

14.1 Randomized Algorithms

This section introduces the notion of a randomized algorithm and presents a randomized version of **Quicksort**.

14.1.1 Quicksort

Quicksort is a sorting algorithm that is based on the divide and conquer paradigm. To sort an array of elements, we pick one element to be the **pivot**. We then split the original array into two smaller arrays, one containing elements that are less than the pivot, and the other containing elements that are greater than the pivot. We then sort two arrays recursively, and recombine the results.

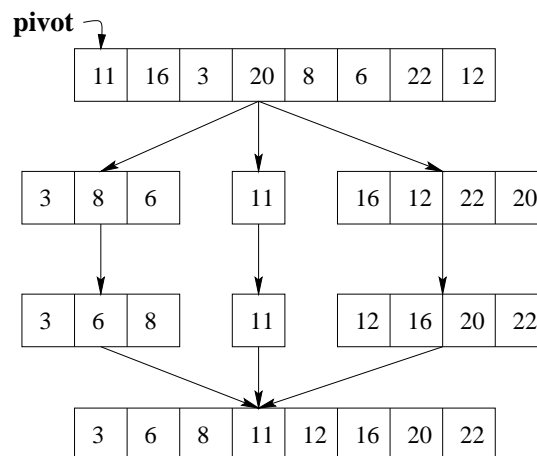


Figure 14.1: An example of Quicksort

QUICKSORT(X)

```

1  if  $|X| = 1$  or 0 then
2    halt
3  else
4    choose  $x^*$  from  $X$  (pivot)
5    compare each element of  $X - x^*$  with  $x^*$ 
6    let  $L = \{x \in X \mid x < x^*\}$ 
7     $U = \{x \in X \mid x > x^*\}$ 
  
```

```

8     QUICKSORT (L)
9     QUICKSORT (U)
10  endif

```

The running time of this algorithm depends on how we choose the pivot element. One approach to do that is to pick it randomly.

Definition 14.1 *Average runtime:* $\bar{T}(n) = \max_{|x|=n} E[T(x)]$, where E is the expected value with respect to random choices made by the algorithm.

Theorem 14.2 *For Quicksort, $\bar{T}(n) = \Theta(n \log n)$. Specifically, for $n \geq 2$, $\bar{T}(n) \leq an \log n$, where a is some constant.*

Proof: As is usual for sorting algorithms, we measure the running time of QUICKSORT in terms of the number of comparisons it performs, that is: $\Theta(\text{number of pairwise comparisons})$.

We consider what is the probability that the i th smallest element is compared with the j th smallest element. Assume ($i < j$): a comparison occurs iff one of these two is chosen as the pivot. That is, in general, once a pivot x is chosen with x between i th smallest and j th smallest, we know that i th smallest and j th smallest cannot be compared at any subsequent time. If, on the other hand, the i th smallest is chosen as a pivot before any other element between the i th smallest and j th smallest, then the i th smallest will be compared to each element in the range from i th smallest to j th smallest, except for itself. A similar argument applies if the j th smallest is chosen as the pivot.

There are $j - i + 1$ elements between the i th smallest and j th smallest, and the chances for these elements to be chosen as the pivot are the same, so the probability that the i th smallest is compared with j th smallest is $\frac{2}{j-i+1}$.

We use an indicator random variable:

$$Z_{ij} = \begin{cases} 1 & \text{if } i\text{th smallest element is compared with the } j\text{th smallest} \\ 0 & \text{otherwise} \end{cases}$$

Thus, Z_{ij} is a count of comparisons between the i th smallest and j th smallest elements. So the total number of comparisons is:

$$Z = \sum_{1 \leq i < j \leq n} Z_{ij}$$

$$Pr[Z_{ij} = 1] = \frac{2}{j - i + 1}$$

$$E[Z_{ij}] = \frac{2}{j - i + 1}$$

$$E[Z] = E\left[\sum_{1 \leq i < j \leq n} Z_{ij}\right] = \sum_{1 \leq i < j \leq n} E[Z_{ij}]$$

This equation uses an important property of expectations called **linearity of expectation**.

$$E \left[\sum_{1 \leq i < j \leq n} Z_{ij} \right] = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-i+1} \leq 2 \sum_{i=1}^{n-1} H_n \leq 2n \ln n$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where H_n is the n th **Harmonic number**, defined by:

$$H_n = \sum_{k=1}^n \frac{1}{k}$$

We have that $H_k \approx \ln k$, so the expected running time of Quicksort is $\Theta(n \ln n)$. ■

The probability that it differs more than some $\varepsilon \cdot E[c(n)] < n^{-\varepsilon(\ln \ln n)}$

$$Pr[| c(n) - E[c(n)] | > \varepsilon \cdot E[c(n)]] < n^{-\varepsilon(\ln \ln n)}$$

where $c(n)$ denotes number of comparisons performed on an input of size n . Shown by Mcdiarmid and Hayward[MH '92]

14.2 “Las Vegas” and “Monte Carlo” Algorithms

The randomized sorting algorithm and the min-cut algorithm exemplify two different types of randomized algorithms. The sorting algorithm always gives the correct solution. The only variation from one run to another is its running time, whose distribution we study. We call such an algorithm a *Las Vegas algorithm*.

In contrast, the min-cut algorithm may sometimes produce a solution that is incorrect. However, we are able to bound the probability of such an incorrect solution. We call such an algorithm a *Monte Carlo algorithm*. Such an algorithm is useful if the probability of producing a correct result is “large enough.”

14.3 Min-Cut Problem

Input: A connected, undirected graph $G = (V, E)$.

Output: A minimum cut in G , i.e., a minimal set of edges whose removal breaks G into two or more components.

14.3.1 Simple Algorithm

A simple algorithm for the Min-Cut problem is to find a max-flow between every pair of vertices (s, t) in the graph. For each pair of vertices, find the minimum s - t cut of the corresponding max-flow. Then return

the minimum of these $s-t$ cuts. Obviously this algorithm has a running time of $O(|V|^5)$ since finding the max-flow takes time $O(|V|^3)$ (using the Karzanov algorithm [K74]), and there are $O(|V|^2)$ pairs of vertices.

An improvement can be made by fixing a single source s . Since in the final min-cut, this particular source has to be on one side of the cut, it is sufficient to find the minimum $s-t$ cut for a single source. This way the algorithm runs in time $O(|V|^4)$.

14.3.2 Karger's Algorithm

1. **while** the number of vertices > 2
2. select an edge $e = (u, v)$ uniformly random;
3. merge u and v into a single vertex, preserve all edges in the graph except those between u and v ;
4. **return** the remaining edges of the resulting graph.

Figure 14.2 shows an example of how this algorithm works. The input graph contains 10 edges labeled from a to k . Suppose at first edge h is picked, randomly, to be eliminated. The two vertices merge together into a single vertex. Now edge e and f have the same vertices, and so do edges i and j . Next edge c is chosen, and then edge e . Note that when merging the vertices of edge e , edge f also disappears. This process continues until two vertices are left. At this point, the remaining edges are b and k .

Luckily, these two edges indeed form a minimum cut of the original graph. In fact, the only two minimum cuts of this graph are $\{a, c\}$ and $\{b, k\}$. If at any two steps, one edge from the $\{a, c\}$ cut and one edge from the $\{b, k\}$ cut are chosen to be eliminated, then the result of the Karger's algorithm would be incorrect. Although we cannot prevent this from happening, we can guarantee a certain probability that a correct cut being returned by Karger's algorithm, as seen in the following theorem:

Theorem 14.3 *Let C be a minimum cut of graph $G = (V, E)$, then*

$$Pr[C \text{ returned by Karger's algorithm}] \geq \frac{2}{n^2},$$

where $n = |V|$

Proof: Let k be the number of edges in C , then every vertex in G must have degree at least k . This is so because otherwise, if some vertex v has degree less than k , then all of the edges of v form a cut which is smaller than C .

Since there are n vertices in G , there are at least $nk/2$ edges in G . Given this, we have

$$Pr[\text{first edge chosen randomly} \in C] \leq \frac{k}{nk/2} = \frac{2}{n}.$$

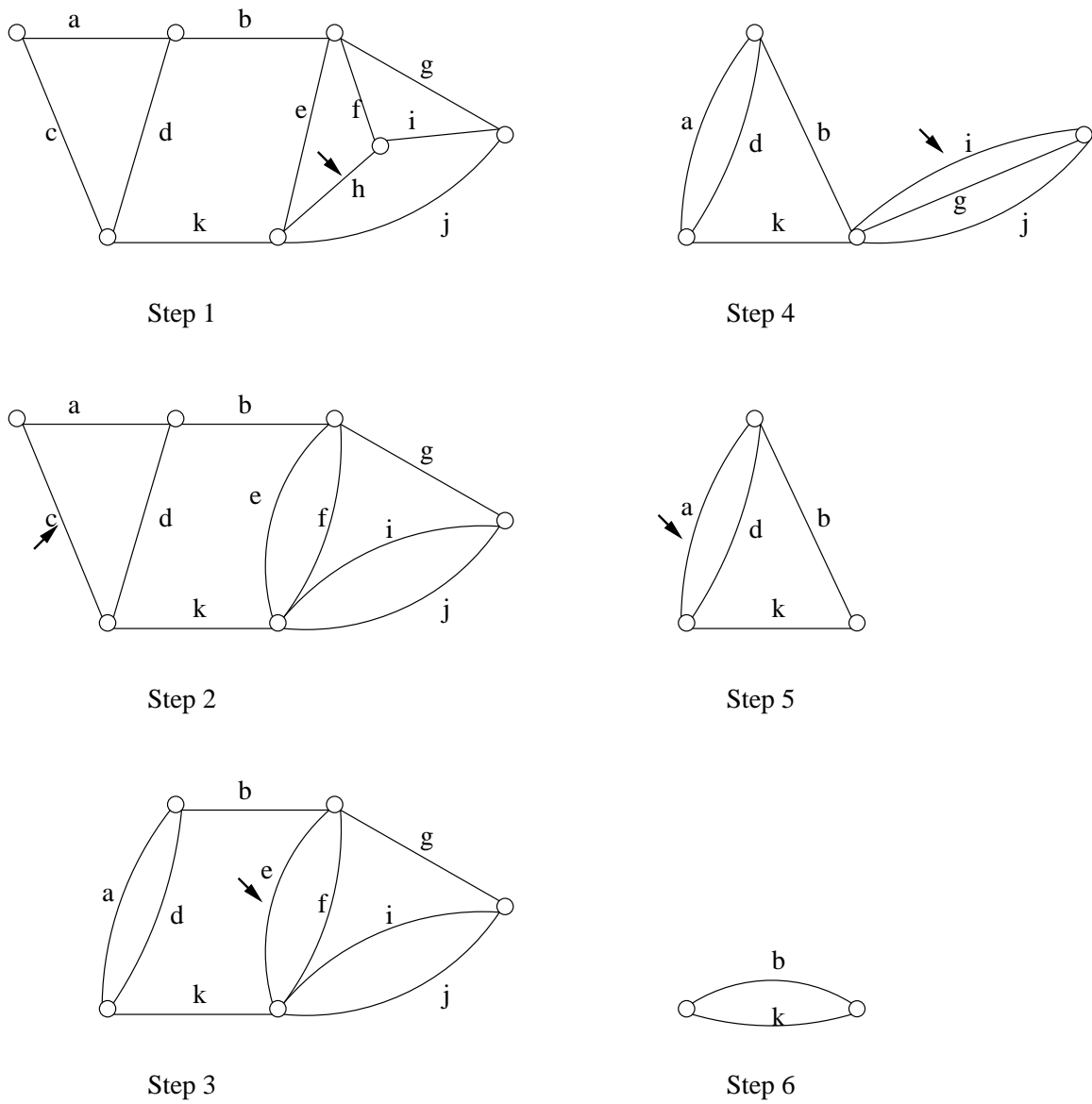


Figure 14.2: Karger's Algorithm

Suppose at some step of the algorithm, there are l vertices left. We call the graph at this step G_l . Since a cut in G_l is necessarily a cut in G , the size of the minimum cut in G_l is at least k . Following the same argument, there are at least $kl/2$ edges in G_l . We say C is "hit" if some edge in C is randomly chosen. Thus:

$$Pr[C \text{ is hit when there are } l \text{ vertices left} \mid C \text{ is not hit before}] \leq \frac{k}{kl/2} = \frac{2}{l}.$$

Then, we have

$$H_l : \text{Event } C \text{ not hit in } G_l$$

and because

$$Pr[A \cap B] = Pr[A \mid B] \cdot Pr[B]$$

where A and B are not necessarily independent and $Pr[A | B]$ denotes conditional probability of A given B.

we get:

$$\begin{aligned}
 Pr[C \text{ is returned by algorithm}] &= Pr\left[\bigcap_{l=3}^n H_l\right] \\
 &= \prod_{l=3}^n Pr[H_l | H_{l-1}] \\
 &= \prod_{l=3}^n Pr[C \text{ is not hit where there are } l \text{ vertices left} | C \text{ is not hit before}] \\
 &\geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{4}\right) \left(1 - \frac{2}{3}\right) \\
 &= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{2}{4} \cdot \frac{1}{3} \\
 &= \frac{2 \cdot 1}{n(n-1)} \geq \frac{2}{n^2}
 \end{aligned}$$

■

References

- 1 DIMITRI LISIN and SERGEI KOVALENKO, Scribe Notes for Lecture12, UMASS CMPSCI 611 *Advanced Algorithms*, Fall2000.
- 2 ZHENGZHU FENG, Scribe Notes for Lecture13, UMASS CMPSCI 611 *Advanced Algorithms*, Fall2000.
- 3 MOTWANI AND RAGHAVAN, *Randomized Algorithms*, Cambridge University Press 1995.
- 4 CORMEN, LEISERSON AND RIVEST, *Introduction to Algorithms*, MIT Press 1990.