Lecture Notes CS:5360 Randomized Algorithms
Lecture 1: Aug 21, 2018
Scribe: Geoff Converse

1 What is a Randomized Algorithm?

These are algorithms that make coin tosses during their execution and take actions that depend on outcomes of these coin tosses. In other words, these algorithms have access to *random bits*. Notice that we are **not** talking about inputs being generated randomly from some probabalistic distribution. The randomness is internal to the algorithm.

2 Classifying Random Algorithms

- Las Vegas algorithms: these make no errors, however, the running time of the algorithm is a random variable. We are typically interested in the *expected* runtime of these algorithms.
- Monte Carlo algorithms: these algorithms make errors, but with some probability that can be controlled. For example, we can say that a certain algorithm has a $\frac{1}{10}$ or maybe a $\frac{1}{100}$ probability of making an error. The runtime of Monte Carlo algorithms can either be deterministic or a random variable.

2.1 Example: Balanced Partition

INPUT: a list L of distinct numbers
OUTPUT: Lists L_1 and L_2 that partition L and satisfy:

1. Every element in L_1 is less than every element in L_2.
2. $\frac{|L|}{3} \leq |L_1| \leq \frac{2|L|}{3}$

In order to solve this problem, we can try to find an approximate median of L by using a randomized step.

```
FUNCTION: randomizedPartition(L=[1..n])
    p <- index chosen uniformly at random from {1,2,...,n}
    for i<- 1 to n do:
        if L[i] <= L[p]:
            L_1 <- L_1 append L[i]
        else:
            L_2 <- L_2 append L[i]
    return (L_1, L_2)
```

Notice that this does not always return a partition that satisfies both requirements above. In fact, it has a $\frac{2}{3}$ chance of error.

Lemma 1 *The function randomizedPartition runs in $O(n)$ time with an error probability of $2/3$.*

Using `randomizedPartition` as a subroutine we can design a Las Vegas algorithm for the BalancedPartition problem and we can also design a Monte Carlo algorithm with a much smaller error probability. First, we state the Las Vegas algorithm.

```
FUNCTION: randomizedPartitionLV(L[1..n])
    repeat:
        (L_1,L_2) <- randomizedPartition(L)
    until: |L|/3 <= |L_1| <= 2|L|/3
    return (L_1, L_2)
```

This algorithm will not make any errors, but its runtime is a random variable. This is because with probability 1/3, the algorithm performs one iteration of the repeat-until loop, with probability (2/3)(1/3) the algorithm performs two iterations of the repeat-until loop, etc.

```
FUNCTION: randomizedPartitionMC(L[1..n])
    for i <- 1 to k do:
        (L_1, L_2) <- randomizedPartition(L)
        if (|L|/3 <= |L_1| <= 2|L|/3) then:
            return (L_1, L_2)
     endfor
     return "Failed"
```

Notice that we are *independently* repeating the `randomizedPartition` algorithm k times. This process of increasing the probability of correctness is called *probability amplification.*

Theorem 2 BalancedPartition *can be solved by a Las Vegas algorithm in expected* $O(n)$ *time.*

Theorem 3 BalancedPartition *can be solved by a Monte Carlo algorithm in* $O(kn)$ *time with probability* $1 - \left(\frac{2}{3}\right)^k$.

3 Why use randomization in algorithms?

- To improve efficiency with faster runtimes. For example, we could use a randomized quicksort algorithm instead of the deterministic quicksort. Deterministic quicksort can be quite slow on certain worst case inputs (e.g., input that is almost sorted), but randomized quicksort is fast on all inputs.
- To improve memory usage. Random sampling as a way to sparsifying input and then working with this smaller input is a common technique.
- To make algorithms simpler. For example, see Karger's min-cut algorithm in the next lecture.
- In parallel/distributed/streaming models of computation, randomization plays an even more critical role. In distributed computing, each machine only has a part of the data, but still has to make decisions that affect global outcomes. Randomization plays a key role in informing these decisions.

4 Classification of Problems Based on Randomization

- **P** = the class of *decision problems* (problems with boolean answers) that can be solved in polynomial time. We typically say that these can be solved *efficiently*.
- **RP** (randomized polynomial) = the class of decision problems L such that L can be solved by a polynomial time algorithm A with the property:
  - If $x \in L$ (x is a "yes" instance of L), then $Pr(A(x) = 1) \geq 1/2$.
  - If $x \notin L$ (x is a "no" instance of L), then $Pr(A(x) = 0) = 1$.

  Note that in this definition the algorithm A has one-sided error, only for "yes" instances. Also, we clearly see that $P \subseteq RP$. Finally, the choice of the constant $1/2$ in the above definition is somewhat arbitrary. By using probability amplification (see below), we can drive the error probability down quite efficiently.

```
FUNCTION: amplifiedA(x)
    for i <- 1 to k do
        if A(x) = 1, then
            return 1
    return 0
```

  In this use of probability amplification, when we input a "yes" instance, the probability of the output begin incorrect is 2^{-k}. So then $Pr(\texttt{amplifiedA}(x) = 1) \geq 1 - 2^{-k}$.

- **CoRP** = $\{L|\overline{L} \in RP\}$ = the class of decision problems such that L can be solved by a polynomial time algorithm A with the property:
  - If $x \in L$, then $Pr(A(x) = 1) = 1$.
  - If $x \notin L$, then $Pr(A(x) = 0) \geq 1/2$.
- **BPP** (bounded error probabalistic polynomial) = the class of decision problems L such that L has a polynomial time algorithm A with the property:
  - If $x \in L$, then $Pr(A(x) = 1) \geq 2/3$.
  - If $x \notin L$, then $Pr(A(x) = 0) \geq 2/3$.

  Notice that for problems in BPP, we can make errors for both positive and negative instances of L.

Not much is known about the relationship between P, RP, $coRP$, and BPP. However – maybe somewhat surprisingly at first glance – many theoretical computer scientists believe the following conjecture.

Conjecture 4 *$BPP = P$.*

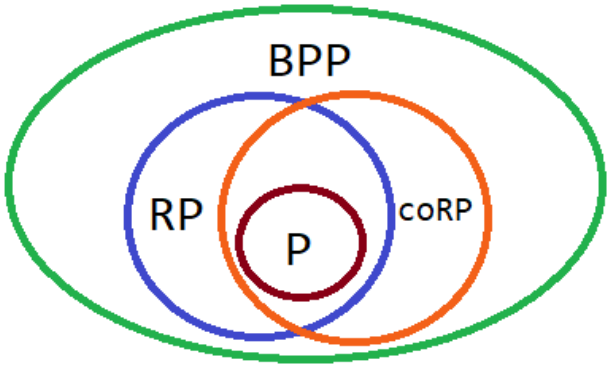

Figure 1: A venn diagram detailing various complexity classes.

The point here is that randomization is not expected to help in a "gross" sense, i.e., it will not help us solve in polynomial time a problem that cannot be solved in polynomial time by deterministic means. However, it can improve a running time that is a high degree polynomial, e.g., $O(n^8)$, to a running time that is a low degree polynomial, e.g., $O(n^2)$. This conjecture is a major open problem in theoretical computer science.

There are at least one well known problem that is not known to be in P, but is in $coRP$. This is the *Polynomial Identity Testing* problem.

5 Polynomial Identity Testing (PIT)

INPUT: multivariate polynomials $P(x_1, x_2, ..., x_m)$ and $Q(x_1, x_2, ..., x_m)$
QUESTION: is $P \equiv Q$?
For example, with $m = 3$, we may be given

$$P(x_1, x_2, x_3) = (3x_1 - 7)(4x_2 - x_3)(3x_2 - 4x_1)$$

$$Q(x_1, x_2, x_3) = 36x_1x_2^2 + 1 - x_1x_2 - 25x_1^2x_3 + 7x_1x_2x_3$$

Obviously, we can easily check whether or not the two polynomials are the same by multiplying everything out. But, multiplying out the terms could lead to a polynomial that has an exponential number (in m) of terms. It is also possible that even though the final coefficients are small, some of the intermediate numbers generated by multiplications/additions can be quite huge. These are some of the reasons why PIT does not have a (deterministic) polynomial-time algorithm yet.

5.1 "Baby version" of PIT

To simplify this problem, let's just consider the case with $m = 1$, so that we are only given single variable polynomials. Further, assume that both polynomials $P(x)$ and $Q(x)$ are given as a product of monomials (for example, $(3x-7)(9x+1)\cdots$). We also will assume that each arithmetic operation take $O(1)$ constant time.

Consider the following simple deterministic algorithm: multiply out both polynomials and express in standard form. It can be checked that the runtime of this algorithm is $O(d^2)$, where d is the degree of the polynomials P and Q.
Now we solve PIT using randomization.

```
Randomized PIT
    (1) Pick a number t uniformly at random from {1,...,100d}
    (2) Evaluate P(t), Q(t)
    (3) If P(t) = Q(t)
           return YES
        else
           return NO
```

The runtime of this algorithm is $O(d)$ because it takes $O(d)$ to evaluate $P(t)$ and $Q(t)$. Note that t can be generated by looking at $O(\log_2(100d)) = O(\log d)$ bits. If we assume that each random bit can be generated in $O(1)$ time, then Step (1) takes only $O(\log d)$ time.

To analyze the error probability of this algorithm, just look at the two possible cases.

- If $P \equiv Q$, then the algorithm returns YES with probability 1.

- If $P \not\equiv Q$, the analysis is slightly more involved. Note that $P(t) = Q(t)$ iff t is a root of $P(x) - Q(x) = 0$. Since $P(x) - Q(x)$ has degree at most d, by the Fundamental Theorem of Algebra, $P(x) - Q(x)$ has at most d roots. Then $Pr(P(t) = Q(t)) \leq \frac{d}{100d} = \frac{1}{100}$. So for $P \not\equiv Q$, the algorithm returns NO with a probability $\geq \frac{99}{100}$.

6 Independence and Conditional Probabilities

Definition 5 *Events E_1 and E_1 are independent iff $Pr(E_1 \cap E_2) = Pr(E_1)Pr(E_2)$.*

Definition 6 *Events $E_1, ..., E_k$ are mutually independent iff for any subset $I \subseteq \{1, 2, ..., k\}$, $Pr(\bigcap_{i \in I} E_i) = \prod_{i \in I} Pr(E_i)$.*

We have already used the notion of mutual independence in analyzing algorithms that amplify correctness probability by independent repititions. In some situations, requiring or expecting mutual independence is too much and a weaker notion of independence suffices.

Definition 7 *Events $E_1, ..., E_k$ exhibit p-wise independence iff for any subset $I \subseteq \{1, 2, ..., k\}$ such that $|I| \leq p$, $Pr(\bigcap_{i \in I} E_i) = \prod_{i \in I} Pr(E_i)$.*

When $p = 2$, the independence we get is called *pairwise independence*. We will encounter this later.

Definition 8 *Conditional probabilitiy:*

$$Pr(E_1|E_2) = \frac{Pr(E_1 \cap E_2)}{Pr(E_2)} \text{ if } Pr(E_2) \neq 0$$

This implies that if $Pr(E_2) \neq 0$, then $Pr(E_1 \cap E_2) = Pr(E_1|E_2) \cdot Pr(E_2)$. More generally,

$$Pr(E_1 \cap E_2 \cap \cdots \cap E_k) = Pr\left(E_1 | \bigcap_{j=2}^{k} E_j\right) \cdot Pr\left(E_2 | \bigcap_{j=3}^{k} E_j\right) \cdots Pr\left(E_{k-1}|E_k\right) \cdot Pr(E_k)$$

We will now use the above formula in the analysis of *Karger's min-cut algorithm*. There are many ways of solving the min-cut problem in polynomial time, but Karger's algorithm showcases the simplicity and elegance one gets by using randomization.

7 Karger's Min-cut Algorithm

INPUT: An undirected multigraph $G = (V, E)$
OUTPUT: A partition (S, T) of V such that the number of edges with one endpoint in S and the other in T is minimized.

In order to understand how Karger's algorithm works, we first need to understand how the *contract* operation works on a graph. This operation takes a graph G and an edge $e = \{u, v\}$ in G and outputs a new graph which "contracts" u and v into a "super-vertex" uv. If a vertex w had an edge to u and an edge to v in G, then after the contract operation w has two edges to the super-vertex uv. See Figure 3 for an illustration.

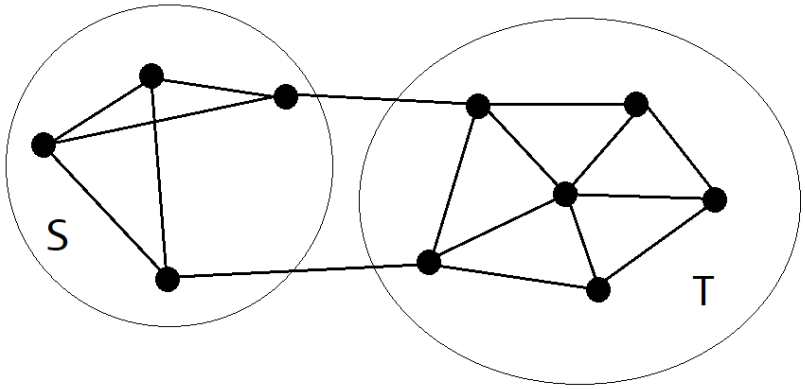

Figure 2: An example of partition which solves the mincut problem. Here the size of the mincut is 2. Notice that the size of the mincut is always $\leq$ the minimum degree, as we can always choose S to contain just one vertex.

7.1 Karger's Mincut Algorithm

```
G_0 <- G
for i <- 1 to n-2 do:
    pick an edge e_i uniformly at random from G_{i-1}
    G_i <- contract(G_{i-1}, e_i)
return number of edges between two remaining vertices
```

Note that after each contract operation, the number of vertices decreases by 1. Therefore, the final graph G_{n-2} only has two vertices. This algorithm does not always return the optimal solution, as demonstrated in Figure 4.

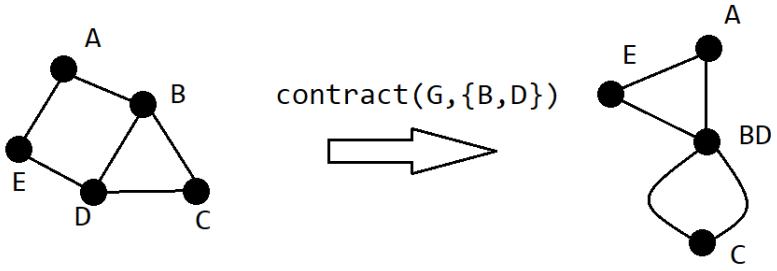

Figure 3: An example of the contract operation in action. Notice that contracting a simple graph can return a multigraph.

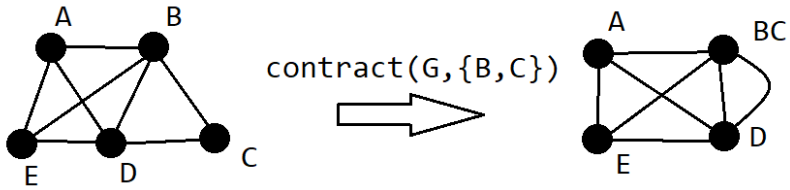

Figure 4: Karger's Mincut Algorithm is not always correct. The original graph has a mincut size equal to 2, but after one iteration, the min-cut size equals 3.