1 Overview

In this lecture, we introduce two kinds of randomized algorithms: Las Vegas and Monte Carlo algorithms.

2 Las Vegas Algorithms

Las Vegas algorithms always output the correct answer. The running time of Las Vegas algorithms is not deterministic. Here we show how Las Vegas algorithms work by an example of randomized quicksort.

2.1 Randomized Quicksort Analysis

Recall that the randomized quicksort algorithm picks a pivot at random, and then partitions the elements into three sets: all the elements less than the pivot, all elements equal to the pivot, and all elements greater than the pivot.

We analyze the runtime using charging. In the computation tree shown above, we count the number times each element appears. The sum over all elements is equal to the sum of the sizes of all the sets in the computation tree, which is proportional to the runtime.

Claim 1. The number of times an element appears in sets in the computation tree is $O(\log n)$ in expectation.

Each set $S$ in the computation tree has three children. Let $S_{=p}$ be child set with elements equal to the pivot. Let $S_{>p}$ and $S_{<p}$ be defined similarly. We color the tree edges to these children as follows:

---

1Some materials are from previous notes by Samuel Haney and Wenshun Liu for this class in Fall 2014.
• Color the edge to \( S_{=p} \) red.
• Color the edge to the larger of \( S_{>p} \) and \( S_{<p} \) black.
• Color the edge to the smaller of \( S_{>p} \) and \( S_{<p} \) blue.

Now, we trace the path of some element \( e \) through the computation tree, and count the number of edges of each color along the path.

• The number of red edges is 1. Only the last edge can be red.
• The number of blue edges is at most \( \log_2 n \), since the smaller of \( S_{>p} \) and \( S_{<p} \) has at most \( \frac{1}{2} \cdot |S| \) elements.
• The number of black edges, using the definitions we have given, could be large.

What is the probability that an edge is black? That is, given that \( e \in S \), what is the probability that the next edge in \( e \)’s path is black? Suppose \( e \) is the smallest element in the array.

For this element, any pivot in the right half of the array, illustrated with the shaded region in the image above, will cause \( e \)’s next edge to black. Therefore, the probability that the next edge is black in this case is roughly \( \frac{1}{2} \).

However, consider some element \( e \) near the center of the array.

The shaded region shows which pivots will cause the next edge in \( e \)’s path to be black. For this element, the probability that the next edge is black is nearly 1. The probability will be high for all elements near the center of the array.

We need to modify our definition of black edges and blue edges to fix the problem. If \( |S_{>p}| > \frac{3}{4} |S| \), color the edge to \( S_{>p} \) black. Otherwise, color the edge to \( S_{>p} \) blue. We color the edge to \( S_{<p} \) using identical rules. Note that now it is possible for both edges to be blue (but not both to be black). We have the following:

• The number of red edges is still 1.
• The number of blue edges is at most \( \log_{4/3} n \).

To calculate the probability of an edge being black, we split the array into four equal parts:
If the pivot is selected from the shaded quadrants, one of the edges will be blue and the other will be black. If the pivot is selected from the non-shaded quadrants, both edges will be blue. Therefore, regardless of which element $e \in S$ we pick, the probability that the next edge on $e$’s path is black is at most $\frac{1}{2}$.

Now, we can bound the number of black edges in a path. Let $X_i$ a random variable, equal to the number of black edges between the $i$th and the $(i+1)$st blue edge. As we have seen,

$$E[X_i] \leq \frac{1}{p_{success}} \leq \frac{1}{1/2} = 2.$$

By linearity of expectation, we have

$$E[\text{total number of black edges}] = E\left[\sum_i X_i\right] = \sum_i E[X_i] \leq 2 \left(\log\frac{4}{3} n\right) = O(\log n).$$

We now need to bound the total running time, $T$. Let $T_e$ be the amount of running time charged to $e$ (which is proportional to the number of sets $e$ appears in).

$$E[T] = E\left[\sum_e T_e\right] = \sum_e E[T_e] = \sum_e O(\log n) = O(n \log n).$$

### 3 Monte Carlo Algorithms

The running time of Monte Carlo algorithms is deterministic. But the output may be wrong with a small probability. Here we show how Monte Carlo algorithms work by an example of contraction algorithm that computes the global minimum cut of a connected graph.

#### 3.1 Contraction Algorithm

##### 3.1.1 Overview

Contraction Algorithm is a Monte Carlo algorithm that computes the global minimum cut of a connected graph. The algorithm has deterministic runtime, but is not guaranteed to output the correct result.

##### 3.1.2 Pseudocode

**Algorithm 1** Contraction Algorithm

1: function CONT($G$)
2: Repeat until there are two vertices left
3: contract an edge chosen uniformly at random
4: remove all self-loops, and keep all parallel edges
5: return the number of edges between the last two nodes
3.2 Correctness Analysis - Probability that the Algorithm will Succeed

As a Monte Carlo algorithm, the mincut outputted by Contraction Algorithm is not guaranteed to be correct. Thus, we are interested in the probability that the algorithm will succeed (correctly output the mincut).

Remark 1. Contraction Algorithm would output a different (and incorrect) cut when somewhere during the randomised process, an edge in the mincut is chosen and contracted.

To find out this probability, we define

$(S, \bar{S})$ to be a mincut containing $C$ edges.

$X_i$ to represent the case where no edges in $(S, \bar{S})$ is contracted in the first $i$ iteration.

(That is, the algorithm stays correct after the first $i$ iterations)

$G_i$ to be the graph after $i$ iterations.

$Pr[\text{Algorithm is successful}]$ is essentially $Pr[X_{n-2}]$, which is when none of the mincut edges is contracted in all $n - 2$ contractions. To find out this probability we take the following steps:

1. $Pr[X_{n-2}|X_{n-3}]$ represents the probability that none of the mincut edges is contracted in the $(n - 2)^{th}$ iteration given that none of the mincut edges has been contracted in the previous $n - 3$ steps.

   We have $Pr[X_{n-2}|X_{n-3}] = 1 - \frac{\#\text{edges in } (S, \bar{S})}{\#\text{edges in } G_{n-3}}$ (which is: $1 - \frac{\#\text{edges in mincut}}{\#\text{edges left in graph}}$)

   $\#\text{edges in } (S, \bar{S})$ is simply $C$, as no edges from the mincut has been contracted.

   Now we want to find out $\#\text{edges in } G_{n-3}$:

   Observation: $\#\text{vertices in } G_{n-3}$ is 3, as we lose exactly 1 vertex during each contraction.
Claim 2. Any contraction (and the removal of self loops) does not decrease the number of edges in a minimum cut.

Proof. We proof the claim by contradiction.

We assume that after the contraction, we have found a cut of length $\lambda$ in the graph smaller than all cuts before the contraction.

We know that the contracted edge $(u, v)$ cannot lie across this new cut, or this cut would not exist anymore as the contraction would merge the partitions together.

Thus, given that vertices $u$ and $v$ must lie on the same side of the cut, if we reverse the contraction, the number of edges of this cut in the original graph would indeed remain to be the same.

As this cut of length $\lambda$ exists in the graph before the contraction, the mincut of the graph before the contraction must have at most $\lambda$ edges.

Thus, we hereby proved that after the contraction, we cannot found a cut smaller than all cuts before the contraction.

Claim 2 shows that the mincut in $G_{n-3}$ has at least $C$ edges. Thus, we know that:

$$\# \text{edges incident on any vertex in } G_{n-3} \geq C.$$

Thus, given that we have 3 vertices, and each has degree of at least $C$, $\# \text{edges in } G_{n-3} = \frac{3c}{2}$

2. We now generalise the case to any iteration. At the $i^{th}$ contraction we have,

$$Pr[X_i|X_{i-1}] = 1 - \frac{c}{\# \text{edges in } G_{i-1}},$$

where

$$\# \text{edges in } G_{i-1} \geq \frac{(n-i+1)c}{2},$$

as $G_{i-1}$ has $n-i+1$ vertices, and $\# \text{edges incident on any vertex} \geq c$.

Thus, we have,

$$Pr[X_i|X_{i-1}] = 1 - \frac{c}{\frac{n-i+1}{2}c} = \frac{n-i-1}{n-i+1}$$

3. Now we have,

$$Pr[X_{n-2}] = Pr[X_{n-2}|X_{n-3}] \cdot Pr[X_{n-3}]$$

$$\geq \frac{1}{n} \cdot \frac{2}{n} \cdot \frac{3}{n-1} \cdot \frac{n-2}{n} \cdots \cdot \frac{n}{n-1} \cdot Pr[X_1]$$

$$= \frac{1}{n} \cdot \frac{2}{n} \cdot \frac{3}{n-1} \cdot \frac{n-2}{n} \cdots \cdot \frac{n}{n-1}$$

$$= \frac{1}{\Theta\left(\frac{1}{n}\right)}$$

We’ve found out the probability that the contraction algorithm would succeed.
3.3 Succeeds with High Probability

From the previous section we could observe that if the contraction algorithm is only run once, the chance that it would output the correct mincut relatively low. Thus, we wish to run the algorithm multiple times to increase the probability of success.

Suppose we run the algorithm \( k \) times, and report the smallest cut found in the \( k \) runs:

\[
Pr[\text{failure in } k \text{ runs}] = Pr[\text{failure in one run}]^k \leq (1 - \frac{1}{\binom{n}{2}})^k
\]

We recall from the previous class that, 

\[
\frac{1}{4} \leq (1 - \frac{1}{N})^N \leq \frac{1}{e} \text{ for } N \geq 2
\]

Thus, we know that \( (1 - \frac{1}{\binom{n}{2}})^k = \Theta(1) \) if \( k = \binom{n}{2} \)

\[
\begin{align*}
... &= \frac{1}{n} \text{ if } k = \binom{n}{2} \log n = O(n^2 \log n) \\
... &= \frac{1}{n^2} \text{ if } k = 4\binom{n}{2} \log n = O(n^2 \log n) \\
... &= \frac{1}{\text{poly}(n)} \text{ if } k = O(n^2 \log n)
\end{align*}
\]

Thus, if we pick \( k = O(n^2 \log n) \), we have,

\[
Pr[\text{Algorithm will succeed}] \geq 1 - \frac{1}{\text{poly}(n)} = 1 - o(1) \text{ (the "little" o)}
\]

We say an algorithm succeeds with high probability if the algorithm succeeds with probability \( \geq 1 - o(1) \).

3.4 Runtime Analysis

To succeed with high probability, we need to run the algorithm \( O(n^2 \log n) \) times. Each run would take \( O(n) \) to finish with some advanced optimisations. Thus, the algorithm would run in \( O(n^3 \log n) \).

Remark 2. The algorithm could be optimised to achieve a runtime of \( O(n^2 \log n) \).