

4. Randomized algorithms

16

References

- Juraj Hromkovič, Algorithms for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics, 2nd Edition, Springer 2003, Chapter 5.
- Jon Kleinberg, Éva Tardos, Algorithm Design, Addison-Wesley 2006, Chapter 13.
- Michael Mitzenmacher, Eli Upfal, Probability and Computing: Randomized Algorithms and Probabilistic Analysis, Cambridge University Press 2005.
- Rajeev Motwani, Prabhakar Raghavan, Randomized Algorithms, Cambridge University Press 1995.
- Heiko Röglin, Melanie Schmidt, Randomized Algorithms and Probabilistic Analysis, Lecture Notes 2 17.
- Norbert Blel, Algorithmen und Datenstrukturen
Algorithms which use random numbers are called randomized or probabilistic. There are different reasons for the use of random numbers. One reason is speeding up the expected run time of the algorithm without affecting the correctness

of the output. Quicksort is a well known example. Such an algorithm is called a Las Vegas algorithm. There are randomized algorithms which use random numbers to improve the run time but may sometimes produce an incorrect output. However, the probability of such an incorrect output can be bounded. Primality testing is a well known example. We call such an algorithm Monte Carlo algorithm.

4.1 Events and probability

We shall review some basic concepts of probability theory needed in the subsequence.

Example 4.1

random experiment: "roll a dice"

The outcome is a number in $\Omega = \{1, 2, 3, 4, 5, 6\}$

◇

The sample space Ω of a random experiment is a finite or countable set of elementary events. An event A is a subset of Ω . The complementary event of A is $\bar{A} := \Omega \setminus A$.

Example 4.1 (continuation)

- 6,
- prime number in Ω ,
- a number larger than two.

◇

The set of all events with respect to Ω is the power set 2^Ω of Ω . Note that $(2^\Omega, \cap, \cup, \bar{\cdot})$ is a Boolean lattice.

Instead of the consideration of the whole power set 2^Ω it is often sufficient to consider set systems $\mathcal{F} \subseteq 2^\Omega$ which have certain properties.

A set system $\mathcal{F} \subseteq 2^\Omega$ is called σ -algebra in Ω if the following properties are fulfilled:

(P1) $\emptyset \in \mathcal{F}$ and $\Omega \in \mathcal{F}$.

(P2) For all finite or countable sets I of indices there holds

$$A_i \in \mathcal{F} \quad \forall i \in I \Rightarrow$$

$$\bigcup_{i \in I} A_i \in \mathcal{F} \quad \text{and} \quad \bigcap_{i \in I} A_i \in \mathcal{F}.$$

(P3) $A, B \in \mathcal{F} \Rightarrow A \setminus B \in \mathcal{F}$.

Each σ -algebra is a Boolean lattice.

Let $\mathcal{F}$ be a σ -algebra. A function $\Pr: \mathcal{F} \rightarrow [0, 1]$ is called a probability measure on Ω if the following axioms are fulfilled:

(K1) $\Pr(\Omega) = 1$.

(K2) For all finite or countable sets I of indices and pairwise disjoint $A_i, i \in I$ hold

$$\Pr\left(\bigcup_{i \in I} A_i\right) = \sum_{i \in I} \Pr(A_i).$$

(K1) and (K2) are called Kolmogorov - axioms. (16)

Note that $\Pr(\emptyset) = 0$ because

$$\Pr(\Omega) = \Pr(\Omega \cup \emptyset) = 1 + \Pr(\emptyset) = 1.$$

In the same way we can show that

$$\Pr(A) = \sum_{a \in A} \Pr(a).$$

This means that the probability of an event A is equal the sum of the probabilities of the elementary events in A .

Example 4.1 (continuation)

$$\Pr(6) = \frac{1}{6},$$

$$\Pr(\text{prime number}) = \frac{1}{2}$$

$$\Pr(\text{larger than two}) = \frac{2}{3}.$$

◆

We have assigned the same probability to each number of the dice. Hence, the elementary events occur uniformly at random.

A simple consequence of the Kolmogorov-axioms is the following Lemma.

Lemma 4.1

For any two events A_1 and A_2 there hold

$$\Pr(A_1 \cup A_2) = \Pr(A_1) + \Pr(A_2) - \Pr(A_1 \cap A_2).$$

Proof:

(K2) $\Rightarrow$

$$i) \Pr(A_1) = \Pr(A_1 \setminus (A_1 \cap A_2)) + \Pr(A_1 \cap A_2),$$

$$ii) \Pr(A_2) = \Pr(A_2 \setminus (A_1 \cap A_2)) + \Pr(A_1 \cap A_2) \text{ and}$$

$$iii) \Pr(A_1 \cup A_2) = \Pr(A_1 \setminus (A_1 \cap A_2)) + \Pr(A_2 \setminus (A_1 \cap A_2)) + \Pr(A_1 \cap A_2).$$

$\Rightarrow$

$$\begin{aligned} \Pr(A_1 \cup A_2) &= \Pr(A_1) - \Pr(A_1 \cap A_2) \\ &\quad + \Pr(A_2) - \Pr(A_1 \cap A_2) \\ &\quad + \Pr(A_1 \cap A_2) \\ &= \Pr(A_1) + \Pr(A_2) - \Pr(A_1 \cap A_2). \end{aligned}$$

■

Lemma 4.1 yields the following observations:

1) The probability of $A_1 \cup A_2$ can be smaller than $\Pr(A_1) + \Pr(A_2)$. We obtain equality iff A_1 and A_2 are disjoint.

2) Always there holds

$$\Pr(A_1 \cup A_2) \leq \Pr(A_1) + \Pr(A_2).$$

These observations can be generalized as follows.

Lemma 4.2 (Union Bound)

For all finite or countable sets I of indices and events $A_i, i \in I$ the following holds

$$\Pr\left(\bigcup_{i \in I} A_i\right) \leq \sum_{i \in I} \Pr(A_i).$$

Proof:

exercise



Note that the events $A_i, i \in I$ must not be mutually disjoint.

By an application of the so-called inclusion-exclusion principle, we can generalize Lemma 4.1 to an arbitrary finite number of events.

-
- Interlude: The principle of inclusion and exclusion.

Reference:

Stasys Jukna, Extremal Combinatorics
With Applications in Computer Science,
Springer, 2001, Ch. 3. (or 2nd Ed. Ch. 1.6)

Assume that we wish to count the number of elements in the union of two sets A and B . Then we obtain

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Given n subsets $A_1, A_2, \dots, A_n$ of a set X , we wish to calculate $\left| \bigcup_{i=1}^n A_i \right|$.

Obviously

$$\left| \bigcup_{i=1}^n A_i \right| \leq \sum_{i=1}^n |A_i|.$$

This upper bound is tight, i.e., we have equality, iff the sets are pairwise disjoint. Otherwise, the upper bound is too large since some elements are counted more than once. If $A_i \cap A_j \neq \emptyset$ then each element in $A_i \cap A_j$ is counted twice, once in $|A_i|$ and once in $|A_j|$.

We can try to correct the upper bound by subtracting from the upper bound $\sum_{1 \leq i < j \leq n} |A_i \cap A_j|$ obtaining the bound

$$(*) \quad \sum_{i=1}^n |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j|.$$

But now, the bound in $(*)$ could be too small since each element in $A_i \cap A_j \cap A_k$ is subtracted three times, once in $|A_i \cap A_j|$, once in $|A_j \cap A_k|$ and once in $|A_i \cap A_k|$.

We can try to correct the bound by adding the sum $\sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k|$ obtaining the bound

$$(**) \sum_{i=1}^n |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j| + \sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| \quad (170)$$

But again, the bound could be too large, etc.

Nevertheless, it appears that after n steps, we shall get the correct result.

Notation:

Let I be a subset of the set $\{1, 2, \dots, n\}$ of indices. Then we define

$$A_\emptyset := X \quad \text{and for } I \neq \emptyset$$

$$A_I := \bigcap_{i \in I} A_i.$$

Theorem 4.1 (Inclusion-Exclusion Principle)

Let $A_1, A_2, \dots, A_n$ be subsets of a set X . Then the number of elements of X which are in none of the subsets A_i , $i \in \{1, 2, \dots, n\}$ is

$$\sum_{I \subseteq \{1, 2, \dots, n\}} (-1)^{|I|} |A_I|.$$

Proof:

The sum is a linear combination of cardinalities of sets A_I with coefficients $+1$ and -1 .

We can rewrite this sum in the following way:

$$\begin{aligned} \sum_I (-1)^{|I|} \cdot |A_I| &= \sum_I \sum_{y \in A_I} (-1)^{|I|} \\ &= \sum_y \sum_{I: y \in A_I} (-1)^{|I|} \end{aligned}$$

We shall calculate for each element $y \in X$ its contribution to this sum. This is the sum of the coefficients of the sets A_I which contain y . We distinguish two cases:

Case 1: $y \notin \bigcup_{i=1}^n A_i$.

Then the only term in the sum to which y contributes is that with $I = \emptyset$.

$\Rightarrow$ Total contribution of y is 1.

Case 2: $y \in \bigcup_{i=1}^n A_i$.

Let $J := \{i \mid y \in A_i\}$.

Then

$$y \in A_I \Leftrightarrow I \subseteq J.$$

Hence, the contribution of y is

$$\sum_{I \subseteq J} (-1)^{|I|} = \sum_{i=0}^{|J|} \binom{|J|}{i} (-1)^i = (1-1)^{|J|} = 0$$

$\uparrow$
binomial Theorem

$\Rightarrow$ Total contribution of y is 0.

For some applications, the following form of the inclusion-exclusion principle is more convenient.

Theorem 4.2

Let $A_1, A_2, \dots, A_n$ be a sequence of (not necessarily distinct) subsets of X . Then

$$|A_1 \cup A_2 \cup \dots \cup A_n| = \sum_{\emptyset \neq I \subseteq \{1, 2, \dots, n\}} (-1)^{|I|+1} |A_I|.$$

Proof:

Note that

$$\begin{aligned} |A_1 \cup A_2 \cup \dots \cup A_n| &= |A_\emptyset| - |\{x \in X \mid x \notin \bigcup_{i=1}^n A_i\}| \\ &\stackrel{\text{Th. 4.1}}{=} |A_\emptyset| - \sum_{I \subseteq \{1, \dots, n\}} (-1)^{|I|} |A_I| \\ &= \sum_{\emptyset \neq I \subseteq \{1, 2, \dots, n\}} (-1)^{|I|+1} |A_I|. \end{aligned}$$

End of interlude.

Now we can generalize Lemma 4.1 using the inclusion-exclusion principle.

Lemma 4.3

Let $A_1, A_2, \dots, A_n$ be any n events. Then

$$\Pr\left(\bigcup_{i=1}^n A_i\right) = \sum_{\emptyset \neq I \subseteq \{1, 2, \dots, n\}} (-1)^{|I|+1} \Pr(A_I)$$

where $A_I := \bigcap_{i \in I} A_i$.

Proof:

exercise



Next we shall introduce the notion of independence.

Two events A_1 and A_2 are independent iff

$$\Pr(A_1 \cap A_2) = \Pr(A_1) \cdot \Pr(A_2).$$

More generally, n events $A_1, A_2, \dots, A_n$ are mutually independent iff for all subsets $I \subseteq \{1, 2, \dots, n\}$

$$\Pr\left(\bigcap_{i \in I} A_i\right) = \prod_{i \in I} \Pr(A_i).$$

Now we shall define conditional probability

The conditional probability that an event A occurs given that event B occurs is

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}.$$

The conditional probability is well-defined only if $\Pr(B) > 0$.

(174)

The definition of conditional probability is consistent with the notion of independent events.
If $A, B \in 2^{\Omega}$ with $\Pr(B) > 0$ are independent then

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)} = \frac{\Pr(A) \cdot \Pr(B)}{\Pr(B)} = \Pr(A).$$

Since the events A and B are independent, the occurrence of B has no influence to the probability of the event A . Hence, the conditional probability $\Pr(A|B)$ has to be $\Pr(A)$.

4.2 A first example: Randomized Quicksort

In the worst case, the run time of randomized quicksort is quadratic. But the expected run time will be $\Theta(n \log n)$ where n is the number of integers which have to be sorted. Why we shall consider randomized quicksort although we know mergesort which does not use randomization and has the worst case run time $\Theta(n \log n)$. In comparison to other sorting algorithms, quicksort has the following advantages:

- 1) It is easy to implement.
- 2) Only a small additional storage is necessary.
- 3) A carefully optimized version of quicksort has on most computers a better run time than each other sorting algorithm.

For example, mergesort needs an additional storage of size $\Omega(n)$ to sort n numbers if one is not willing to do a considerable extra effort.

Exercise:

Guess to reduce the additional storage needed by mergesort.

The idea of quicksort is simple. Assume that we intend to sort a set $S = \{s_1, s_2, \dots, s_n\}$ of n numbers. If we could $s_k \in S$ such that the half of the numbers in S are smaller or equal than s_k and the half of the numbers in S are larger than s_k then we could sort S in the following way:

- (1) Partition $S \setminus \{s_k\}$ into two sets S_1 and S_2 such that S_1 contains exactly the numbers in $S \setminus \{s_k\}$ which are smaller or equal s_k and S_2 contains exactly the other numbers.
- (2) Apply the method to S_1 and to S_2 .

Step (1) can be performed using exactly $n-1$ comparisons. If s_k could be in linear time; i.e., in time cn for a constant c then we obtain for the total time $T(n)$ used by the algorithm

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + (c+1)n \\ &= O(n \log n) \end{aligned}$$

where the constant hidden in $O(\cdot)$ is larger

(17)

then $c+1$. Indeed, such a linear time algorithm exists but the constant c is too large to improve the expected run time in comparison to the run time of quicksort.

Observation:

If we could determine s_k such that none of S_1 and S_2 has more than $3/4 \cdot n$ then the expected run time would also be $O(n \log n)$. But such an s_k can also not be easily determined.

~>

Idea:

Choose s_k randomly!

The resulting algorithm is randomized quicksort.

Implementation of quicksort:

Assume that an array $S[1:n]$ contains the n numbers which have to be sorted and that $S[k]$ contains that element which is used for the partition of S into two sets S_1 and S_2 . We use two pointers i and t for the inspection of the array $S[1], \dots, S[k-1], S[k+1], \dots, S[n]$ as follows:

- i checks $S[1], S[2], \dots, S[n]$ from left to right until $S[i]$ with $S[i] > S[k]$ is found.

- t checks $S[1], S[2], \dots, S[n]$ from right to left until $S[t]$ with $S[t] \leq S[i]$ is found
- If $t > i$ then exchange the contents of $S[i]$ and $S[t]$. Repeat the whole for $S[i+1], \dots, S[t-1]$.
- If $t < i$ then there holds $t = i-1$. Exchange the contents of $S[i]$ and $S[i-1]$. The effect of this exchange is that the element which is used for the partition has its correct position in the sorted list.

Apply the method recursively to $S[1], S[2], \dots, S[i-2]$ and to $S[i], S[i+1], \dots, S[n]$.

We do not need a test for the index exceeding of the pointers i and t if we use two additional elements $S[0] = -\infty$ and $S[n+1] = \infty$.

Let $QS(n)$ denote the maximal number of comparisons used by quicksort for sorting n numbers. Then we obtain

$$QS(0) = QS(1) = 0 \quad \text{and}$$

$$QS(n) \leq n+1 + \max \{ QS(k-1) + QS(n-k) \mid 1 \leq k \leq n \}.$$

For $n \geq 1$, we can show by induction

$$QS(n) \leq \frac{(n+1)(n+2)}{2} - 3.$$

Exercise

Work out the analysis of $QS(n)$. Show that the obtained upper bound is tight.

Now we shall analyse the expected number of comparisons performed by randomized quick-sort. W. l. o. g., we can assume that we sort n distinct numbers which are the numbers $1, 2, \dots, n$.

Exercise

Show that the result of the analysis would not be worse if the assumption above does not hold.

Each position of the array is chosen with the same probability to contain s_k .

$\Rightarrow$

For $1 \leq t \leq n$, $s_k = t$ with probability $\frac{1}{n}$.

If $s_k = t$ then we obtain after the partition subproblems of sizes $t-1$ and $n-t$.

Let $QS_e(n)$ denote the expected number of comparisons used for an input of n elements.

We obtain the following recurrence equation:

$$QS_e(0) = QS_e(1) = 0$$

$$\begin{aligned} QS_e(n) &= \frac{1}{n} \sum_{t=1}^n (n+1 + QS_e(t-1) + QS_e(n-t)) \\ &= n+1 + \frac{2}{n} \sum_{t=0}^{n-1} QS_e(t). \end{aligned}$$

After multiplication with n , we obtain

$$n \cdot QS_e(n) = n(n+1) + 2 \sum_{t=0}^{n-1} QS_e(t).$$

$\Rightarrow$

$$(n+1) QS_e(n+1) - n QS_e(n) =$$

$$(n+1)(n+2) + 2 \sum_{t=0}^n QS_e(t) - \left(n(n+1) + 2 \sum_{t=0}^{n-1} QS_e(t) \right)$$

After the simplification of the right side of the equation, the addition of $n QS_e(n)$ and the division by $(n+1)(n+2)$, we obtain

$$(*) \quad \frac{QS_e(n+1)}{n+2} = \frac{2}{n+2} + \frac{QS_e(n)}{n+1}.$$

Goal:

Transformation of the equation above to a recurrence equation which is easier to handle.

Define

$$A_n := \frac{QS_e(n)}{n+1}.$$

Then we can write the equation (*) in the following way:

$$A_{n+1} = \frac{2}{n+2} + A_n.$$

This yields the following recurrence equation:

$$A_1 = 0$$

$$A_{n+1} = \frac{2}{n+2} + A_n$$

Hence, we obtain

$$\begin{aligned}
A_n &= \frac{2}{n+1} + A_{n-1} \\
&= \frac{2}{n+1} + \frac{2}{n} + A_{n-2} \\
&= \frac{2}{n+1} + \frac{2}{n} + \dots + \frac{2}{3} + A_1 \\
&= 2 \cdot \sum_{i=3}^{n+1} \frac{1}{i} \\
&= 2 \left(H_{n+1} - \frac{3}{2} \right) \\
&< 2 \ln(n+1),
\end{aligned}$$

wobei $H_{n+1} = \sum_{i=1}^{n+1} \frac{1}{i}$ is the $(n+1)$ st harmonic number.

$$\Rightarrow QSe(u) = (n+1)A_n < 2(n+1) \ln(n+1).$$

Altogether, we have proved the following Theorem.

Theorem 4.3

The expected number of comparisons used by randomized quicksort for sorting of n numbers is smaller than $2(n+1) \ln(n+1)$.

4.3 Universal hashing

Let us review hashing, a method for storing data. To realize hashing, we use

- 1) an array $T[0:m-1]$, the hash table and
- 2) a function $h: U \rightarrow \{0, 1, \dots, m-1\}$ where

U is the universe containing all possible datas.

Let $S \subseteq U$ be the set which we wish to store in the data structure. Our goal is to store each element $x \in S$ in $T[h(x)]$.

Example:

$$m = 5, S = \{3, 15, 22, 24\} \text{ and } h(x) = x \bmod 5$$



0	15
1	
2	22
3	3
4	24

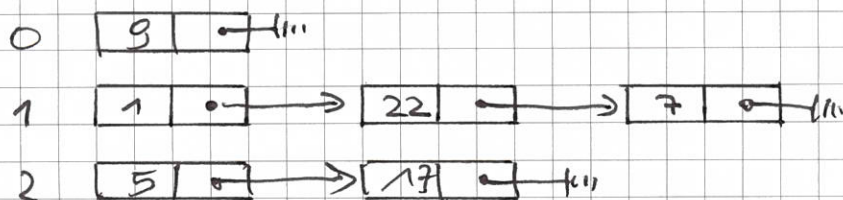
What to do if $h(x) = h(y)$ for $x, y \in S, x \neq y$?
Such an event is called a collision.

In our example, a collision occurs if we would replace 22 by 20.

A possible solution of the collision problem is the use of chained lists.

Example:

$$m = 3, S = \{1, 5, 7, 9, 17, 22\}, h(x) = x \bmod 3$$



We assume that the time to evaluate the hash function h is constant. The cost for an operation with respect to an $x \in U$ is defined by

$$1 + \delta_h(x, S)$$

where $\delta_h(x, S)$ is the number of elements y in S with $h(y) = h(x)$; i.e.,

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$$

where

$$\delta_h(x, y) = \begin{cases} 1 & \text{if } h(x) = h(y) \text{ and } x \neq y \\ 0 & \text{otherwise.} \end{cases}$$

worst case:

Each key $x \in S$ is stored in the same list

$$\leadsto \Omega(|S|).$$

The expected cost of hashing is much better. Usually, the expected cost is analyzed under the following assumptions:

- 1) The hash function $h: U \rightarrow \{0, 1, \dots, m-1\}$ distributes the universe U uniformly over the set $\{0, 1, \dots, m-1\}$; i.e., $\forall i, j \in \{0, 1, \dots, m-1\}$ there hold $|h^{-1}(i)| - |h^{-1}(j)| \leq 1$.

- 2) All keys in U are with the same probability argument of the next operation; i.e., for all $x \in U$ and $k \geq 1$ is x with probability $\frac{1}{|U|}$ argument of the k -th operation.

Let x_k denote the argument of the k -th operation.
Assumptions above $\Rightarrow$

$$\Pr(h(x_k) = i) = \frac{1}{m} \quad \forall i \in \{0, 1, \dots, m-1\}.$$

Using this implication of the assumptions above, the following theorem is not hard to prove.

Theorem 4.4

Under the assumptions above, a sequence of n insertion-, deletion- and access-operations takes expected time

$$T(n, m) \leq (1 + \beta/2) n$$

where $\beta = \frac{n}{m}$ is the maximal possible load factor of the hash table.

The first assumption is easily to obtain. But in practise, mostly the second assumption is not fulfilled.

Question:

How to omit the second assumption without changing the expected time complexity?

Observation

In practise, the system has no influence to the sequence of arguments. But the system could have influence to the choice of the hash function.

~>

Idea:

Define a suitable class H of hash functions and choose one function in H randomly.

Needed property:

For all $S \subseteq \{0, 1, \dots, N-1\}$ where $|S| = N$

"nearly all" functions in H are good; i.e., for all $x, y \in \{0, 1, \dots, N-1\}$, $x \neq y$, a randomly chosen function $h \in H$, the probability that $h(x) = h(y)$ is small.

~>

Let $c \in \mathbb{R}$, $N, m \in \mathbb{N}$. A collection

$$H \subseteq \{h \mid h: \{0, 1, \dots, N-1\} \rightarrow \{0, 1, \dots, m-1\}\}$$

of functions is called c-universal if for all $x, y \in \{0, 1, \dots, N-1\}$ with $x \neq y$

$$|\{h \mid h \in H \text{ and } h(x) = h(y)\}| \leq c \cdot \frac{|H|}{m}.$$

First, we shall construct a c -universal class of hash functions.

Theorem 4.5

Let $m, N \in \mathbb{N}$ and N be a prime number. Then

$$H_1 := \{ h_{a,b} \mid h_{a,b}(x) = [(ax + b) \bmod N] \bmod m \\ \text{and } a, b \in \{0, 1, \dots, N-1\} \}$$

is a c -universal class of hash functions, where
 $c = \left(\lceil \frac{N}{m} \rceil / \frac{N}{m} \right)^2$.

Proof:

Let $x, y \in \{0, 1, \dots, N-1\}$, $x \neq y$. Since
 $|H_1| = N^2$ it suffices to prove

$$|\{(a, b) \mid h_{a,b}(x) = h_{a,b}(y)\}| \leq c \cdot \frac{N^2}{m}.$$

Assume that

$$h_{a,b}(x) = h_{a,b}(y).$$

$$\Rightarrow \exists q \in \{0, 1, \dots, m-1\}, r, s \in \{0, 1, \dots, \lceil \frac{N}{m} \rceil - 1\} \\ \text{such that}$$

$$\begin{aligned} ax + b &= q + rm \bmod N & \text{and} \\ ay + b &= q + sm \bmod N. \end{aligned}$$

N is a prime number $\Rightarrow \mathbb{Z}_N$ is a field.

$\Rightarrow$

For each choice of q, r and s , the equations above have exactly one solution in a and b .

$\Rightarrow$

$$|\{(a, b) \mid h_{a,b}(x) = h_{a,b}(y)\}| = m \cdot \left\lceil \frac{N}{m} \right\rceil^2.$$

Since

$$\left(\frac{\lceil N/m \rceil}{N/m} \right)^2 \cdot \frac{N^2}{m} \geq m \cdot \left\lceil \frac{N}{m} \right\rceil^2$$

$$\Leftrightarrow \left(\frac{m}{N} \right)^2 \cdot \frac{N^2}{m} \geq m$$

$$\Leftrightarrow m \geq m$$

there holds: H_1 is c -universal

□

The size of H_1 is N^2 . Hence $2 \log N$ bits are necessary and sufficient for the specification of a function in H_1 . We can choose an element of H_1 randomly by generating randomly two numbers in $\{0, 1, \dots, N-1\}$.

Goal:

An analysis of universal hashing under the following assumptions:

- a) The hash function h is chosen at random from some c -universal class H of hash functions; i.e., each element $h \in H$ is chosen with probability $\frac{1}{|H|}$.
- b) We use chained lists for the treatment of collisions.

Theorem 4.6

Let $c \in \mathbb{N}$ and H be a c -universal class of

hash functions.

- a) Let $S \subseteq \{0, 1, \dots, N-1\}$, $|S| = n$ and $x \in \{0, 1, \dots, N-1\}$.
Then

$$\frac{1}{|H|} \cdot \sum_{h \in H} (1 + \delta_h(x, S)) \leq \begin{cases} 1 + \frac{cn}{m} & \text{if } x \notin S \\ 1 + \frac{c(n-1)}{m} & \text{if } x \in S. \end{cases}$$

- b) The expected cost of an access-, insert- or delete operation is $O(1 + \frac{cn}{m})$.
- c) The expected cost of a sequence of n access-, insert- and delete-operations starting with an empty hash table is $O((1 + \frac{cn}{2m})n)$.

Proof:

$$\begin{aligned} \text{a) } \sum_{h \in H} (1 + \delta_h(x, S)) &= |H| + \sum_{h \in H} \sum_{y \in S} \delta_h(x, y) \\ &= |H| + \sum_{y \in S} \sum_{h \in H} \delta_h(x, y) \\ &\leq |H| + \sum_{y \in S \setminus \{x\}} c \cdot \frac{|H|}{m} \\ &\leq \begin{cases} |H| (1 + \frac{cn}{m}) & \text{if } x \notin S \\ |H| (1 + \frac{c(n-1)}{m}) & \text{if } x \in S. \end{cases} \end{aligned}$$

- b) is an immediate consequence of a)

- c) follows immediately from b). Note that the expected cost of the i -th operation is $O(1 + \frac{c_i}{m})$.

The expected cost of c -universal hashing differs from "standard" hashing only by the factor c . But now, the assumptions for the analysis are controlled by the system and not by the user.

4.4 Contention resolution

I follow the lecture notes of Thomas Kesselheim given in Summer 2016 at the MPI Saarbrücken

<https://www.mpi-inf.mpg.de/departments/algorithms-complexity/teaching/Summer16/random/>

4.4.1 Waiting for one success

Assume you are rolling a dice over and over again. How long to expect until it shows "6" for the first time. More generally, we observe a random process which is "successful" with probability p and "unsuccessful" with probability $1-p$. Let T denote the random variable indicating the number of rounds it takes until we see "successful" for the first time.

T is a non-negative integer random variable

$$\Rightarrow \underset{\substack{\text{expected value} \\ \uparrow}}{E[T]} = \sum_{t=1}^{\infty} \Pr[T \geq t]$$

Observation:

$T \geq t \Leftrightarrow$ The first $t-1$ rounds are unsuccessful.

$$\Rightarrow \Pr[T \geq t] = (1-p)^{t-1}$$

and hence,

$$\begin{aligned} E[T] &= \sum_{t=1}^{\infty} \Pr[T \geq t] \\ &= \sum_{t=1}^{\infty} (1-p)^{t-1} \\ &= \frac{1}{p}. \end{aligned}$$

4.4.2 Collecting coupons

In contrast to above where we wait until we see a specific number for the first time, we wait now until we have seen each number of the dice at least once. This is the so-called coupon collector's problem. There are n coupons. In every round, one of these coupons is drawn uniformly with replacement. Let T be the number of rounds until each coupon is selected at least once.

Goal:

Calculation of $E(T)$.

For doing this, we divide the rounds into phases. The first round is also Phase 1.

Phase j starts after Phase $j-1$ and ends in the next round where we see a new coupon.

Let X_j denote the length of Phase j .

$\Rightarrow$

The time it takes until we see k different coupons is

$$\sum_{j=1}^k X_j.$$

$\Rightarrow$

$$T = \sum_{j=1}^n X_j.$$

When we are in Phase j , we have already seen $j-1$ different coupons. Hence, $n - (j-1) = n - j + 1$ coupons are still new.

$\Rightarrow$

The probability that a new coupon is seen in a round of Phase j is $\frac{n-j+1}{n}$.

Section 4.4.1 $\Rightarrow$

$$E[X_j] = \frac{n}{n-j+1}.$$

For the calculation of $E[T]$, we use the following lemma.

Lemma 4.4 (Linearity of expectation)

Let $X_1, X_2, \dots, X_n$ be (not necessarily independent) random variables. Then

$$E\left[\sum_{j=1}^n X_j\right] = \sum_{j=1}^n E[X_j].$$

191

Proof:

exercise

III

Lemma 4.4 $\Rightarrow$

$$\begin{aligned} E[T] &= E\left[\sum_{j=1}^n X_j\right] \\ &= \sum_{j=1}^n E[X_j] \\ &= \sum_{j=1}^n \frac{n}{n-j+1} \\ &= n \sum_{i=1}^n \frac{1}{i} \\ &= n \cdot H_n \end{aligned}$$

where H_n is the n -th harmonic number.

Remember that

$$\ln(n+1) < H_n < 1 + \ln n \quad \text{for all } n.$$

$\Rightarrow$

$$H_n = \Theta(\log n).$$

4.4.3 Contention resolution

The following contention resolution problem has some similarities with the coupon collectors problem.

Let $P_1, P_2, \dots, P_n$ be n processes in a distributed system. Each process wish to access a single shared data-base. The time of the distributed system is divided into discrete rounds. At most one process can access the data base in a single round. If more than one process attempt to access the data-base simultaneously then no process can access the database during the current round. Hence, it make no sense if all processes try to access the database in every round. How to resolve this problem? If the processes can communicate then a solution should be found. But what to do if the processes cannot communicate? This means how to get a protocol which manages the access to the shared data-base?

Randomization provides a natural protocol for this problem



- Each process will attempt to access the database in each round with probability p , independently of the decisions of the other processes.

Three situations can arise:

- 1) Exactly one process decides to make the attempt in a given round

~> It will succeed.

- 2) More than one process make an attempt

~> None will succeed.

- 3) No process make an attempt.

~> The round is "wasted."

The randomized algorithm above is extremely simple. But the analysis of its performance is more involved.

For the analysis of such a probabilistic system it is useful to extract some basic events and to investigate their probabilities.

- For P_i , $1 \leq i \leq n$ and round t , $A[i, t]$ denotes the event that P_i attempts to access the database in round t .

$$\Pr(A[i, t]) = p.$$

196

The complementary event $\overline{A[i,t]}$ indicates that $A[i,t]$ does not occur.

$$\Pr(\overline{A[i,t]}) = 1 - \Pr(A[i,t]) = 1 - p.$$

- $S[i,t]$ denotes the event that P_i succeeds in accessing the database in round t . Then

$$S[i,t] = A[i,t] \cap \left(\bigcap_{j \neq i} \overline{A[j,t]} \right).$$

By the definition of the contention resolution protocol, all events in this intersection are mutually independent.

$\Rightarrow$

$$\begin{aligned} \Pr(S[i,t]) &= \Pr(A[i,t]) \cdot \prod_{j \neq i} \Pr(\overline{A[j,t]}) \\ &= p(1-p)^{n-1} \end{aligned}$$

Question:

How to set p such that $\Pr(S[i,t])$ is maximized?

$$p=0 \text{ or } p=1 \Rightarrow \Pr(S[i,t]) = 0$$

The function

$$f(p) = p(1-p)^{n-1}$$

is positive for $p \in (0,1)$ and its derivative

$$f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$$

has a single zero at the value $p = \frac{1}{n}$

$\Rightarrow$

$\Pr(S[i, t])$ achieves its maximum at $p = \frac{1}{n}$.

For $p := \frac{1}{n}$, we get

$$\Pr(S[i, t]) = \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1}$$

Question:

What is the asymptotic value of this expression?

basic calculus $\leadsto$

Lemma 4.5

$$a) \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$$

$$b) \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^{n-1} = \frac{1}{e}.$$

Note that $\left(1 - \frac{1}{2}\right)^{2-1} = \frac{1}{2}$. Applying Lemma 4.5, we obtain

$$\frac{1}{en} \leq \Pr(S[i, t]) \leq \frac{1}{2n}.$$

Hence, $\Pr(S[i, t])$ is asymptotically equal to $\Theta\left(\frac{1}{n}\right)$.

Question

How many rounds are needed until process P_i succeeds to access the database at least once?

Let $F[i, t]$ denote the failure event that process P_i does not succeed in any of the first t rounds? Then

$$F[i, t] = \bigcap_{r=1}^t \overline{S[i, r]}.$$

Since all events in this intersection are mutually independent, there holds

$$\begin{aligned} \Pr(F[i, t]) &= \prod_{r=1}^t \Pr(\overline{S[i, r]}) \\ &= \left[1 - \frac{1}{n} \left(1 - \frac{1}{n} \right)^{n-1} \right]^t. \end{aligned}$$

This expression looks complicated. Hence, it is important to think asymptotically.

~>

$$\begin{aligned} \Pr(F[i, t]) &= \prod_{r=1}^t \Pr(\overline{S[i, r]}) \\ &\leq \left(1 - \frac{1}{en} \right)^t. \end{aligned}$$

After setting $t := \lceil en \rceil$, we get

$$\Pr(F[i, t]) \leq \left(1 - \frac{1}{en} \right)^{\lceil en \rceil} \leq \frac{1}{e}$$

After setting $t := \lceil en \rceil \cdot (c \cdot \ln n)$, we obtain

$$\Pr(F[i, t]) \leq e^{-c \cdot \ln n} = n^{-c}.$$

Question

How many rounds are needed until all processes will have succeeded in accessing the database at least once with high probability?

We say that the protocol fails after t rounds if at least one process has not yet succeeded in accessing the database. Let F_t denote the event that the protocol fails after t rounds.

F_t occurs if at least one of the events $F[i, t]$ occurs. Hence

$$F_t = \bigcup_{i=1}^n F[i, t].$$

Until now, we have obtained intersections of mutually independent events. But now, we have the union of events that are not independent.

Probabilities of such unions can be very hard to compute exactly. Often, it is sufficient to use the Union Bound to get an upper bound for this probability.



$$\Pr(F_t) \leq \sum_{i=1}^n \Pr(F[i, t]).$$

If we choose $t := 2 \lceil \frac{1}{\epsilon} \ln n \rceil$ then we obtain

$$\Pr(F_i) \leq n \cdot n^{-2} = n^{-1}.$$

Altogether, we get the following answers to our last question:

- With probability at least $1 - n^{-1}$, all processes succeed in accessing the database at least once within $t = 2 \lceil \log n \rceil \ln n$ rounds.

4.5 A first Monte Carlo algorithm: Finding the global minimum cut

Given an undirected graph $G = (V, E)$, we define a cut of G to be a partition of V into two nonempty set A and B . The size of a cut (A, B) is the number of edges with one end node in A and the other end node in B . A (global) minimum cut is a cut of minimum size. We shall consider the following problem:

Input: An undirected graph $G = (V, E)$.
task: Compute a minimum cut of G .

Using algorithms for the maximum flow problem, it is not hard to solve this problem in polynomial time. There are better deterministic algorithms. See for example

M. Stoer, F. Wagner, A simple min-cut algorithm, JACM 44 (1997), 585-591.

The best known deterministic algorithm for the minimum cut problem has run time $O(n \cdot m + n^2 \log n)$ where $n = |V|$ and $m = |E|$. We shall discuss two randomized algorithms where the second algorithm has a better running time than each known deterministic algorithm.

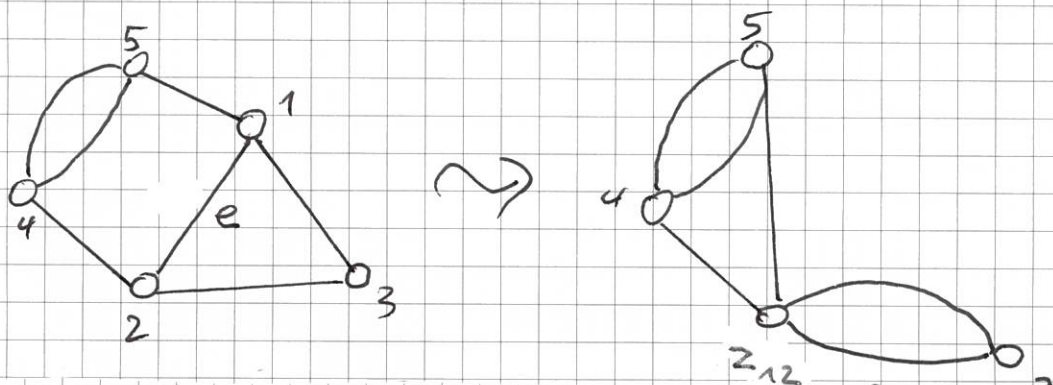
Reference

D. R. Karger, C. Stein, A new approach to the minimum cut problem, JACM 43 (1996), 601-640.

A multigraph is a graph which may contain multiple edges between any pair of different nodes.

The main operation of both algorithms is edge contraction. For the contraction of the edge $e = (u, v)$ both nodes u and v are replaced by one node z_{uv} , the edges between u and v are deleted and edges which have either u or v as an end node obtain instead of the end node u or v the end node z_{uv} .

Example 4.2



Effect of contracting the edge $e = (1, 2)$.

As example 4.2 shows, the contraction of an edge can produce parallel edges. Hence we have to consider multigraphs.

Let $G = (V, E)$ be a multigraph. The operation $\text{CONTRACT-EDGE}(E, e)$ replaces the multigraph G by the multigraph $G' = (V', E')$ where

- $V' = V \setminus \{u, v\} \cup \{z_{uv}\}$ and
- $E' = E \setminus \{(x, y) \mid \{x, y\} \cap \{u, v\} \neq \emptyset\}$
 $\cup \{(x, z_{uv}) \mid x \notin \{u, v\}, (x, u) \in E\}$
 $\cup \{(x, z_{uv}) \mid x \notin \{u, v\}, (x, v) \in E\}.$

We assign to each node x of the current multigraph a subset $S(x)$ of V . At the beginning, we set for each node $x \in V$

$$S(x) := \{x\}.$$

After the contraction of an edge $e = (u, v)$ we set

$$S(z_{uv}) := S(u) \cup S(v).$$

The contraction algorithm performs $n-2$ iterations. In each iteration, an edge is chosen uniformly at random from the edges which are still present. Note that the number of parallel edges between two nodes u and v is important for the probability that the pair (u, v) is chosen by the algorithm. Then, the chosen edge is contracted. This reduces the number of nodes in the current multigraph by one. The algorithm terminates

with two nodes x and y . The corresponding sets $S(x)$ and $S(y)$ form a partition of V . The number of parallel edges between x and y is exactly the size of the cut $(S(x), S(y))$.



Algorithm Cut

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

Output: cut $(S(x), S(y))$

Method:

for all $v \in V$

do

$S(v) := \{v\}$

od;

$V' := V$; $E' := E$;

while $|V'| > 2$

do

choose an edge $e = (u, v) \in E'$

uniformly at random;

CONTRACT-EDGE(E', e);

$V' := V' \setminus \{u, v\} \cup \{z_{uv}\}$;

$E' := E' \setminus \{(x, y) \mid \{x, y\} \cap \{u, v\} \neq \emptyset\}$
 $\cup \{(x, z_{uv}) \mid x \notin \{u, v\}, (x, u) \in E'\}$
 $\cup \{(x, z_{uv}) \mid x \notin \{u, v\}, (x, v) \in E'\}$;

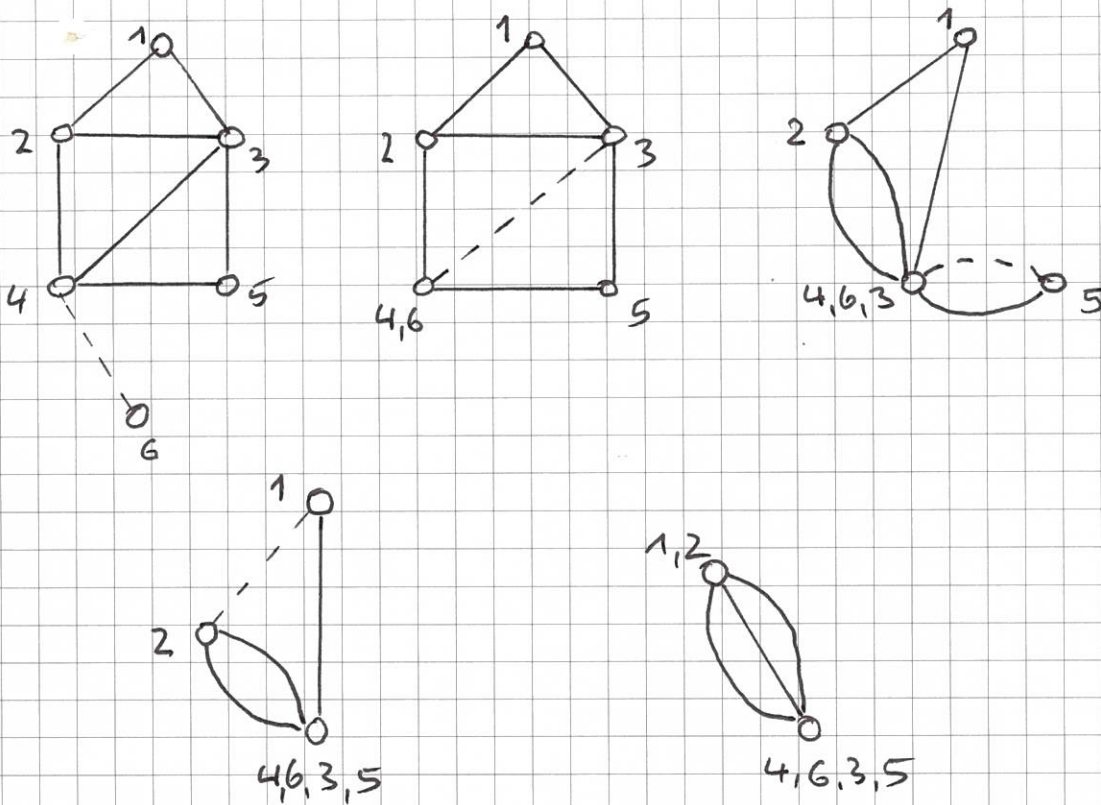
$S(z_{uv}) := S(u) \cup S(v)$

od (* Assume $V' = \{x, y\}$)

return $(S(x), S(y))$.

Example 4.3 (Run of Cut)

The dashed line is the edge which is contracted in the next step. We write at each node x the set $S(x)$.



The algorithm returns the cut $(\{1,2\}, \{4,6,3,5\})$ which has the size three. Note that it is not a minimum cut since the size of the cut $(\{\emptyset\}, \{1,2,3,4,5\})$ is one.

Exercise

Show that the output of the algorithm Cut is always a cut of the input graph $G = (V, E)$.

Theorem 4.7

The algorithm Cut returns a minimum cut of the input graph $G = (V, E)$ with probability at least $1/\binom{n}{2}$.

Proof:

Let (A, B) be any minimum cut of G and suppose that it has size k . This means that there is a set F of k edges with one end node in A and the other end node in B .

We want to derive a lower bound on the probability that the algorithm Cut returns the cut (A, B) .

Consider what could go wrong in the first step of the algorithm Cut:

- If an edge in F would be contracted, both end nodes of e would enter the same set $S(z_e)$ such that the cut (A, B) could not be the output of the algorithm.

Conversely, if an edge in $E \setminus F$ is contracted, then there is still a chance that the cut (A, B) would be the output of the algorithm.

↪

Goal:

An upper bound on the probability that an edge in F is contracted.

To get such an upper bound, we need a lower bound on the number of edges in E .

Note that for each node $v \in V$, the cut $(\{v\}, V \setminus \{v\})$ has the size $\text{degree}(v)$. Since the minimum cut (A, B) has size k , the degree of v has to be at least k .

$\Rightarrow$

$$\text{degree}(v) \geq k \quad \text{for all } v \in V.$$

$\Rightarrow$

$$|E| \geq \frac{1}{2} \cdot k \cdot n.$$

Hence, the probability that an edge in E is contracted is

$$\leq \frac{|E|}{|E|} = \frac{k}{\frac{1}{2}kn} = \frac{2}{n}.$$

Now we consider the situation after j iterations. Then we have $|V'| = n - j$.

Suppose that no edge in E has been contracted during the first j iterations.

Every cut of G' is a cut of G . Hence, each node in V' has degree at least k .

$\Rightarrow$

$$|E'| \geq \frac{1}{2} \cdot k \cdot (n - j).$$

$\Rightarrow$

The probability that an edge in E is contracted in the $(j+1)$ st iteration is

$$\leq \frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}.$$

The cut (A, B) is returned by the algorithm Cut iff no edge of F is chosen in any of the iterations $1, 2, \dots, n-2$.

Let E_j denote the event that an edge of F is not contracted in iteration j . Then we have shown

$$\Pr(E_1) \geq 1 - \frac{2}{n} \quad \text{and}$$

$$\Pr(E_{j+1} | E_1 \cap E_2 \cap \dots \cap E_j) \geq 1 - \frac{2}{n-j}$$

Goal:

A lower bound for $\Pr(E_1 \cap E_2 \cap \dots \cap E_{n-2})$.

By unwinding the formula for conditional probability, we obtain

$$\begin{aligned} & \Pr(E_1 \cap E_2 \cap \dots \cap E_{n-2}) \\ &= \Pr(E_1) \cdot \Pr(E_2 | E_1) \cdot \Pr(E_3 | E_1 \cap E_2) \cdot \dots \cdot \\ & \quad \cdot \Pr(E_{j+1} | E_1 \cap E_2 \cap \dots \cap E_j) \cdot \dots \cdot \Pr(E_{n-2} | E_1 \cap \dots \cap E_{n-3}) \\ &\geq \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \dots \cdot \left(1 - \frac{2}{n-j}\right) \cdot \dots \cdot \left(1 - \frac{2}{3}\right) \\ &= \left(\frac{n-2}{n}\right) \left(\frac{n-3}{n-1}\right) \left(\frac{n-4}{n-2}\right) \cdot \dots \cdot \left(\frac{2}{4}\right) \cdot \left(\frac{1}{3}\right) \\ &= \frac{2}{n(n-1)} = \binom{n}{2}^{-1}. \end{aligned}$$

Theorem 4.7 $\Rightarrow$

A single run of the algorithm Cut fails to find a minimum cut with probability

$$\leq \left(1 - \left(\frac{n}{2}\right)^{-1}\right).$$

This upper bound on the probability of success is very close to one. But we can improve the upper bound by repeatedly running the algorithm, with independent random choices, and taking the best cut we find.

$\Rightarrow$ If we run the algorithm $\left(\frac{n}{2}\right)$ times, then the probability that we fail to find a minimum cut of G in any run is

$$\leq \left(1 - \frac{1}{\left(\frac{n}{2}\right)}\right)^{\left(\frac{n}{2}\right)} \leq \frac{1}{e}.$$

We can improve the failure probability with further repetitions. If we run the algorithm Cut $\left(\frac{n}{2}\right) \cdot \ln n$ times, then the probability we fail to find a minimum cut of G is

$$\leq e^{-\ln n} = \frac{1}{n}.$$

Using an appropriate data structure, one edge contraction can be implemented such that the used time is $O(n)$. It is also possible to

choose an edge uniformly at random in $O(n)$ time. Hence, the overall running time of the algorithm Cut is bounded by $O(n^2)$.

(see Rajeev Motwani, Prabhakar Raghavan, Randomized Algorithms, Ch. 10.2)

$\Rightarrow$

We obtain a failure probability at most $\frac{1}{n}$ in time $O(n^2 \cdot n^2 \log n)$.

This time complexity is too high such that the algorithm Cut could be a good alternative to the best deterministic algorithms. Therefore, we wish to decrease the running time of the algorithm Cut.

The main problem with algorithm Cut is that it succeeds in finding a minimum cut only with probability $\Omega(n^{-2})$. Hence we have to repeat the algorithm at least $\Omega(n^2)$ times to ensure a reasonable probability of success. Thus, the obvious approach to improving the running time is to increase the probability that a minimum cut is found by the algorithm.

Suppose we consider a specific minimum cut (A, B) and wish that the algorithm produces this cut as output. This means that the algorithm never choose an edge crossing this cut for the

contraction. If such an edge is chosen then the cut (A, B) is destroyed.

Let $G_n = G, G_{n-1}, G_{n-2}, \dots, G_3, G_2$ be the sequence of graphs generated by the algorithm Cut with input $G = (V, E)$, $|V| = n$.

Note that in the first iteration the probability that the algorithm does not destroy the cut (A, B) is

$$\geq 1 - \frac{2}{n}.$$

This is close to one. The probability that the cut (A, B) still exists in G_t is

$$\begin{aligned} & \Pr(E_1) \cdot \Pr(E_2 | E_1) \cdot \dots \cdot \Pr(E_{n-t} | E_1, E_2, \dots, E_{n-t+1}) \\ & \geq \left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \cdot \dots \cdot \left(1 - \frac{2}{n - (n-t+1)}\right) \\ & = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \dots \cdot \frac{t-3}{t-1} \\ & = \frac{(t-2)(t-3)}{n(n-1)}. \end{aligned}$$

This shows the following lemma:

Lemma 4.5

Let (A, B) be a minimum cut in G . The probability that the cut still exists in G_t is $\Omega\left(\left(\frac{t}{n}\right)^2\right)$.

In order to obtain a reasonable success guarantee for the algorithm Cut, one has

to repeat the algorithm many times.

The probability that the algorithm destroys the cut (A, B) is low during the first iterations and high during the last iterations of the algorithm Cut.

After setting

$$t := 1 + \lceil \frac{n}{T} \rceil$$

and applying Lemma 4.6, we obtain:

- The probability that the cut (A, B) still exists in G_t is at least $\frac{1}{2}$.

Now we change the algorithm Cut in the sense that the algorithm terminates earlier.

Let Cut_t be that algorithm which we obtain if we replace the condition

$$|V'| > 2$$

of the while-loop by

$$|V'| > t$$

and the algorithm returns the current graph $G' = (V', E')$ instead of a cut.

The idea behind the algorithm FastCut is to include the repetitions into the algorithm instead of simply repeating the whole algorithm Cut.

For doing this, FASTCut makes two copies of the input graph G and let run Cut_{t_0} for $t_0 = 1 + \lceil \frac{n}{2} \rceil$ on both copies. Then the algorithm is applied recursively on the obtained graphs; i.e., each graph is duplicated and Cut_{t_0} performed on all four graphs. Then FASTCut duplicates each resulting graph and let run Cut_{t_0} on all eight graphs and so on. The recursion ends at the moment when the resulting graphs have only six nodes. A minimum cut of these graphs is obtained by brute-force enumeration.



Algorithm FASTCut(G)

Input: A multigraph $G = (V, E)$, $n = |V|$.

Output: A cut (V_1, V_2) of G .

Method:

If $n \leq 6$

then

compute a minimum cut of G by
brute-force enumeration

else

$t_0 := 1 + \lceil \frac{n}{2} \rceil$;

$H_1 := \text{Cut}_{t_0}(G)$; $H_2 := \text{Cut}_{t_0}(G)$;

FASTCut(H_1); FASTCut(H_2);

return the smaller of the two cuts
produced by $\text{FASTCUT}(H_1)$ and
 $\text{FASTCUT}(H_2)$

fi.

It remains the analysis of the algorithm. First we shall estimate the time needed by $\text{CUT}_{t_0}(G)$.

As remarked above, choosing an edge uniformly at random in a graph with n nodes and the contraction of the chosen edge can be performed in $O(n)$ time.

$\Rightarrow$

$\text{CUT}_{t_0}(G)$ can be performed in $O(t_0 \cdot n)$ time where n is the number of nodes in G .

$\Rightarrow$

Performing two $\text{CUT}_{t_0}(G)$ can be performed in $O(n^2)$ time.

Therefore, we obtain the following recurrence for the running time $T(n)$ of the algorithm $\text{FASTCUT}(G)$ where n is the number of nodes in G :

$$T(n) = 2 \cdot T\left(1 + \left\lceil \frac{n}{2} \right\rceil\right) + O(n^2)$$

The solution of this recurrence is $O(n^2 \log n)$

Exercise

a) Show that the solution of the recurrence

$$T(n) = 2 \cdot T\left(1 + \left\lceil \frac{n}{2} \right\rceil\right) + O(n^2)$$

is $O(n^2 \log n)$.

b) Show that FASTCUT can be implemented such that it uses $O(n^2)$ space.

It remains the analysis of the probability that FASTCUT(G) returns a minimum cut of the input graph G .

Theorem 4.8

The algorithm FASTCUT returns a minimum cut of the input graph $G = (V, E)$ with probability $\Omega\left(\frac{1}{\log n}\right)$ where $n = |V|$.

Proof:

Suppose that the size of a minimum cut of G is k . FASTCUT(G) returns a best cut found by any of its branches.

Assume that a cut of size k has survived up to some point in the recursion where the size of the residual graph H is t .

Let $H_1 = \text{CUT}_{t'}(H)$ and $H_2 = \text{CUT}_t(H)$, where $t' = 1 + \left\lceil \frac{t}{2} \right\rceil$ be the two contracted versions of H produced by FASTCUT.

FASTCUT (H) will return a minimum cut for G if the following two properties hold:

- 1) A cut of size k survives at least one of the two contraction sequences producing the graphs H_1 and H_2 , respectively.
- 2) FASTCUT succeeds in finding a cut of size k in the graph H_i in which a cut of size k has survived.

As observed above, the probability that any specific minimum cut in H survives a contraction sequence that reduces the number of nodes from t to $1 + \lceil \frac{t}{\sqrt{2}} \rceil$ is at least $\frac{1}{2}$.

Let $P(t)$ denote the probability that FASTCUT succeeds in finding a minimum cut in a graph with t nodes. Then

$$\frac{1}{2} \cdot P\left(1 + \lceil \frac{t}{\sqrt{2}} \rceil\right)$$

is a lower bound for the conditional probability that a cut of size k survives the contraction sequence from H to H_i and FASTCUT(H_i) returns a cut of size k .

$$\Rightarrow \left(1 - \frac{1}{2} \cdot P\left(1 + \lceil \frac{t}{\sqrt{2}} \rceil\right)\right)^2$$

is an upper bound for the probability that $\text{FastCut}(H)$ does not return a cut of size k assuming that H contains a cut of size k .

$\Rightarrow$

$$P(t) \geq 1 - \left(1 - \frac{1}{2} \cdot P\left(1 + \left\lceil \frac{t}{2} \right\rceil\right)\right)^2$$

How to solve such a recurrence? As a first step, it is useful to perform a change of variables and to turn the recurrence into an equality. Instead of writing a recurrence with respect to the problem size, we shall perform the recurrence with respect to the depth of the recursion. Define

$$p(k) := P(t),$$

where $k = 2 \log t - O(1)$ is the depth of the recursion.

Since for $t \leq 6$, a minimum cut is computed, there holds

$$p(0) = 1.$$

Furthermore, we obtain the following recurrence

$$\begin{aligned} p(k) &= 1 - \left(1 - \frac{1}{2} p(k-1)\right)^2 \\ &= p(k-1) - \frac{p(k-1)^2}{4}. \end{aligned}$$

Note that $p(k)$ is decreasing and converges to zero.

Usually, it is easier to work with large quantities than with quantities which get smaller and smaller. Hence, we define

$$q(k) := \frac{1}{p(k)}.$$

Then (here hold

$$\bullet \quad q(0) = \frac{1}{p(0)} = 1 \quad \text{and}$$

$$\begin{aligned} \bullet \quad q(k) &= \frac{1}{p(k-1) - \frac{p(k-1)^2}{4}} \\ &= \frac{4}{4p(k-1) - p(k-1)^2} \\ &= \frac{4}{\frac{4}{q(k-1)} - \frac{1}{q(k-1)^2}} \\ &= \frac{4q(k-1)^2}{4q(k-1) - 1} \\ &= \frac{[4q(k-1) - 1]q(k-1) + q(k-1)}{4q(k-1) - 1} \\ &= q(k-1) + \frac{q(k-1)}{4q(k-1) - 1} \\ &\geq q(k-1) + \frac{1}{4} \end{aligned}$$

Hence

$$q(k) \geq 1 + \frac{k}{4}.$$

Since

$$\frac{q(k-1)}{4q(k-1) - 1} \leq \frac{1}{3}$$

there holds also

$$q(k) \leq 1 + \frac{k}{3}.$$

Altogether, we have obtained

$$q(k) = \Theta(k)$$

and hence

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

$\Rightarrow$

$$P(t) = \Theta\left(\frac{1}{\log t}\right).$$

Using n instead of t in the last expression, we obtain

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

Exercise

How often, we have to run the algorithm FASTCUT for increasing the success probability to $\frac{1}{2}$?

4.6 Primality testing

Given a number $n \in \mathbb{N}$, we wish to test if n is a prime number. For doing this one could divide n by all prime numbers $p \leq \sqrt{n}$. If one find such a number which divides n then n is composite and prime otherwise. If n is large then such an algorithm would be too inefficient.

Goal:

The development of an efficient Monte-Carlo-algorithm for primality testing.

Idea:

Given any composite number n and any number $a \in \mathbb{N}$ with $\gcd(a, n) = 1$, we develop a criterion which, in the case that it is fulfilled, proves that n is composite.

Then we show that for any composite number n , at most a quarter of the natural numbers $a \leq n$ with $\gcd(a, n) = 1$ do not fulfill the criterion. In the case that n is prime, the criterion is never fulfilled.



The Monte-Carlo-algorithm works as follows:

The algorithm chooses at random $a < n$ and checks if $\gcd(a, n) = 1$. If $\gcd(a, n) \neq 1$ then a is a divisor of n such that n is composite. Otherwise, the algorithm checks if the criterion is fulfilled. In that case, the algorithm has found a proof that n is composite. Otherwise, the algorithm repeats the whole t -times. If the algorithm does not find a proof that n is composite then the algorithm returns that n is prime. The probability that this output is wrong is $\leq (\frac{1}{4})^t$. First, we shall derive the criterion.

Theorem 4.9

Let n be a prime number and $a \in \mathbb{N}$ with $\gcd(a, n) = 1$. Let $s = \max\{r \in \mathbb{N} \mid 2^r \text{ divides } n-1\}$ and let $d = \frac{n-1}{2^s}$. Then

$$a^d \equiv 1 \pmod{n}$$

or $\exists r \in \{0, 1, \dots, s-1\}$ with

$$a^{2^r d} \equiv -1 \pmod{n}.$$

Proof:

Consider any $a \in \mathbb{N}$ with $\gcd(a, n) = 1$.

Let $\phi(n)$ be the number of $a \in \{1, 2, \dots, n-1\}$ with $\gcd(a, n) = 1$.

Since n is a prime number there holds

$$\phi(n) = n-1.$$

Definition of s and $d \Rightarrow$

$$n-1 = 2^s \cdot d.$$

- The order of an element of a finite multiplicative group divides the order of the group.

$\Rightarrow$

The order k of the residue class

$$a^d + n\mathbb{Z}$$

is a power of 2.

We distinguish two cases:

- If $k = 2^0 = 1$ then $(a^d)^1 = a^d \equiv 1 \pmod{n}$
- If $k = 2^l$ with $1 \leq l \leq s$ then the residue class

$$a^{2^{l-1}d} + n\mathbb{Z}$$

has order 2.

Otherwise, k would be $< 2^l$.

The only element of order 2 is

$$-1 + n\mathbb{Z}.$$

$\Rightarrow$

$$a^{2^{s-1}d} \equiv -1 \pmod{n}.$$

Theorem 4.9 $\Rightarrow$

For a number n , an integer a with $\gcd(a, n) = 1$ such that $\forall r \in \{1, 2, \dots, s-1\}$

$$a^d \not\equiv 1 \pmod{n} \quad \text{and}$$

$$a^{2^{r-1}d} \not\equiv -1 \pmod{n}$$

is a witness against the primality of n .

The following theorem gives an upper bound for the number of integers in $\{1, 2, \dots, n-1\}$ which are not a witness against the primality of n .

Theorem 4.10

Let $n > 9$ be an odd composite integer.

Then $\{1, 2, \dots, n-1\}$ contains at most $\frac{1}{4}(n-1)$ integers a such that

- i) $\gcd(a, n) = 1$ and
- ii) a is not a witness against the primality of n .

The proof of Theorem 4.10 needs some basics in number theory. Furthermore, the proof is not simple. Therefore, the proof is omitted.


```

else
    if  $a_{k-1} \not\equiv -1 \pmod{n}$ 
        then
            return "composite"
        else
            return "prime"
    fi
fi
fi
fi

```

The following theorem summarizes the properties of the algorithm PRIMALITY TEST.

Theorem 4.11

If the input n of the algorithm PRIMALITY TEST is a prime number then the algorithm returns always "prime". If n is composite then the output is "composite" with probability $\geq \frac{3}{4}$.

If we repeat the algorithm with respect to an input t - times, the error probability would be reduced to $\leq 4^{-t}$.

For a long time, it was an open problem if primality testing is in P . In 2002, Agrawal, Kayal and Saxena have shown that there is a deterministic polynomial time algorithm for primality testing. An excellent book

describing the development of primality testing from randomized algorithms to "PRIMES is in P^h " is

Martin Dietzfelbinger, Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES is in P^h ", LNCS 3000, Springer 2004.