** S if one of u and v is in S and the other is not.

The global min-cut asks us to find a cut with the smallest number of crossing edges. The number of crossing edges is sometimes referred to as the value of a cut. Formally, the problem is as follows:

Given: A graph $G = (V, E)$

Output: A strict, nonempty subset of the vertices $S \subsetneq V$, $S \neq \emptyset$, that minimizes the number of edges crossing the cut:

$$\arg \min_{S \subsetneq V, S \neq \emptyset} |\{(u, v) \in E, u \in S, v \in \overline{S}\}|$$

4.2 The s, t Min Cut Problem

In undergraduate algorithms, you likely saw the Ford-Fulkerson algorithm [**?**] to solve the s, t min cut problem. In the s, t min-cut problem, we are given a graph and two distinguished nodes s and t. We are asked to find the smallest cut in which s and t are in opposite parts of the partition. Formally:

Given: A graph $G = (V, E)$ and distinguished nodes $s, t \in V$

Output: A strict, nonempty subset of the vertices $S \subseteq V$ containing s and not t that minimizes the number of edges crossing the cut:

$$\arg \min_{S \subseteq V, s \in S, t \notin S} |\{(u, v) \in E, u \in S, v \in \overline{S}\}|$$

Of course, this problem is highly related to the global min-cut problem. The value of the global min cut is the minimum, over all possible s, t pairs, of the value of the s, t min cut. This gives an equivalent definition to the one given in the previous section.

4.3 A Deterministic Algorithm

The Ford-Fulkerson algorithm [**?**] uses the max-flow min-cut theorem, which says that the value of the max (s,t)-flow in a graph is equal to the value of the min (s,t)-cut. The algorithm finds the max-flow, which simultaneously find the min cut.

4.3.1 Runtime Analysis

How can we use this algorithm to find the global min cut? Naively, we can simply find the min cut for all n^2 pairs of vertices and then return the smallest.

Recall that each iteration of the Ford-Fulkerson algorithm [**?**] finds an $s - t$ path in the residual graphs. This can be done in $O(m)$ with one run of BFS (and the residual graph can also be updated in $O(m)$). How many iterations do we need to do? Well we increase the value of the flow by one each iteration, so our number of iterations is at most the value of the min cut. This can be upper bounded by the degree of s, which is less than n.

Our overall runtime is $O(n^2)$ (s,t) pairs times times $O(n)$ iterations per pair times $O(m)$ per BFS iteration, so $O(n^3m) = O(n^5)$.

We can improve this naive algorithm slightly be fixing one vertex as s and trying every other vertex for$t$. This works because s must be in either S or $\overline{S}$ in any min cut, and we can assume without loss of generality that is it in S. This means we only have to check $O(n)$ pairs, bringing our overall runtime down to $O(n^4)$.

Can we do better than this with a randomized algorithm? We'll see that we can.

4.4 A Naive Randomized Algorithm

The basic idea of our randomized algorithm is that if we pick a random edge, it is unlikely to be in the min cut, since the min cut is "small". For our analysis, we'll fix a min cut and focus on our probability of finding that specific min cut (which is at least the probably of finding any many cut).

Basic Algorithm:

1. Let S be a min cut
2. Repeat the following until only two vertices remain:
   (a) Pick a random edge e
   (b) Contract along the edge e
3. Return the pre-image of each of the two remaining vertices as the cut

When we say **contract**along $e = (u, v)$, we mean combine u and v into one super-node. Every edge that was previously incident to u or v should now be incident to the new node (with the exception of e itself) even if this creates multi-edges.

Observation: After each iteration of step 2, every cut that does not contain e has the same size in the new graph as it did in the old graph. Therefore, in particular, step 2 preserves the size of the minimum cut S as long as $e \notin S$.

When we say the **pre-image of a vertex** v, we mean all vertices that have been contracted into v. So when we stop with two remaining vertices, the pre-image of one corresponds to a S (if we are lucky and the each step was correct) and the pre-image of the other corresponds $\overline{S}$. It's also good that we stop with two vertices because if we iterated again, we would surely contract the min cut as only one cut remains.

4.4.1 Analysis of Correctness

First, we analyze the probability of picking an incorrect edge in any step. What is the probability that we pick an edge on the minimum cut when there are n vertices and m edges remaining? Well, we pick a random one of the m vertices, and $|cut(S)|$ are in the min cut. Then, we notice that we can upper bound the size of the min cut $|cut(S)|$ with the minimum degree of the graph since if v has the minimum degree d, the cut $\{v\}, V \setminus \{v\}$ has value d. Next, we notice that the average degree is an upper bound on the minimum degree. Finally, we compute that the average degree is $2m/n$ since each edge contributes 2 to the total degree and simplify.

$$\begin{aligned} Pr[e \in cut(S)] &= \frac{|cut(S)|}{m} \\ &\leq \frac{(min\ degree\ of\ G)}{m} \\ &\leq \frac{(average\ degree\ of\ G)}{m} \\ &= \frac{2m/n}{m} \\ &= \frac{2}{n} \end{aligned}$$

Note that our probability of picking an incorrect edge depends only on n and is independent of m.

Now, we're ready to analyze the overall probability our algorithm finds the min cut.

Attempt 1: Union Bound

The probability of failing ever can be upper bounded by just summing the probabilities that we fail at any step.

$$Pr[Failure] \leq \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{4} + \frac{2}{3}$$

This is roughly equal to $2 \log n$, which is greater than 1 (even if you just take the last two terms). Unfortunately, this is worthless– we already knew probability of failure was not more than 1.

Attempt 2:

Instead, let's consider the probabilities of success at each step, multiplying them together to find the probability of succeeding at every step. We can simplify this by multiplying over the numerators and over the denominators and then simplifying the resulting factorials:

$$\begin{aligned}
Pr[Success] &\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{5}\right)\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{(n-2)!}{n!/2} \\
&= \frac{2}{n(n-1)} \\
&\geq \frac{2}{n^2}
\end{aligned}$$

An alternative analysis is to notice that product is telescoping: the $(n-2)$ in the numerator of the first factor cancels with the $(n-2)$ in the denominator of the third factor, the $(n-3)$ in the numerator of the second with the $(n-5)$ in the denominator of the fifth, etc. The cancellation leaves $\frac{2\cdot 1}{n(n-1)}$, as before.

In either case, we have an algorithm with at least $\frac{2}{n^2}$ probability of being correct.

4.4.2 Amplifying Success Probability

The basic algorithm only has a probability of at least $\frac{2}{n^2}$. Can we run it multiple times to improve the success probability? We can, since if we a smaller cut, we know any iteration that found a larger one was incorrect. In other words, if we do several trials, we can find the best trial by just taking the smallest cut found in any trial. This leads to an improved algorithm:

Algorithm (Karger '93 [?]):

1. Repeat the following $O(n^2)$ times and return the smallest cut found in any of the repetitions:
   (a) Let S be a min cut
   (b) Repeat the following until only two vertices remain:
      i. Pick a random edge e
      ii. Contract along the edge e
   (c) Return the pre-image of each of the two remaining vertices as the cut

Notice that we have a n^2 in the denominator of our success probability and we run $O(n^2)$ trials. This is not a coincidence. What is the overall failure probability? Well the entire algorithm only fails if every single iteration fails. Let's say we do $n^2/2$ iterations. Since the iterations are independent, the overall failure probability is less than:

$$(1 - 2/n^2)^{(n^2/2)}$$

This probability approaches $1/e$, giving us a constant probability of success which is good enough. (In general, $(1 - 1/x)^x \to 1/e$ as $x \to \infty$.)

4.4.3 Runtime Analysis

It takes $O(m \log n)$ time to run each trial. The complexity boils down to doing $O(m)$ contractions of edges. When we contract edges, we need to keep track of which vertices have been contracted into each other. We can do this with simple data structures in $O(\log n)$ (and more quickly with more complicated data structures, see the disjoint-set data structure).

Since we do $O(n^2)$ trials, our overall runtime is $\tilde{O}(n^2 m) = \tilde{O}(n^4)$. (The $\sim$ means we are dropping polylogarithmic factors.)

Note: Our simple randomized algorithm has already matched the performance of Ford-Fulkerson. And it didn't require any complicated residual graph constructions or data structures or the max-flow min-cut theorem. However we can do ever better.

4.5 An Improved Randomized Algorithm

Let's think about where our algorithm is likely to fail. The early contractions have much lower failure probability (like $\frac{2}{n}$) since the the number of nodes is much higher, while at the last step the failure probability is a whopping 2/3. On the other hand, it is also more work to do the contractions earlier on in the algorithm since the cost of contractions corresponds to the size of the graph.

Hence, if we fail (for the first time) towards the end, we've already done most of the work. It doesn't make sense to just restart from the beginning, and instead, we should try the later steps multiple times. Let's rejigger our algorithm to retry later steps. We're going to do something recursive.

Algorithm (Karger-Stein '96 [?]):

1. Repeat the following twice:
   (a) Do $n - \frac{n}{\sqrt{2}}$ steps of the basic algorithm (that is, repeatedly picking a random edge and contracting along it)
   (b) This leaves a graph with $\frac{n}{\sqrt{2}}$ vertices
   (c) Recursively find the min cut in the remaining (post-contractions) graph [1]
2. Return the better of the two min cuts found

[1] Scribe comments: although we didn't discuss it in class, the recursive algorithm should have a base case. WE might just use $n = 2$, where there is only one cut remaining. We could also just decide to find the min cut by brute force once n is small enough.

4.5.1 Analysis of Correctness and Runtime

First, let's consider the runtime. Let $T(n)$ be the runtime of the algorithm with n nodes.

It takes $O(n^2)$ time to do the $n - \frac{n}{\sqrt{2}}$ steps (this is just the same bound as before for the whole trial, not just the first steps). Additionally, we have the two recursive calls on instances of size $n/\sqrt{2}$. Hence, $T(n) = O(n^2) + 2T\left(\frac{n}{\sqrt{2}}\right)$. Solving this recurrence gives a runtime of $O(n^2 \log^2 n)$.

Runtime: $O(n^2 \log^2 n)$.

Next, let's consider the probability that our algorithm succeeds. Let $P(n)$ be the probability that our algorithm fails when run on an instance with n vertices.

First, consider the non-recursive part. What is the probability that we succeed in each of those $n/\sqrt{2}$ steps? It turns out to be at least $1/2$. We can do similar analysis from before, leaving out some details such as the rounding of $n/\sqrt{2}$ and other similar issues:

$$\begin{aligned}
Pr[Success] &\geq \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right)\cdots\left(\frac{n/\sqrt{2}}{n/\sqrt{2}+2}\right)\left(\frac{n/\sqrt{2}-1}{n/\sqrt{2}+1}\right)\left(\frac{n/\sqrt{2}-2}{n/\sqrt{2}}\right) \\
&= \frac{(n-2)!/(n/\sqrt{2}-2)!}{n!/(n/\sqrt{2})!} \\
&= \frac{(n-2)!}{n!} \cdot \frac{(n/\sqrt{2})!}{(n/\sqrt{2}-2)!} \\
&\approx \frac{1}{n^2} \cdot (n/\sqrt{2})^2 \\
&= 1/2
\end{aligned}$$

Now, we know that our algorithm fails only if both of the two repetitions fails. And one of those repetitions succeeds only if both the "first part" (the non-recursive part) and the recursive call succeed. Writing this with the recurrence, this gives:

$$\begin{aligned}
\mathbb{P}[success] = P(n) &= 1 - \mathbb{P}[\text{both tries fail}] \\
&= 1 - \mathbb{P}[\text{one try fails}]^2 \\
&= 1 - \mathbb{P}[1 - \mathbb{P}[\text{first part fails}] \cdot \mathbb{P}[\text{recursive part fails}]]^2 \\
&= 1 - \left[1 - \frac{1}{2}P(n/\sqrt{2})\right]^2
\end{aligned}$$

Solving this recurrence[2] gives a success probability $P(n) = O\left(1/\log n\right)$. This can be amplified to make an algorithm with a good probability of success.

[2]Scribe comments: again, we probably need to assume some base case for the recursive algorithm, corresponding to an initial condition on the recurrence like $P(2) = 1$ or similar.

5 Bibliography

References

[FF56] L. R. Ford Jr., D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Mathematics* 8: 399–404. 1956

[K93] D. Karger. Global min-cuts in RNC, and other ramifications of a simple min-out algorithm *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms* 1993

[KS96] D. Karger, C. Stein. A new approach to the minimum cut problem *Journal of the ACM* 43 (4): 601. 1996