CSC2710 Analysis of Algorithms — Fall 2022
Unit 13 - Randomized Algorithms

13.1 Randomized Algorithms

- Las Vegas algorithms are randomized algorithms that always provide the correct solution, but get there through non-deterministic means
  - We will use this method to analyze random quicksort, or `RandQS`.
- Monte Carlo algorithms are such that they do not always provide the correct solution, but a solution with a probability of being correct.
  - With repetitions of the algorithm, the "correctness" of the solution approaches 100%.
  - We will use this method to solve the MinCut problem.

13.1.1 Remember the Probability Rules

- For events ε_1 and ε_2, they are said to be *independent* if and only if $\mathbf{Pr}[\varepsilon_1 \cap \varepsilon_2] = \mathbf{Pr}\left[\varepsilon_1\right] \times \mathbf{Pr}\left[\varepsilon_2\right]$
- More generally, if ε_1 and ε_2 are not necessarily independent, $\mathbf{Pr}[\varepsilon_1 \cap \varepsilon_2] = \mathbf{Pr}\left[\varepsilon_1 | \varepsilon_2\right] \times \mathbf{Pr}\left[\varepsilon_2\right] = \mathbf{Pr}\left[\varepsilon_2 | \varepsilon_1\right] \times \mathbf{Pr}\left[\varepsilon_1\right]$
- In general: $\mathbf{Pr}[\cap_{i=1}^{n} \varepsilon_i] = \mathbf{Pr}[\varepsilon_1] \times \mathbf{Pr}\left[\varepsilon_2 | \varepsilon_1\right] \times \mathbf{Pr}\left[\varepsilon_3 | \varepsilon_1 \cap \varepsilon_2\right] ... \mathbf{Pr}\left[\varepsilon_k | \cap_{i=1}^{k-1} \varepsilon_i\right]$

13.2 Analysis of Randomized QuickSort (Las Vegas)

- Recall that the QuickSort algorithm requires determining a pivot element at each step.
  - In the variation we discussed in class, we said that this would be the first element every step.
  - However, you can randomly select this pivot. This variation we will refer to as RandQS, and it belongs to the family of *randomized algorithms*.

```
# Input: A subarray of A defined by indices 'left' and 'right'
# Output: The subarray is in sorted increasing order
def RandQS( A, left, right ):
    if( left < right ):                               # Line 1
        # Partition using random pivot
        s = RandomPartition( A[ left : right ] )      # Line 2
        # Sort the lower and upper halves separately
        RandQS( A[ left : s-1 ] )                     # Line 3
        RandQS( A[ s+1 : right ] )                    # Line 4
    return A                                          # Line 5
```

- We did not do in-depth analysis of `QuickSort` earlier in the semester[1], but now we will introduce some tools that will allow us to examine its performance.
  - We want to analyze the *expected number of comparison* during the partitioning step of the algorithm.
  - Let A_i denote the element of *rank* i, or the i^{th} smallest element of `A`, for $1 \leq i \leq n$.
  - We will use a *random variable* `X` to denote comparisons.
    - Let X_{ij} be equal to 1 if elements A_i and A_j are compared during the execution of `RandQS`, and 0 otherwise.
    - **Practice:** Can two elements ever be compared more than once?
  - Thus, a count of the *total number of comparisons* in `RandQS` is given by:
    $$\sum_{i=1}^{n} \sum_{j>i} X_{ij}$$
  - We are interested in the *expected number of comparison* for `RandQS`:
    $$E\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} E\left[X_{ij}\right]$$
    - **Practice:** What is the *expected value* of an expression?
  - Let p_{ij} denote the probability that $X_{ij} = 1$. Since X_{ij} can only be 1 or 0, $E\left[X_{ij}\right] = p_{ij} \times 1 + (1 - p_{ij}) \times 0 = p_{ij}$
    - **Practice:** Do you see why?
  - This means we can calculate the number of comparisons once we know the probability that A_i and A_j are compared during an execution of `RandQS`. How can we accomplish this?

[1] The hand-wave method!

13.2.1 Analyzing RandQS Using BST

- Treat a single execution of `RandQS` as a binary tree `T`.
  - Each node A_i of the tree represents a distinct element of `A` chosen as the pivot for one partitioning.
  - The node's left subtree are the elements less than A_i (partitioned to the left) and the right subtree are the elements greater than A_i, so the result of the partitioning.
    - This is a binary search tree.
    - A_i is compared with every element of each of its subtrees, but elements of each subtree are not compared with each other at the i^{th} iteration of the algorithm.
      - **Practice:** Do you see why?
    - This implies that elements A_i and A_j are compared if and only if one is an ancestor of the other.
  - Let π be the *level-order traversal* of `T`, or the order of the chosen pivots. We can make the following observations:
    1. A_i and A_j are compared if and only if A_i or A_j occurs earlier in π than *any* element A_k such that $i < k < j$.
       - Otherwise, elements A_i and A_j will be in opposite subtrees. However, if either element A_i or A_j is selected first, then they are compared.
       - So we need to know the probability that A_i or A_j is selected first of all the elements between A_i and A_j
    2. Any of the elements $A_i, A_{i+1}, ..., A_j$ is *equally likely* to be the first chosen to be a pivot, and therefore the first to appear in the squence π. Thus, the probability of the first being chosen being either A_i or A_j is $\frac{2}{j-i+1}$.
  - Thus, $p_{ij} = \frac{2}{j-i+1}$
    $$\begin{aligned}\sum_{i=1}^{n}\sum_{j>i} p_{ij} &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1} \\ &\leq \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k} \\ &\leq 2\sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{1}{k} \\ &\leq 2nH_k\end{aligned}$$
  - $H_k \approx \ln n + \Theta(1)$
  - Therefore, `RandQS` $\in O(n \ln n)$

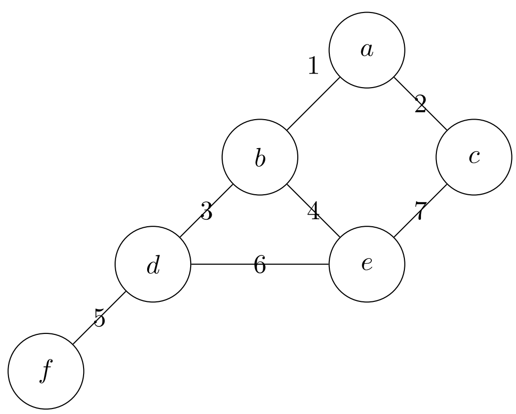
13.3 Analysis of Min-Cut (Monte Carlo)

13.3.1 RandMinCut Algorithm

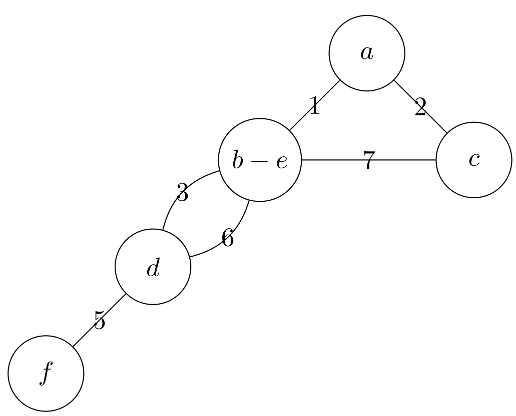
- Let `G` be a connected, undirected multigraph (with multiple edges allowed between pairs of vertices).
- A *cut* in the graph is a set of edges whose removals from `G` result in `G` being broken into two or more components.
- The MinCut is the set of such edges with smallest cardinality.
  - The following algorithm is a *randomized algorithm* for finding the minimum cut:
    1. Repeat the following until only two vertices remain:
       * Remove an edge `e` uniformly at random
       * Collapse (*contract*) the two vertex endpoints of `e`
         · All extra edges between the vertices collapsed are removed
         · All other edges are retained
    2. The edges that remain are a cut of G. But is it the minimum cut?

13.3.2 Example Walkthrough

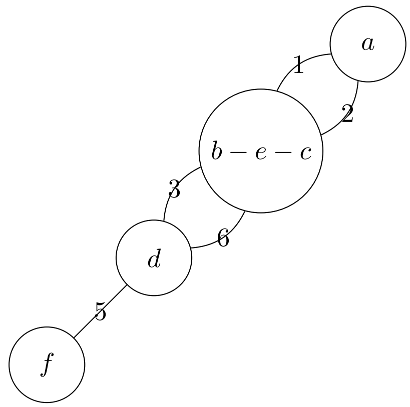
1. Begin with the following graph



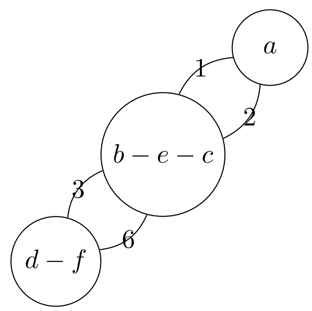
2. Select edge (b, e)



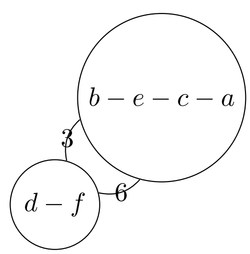
3. Now we selected edge $(b-e, c)$



4. Remove edge (f, d)



5. Finally, collape $(b-e-c, a)$



6. The minimum cut of the graph `G` is $\{3, 6\}$

   - This is clearly not the real minimum cut for `G`.
   - **Practice:** What is the running time of this algorithm?

13.3.3 Analysis of RandMinCut

- Let k be the size of the actual minimum cut, and let `C` be a cut of size k of a graph `G` (whose number of vertices is at least $\frac{kn}{2}$).
- We will bound from below the probability that no edge of `C` is ever removed during the execution of the algorithm, so that the set of edges left is a minimum cut.

1. Let ε_i denote the event that the i^{th} step of the algorithm does *not* remove an edge of `C`, for $1 \le i \le n-2$. Note that n dictates the number of steps of the algorithm.
2. The probability that an edge of `C` randomly chosen in the first step is *at most* $\dfrac{k}{\left(\dfrac{nk}{2}\right)} = \dfrac{2}{n}$.
   - This implies $\mathbf{Pr}[\varepsilon_1] = 1 - \dfrac{2}{n}$
3. The probability that the edge randomly chosen in the second step is *at most* $\dfrac{2}{n-1}$.
   - Therefore, $\mathbf{Pr}[\varepsilon_2|\varepsilon_1] \ge 1 - \dfrac{2}{n-1}$
4. At the i^{th} step, there are $n-i+1$ vertices remaining, and the size of the minimum cut is still k, so the graph has at least $\dfrac{k\,(n-i+1)}{2}$ edges remaining.
5. Thus,
   $\mathbf{Pr}\left[\varepsilon_i | \cap_{j=1}^{i-1} \varepsilon_j\right] \ge 1 - \dfrac{2}{n-i+1}$
6. What's the probability of never picking an edge of `C`?
   - From our equation above, we have that
     $\mathbf{Pr}[\cap_{i=1}^{n} \varepsilon_i] = \mathbf{Pr}[\varepsilon_1] \times \mathbf{Pr}\ [\varepsilon_2|\varepsilon_1] \times \mathbf{Pr}\ [\varepsilon_3|\varepsilon_1 \cap \varepsilon_2] ... \mathbf{Pr}\ \left[\varepsilon_k | \cap_{i=1}^{k-1} \varepsilon_i\right]$
   - This implies that
     $\mathbf{Pr}\left[\cap_{i=1}^{n-2} \varepsilon_i\right] \ge \displaystyle\prod_{i=1}^{n-1} \left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n\,(n-1)}$
7. Therefore, the probability of discovering a minimum cut is at least $\dfrac{2}{n^2}$, which means we may not (and probably won't) find the minimum cut for `G`.

- Suppose we performed the same algorithm $\frac{n^2}{2}$ times, each independent.
  - Recall:
    $\mathbf{Pr}[\varepsilon_1 \cap \varepsilon_2] = \mathbf{Pr}\ [\varepsilon_1] \times \mathbf{Pr}\ [\varepsilon_2]$
  - Therefore, the probability of *not* finding a a minimum cut with $\frac{n^2}{2}$ runs of the algorithm, we get
    $\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{\epsilon}$
  - Further executions of the algorithm allow for *arbitrarily small* failure probabilities [2].
    * Instead of $\frac{n^2}{2}$ executions, pick an arbitrarilty small failure change (ϵ) and solve for the number of executions necessary.
    * For example, for 0.0001 failure chance, make episode 10000:
      $\left(1 - \frac{2}{n^2}\right)^X < \frac{1}{10000}$
      $X \lg\left(1 - \frac{2}{n^2}\right) \leq \lg\left(\frac{1}{10000}\right)$
      $X \geq \frac{-\lg(10000)}{\lg\left(1 - \frac{2}{n^2}\right)}$

[2] This is the essence of Monte Carlo algorithms

13.4 Challenge Problem - Analysis of Vector Ordering

- We are posed the following question: Given an array A of length n of integers, is A in sorted order?
- We want a *sublinear time* algorithm for determining, with some failure probability, whether `A` is sorted.
- To accomplish this, we will use the following randomized algorithm:
  - Select a pair of elements at random
  - If they are in sorted order, return `true`. Otherwise, return `false`.
- Analyze the algorithm by answering the following questions:
  1. What is the total number of pairs from which to choose? Assume that two unique elements are chosen, so an element can't be paired with itself.
  2. Assume that m is the total number of *unsorted pairs* in A. That is, given the list of all of the pairs you may choose from, m is the total number of them that are not in sorted order.
     - What does it mean when $m = 0$?
     - What does it mean when $m = 1$?
  3. Let X_i be the i^{th} pair chosen, and let X be the total number of unsorted pairs found after n random pairs are drawn.
     - $X_i = 0$ when the i^{th} pair chosen is not sorted.
       * In terms of n and m, what is the probability $X_i = 0$?
     - $X_i = 1$ when the i^{th} pair chosen is sorted.
       * In terms of n and m, what is the probability $X_i = 1$?
     - What is the expected value of X?
  4. What is an expression for the number of trials of the algorithm necessary to meet arbitrary success probability $\frac{1}{\epsilon}$, provided a value for m? In otherwords, finish the phrase:
     - "If A has 5 unsorted pairs, I would need to select `<some number>` pairs to have a probability of $\frac{1}{\epsilon}$ of finding them."