Topics in Algorithm Design - Lecture Notes for COMP 5703

Anil Maheshwari
School of Computer Science
Carleton University
anil@scs.carleton.ca

November 9, 2015

Contents

Preface + Acknowledgments

These notes extensively use material from the course notes of Lars Arge, David Mount, COMP 2805/3803 Notes of myself and Michiel Smid [69], CLRS book [21], Knuth's Art of Computer Programming [61], Kleinberg and Tardos Algorithms book [60], Ullman et al's book on Algorithms for Massive Data Sets [85]. These notes are updated occasionally. A substantial update was done in the Fall Term of 2013. Chapters on elementary probability, locality sensitive hashing, dimensionality reduction, and several exercises have been added. Moreover, as part of the offering of COMP 5703 in the Fall of 2013, several students have contributed significantly. Gregory Bint updated the chapter on the Minimum Spanning Trees, and has added a completely new Section 3.5 on the Spanning Tree Verification. Alexis Beingessner has provided Section 5.3 on extension of the planar separator theorem. Andrew Wylie has updated the chapter on Locality Sensitive Hashing and has added a Section 7.5.5 on Image Similarities and provided an extensive bibliography on locality sensitive hashing and its applications. Michael St. Jules has provided an entire chapter on Stable Matchings. I thank the Fall 2013 batch of COMP 5703 for providing valuable contributions. In Fall 2015 term I started to work on the Chapter on Second Moment Method. The addition of this chapter was inspired by a comment from one of the referees of our paper [8] in ALGOSENSORS 2015, where he/she mentioned that the paper is a good introduction to the Second Moment Method at graduate level. So I have pasted the whole paper in that chapter and added a section on Cliques. Over time, this chapter will evolve.

I have used parts of this material for the graduate course COMP 5703 at Carleton. The aim of these notes is mainly to summarize the discussion in the lectures. They are not designed as stand alone chapters or a comprehensive coverage of a topic. The exercises are from numerous sources and most of them have been asked in either an exam or in an assignment! Since these chapters have been written over time, and many of the TeX tools weren't available in olden times, you will see that the initial chapters don't have elegant figures or texts. Hopefully, volunteers in future will modernize this part!

If you spot any errors, or have suggestion which can help me to improve, I will be glad to hear them. If you wish to add material to these notes, including exercises, please do get in touch. Thanks in advance!

Anil Maheshwari
anil@scs.carleton.ca

Chapter 1

Preliminaries

1.1 Introduction

- Class is about *designing* and *analyzing algorithms*
  - *Algorithm*:
    - Mis-spelled *logarithm!*.
    - The first most popular algorithm is the Euclid's algorithm for computing the GCD of two numbers.
    - A well-defined procedure that transfers an input to an output.
    - Not a program (but often specified like it): An algorithm can often be implemented in several ways.
    - Knuth's, Art of Computer Programming, vol.1, is a good resource on the history of algorithms!. He says that an algorithm is a finite set of rules that gives a sequence of operations for solving a specific type of problem. Algorithm has five important features:
      Finiteness: must terminate after finite number of steps.
      Definiteness: each step is precisely described.
      Input: algorithm has zero or more inputs.
      Output: has at least one output!.
      Effectiveness: Each operation should be sufficiently basic such that they can be done in finite amount of time using pencil and paper.
  - *Design*: The focus of this course is on how to design good algorithms and how to analyze their efficiency. We will study methods/ideas/tricks for developing fast and efficient algorithms.
  - *Analysis*: Abstract/mathematical comparison of algorithms (without actually implementing, prototyping and testing them).
- This course will require proving the correctness of algorithms and analyzing the algorithms. Therefore MATH is the main tool. Math is needed in three ways:
  - Formal specification of problem
  - Analysis of correctness

  - Analysis of efficiency (time, memory use,...)

  Please review mathematical induction, what is a proof?, logarithms, sum of series, elementary number theory, permutations, factorials, binomial coefficients, harmonic numbers, Fibonacci numbers and generating functions [Knuth vol 1. or his book Concrete Mathematics is an excellent resource].

- Hopefully the course will show that **algorithms matter!**

1.2 Model of Computation

- Predict the resources used by the algorithm: running time and the space.
- To analyze the running time of an algorithm, we need a mathematical model of a computer:

  Random-access machine (RAM) model:
  - Memory consists of an infinite array of cells.
  - Each cell can store at most one data item (bit, byte, a record, ..).
  - Any memory cell can be accessed in unit time.
  - Instructions are executed sequentially
  - All basic instructions take unit time:
    * Load/Store
    * Arithmetic's (e.g. $+, -, *, /$)
    * Logic (e.g. $>$)

- Running time of an algorithm is the number of RAM instructions it executes.
- RAM model is not realistic, e.g.
  - memory is finite (even though we often imagine it to be infinite when we program)
  - not all memory accesses take the same time (cache, main memory, disk)
  - not all arithmetic operations take the same time (e.g. multiplications are expensive)
  - instruction pipelining
  - other processes
- But RAM model often is enough to give relatively realistic results (if we don't cheat too much).

1.3 Asymptotics

We do not want to compute a detailed expression of the run time of the algorithm, but rather will like to get a feel of what it is like? We will like to see the trend - i.e. how does it increase when the size of the input is increased - is it linear in the size of the input? or quadratic? or exponential?

or who knows? The asymptotics essentially capture the rate of growth of the underlying functions describing the run-time. Asymptotic analysis assumes that the input size is large (since we are interested how the running time increases when the problem size grows) and ignores the constant factors (which are usually dependent on the hardware, programming smartness or tricks, compile-time-optimizations).

David Mount suggests the following simple definitions based on the limits for functions describing the running time of algorithms. We will describe the formal definitions from [21] later.

Let $f(n)$ and $g(n)$ be two positive functions of n. What does it mean when we say that both f and g grow at roughly the same rate for large n (ignoring the constant factors), i.e.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,$$

where c is a constant and is neither 0 or ∞. We say that $f(n) \in \theta(g(n))$, i.e. they are asymptotically equivalent. What about $f(n)$ does not grow significantly faster than $g(n)$ or grows significantly faster? Here is the table of definitions from David Mount.

Asymptotic Form	Relationship	Definition
$f(n) \in \Theta(g(n))$	$f(n) \equiv g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in O(g(n))$	$f(n) \leq g(n)$	$0 \leq \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in \Omega(g(n))$	$f(n) \geq g(n)$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)}$
$f(n) \in o(g(n))$	$f(n) < g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \omega(g(n))$	$f(n) > g(n)$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Example: $T(n) = \sum_{x=1}^{n} x^2 \in \Theta(n^3)$. Why?

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{(n^3 + 3n^2 + 2n)/6}{n^3} = 1/6,$$

and $0 < 1/6 < \infty$.
Just for fun show that $T(n) \in O(n^4)$ or $T(n) = n^3/3 + O(n^2)$.

1.3.1 O-notation

$O(g(n)) = \{f(n) : \exists\ c, n_0 > 0 \text{ such that } f(n) \leq cg(n)\ \forall n \geq n_0\}$

- $O(\cdot)$ is used to asymptotically *upper bound* a function.
- $O(\cdot)$ is used to bound *worst-case* running time (see Figure 1.1).

- Examples:
  - $1/3n^2 - 3n \in O(n^2)$ because $1/3n^2 - 3n \leq cn^2$ if $c \geq 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in O(n^k)$, where $k \geq d$ is a constant. What are c and n_0 for this?

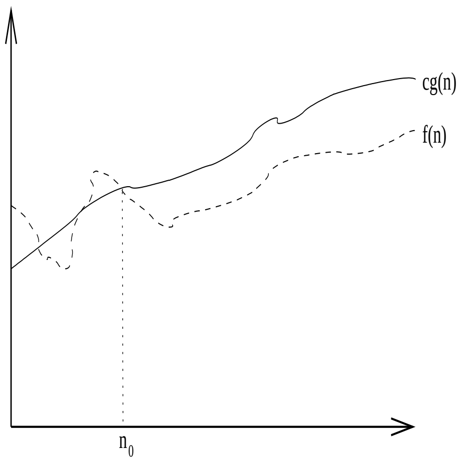


Figure 1.1: Illustration of $O()$ notation.

- Note:
  - When we say "the running time is $O(n^2)$", we mean that the *worst-case* running time is $O(n^2)$ — best case might be better.
  - We often abuse the notation:
    - We write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$!
    - We often use $O(n)$ in equations: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$).
    - We use $O(1)$ to denote a constant.

1.3.2 Ω-notation (big-Omega)

$\Omega(g(n)) = \{f(n) : \exists\ c, n_0 > 0 \text{ such that } cg(n) \leq f(n)\ \forall n \geq n_0\}$

- $\Omega(\cdot)$ is used to asymptotically *lower bound* a function (see Figure 1.2).

- Examples:
  - $1/3n^2 - 3n = \Omega(n^2)$ because $1/3n^2 - 3n \geq cn^2$ if $c \leq 1/3 - 3/n$ which is true if $c = 1/6$ and $n > 18$.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in \Omega(n^k)$, where $k \leq d$ is a constant. What are c and n_0 for this?
  - Prove or disprove: $g(n) = \Omega(f(n))$ if and only if $f(n) = O(g(n))$.

- Note:
  - When we say "the running time is $\Omega(n^2)$", we mean that the *best case* running time is $\Omega(n^2)$ — the worst case might be worse.

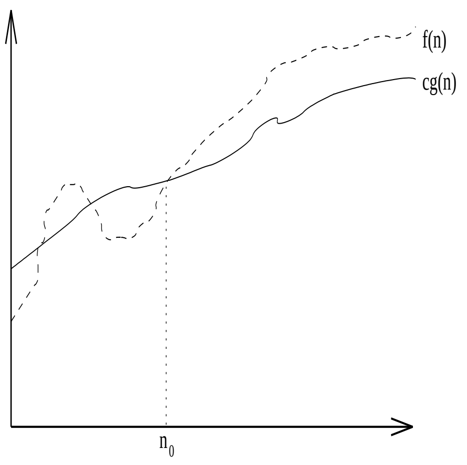


Figure 1.2: Illustration of $\Omega()$ notation.

1.3.3 Θ-notation (Big-Theta)

$\Theta(g(n)) = \{f(n) : \exists\ c_1, c_2, n_0 > 0$ such that $c_1 g(n) \le f(n) \le c_2 g(n)\ \forall n \ge n_0\}$

- $\Theta(\cdot)$ is used to asymptotically *tight bound* a function.

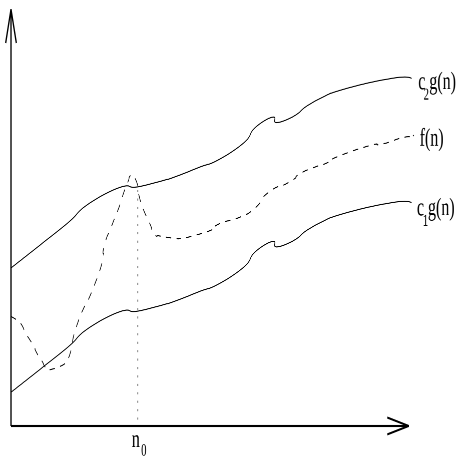


Figure 1.3: Illustration of $\Theta()$ notation.

$f(n) = \Theta(g(n))$ *if and only if* $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (see Figure 1.3)

- Examples:
  - $6n \log n + \sqrt{n} \log^2 n = \Theta(n \log n)$:

    - We need to find n_0, c_1, c_2 such that $c_1 n \log n \leq 6n \log n + \sqrt{n} \log^2 n \leq c_2 n \log n$ for $n > n_0$ $c_1 n \log n \leq 6n \log n + \sqrt{n} \log^2 n \Rightarrow c_1 \leq 6 + \frac{\log n}{\sqrt{n}}$. Ok if we choose $c_1 = 6$ and $n_0 = 1$. $6n \log n + \sqrt{n} \log^2 n \leq c_2 n \log n \Rightarrow 6 + \frac{\log n}{\sqrt{n}} \leq c_2$. Is it ok to choose $c_2 = 7$? Yes, $\log n \leq \sqrt{n}$ if $n \geq 2$.
    - So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in \Theta(n^k)$, where $k = d$ is a constant.

1.4 How to analyze Recurrences?

There are many ways of solving recurrences. I personally prefer the recursion tree method, since it is visual! Here the recurrence is depicted in a tree, where the nodes of the tree represent the cost incurred at the various levels of the recursion. We illustrate this method using the following recurrence (so called the recurrence used in the Masters method).

Let $a \geq 1, b > 1$ and $c > 0$ be constants and let $T(n)$ be the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k,$$

defined for integer $n \geq 0$. Then

Case 1 $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$.

Case 2 $a = b^k$ then $T(n) = \Theta(n^k \log_b n)$.

Case 3 $a < b^k$ then $T(n) = \Theta(n^k)$.

The proof is fairly simple. We need to visualize the levels of the underlying recursion tree.

Level 1: a subproblems are formed, each of size n/b, and the total cost is cn^k.

Level 2: a^2 subproblems are formed, each of size n/b^2, and the total cost is $a * c(n/b)^k$.

Level 3: a^3 subproblems are formed, each of size n/b^3, and the total cost is $a^2 * c(n/b^2)^k$.

...

Level $\log_b n$: $a^{\log_b n}$ subproblems are formed, each of constant size and the total cost is about $a^{\log_b n} c(\frac{n}{b^{\log_b n}})^k$.

Therefore the total cost is

$$T(n) = O(n^{\log_b a}) + \sum_{i=0}^{\log_b n} a^i c \left(\frac{n}{b^i}\right)^k.$$

Now apply the various cases.

1.5 Strassen's Matrix Multiplication

1.5.1 Matrix Multiplication

This is a classical example to illustrate the recurrences as well as the divide and conquer method. Consider Strassen's matrix multiplication method [64, 90] as illustrated in the following. Let A, B and C be three $n \times n$ matrices, where $Z = X \cdot Y$. There are n rows, n columns and $n \times n$ entries in each of the matrices.

- $X = \left\{ \begin{array}{cccc} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ x_{n1} & x_{n2} & \cdots & x_{nn} \end{array} \right\}$

- $Y = \left\{ \begin{array}{cccc} y_{11} & y_{12} & \cdots & y_{1n} \\ y_{21} & y_{22} & \cdots & y_{1n} \\ \cdots & \cdots & \cdots & \cdots \\ y_{n1} & y_{n2} & \cdots & y_{nn} \end{array} \right\}$

- We want to compute $Z = X \cdot Y$, where

$$z_{ij} = \sum_{k=1}^{n} x_{ik} \cdot y_{kj}.$$

- How many operations we require?

- In all we generate n^2 entries in the matrix Z and each entry requires n multiplications and $n-1$ additions. So the total number of operations can be bounded by $\Theta(n^3)$.

- Next we want to discuss a divide and conquer solution by Strassen which requires only $O(n^{\log_2 7})$ operations.

- Let's first analyze the recurrence

$$T(n) = 7T(n/2) + cn^2,$$

  where c is a constant, n is a positive integer, and $T(constant) = O(1)$.

- Using the simplified master method, $a = 7$, $b = 2$, $c = c$, $k = 2$ and $a > b^k$. Hence $T(n) = O(n^{\log_2 7})$.

1.5.2 Strassen's Algorithm

- Divide each of the matrices into four sub-matrices, each of dimension $n/2 \times n/2$. Strassen observed the following:

$$Z = \left\{ \begin{array}{cc} A & B \\ C & D \end{array} \right\} \cdot \left\{ \begin{array}{cc} E & F \\ G & H \end{array} \right\} = \left\{ \begin{array}{cc} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{array} \right\}$$

  where

$$\begin{aligned}
S_1 &= (B-D)\cdot(G+H)\\
S_2 &= (A+D)\cdot(E+H)\\
S_3 &= (A-C)\cdot(E+F)\\
S_4 &= (A+B)\cdot H\\
S_5 &= A\cdot(F-H)\\
S_6 &= D\cdot(G-E)\\
S_7 &= (C+D)\cdot E
\end{aligned}$$

- Lets test that for S_4+S_5, which is supposed to be $AF+BH$.

$$\begin{aligned}
S_4+S_5 &= (A+B)\cdot H + A\cdot(F-H)\\
&= AH+BH+AF-AH\\
&= AF+BH
\end{aligned}$$

- This leads to a divide-and-conquer algorithm with running time $T(n) = 7T(n/2)+\Theta(n^2)$, since
  - We only need to perform 7 multiplications recursively. Additions/Subtractions only take $\Theta(n^2)$ time, and we need to do 18 of them for $n/2 \times n/2$ matrices for each step of the recursion.
  - Division/Combination can still be performed in $\Theta(n^2)$ time.

Matrix multiplication is a fundamental problem and it arises in almost all branches of Sciences, Social Sciences and Engineering. For example, high energy physicists multiply monstrous matrices. Strassen's is not the currently fastest known algorithm, there have been numerous improvements over that method. It is obvious that any algorithm for matrix multiplication needs to perform $\Omega(n^2)$ operations, since the output matrix Z has that many entries. But only lower bound that is known for this problem is the trivial one, i.e. the $\Omega(n^2)$ bound. The currently best known upper bound is significantly larger than this (its about $O(n^{2.37..})$). So a major open problem, whose solution will be of immense importance will be either to raise the lower bound or drop down the upper bound!

Here is something which may be interesting to look into. This is regarding verifying given three $n\times n$ matrices A, B and C, whether $AB = C$? It turns out that there is a nice randomized algorithm that can do this and it is stated in the following Theorem (See Motwani and Raghavan [78] for details).

Theorem 1.5.1 *Let A, B, and C be $n\times n$ matrices over a Field F such that $AB \neq C$. Then for r chosen uniformly at random from $\{0,1\}^n$, probability that $Pr[ABr = Cr] \leq 1/2$.*

1.6 Elementary stuff from probability

This material is adapted from the book of Meyer [74]. (BTW, this was my textbook for the first undergraduate course in probability - way back in the winter of 1982-83!)

With each probabilistic experiment, the *sample space* is the set of all possible outcomes of that experiment. For example, for rolling a dice the set of all possible outcomes are $\{1, 2, 3, 4, 5, 6\}$. An *event* is also a set of possible outcomes, and is a subset of the sample space. For example, for rolling a dice and getting an even number, the events are $\{2, 4, 6\}$. If the sample space consists of n elements, then the total number of all possible events are 2^n. Since events are sets, we can use associated operations on sets. For example, if A and B are events for a sample space S, then we can define $A \cup B$, $\bar{A}$, $A \cap B$, with the usual meaning.

Two events A and B are said to be *mutually exclusive* if they cannot occur simultaneously, i.e. $A \cap B = \emptyset$. Let S be a sample space associated with an experiment. For each event $A \subseteq S$, associate a real number $0 \leq P(A) \leq 1$, called as the *probability* of A, and $P(A)$ satisfies following natural conditions

1. $P(S) = 1$,
2. If events A and B are mutually exclusive, $P(A \cup B) = P(A) + P(B)$ and $P(A) \cap P(B) = 0$.
3. $P(\bar{A}) = 1 - P(A)$, where $\bar{A} = S \setminus A$.
4. In general, for two events A and B, $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
5. If $A \subset B$, $P(A) \leq P(B)$.

Let A and B be two events associated with an experiment. We say $P(B|A)$ to be the *conditional probability* of occurrence of event B given that A has occurred. Intuitively, computation of $P(B)$ is done with respect to a reduced sample space. For example, let B be the event of getting 2 after rolling a dice. Then $P(B) = 1/6$. Let A be the event of getting an even number after rolling a dice. Then $P(A) = 1/2$. What about $P(B|A)$?. It is 1/3, since the reduced sample space is $\{2, 4, 6\}$, and the probability of drawing a 2 is 1/3 from this space. Note that $P(B|A) = \frac{P(A \cap B)}{P(A)}$ or equivalently $P(A \cap B) = P(B|A)P(A)$. Similarly we can define $P(A|B) = \frac{P(A \cap B)}{P(B)}$, or equivalently $P(A \cap B) = P(A|B)P(B)$. Note that if two events A and B are mutually-exclusive than $P(A|B) = P(B|A) = 0$. If $A \subset B$, than $P(B|A) = 1$. (For example, if B is the event of obtaining an even number and A is the event of getting a 2 on rolling a dice, than if A has occurred, than for sure B has occurred.) Now a classical theorem on conditional probabilities, called *Bayes Theorem*.

Let $B_1, B_2, \cdots, B_k$ represent a partition of sample space S. Let A be an event in S. Then

$$P(B_i|A) = \frac{P(A \cap B_i)}{P(A)} = \frac{P(A|B_i)P(B_i)}{\sum_{i=1}^{k} P(A|B_i)P(B_i)}.$$

Here is a classical example from Meyer [74] showing an application of this theorem.

Example 1.6.1 *An item is produced by three factories - F1, F2, and F3. F2 and F3 produce the same number of items. F1 produces twice many items as F2 and F3. 2% of items produced by F1 and F2 are defective, and 4% of items produced by F3 are defective. All of these items are indistinguishable in terms of which factory they come from. First let us understand the partitioning by the following.*
a) What is the probability that an item is defective?
Here $A = \{$*item is defective*$\}$. $B_1 = \{$*item came from F1*$\}$ *and likewise define* B_2 *and* B_3. *Note*

that $P(B_1) = 1/2$ and $P(B_2) = P(B_3) = 1/4$. $P(A|B_1) = P(A|B_2) = 0.02$ and $P(A|B_3) = 0.04$. Then

$$P(A) = P(A|B_1)P(B_1) + P(A|B_2)P(B_2) + P(A|B_3)P(B_3) = 0.025.$$

Hence there is a 2.5% chance that an item is defective.
b) Suppose that an item is defective. What is the chance that it is made in F1?
Let us apply Bayes Theorem.

$$P(B_1|A) = \frac{P(A|B_1)P(B_1)}{\sum_{i=1}^{k} P(A|B_i)P(B_i)} = \frac{0.02 * 1/2}{0.02 * 1/2 + 0.02 * 1/4 + 0.04 * 1/4} = 0.40.$$

Hence there is a 40% chance.

We say two events A and B are *independent* if $P(A \cap B) = P(A)P(B)$. Intuitively, this implies that occurrence and nonoccurence of A has no effect on the occurrence or non-occurrence of B. An example from Meyer [74] - its insightful.

Example 1.6.2 *We are rolling two dices and have three types of events.*
$A =\{$*The first die shows an even number*$\}$
$B =\{$*The second die shows an odd number*$\}$
$C =\{$*Both show an odd or both show an even number*$\}$
Observe that $P(A) = P(B) = P(C) = 1/2$. Note that $P(A \cap B) = 1/4 = P(A)P(B)$. $P(A \cap C) = 1/4 = P(A)P(C)$. $P(B \cap C) = 1/4 = P(B)P(C)$. But what about $P(A \cap B \cap C)$? Note that $P(A \cap B \cap C) = 0 \neq P(A)P(B)P(C)$.

Hence three events A, B and C are said to be *mutually independent* if and only if they are pairwise independent as well as $P(A \cap B \cap C) = P(A)P(B)P(C)$. In general n events are said to be mutually independent, if and only if all combinations of them are mutually independent.

A function X assigning every element of a sample space of an experiment to a real number is called a *random variable* (r.v.), i.e. $X : S \rightarrow \Re$. For example, X may count number of 1's in a random bit string of length n. A r.v. is called *discrete* if the number of possible values it can take is either finite or countably infinite. For each of those values, associate a real value between 0 and 1, i.e. its probability $P(x_i) = P(X = x_i)$. Each $P(x_i) \geq 0$ and $\sum_{i=1}^{\cdots} P(x_i) = 1$. The function p is called the *probability function* and the collection of pairs $(x_i, P(x_i))$ is called the *probability distribution* of X. For example, if X is a r.v. describing number of heads in three tosses of a coin, than the range of X is $\{0, 1, 2, 3\}$ and $P(0) = P(3) = 1/8$, $P(1) = P(2) = 3/8$. Consider the following example.

Example 1.6.3 *Suppose you want to generate several strings, each string consists of several a's followed by a single b. The random character generator spits out an a with probability $2/3$ and b with probability $1/3$. The string is generated by repeatedly invoking the 'sputter' till it outputs the first b. Let X be the random variable denoting the number of times the sputter is invoked. Note that X can take values $1, 2, 3, \ldots$. Observe that $P(X = 1) = 1/3$, $P(X = 2) = 2/3 * 1/3$, and $P(X = i) = (2/3)^{i-1}1/3$. Note that $\sum_{i=1}^{\infty}(2/3)^i = 1$.*

The classical *Binomial distribution* is defined as follows. Consider an experiment, where A is an event. Let $P(A) = p$ and $P(\bar{A}) = q = 1 - p$. Let us repeat the experiment n times and define a random variable X indicating the number of times A occurs. It is assumed that each experiment is independent of others. What is probability that $X = k$, for some $1 \leq k \leq n$? It is easy to see that $P(X = k) = \binom{n}{k} p^k q^{n-k}$. X is called a *binomial random variable* with parameters p and n. Individual experiments (trials) are called *Bernoulli trials.*

A r.v. X is said to be *continuous* if there exists a function f satisfying (a) $f(x) \geq 0$, for all x. (b) $\int_{-\infty}^{+\infty} f(x)dx = 1$ and (c) For all $-\infty < a < b < +\infty$, $P(a < x < b) = \int_a^b f(x)dx$, i.e. the area of the curve under $f(x)$ between $x = a$ and $x = b$.

Expected value of a discrete r.v. X is defined as $E[X] = \sum_{i=1}^{\cdots} x_i P(x_i)$. If this series converges, than $E[X]$ is also called the mean value of X.

Expected value of a continuous r.v. X is defined analogously, i.e., $E[X] = \int_{-\infty}^{+\infty} x f(x)dx$. For example, consider X to be uniformly distributed over the interval $[a, b]$. We know that the probability distribution function $f(x) = \frac{1}{b-a}$, $a \leq x \leq b$. Hence $E[X] = \int_a^b \frac{x}{b-a}dx = (a + b)/2$.

The *variance* of a r.v. X is defined to be $V[X] = E[X - E[X]]^2$. Let us calculate the variance of the r.v. that is uniformly distributed over the interval $[a, b]$. We know that $E[X] = (a + b)/2$. Note that $E[X^2] = \int_a^b \frac{x^2}{b-a}dx = \frac{b^3-a^3}{3(b-a)}$. Since $V[X] = E[X^2] - [E[X]]^2$ (see Exercises), we obtain $V[X] = \frac{b^3-a^3}{3(b-a)} - (\frac{a+b}{2})^2 = \frac{(b-a)^2}{12}$.

Now let us look at a famous inequality due to Chebyshev.

Theorem 1.6.4 *Let X be a random variable and let c be a real number. If $E[X - c]^2$ is finite and $\epsilon > 0$, then*

$$P(|X - c| \geq \epsilon) \leq \frac{1}{\epsilon^2} E[X - c]^2.$$

Proof. Let X be a continuous r.v. Let $R = \{x : |x-c| \geq \epsilon\}$. Note that $P(|X-c| \geq \epsilon) = \int_R f(x)dx$. Observe that $|x - c| \geq \epsilon$ implies $\frac{(x-c)^2}{\epsilon^2} \geq 1$. Thus

$$P(|X - c| \geq \epsilon) \leq \int_R \frac{(x-c)^2}{\epsilon^2} f(x)dx \leq \int_{-\infty}^{+\infty} \frac{(x-c)^2}{\epsilon^2} f(x)dx = \frac{1}{\epsilon^2} E[X - c]^2.$$

∎

Next, lets talk about Normal Distributions, as we will need them in the chapter on Dimensionality Reduction. Random variable X has a *normal distribution* if its probability density function is of the form

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\lceil\frac{x-\mu}{\sigma}\rceil^2}, -\infty < x < \infty.$$

Usuall it is denoted by $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean (expected value) and σ is the standard deviation, i.e. positive squareroot of the variance. It is also referred to as Gaussian or bell-shaped distribution. It is easy to see that it is a valid distribution, as $f(x) > 0$ for all values of x and $\int f(x)dx = 1$. The function f is symmetric around $x = \mu$. If we trace the boundary of the function, it changes from being convex to concave, and these points of inflection are at $x = \mu \pm \sigma$. The distribution $\mathcal{N}(0, 1)$ is referred to as a *standardized normal distribution.* If X is $\mathcal{N}(\mu, \sigma^2)$ and $Y = aX + b$ than Y is $\mathcal{N}(a\mu + b, a^2\sigma^2)$.

Lastly, a bit on law of large numbers. Suppose we repeat an experiment multiple times. Then the relative frequency of occurrence of an event should converge to its actual probability, if the experiment is repeated for sufficiently large number of times. For example, a factory is producing items. We don't know what is the failure probability. Then one way to estimate this probability is to take a large sample and see what percentage are faulty. This percentage will be a good estimate of the failure probability. Of course this should be taken with a grain of salt. This really depends upon what kind of sample is chosen, etc. Essentially what we are heading towards is that if the elements in the sample are chosen randomly, than the percentage of faulty items will be a true indicator of failure probability.

Consider a particular event A in an experiment. This experiment is repeated n times, and each run is independent of other runs. Let n_A be the number of times A occurs in the runs. Let $f_A = n_A/n$. Let $P(A) = p$. For a positive $\epsilon > 0$,

$$P(|f_A - p| \geq \epsilon) \leq \frac{p(1-p)}{n\epsilon^2},$$

or equivalently,

$$P(|f_A - p| < \epsilon) \geq 1 - \frac{p(1-p)}{n\epsilon^2}.$$

This can be shown as follows. Observe that n_A is a binomially distributed random variable. $E[n_A] = np$ and $V[n_A] = np(1-p)$. Since $f_A = n_A/n$, $E[f_A] = p$ and $V[f_A] = p(1-p)/n$. Recall Chebyshev's inequality (see Exercise 1.16) which states that $P(|X - \mu| < \epsilon) \geq 1 - \frac{Var(X)}{\epsilon^2}$. Substituting $X = f_A$, $p = \mu$, we obtain

$$P(|f_A - p| < \epsilon) \geq 1 - \frac{V[f_A]}{\epsilon^2}.$$

Set $\epsilon = k\sqrt{p(1-p)/n}$ and $V[f_A] = p(1-p)/n$, and we obtain

$$P(|f_A - p| < \epsilon) \geq 1 - \frac{p(1-p)}{n\epsilon^2}.$$

Note that

$$\lim_{n \to \infty} P(|f_A - p| < \epsilon) = 1.$$

This is sort of the meaning that the relative frequency converges to $P(A)$, when the experiment is repeated for a large number of times.

Here is a different type of question. How many times should we repeat the experiment, so that the relative frequency f_A differs from $P(A)$ by at most 0.01 with probability at least 0.9? We need to choose n so that for $\epsilon = 0.01$, $1 - \frac{p(1-p)}{n\epsilon^2} = 0.9$. This implies that $n = \frac{p(1-p)}{0.1\epsilon^2}$. For example if $p = 1/2$, than $n = 25000$. What this means is that if we toss a coin 25000 times, than we are 90% sure that the relative frequency of getting a head is within 0.01 of the theoretical probability.

Lastly, we state a theorem (without proof), which is a special case of the Central Limit Theorem (see Meyer [74]). Intuitively, the following theorem states that the arithmetic mean of n observations from the same random variable, has for large n, a normal distribution.

Theorem 1.6.5 *Let $X_1, X_2, \cdots, X_n$ be n independent r.v. all of which have the same distribution. Let $\mu = E[X_i]$ and $\sigma^2 = V[X_i]$ be their common expectation and variance, respectively. Define a r.v. $S = \sum_{i=1}^{n} X_i$. Then $E[S] = n\mu$ and $V[S] = n\sigma^2$. Moreover, for large n, $T_n = \frac{S - n\mu}{\sqrt{n}\sigma}$ has essentially the distribution $\mathcal{N}(0,1)$.*

Intuitively, the above theorem states that the arithmetic mean of n observations from the same random variable, has for large n, a normal distribution. Here is an example from Meyer's [74] to get some idea of this theorem.[1]

Example 1.6.6 *Consider a box containing three types of balls - 20 balls labelled with a zero, 30 with a one and 50 with a two. In this experiment, we draw a random ball, note its label, and then place it back. We will repeatedly pick the balls, and eventually report the average of their labels. We are interested in understanding the distribution of the averages when we repeat this experiment for a large number of rounds. The above theorem claims that the average behaves like a normal distribution. Let X_i be the label of the ball drawn in Round i of this experiment. We are interested in the random variable $M_i = \frac{1}{i}\sum_{k=1}^{i} X_k$, denoting the averages of the labels drawn up to and including the Round i. Note that each X_i takes values $0, 1,$ and 2, with probabilities $0.2, 0.3$ and 0.5, respectively. Observe that $M_1 = X_1$ and it takes values $0, 1$ or 2 with probability $0.2, 0.3$ and 0.5, respectively. Next let us look at M_2.*

$m = \frac{X_1+X_2}{2}$	*0*	*1/2*	*1*	*3/2*	*2*
$P(M_2 = m)$	*0.04*	*0.12*	*0.29*	*0.30*	*0.25*

$m = \frac{X_1+X_2+X_3}{3}$	*0*	*1/3*	*2/3*	*1*	*4/3*	*5/3*	*2*
$P(M_3 = m)$	*0.008*	*0.036*	*0.114*	*0.207*	*0.285*	*0.225*	*0.125*

Hopefully, this example convinces us that for large values of n, M_n converges to a Normal distribution.

1.6.1 Chernoff Bounds

Suppose we toss a fair coin 1000 times. We expect about 500 tails. What is the probability that we will get over 750 tails? or over 900 tails? Chernoff bounds help us in determining probabilities of extreme events in a large collection of independent events. In this section we will establish these bounds. This section is derived from [43] and notes of Michiel Smid. First let us establish Markov's inequality.

Theorem 1.6.7 *Let X be a non-negative discrete r.v. and $s > 0$ be a constant. Then $P(X \geq s) \leq E[X]/s$.*

Proof. Note that $E[X] = \sum_{i=0}^{\infty} i.P(X = i) \geq \sum_{i=s}^{\infty} i.P(X = i) \geq s\sum_{i=s}^{\infty} P(X = i) = sP(X \geq s)$. Hence $P(X \geq s) \leq E[X]/s$. ■

Let $X_1, \cdots X_n$ be identical 0-1 independent r.v's, such that $P(X_i = 1) = p_i$ and $P(X_i = 0) = 1 - p_i$. Define $X = \sum_{i=1}^{n} X_i$ and let $m = \sum_{i=1}^{n} p_i$. Note that

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p_i = m.$$

[1]BTW, I never 'really' understood the proof of this theorem, except that I can verify each of the steps in the proof. I hope one day I will get it!

Suppose we have a fair coin, which we toss n times. We define 0-1 r.v's $X_1, \cdots, X_n$, where $X_i = 1$ if the i-th toss was a head otherwise $X_i = 0$. $P(X_i = 0) = P(X_i = 1) = 1/2$. Let $X = \sum_{i=1}^n X_i$. Now $E[X] = n/2 = m$. We are interested in estimating the probability of X deviating from $(1 \pm \epsilon)m$, for $0 < \epsilon < 1$. From Markov's inequality we obtain $P(X > (1+\epsilon)m) \leq 1/(1+\epsilon)$. We will show the following, which are collectively known as *Chernoff bounds* in the algorithms community.

$$P(X \geq (1+\epsilon)m) \leq \exp(-\epsilon^2 m/3)$$

$$P(X \leq (1-\epsilon)m) \leq \exp(-\epsilon^2 m/2).$$

Let $t > 0$. Let us first deal with proving $P(X \geq (1+\epsilon)m) \leq \exp(-\epsilon^2 m/3)$.

$$\begin{aligned} P(X \geq (1+\epsilon)m) &= P(e^{tX} \geq e^{t(1+\epsilon)m}) && (1.1)\\ &= e^{-t(1+\epsilon)m} e^{t(1+\epsilon)m} P(e^{tX} \geq e^{t(1+\epsilon)m}) && (1.2)\\ &\leq e^{-t(1+\epsilon)m} E[e^{tX}] && (1.3)\end{aligned}$$

Equation 1.3 follows from Markov inequality applied to $P(e^{tX} \geq e^{t(1+\epsilon)m}) \leq e^{-t(1+\epsilon)m}E[e^{tX}]$. Observe that $E[e^{tX}] = E[e^{t\sum_{i=1}^n X_i}] = E[\Pi_{i=1}^n e^{tX_i}] = \Pi_{i=1}^n E[e^{tX_i}]$, as X_i's are independent. $E[e^{tX_i}] = p_i e^t + (1-p_i)e^0$. Hence $E[e^{tX}] = \Pi_{i=1}^n (p_i e^t + (1-p_i)) = \Pi_{i=1}^n (1 + p_i(e^t - 1)) \leq \Pi_{i=1}^n e^{p_i(e^t-1)} = e^{\sum_{i=1}^n p_i(e^t-1)} = e^{m(e^t-1)}$. (Note that for this we used the fact that $e^x \geq 1 + x$.) Hence,

$$P(X \geq (1+\epsilon)m) \leq e^{-t(1+\epsilon)m + m(e^t-1)}.$$

This expression holds for all values of $t \in R$. It is minimized for $t = \ln(1+\epsilon)$. To see this, one can write $e^{-t(1+\epsilon)m+m(e^t-1)} = [e^{-t(1+\epsilon)+(e^t-1)}]^m$. To minimize this, one needs to minimize $-t(1+\epsilon) + (e^t - 1)$. Differentiate this with respect to t, and we obtain that $-(1+\epsilon) + e^t = 0$ or $t = \ln(1+\epsilon)$.

Thus,

$$P(X \geq (1+\epsilon)m) \leq (1+\epsilon)^{-(1+\epsilon)m} e^{m\epsilon} = \left(\frac{e^\epsilon}{(1+\epsilon)^{1+\epsilon}}\right)^m \leq e^{-\epsilon^2 m/3}.$$

To show the last inequality, one needs to prove that $\epsilon - (1+\epsilon)\ln(1+\epsilon) \leq -\frac{1}{3}\epsilon^2$, which is left as an exercise.

Next, we show that $P(X \leq (1-\epsilon)m) \leq \exp(-\epsilon^2 m/2)$. The proof is along the same lines as the previous one.

$$\begin{aligned} P(X \leq (1-\epsilon)m) &= P(m - X \geq \epsilon m) && (1.4)\\ &= P(e^{t(m-X)} \geq e^{t\epsilon m}) && (1.5)\\ &\leq e^{-t\epsilon m} E[e^{t(m-X)}] && (1.6)\\ &= e^{tm(1-\epsilon)} E[e^{-tX}] && (1.7)\end{aligned}$$

Note that

$$E[e^{-tX}] = \Pi_{i=1}^{n} E[e^{-tX_i}] \tag{1.8}$$
$$= \Pi_{i=1}^{n}[p_i e^{-t} + (1-p_i)] \tag{1.9}$$
$$. = \Pi_{i=1}^{n}[1 - p_i(1-e^{-t})] \tag{1.10}$$
$$\leq \Pi_{i=1}^{n}[e^{-p_i(1-e^{-t})}] \tag{1.11}$$
$$= e^{-\sum_{i=1}^{n} p_i(1-e^{-t})} \tag{1.12}$$
$$= e^{-m(1-e^{-t})} \tag{1.13}$$

Therefore,

$$P(X \leq (1-\epsilon)m) \leq e^{-m(1-e^{-t})} e^{tm(1-\epsilon)}.$$

This is minimized for $t = -\ln(1-\epsilon)$. Hence,

$$P(X \leq (1-\epsilon)m) \leq (1-\epsilon)^{-m(1-\epsilon)} e^{-m\epsilon} = \left[\left(\frac{1}{1-\epsilon}\right)^{1-\epsilon} e^{-\epsilon}\right]^{m} \leq e^{-\epsilon^2 \frac{m}{2}}.$$

1.7 Exercises

1.1 *Provide a proof for the simplified Masters theorem.*

1.2 *Present examples for each of the three cases and present an example where this theorem is not applicable.*

1.3 *This is from [21] and is based on Divide-and-Conquer Multiplication. (Do not use FFTs as such for this)*

1. *Show how to multiply two polynomials of degree 1, namely $ax + b$ and $cx + d$ using only three multiplications. (Note that $(a + b).(c + d)$ is considered as 1 multiplication.)*
2. *Give a divide-and-conquer algorithm for multiplying two polynomials of degree n that runs in $\Theta(n^{\log_2 3})$. You may think of dividing the coefficients into a high half and a low half, or in terms of whether the index is even or odd.*
3. *Show that two n-bit integers can be multiplied in $O(n^{\log_2 3})$ steps, where each step operates on at most a constant number of 1-bit values.*

1.4 *Show that $1^d + 2^d + 3^d + ... + n^d$ is $O(n^{d+1})$ for a constant d? What is the value of c and n_0.*

1.5 *Solve the recurrence relation*

$$T(n) = T(xn) + T((1-x)n) + cn$$

in terms of x and n where x is a constant in the range $0 < x < 1$. Is the asymptotic complexity the same when $x = 0.5, 0.1$ and 0.001? What happens to the constant hidden in the O() notation.

1.6 *Analyze the recurrence relation*

$$T(n) = \sqrt{n}T(\sqrt{n}) + n.$$

(This is called as the rootish-divide-and-conquer and for example plays an important role in parallel computing.)

1.7 *V. Pan has discovered a way to multiply two* $70*70$ *matrices using only* 143640 *multiplications. Ignoring the additions, what will the asymptotic complexity of Pan's algorithm for multiplying two* $n*n$ *matrices? Is it better than Strassen's? Justify your answer.*

1.8 *Suppose we have* n *real numbers, where no two are the same. We want to report the smallest* i *numbers in the sorted order, where* $i < n$*. Which algorithm you think is the best option (Justify your answer)?*
A. Sort the numbers and list the first i*.*
B. Build a priority queue and then call Extract-Min i *times.*
C. Use the order statistics to find the i*-th smallest number, partition the set according to this value, and then sort the* i *smallest numbers.*

1.9 *Let* A *and* B *be two arrays, each consisting of* n *distinct elements in sorted order (in an increasing order). Report the median of the set* $A \cup B$ *in* $O(\log n)$ *time.*

1.10 *Given two binary strings* $a = a_0a_1 \ldots a_p$ *and* $b = b_0b_1 \ldots b_q$*, where each* a_i *and* b_j *are either 0 or 1. We say that* $a \leq b$ *if either of the following holds*
(1) there exists an integer j*,* $1 \leq j \leq \min(p, q)$*, such that* $a_i = b_i$ *for all* $i = 0, 1, ..., j-1$ *and* $a_j < b_j$*.*
(2) $p < q$ *and* $a_i = b_i$ *for all* $i = 0, 1, 2, , , , p$*.*
Let $A \subseteq \Sigma^*$ *be a set of distinct binary strings whose lengths sums up to* n*. Present an algorithm that can sort the binary strings in* $O(n)$ *time. (All the strings are not of the same length!)*

1.11 *Let* X *be a binomially distributed r.v. with parameters* n *and* p*. Then* $E[X] = \sum_{k=0}^{n} k\binom{n}{k}p^k(1-p)^{n-k} = np$*.*

1.12 *Let* X *and* Y *be two r.v. Show that* $E[X+Y] = E[X] + E[Y]$*.*

1.13 *Let* X *and* Y *be two independent random variables. Show that* $E[XY] = E[X]E[Y]$*.*

1.14 *Show that the variance of a r.v.* $V[X] = E[X^2] - [E[X]]^2$*.*

1.15 *If* X *and* Y *are independent r.v. than* $V[X+Y] = V[X] + V[Y]$*.*

1.16 *Let* $E[X] = \mu$ *and* $\epsilon > 0$*. Show that* $p(|X - \mu| \geq \epsilon) \leq \frac{V[X]}{\epsilon^2}$*.*

1.17 *For any value of* x*,* $0 < x < 1$*, show that* $x - (1+x)\ln(1+x) + \frac{1}{3}x^2 \leq 0$*.*

Chapter 2

Introduction to Graphs and their Representation

2.1 Introduction and Definitions

Graphs were discovered by Euler (Königsberg bridge problem[1]), Kirchoff (electrical networks[2]) and Cayley (enumeration of organic chemical isomers[3]) in different contexts. Graphs are combinatorial structures used in computer science. Lists, Trees, Directed Acyclic Graphs, Flow Charts, Control Flow Graphs, Planar Graphs, web, unit disk graphs, and communication networks are examples of graphs that are widely used in computer science. Most often, practical problems, can be cast into some sort of graph problem. Examples include the Traveling Salesperson problem (finding a route of the cheapest cost through many cities), or coloring a map so that no two neighboring countries receive the same color or finding shortest path from Carleton to National Art Gallery, or navigating hyperlinks in webpages. There are excellent books and thousands of papers discussing several aspects of graphs (definitions, connectivity, coloring, independent sets, matchings, Kuratowski's theorem, four color theorem, ...). Some of the classical books include the book by Harary [45], Bollobas [11], Bondy and Murty [12], Lovasz [68]. We need to get used to some of the basic definitions.

[1]: Leonhard Euler, 1707-1783

[2]: Gustav Kirchoff, 1824-1887: At every node an electrical circuit the s of all currents should be equal to zero, i.e., the charge cannot accumulat a node - or what comes i must go out!

[3]: Arthur Cayley, 1821-1895

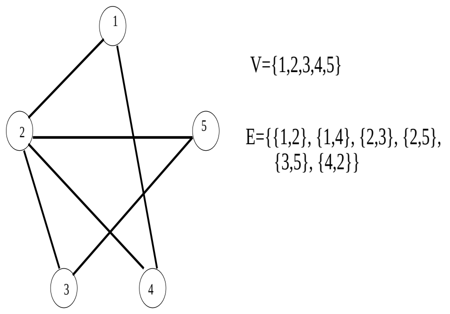


Figure 2.1: An example of a undirected graph. The edge {1,4} is incident to the vertex labelled 1 and to the vertex labelled 4. The degree of vertex 1 is 2, degree of vertex 2 is 4. A path from the vertex $u = 1$ to the vertex $v = 5$ consists of vertices $< u = 1, 4, 2, 5 = v >$. This graph is connected and has only one connected component. This graph is simple.

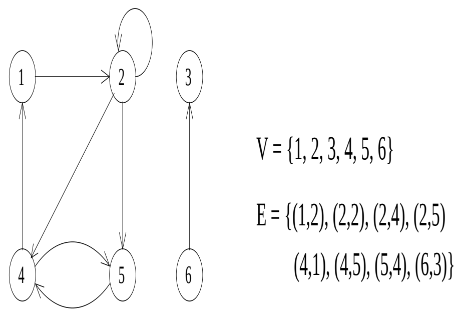


Figure 2.2: An example of a directed graph. This graph is not strongly connected since there is no way to reach from the vertex labelled 3 to the vertex labelled 4 or from the vertex labelled 1 to the vertex labelled 6. The strongly connected components are {{1,2,5,4},{3},{6}}.

Graph A graph $G = (V, E)$ consists of a finite set of *vertices* V and a finite set of *edges* E.

- *Undirected graph*: E is a set of unordered pairs of vertices $\{u, v\}$ where $u, v \in V$ (see Figure 2.1).
- *Directed graph*: E is a set of ordered pairs of vertices (u, v) where $u, v \in V$ (see Figure 2.2).

Incidence An edge $\{u, v\}$ is *incident* to u and v.

Degree of vertex in undirected graph is the number of edges incident to it.

In (Out) degree of a vertex in directed graph is the number of edges entering (or leaving) it.

Path A *path* from u to v is a sequence of vertices $< u = v_0, v_1, v_2, \cdots, v_k = v >$ such that $(v_i, v_{i+1}) \in E$ (or $\{v_i, v_{i+1}\} \in E$)

- We say that v is reachable from u
- The *length* of the path is k
- It is a *cycle* if $u = v$

Connected An undirected graph is connected if every pair of vertices are joined by a path.

Component The connected components are the equivalence classes of the vertices under the "reachability" relation.

Strongly Connected A directed graph is *strongly connected* if every pair of vertices are reachable from each other.

Strongly connected components The *strongly connected components* are the equivalence classes of the vertices under the "mutual reachability" relation.

Simple connected undirected graph An undirected connected graph is called simple, if between every pair of vertices there is at most one edge, and no vertex contains a self loop (i.e. a vertex connected to itself by an edge). In this course, a graph specified without any qualifications is an undirected, connected, and a simple graph!

Complete Graphs An undirected graph is called a complete graph, if every pair of (distinct) vertices are joined by an edge. Examples include K_1 (just a single vertex), K_2 (a pair of vertices joined by an edge), K_3 (a triangle), K_5 (graph on five vertices), ... K_5 is the smallest (in terms of vertices) non-planar graph (i.e. no matter how one draws it in the plane, there is a crossing). See Figure 2.3.

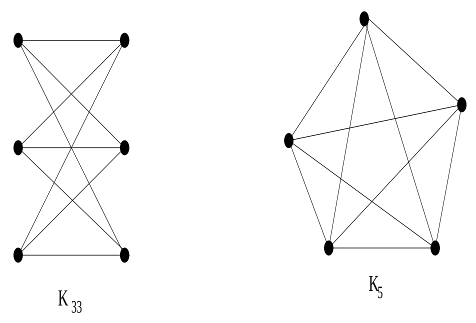


Figure 2.3: A complete bipartite graph K_{33} and a complete graph K_5.

Bipartite Graphs A graph is called bipartite if the vertex set V can be partitioned into two subsets $S \cup T = V$, such that for any edge $\{a, b\}$, $a \in S$ and $b \in T$. A bipartite graph is complete if every vertex in S is connected to each vertex in T by an edge. For example, K_{mn} refers to a complete bipartite graph consisting of vertices $V = S \cup T$, where $|S| = m$ and $|T| = n$. Interestingly $K_{3,3}$ is the smallest graph (in terms of edges) which is non-planar.

Kuratowski's Theorem A graph is planar if and only if it has no subgraph homeomorphic to K_5 or $K_{3,3}$.[4] Two graphs are homeomorphic if both can be obtained from the same graph by a sequence of subdivisions of edges (insertion of a vertex on an edge). For example any two cycles are homeomorphic.

[4]: This is one of the fundamental theorems in Graph Theory.

2.2 How to represent graphs in a computer?

There are two standard ways of representing graphs in computers: Adjacency list and Adjacency Matrix. Let $G = (V, E)$ be the graph under consideration (assume that it is undirected - for directed the same representation works as well).

In the adjacency matrix representation, we form a $|V| \times |V|$ matrix A of 0s and 1s, where ij-th entry is 1 if and only if there is an edge from vertex i to vertex j. It is easy to see that this matrix will be symmetric for undirected graphs. Also given a pair of vertices v_i and v_j, it takes constant time to check whether there is an edge joining them by inspecting the ij-th entry in the matrix A. Moreover, this representation is independent of the number of edges in G. The main drawback is that this representation requires $O(|V|^2)$ memory space whereas the graph G may have very few edges! Just for fun and to get some insight, try to see what it means by taking products $A \times A$ or $A \times A \times A$,..., where the '.' refers to the boolean AND and '+' refers to the boolean OR? Try it for small graphs. We will come back to that later!

The other most common representation is the adjacency list representation. The adjacency list for a vertex v is a list of all vertices w that are adjacent to v. To represent the graph we have in all $|V|$ lists, one for each vertex. This representation requires optimal storage, i.e. $O(|V| + |E|)$.

But to check whether the two vertices v and w are connected, we need to check in lists of v (or w) whether the vertex w (resp. v) is present. Searching in a list requires, in the worst case, time proportional to the size of the list, and hence searching will require $O(\min\{\text{degree}(v), \text{degree}(w)\})$ time.

2.3 Graph traversal

Once we have a graph, represented inside the computer, what do we do with it? When we go to a new (small!) city what is the best way to explore all the bars? What is the best way to search for a particular web page just following the hyperlinks (assume no search engine and assume that we can go from any web page to any other web page)? How to solve those maze problems?

- There are two standard and simple ways of traversing all vertices/edges in a graph in a systematic way
  - Breadth-first search (bfs)
  - Depth-first search (dfs)
- They are used in many fundamental algorithms as a preprocessing step.

2.3.1 Breadth-first search (BFS)

- Main idea (see Figure 2.4):
  - Start at a source vertex and visit,
  - All vertices at distance 1 (i.e. vertices that are neighbors of source),
  - Followed by all vertices at distance 2 (neighbors of neighbors of source),
  - Followed by all vertices at distance 3 (neighbors of neighbors of neighbors of source),

    $\vdots$
- BFS corresponds to computing *shortest path* distance (number of edges) from s to all other vertices in the graph.

Here is a pseudo-code of a BFS algorithm.

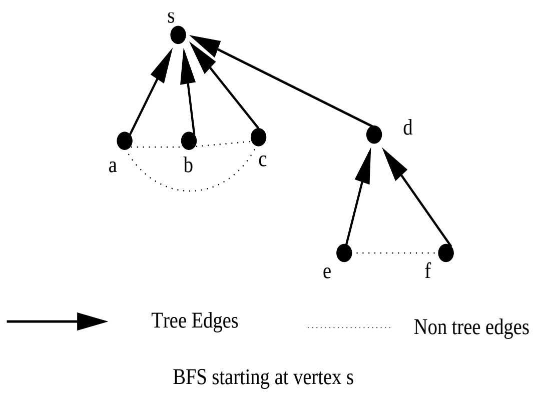


Figure 2.4: Illustration of BFS.

```
procedure BFS(v)
Input: A simple connected undirected graph G=(V,E), in adjacency list representation,
       and the start (source) vertex v.
Output : A BFS tree T, where each vertex u knows its parent p(u).

Variables: Q: Queue of vertices, maintained in the FIFO order
           x, y: vertex.
Begin
   T := empty; // BFS Tree initialized as empty.
   mark[v] := visited;
   Enqueue (v,Q); //Insert v in the Queue Q.
   While not empty (Q) do
      x:=front(Q); // Set x to be the first vertex in Q.
      Dequeue(Q); // Remove that vertex from Q.
      For each vertex y adjacent to x do
          if mark[y]=unvisited then
              mark[y]:= visited;
              Enqueue(y,Q);
              Insert((x,y),T);//Insert the directed edge (x,y) in the tree T.
          end if
      end for
    end while
End
```

It is easy to see that this algorithm runs in $O(|V| + |E|)$ time since each vertex is inserted (enqueued) once in the queue Q and each edge $\{x, y\}$ is explored twice, once when the vertex x is dequeued from Q and once when the vertex y is dequeued. Insertion and Deletion of a vertex in Q can be achieved in constant time since Q is a FIFO Queue, and can be maintained as a doubly-connected list. Also adjacency list representation will suffice and the correctness is left as an assignment problem. We can summarize this as follows:

Theorem 2.3.1 *Let $G = (V, E)$ be a simple graph. A breadth-first search traversal of G, and its corresponding tree, can be computed in $O(|V| + |E|)$ time.*

2.4 Topological sort and DFS

2.4.1 Topological Sort

Let $G = (V, E)$ be a directed acyclic graph (DAG) on the vertex set V with directed edge set E. In a DAG, there are no directed cycles. But, between a pair of nodes, there may be multiple directed paths. A *topological sort* of a DAG is a linear ordering of all its vertices such that if G contains a directed edge (u, v), then u appears before v in the ordering. One can think of this process as assigning a number $f : V \rightarrow \{1, \cdots, |V|\}$ to each vertex such that for every directed edge (u, v), $f(u) < f(v)$ (see Figure 2.5). We will sketch two algorithms, first a slower one followed by an

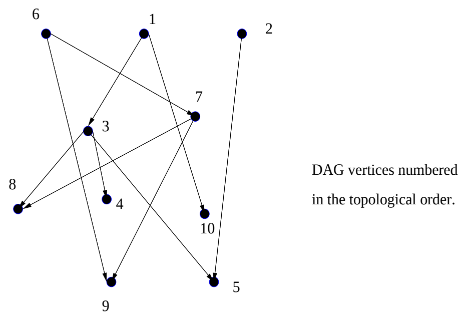


Figure 2.5: Illustration of a Topological Sort

optimal one. You need to verify the correctness as well as the complexities of both of them. Try to decide yourself what should be a suitable graph representation for these algorithms.

```
Algorithm 1: Topological sort in $O(|V|^2)$ time.
Step 1:  Start from any vertex and follow edges backwards until a vertex $v$ is found, such that
         $v$ has no incoming edges.
Step 2:  Make $v$ the next vertex in the total order.
Step 3:  Delete $v$ and all of its outgoing edges.
Step 4:  If the graph is non-empty, go to Step 1.
```

Observe that in Step 1 we will find a vertex v, having no incoming edges, as DAGs are acyclic and have a finite number of vertices. Moreover a vertex is assigned a number when it has no incoming edges. This should be sufficient to prove that the generated linear ordering of vertices satisfy the requirements of topological sort.

Algorithm 2: Topological sort in $O(|V| + |E|)$ time using adjacency list representation, where for each vertex maintain a separate list of incoming edges and outgoing edges.

Step 1: Form a queue Q of vertices which have no incoming edges.

Step 2: Pick a vertex v from Q, and make v the next vertex in the order.

Step 3: Delete v from Q and delete all of its outgoing edges. Let (v, w) be an outgoing edge. If the list of incoming edges of w becomes empty then insert w in Q.

Step 4: If Q is not empty then GOTO Step 2.

Which invariant(s) are maintained by Algorithm 2? Why is it correct? Why does it run in $O(|V| + |E|)$ time?

2.4.2 Depth First Search

Depth First Search (DFS) is another way of exploring a graph. Like BFS, DFS traversal will take linear time, will produce a DFS spanning tree and this tree will posses very interesting, useful and beautiful properties.

Assume that we have an undirected connected simple graph $G = (V, E)$. Informally DFS on G performs the following steps:

1. Select a vertex v of G which is initially unvisited.
2. Make v visited.
3. Each unvisited vertex adjacent to v is searched in-turn using DFS recursively.

DFS partitions the edges in G into two sets, the set of DFS spanning tree edges, say T, and the set of back edges, say B, where $E = T \cup B$, and $T \cap B = \emptyset$. Next we formally describe the algorithm of Aho, Hopcroft, Ullman [1] for DFS. Each vertex in G will be assigned a DFS-number, i.e., the order in which they are first visited in the DFS (see Figure 2.6).

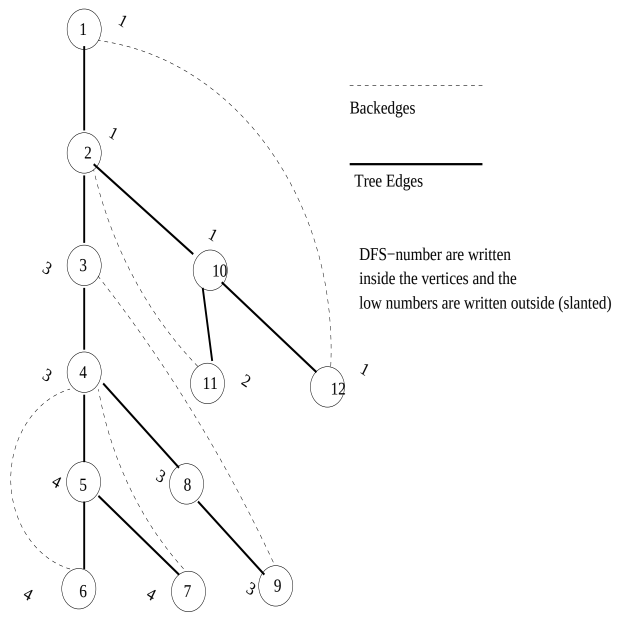

Backedges
Tree Edges
DFS−number are written
inside the vertices and the
low numbers are written outside (slanted)

Figure 2.6: Illustration of dfs and low-numbers.

DFS Algorithm

Input: A (undirected simple connected) graph $G = (V, E)$, represented by adjacency list $L[v]$ for each vertex $v \in V$.

Output: Partition of $E = T \cup B$. Tree edges are given as directed edges from a child to its parent. All edges not in T are considered to be in B. DFS-number, an integer in the range $1..|V|$, assigned to each vertex.

```
1. $T := \emptyset$; COUNT:=1;
2. for all $v \in V$ do mark $v$ as *unvisited*;
3. while there exists an unvisited vertex do SEARCH(v)
```

procedure SEARCH(v)

```
1. mark $v$ as *visited*;
2. DSF-number[v]:=COUNT;
3. COUNT:=COUNT+1;
4. for each vertex $w$ on $L[v]$ do
   if $w$ is unvisited then
```

```
        (a) add $(w, v)$ to $T$; /*edge $(w, v)$ is a DFS tree edge */
        (b) SEARCH(w); /* Recursive call */
```

Complexity Analysis: Why does the above algorithm runs in $O(|V| + |E|)$ time?
We call the procedure SEARCH(v), $|V|$ times, once for each vertex. The total running time of SEARCH(v), exclusive of the recursive calls, is proportional to the degree of v. Hence the total time complexity is $O(|V| + \sum_{v \in V}(degree(v))) = O(|V| + |E|)$.

Property of Back edges: If $\{w, v\} \in B$ is a back edge, then either w is an ancestor of v or v is an ancestor of w in the DFS tree T. Why?
Suppose, without loss of generality, v has a lower DFS-number than w, i.e., the vertex v is visited before the vertex w. Therefore, when SEARCH(v) is invoked, the vertex w is labeled *unvisited*. All the *unvisited* vertices visited by SEARCH(v) will become descendants of v in the DFS tree. Therefore, w will become descendant of v, since $w \in L[v]$ and each vertex in $L[v]$ is looked at while executing SEARCH(v).

2.4.3 Computation of $low(v)$

We introduce a quantity, called $low(v)$, for each vertex $v \in V$ with respect to the DFS tree T and the back edges B. This quantity will be used in checking whether a graph is biconnected and finding its biconnected components. We will deal with biconnectivity in the next section.

Let us first define $low(v)$. Relabel the vertices of G by their DFS-numbers. For each vertex $v \in V$, define $low(v)$ as follows:
$low(v) = MIN(\{v\} \cup \{w |$there exists a back edge $(x, w) \in B$ such that x is a descendant of v and w is an ancestor of v in the DFS tree$\})$

Intuitively, $low(v)$ is trying to capture the following. Consider the subtree T_v of the DFS-tree T, rooted at the vertex v. What is the vertex closest to the root of T which we can reach by using back edges emerging in T_v, and going to the ancestors of v? If there are no back edges going out of T_v, then $low(v) = v$; otherwise it is the minimum (i.e. closest to the root) among the set of ancestors of v, which are joined by back edges from the vertices in T_v.

To compute $low(v)$, we will compute three quantities. These quantities can be computed by simple modification to the DFS algorithm. The three quantities are

1. $w = v$; i.e. the case when there are no back edges going out of the subtree T_v.
2. $w = low(c)$ and c is a child of v; i.e. the case when $low(v)$ is the same as *low* value of one of its children.
3. (v, w) is a back edge in B; i.e. the back edges associated to vertex v itself.

Then, the $low(v)$ value is given by
$low(v) = MIN(\{v\} \cup \{low(c) |$ c is a child of $v\} \cup \{w | (v, w) \in B\})$.

The modified SEARCH(v) procedure that computes the low values is as follows:

```
procedure SEARCH(v)
    1. mark $v$ as *visited*;
    2. DSF-number[v]:=COUNT;
    3. COUNT:=COUNT+1;
    4. $low(v)$:=DFS-number[v]; /* $low(v)$ is at least equal to the DFS-number of $v$ */
    5. for each vertex $w$ on $L[v]$ do
       if $w$ is unvisited then
          (a) add $(w, v)$ to $T$; /*edge $(w, v)$ is a DFS tree edge */
          (b) SEARCH(w); /* Recursive call */
          (c) $low(v) := \min(low(v), low(w))$ /*Compare the low value of $v$ with its child $w$ */
       else if $w$ is not the parent of $v$ then
       $low(v) := \min(low(v),$ DFS-number$[w])$; /* $(v, w)$ is a back edge */
```

Given that the DFS algorithm runs in $O(|V| + |E|)$ time, it is easy to see that this algorithm runs within the same time complexity.

2.5 Biconnectivity

2.5.1 Equivalence Relation

Before we talk about biconnectivity, we need to recall what is an equivalence relation.

Relation Let A and B be finite sets. A binary relation R from A to B is a subset of the cross product of A and B, i.e. $R \subseteq A \times B$. A relation on a set A is a relation from A to A. Example: Let $A = \{1, 2, 3, 4\}$. Let $R = \{(a, b) | a$ divides b, where $a, b \in A\}$, i.e. $R = \{(1, 1), (2, 2), (3, 3), (4, 4), (1, 2), (1, 3), (1, 4), (2, 4))\}$.

Reflexive A relation R on A is reflexive if $(a, a) \in R$ for every element $a \in A$. The relation in the divide example is reflexive.

Symmetric A relation R on A is called symmetric if $(b, a) \in R$ whenever $(a, b) \in R$, where $a, b \in A$. The relation in the divide example is not symmetric.

Transitive A relation R on A is called transitive if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$, for $a, b, c \in A$. The relation in the divide example is transitive.

Equivalence Relation A relation on a set A is an equivalence relation if it is reflexive, symmetric, and transitive.

Equivalence Classes Let R be an equivalence relation on a set A. The set of all elements that are related to an element $a \in A$ is called the equivalence class of a, denoted by $[a]$.

Property of Equivalence Classes Let R be an equivalence relation on A. Then if $(a, b) \in R$, then $[a] = [b]$.

Partition of A Let R be an equivalence relation on A. Then the equivalence classes of R form a partition of A.

Example: Let R be a relation on the set of integers such that $(a, b) \in R$ if and only if $a = b$ or $a = -b$. The equivalence class of integer 4 will be $[4] = \{4, -4\}$. Similarly, $[7] = \{-7, 7\}$. Observe that since $(4, -4) \in R$, then $[4] = [-4] = \{4, -4\}$. Also observe that the set of integers can be partitioned by R as follows: $\{(-1, 1), (0), (-2, 2), (-3, 3), ...\}$.

There are many books in Discrete Mathematics discussing equivalence relations, for example, see Rosen [87].

2.5.2 Biconnectivity

Most of the material in this section is from Kozen [64] and Aho, Hopcroft and Ullman [1]. Assume that the graph $G = (V, E)$ is undirected and connected. We start with definitions.

Articulation vertex A vertex $v \in V$ is called an articulation vertex, if its removal disconnects the graph. Equivalently, vertex v is an articulation vertex if there exists vertices a and b, so that every path between a and b goes through v and a, b, and v are all distinct.

Biconnected A connected graph is biconnected if any pair of distinct vertices lie on a simple cycle (one with no repeated vertices). Equivalently, for every distinct triple of vertices v, a, and b, there exists a path between a and b not containing v. Observe that G is biconnected if and only if it has no articulation vertices. Note that a graph just consisting of a single edge (two vertices joined by an edge) is biconnected!

Relation on edges For edges $e, e' \in E$, define that the two edges are equivalent, $e \equiv e'$, if e and e' lie on a simple cycle.

Lemma 2.5.1 *The relation $\equiv$ defined above is an equivalence relation.*

Proof. [64]: To prove that it is an equivalence relation, we need to show that it is reflexive, symmetric and transitive.
Reflexive: Obviously $e \equiv e$, $\forall e \in E$, since an edge by itself lies on a cycle.

Symmetric: If $e \equiv e'$, then $e' \equiv e$, since they are on the same cycle.

Transitivity: Suppose that $e = (u, v) \equiv e' = (u', v')$ and $e' = (u', v') \equiv e'' = (u'', v'')$. We want to show that $e \equiv e''$. Suppose that the simple cycle c contains e and e' and the simple cycle c' contains e' and e''. See Figure 2.7. Suppose that u, u', v', v appear in that order around c. Let x be the first vertex in c from u to u' that also is in c'. Similarly let y be the first vertex in c from v to v' which also lies in c'. The vertices x and y exists, since u' and v' are in c' as well. Construct a new cycle c'' consisting of
(a) Path from x to y in c containing uv, and
(b) Path from x to y in c' containing $u''v''$. Observe that c'' is a simple cycle containing e and e'', and hence $e \equiv e''$.

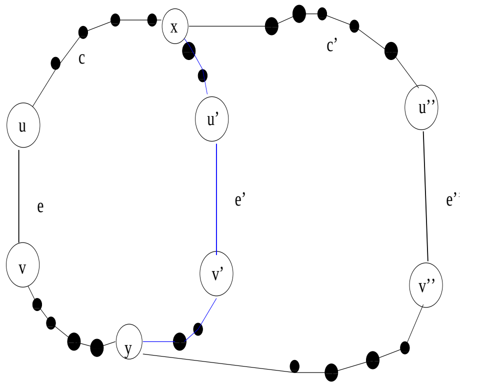


Figure 2.7: Illustration of Transitivity

Remark 2.5.2 *Professor Danny Sleator pointed out that the above proof is not complete. Here is an excerpt below (bit modified) from his e-mail:*
"... you claim to prove that the edges of a bi-connected component form an equivalence relation by the equivalence defined as edge (u, v) and (u', v') are equivalent if there is a simple cycle through them.

The proof is not really complete, at least from my perspective. The vertices x and y as you define them may not exist. E.g. suppose that the simple cycle using (u', v') and (u'', v'') only overlaps the other cycle on the bottom path from v to v'. (At the very least there are cases you are not considering.)

Here's how to correct the proof. Consider edges a, b, and c. Say a and b are on a simple cycle X and a and c are on a simple cycle Y. If c is also on the cycle X then we're done because b is on that cycle too. Otherwise, follow the edges of Y from one end of c until it hits X. Call that intersection vertex p. Do the same for the other end of c, and call that intersection vertex q. Now p and q are distinct and there are two disjoint paths between them using edges of X. One of these paths contains edge b. Combing this path with the ones of Y we used to get from c to X gives us a simple cycle containing both b and c."

■

Now we have an equivalence relation. What are the equivalence classes of this relation with respect to the edge set of G.

Biconnected Components The equivalence classes of the relation $\equiv$ are the biconnected components of G.

Now we discuss critical lemmas, which relate articulation vertices, biconnected components, DFS number and low values. These lemmas are '*if and only if*' type - or '*necessary and sufficient*' type. Such types of lemmas are extremely useful, especially in computer science, since they provide a complete characterization of the object/structure under consideration, and often lead to an algorithm.

Lemma 2.5.3 *A vertex v is an articulation vertex if and only if it is contained in at least two distinct biconnected components.*

Proof. Suppose v is an articulation vertex. Then its removal disconnects G. That means that there are two vertices a and b neighboring v so that each path between a and b goes through v. See Figure 2.8. Then the edges (a, v) and (v, b) cannot lie on a simple cycle, and hence they belong to two distinct biconnected components. This implies that v is contained in at least two distinct biconnected components.

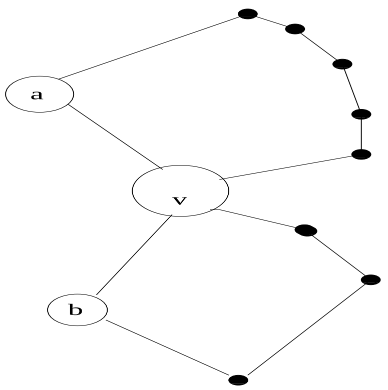


Figure 2.8: An articulation vertex in two distinct biconnected components.

Now suppose that v is contained in two distinct biconnected components, and is adjacent to vertices a and b in these components, respectively. Then $(va) \not\equiv (vb)$. Then all paths between a and b goes through v, and hence removing v disconnects G. So v is an articulation vertex. ■

Lemma 2.5.4 *Let (uv) and (vw) be two adjacent tree edges in a DFS tree T of G. Then $(uv) \equiv (vw)$ if and only if there exists a back edge from some descendant of w to some ancestor of u.*

Proof. Recall that the descendants of w are w and all the vertices in the subtree rooted at w. Ancestors of u include u and all vertices on the path from u to the root of the DFS tree.

If there exists a backedge from some descendant w' of w to some ancestor u' of u, then $(uv) \equiv (vw)$, since there is a simple cycle consisting of the tree path between u' and w' and the backedge $w'u'$. See Figure 2.9.

Suppose $(uv) \equiv (vw)$. By definition, there is a simple cycle that contains both of them. The edges (uv) and (vw) must appear in this order in the cycle, as the vertex v appears exactly once on the cycle. This implies that there is an edge (actually a backedge) from some vertex w' in the subtree rooted at w to some ancestor u' of u. (This ancestor could be the vertex u or a vertex on the path from u to the root of the DFS tree.) ■

Lemma 2.5.5 *Vertex v is an articulation vertex if and only if either*
(a) v is the root of the DFS tree and has more than one child.
(b) v is not the root, and for some some child w of v there is no backedge between any descendant of w (including w) and a proper ancestor of v.

Part (a) is easy to prove and part (b) follows from the previous lemma! The modifications to the DFS procedure to compute the biconnected components are as follows:

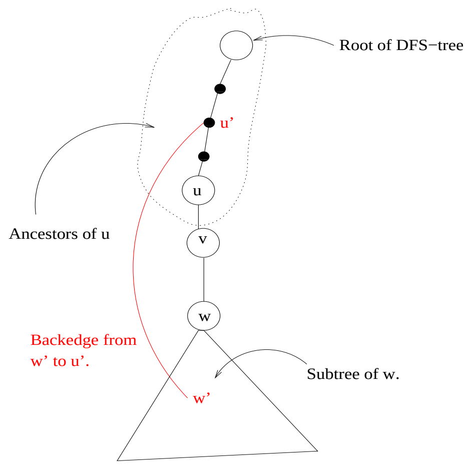


Figure 2.9: Illustration of existence of a back edge

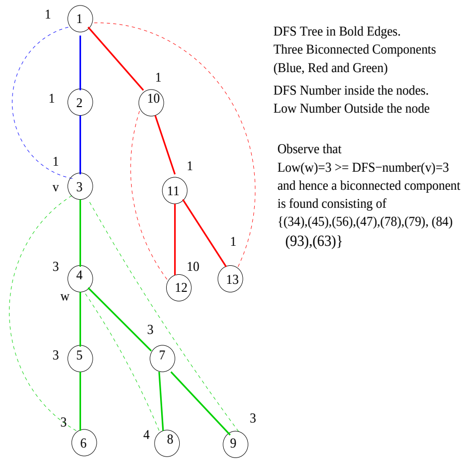

DFS Tree in Bold Edges.
Three Biconnected Components
(Blue, Red and Green)
DFS Number inside the nodes.
Low Number Outside the node
Observe that
Low(w)=3 >= DFS−number(v)=3
and hence a biconnected component
is found consisting of
{(34),(45),(56),(47),(78),(79), (84)
(93),(63)}
v
w

Figure 2.10: Illustration of biconnected components

See Figure 2.10 for an illustration for the biconnected components computed by the following search procedure.

```
procedure SEARCH(v)
begin

   1. mark $v$ as *visited*;
   2. DSF-number[v]:=COUNT;
   3. COUNT:=COUNT+1;
   4. $low(v)$:=DFS-number[v]; /* $low(v)$ is at least equal to the DFS-number of $v$ */
```

```
    5. for each vertex $w$ on $L[v]$ do
       if $w$ is unvisited then
           (a) add $(w, v)$ to $T$; /*edge $(w, v)$ is a DFS tree edge */
           (b) SEARCH(w); /* Recursive call */
           (c) If $low(w) \geq DFS - number[v]$ then a biconnected component has been found;
           (d) $low(v) := \min(low(v), low(w))$ /*Compare the low value of $v$ with its child $w$ */
       else if $w$ is not the parent of $v$ then
       $low(v) := \min(low(v), DFS - number[w])$; /* $(v, w)$ is a back edge */
end
```

Why does the above algorithm compute biconnected components? Actually we will need a STACK to figure out the edges in a biconnected component! How do we do that? When a vertex w is encountered in the SEARCH procedure, put the edge (v, w) on the STACK if it is not there. After discovering a pair (v, w), such that w is a child of v and $low(w) \geq DFS - number[v]$, POP all the edges from the STACK up to and including (v, w). These edges form a biconnected component. This extra step can be accomplished in linear time as well. To prove that the above SEARCH procedure indeed computes the biconnected components, we need to argue by induction on the number of biconnected components.

2.6 Exercises

2.1 *This problem is related to the representation of graphs. Assume that the number of edges in the graph $G = (V, E)$ is small, i.e., it is a sparse graph. In the adjacency matrix representation of G, the normal tendency is to first initialize the matrix, requiring $O(|V|^2)$ time. Is there any way we can initialize the adjacency matrix in time proportional to $O(|E|)$ and still have $O(1)$ adjacency test?*

2.2 *Provide a clear, concise, and complete proof for the correctness of the BFS algorithm.*

2.3 *Let G=(V,E) be a directed acyclic graph with two designated vertices, the start and the destination vertex. Write an algorithm to find a set of paths from the start vertex to the destination vertex such that (a) no vertex other than the start or the destination vertex is common to two paths. (b) no additional path can be added to the set and still satisfy the condition (a). Note that there may be many sets of paths satisfying the above conditions. You are not required to find the set with the most paths but any set satisfying the above conditions. Your algorithm should run in $O(|V| + |E|)$ time. State the algorithm, its correctness and analyze the complexity.*

2.4 *Clearly describe the modifications you need to make in the SEARCH procedure to compute and output the biconnected components. Prove that your algorithm is correct, i.e. it computes all the biconnected components.*

2.5 *How can we find in $O(|V| + |E|)$, whether a graph $G = (V, E)$ is a bipartite graph (Hint: Use BFS).*

2.6 *An Euler circuit for an undirected graph is a path that starts and ends at the same vertex and uses each edge exactly once (vertices might be repeated). A connected, undirected graph G has an Euler circuit if and only if every vertex is of even degree. Give an $O(|E|)$ algorithm to find an Euler circuit in G, if it exists.*

2.7 *Assume that you are given n positive integers, $d_1 \geq d_2 \geq \cdots \geq d_n$, each greater than 0. You need to design an algorithm to test whether these integers form the degrees of an n vertex simple undirected graph $G = (V, E)$ (Think of a greedy algorithm.)*

2.8 *Show that in a depth-first search, if we output a left parenthesis '(' when a node is accessed for the first time and output a right parenthesis ')' when a node is accessed for the last time, then resulting parenthesization (or bracketing sequence) is proper. Each left '(' is properly matched with each right ')'.*

2.9 *Assume that $G = (V, E)$ is biconnected. Our task is to identify those edges $E' \subseteq E$, so that if we remove any edge $e \in E'$ from G, then the resulting graph is not biconnected. Intuitively, edges in E' are essential in maintaining the biconnectivity of G. It is fairly straightforward to test whether an edge $e \in E$ is critical, by just removing e from G and running the biconnectivity algorithm to test whether the resulting graph is biconnected. A question worth trying, but is likely to be nontrivial, is to compute the set $E' \subseteq E$ in $o(|E|(|V| + |E|))$ time.*

Chapter 3

Minimum Spanning Trees

3.1 Minimum Spanning Trees

Let $G = (V, E)$ be an undirected connected graph with a cost function w mapping edges to positive real numbers. A spanning tree is an undirected tree connecting all vertices of G. The *cost* of a spanning tree is equal to the sum of the costs of the edges in the tree. A *minimum* spanning tree (MST) is a spanning tree whose cost is minimum over all possible spanning trees of G. It is easy to see that a graph may have many MSTs with the same cost (e.g., consider a cycle on 4 vertices where each edge has a cost of 1; deleting any edge results in a MST, each with a cost of 3).

As in the CLRS book[22], we will describe the two main algorithms for building MSTs, Kruskal's and Prim's. Both of these algorithms are greedy algorithms and are based on the following generic algorithm (Algorithm 3.1). The algorithm maintains a subset of edges A, which is a subset of some MST of G.

```
Algorithm 3.1: Generic-MST
  Input: Graph $G$, cost function $w$
  Output: A minimum spanning tree of $G$
1 $A \leftarrow 0$
2 while $A \neq MST$ do
3     find a safe edge $\{u, v\}$ for $A$
4     $A \leftarrow A \cup \{u, v\}$
5 end
6 return $A$
```

Intuitively this algorithm is straight-forward except for two pressing questions: What is a safe edge, and how do we find one? To answer these questions, we first need a few definitions.

Cut A cut $(S, V \setminus S)$ of $G = (V, E)$ is a partition of vertices of V.

Edge crossing a cut
An edge $(u, v) \in E$ crosses the $cut(S, V \setminus S)$ if one of its end point is in the set S and the other one in the set $(V \setminus S)$.

Cut respecting A
A cut $(S, V \setminus S)$ respects the set A if none of the edges of A crosses the cut.

Light edge An edge which crosses the cut and which has the minimum cost of all such edges.

Theorem 3.1.1 *Let A be a subset of the edges of E which is included in some MST, and let $(S, V \setminus S)$ be a cut which respects A. Let (u,v) be a light edge crossing the cut $(S, V \setminus S)$, then (u,v) is safe for A.*

Proof.

Assume that T is a MST that includes A (similarly, you may think of A as being a subset of the edges of T, or being "the makings of" a MST). If T includes the edge (u,v) then (u,v) is safe for A. If T does not include (u,v), then we will show that there is another MST, T', that includes $A \cup \{(u,v)\}$, and this will prove that $\{(u,v)\}$ is safe for A. Since T is a spanning tree, there is a path, say $P_T(u,v)$, from the vertex u to vertex v in T. By inserting the edge (u,v) in T we create a cycle. Since u and v are on different sides of the cut, there is at least one edge $(x,y) \in P_T(u,v)$ that crosses the $cut(S, T \setminus S)$. Moreover $(x,y) \notin A$, since the cut respects A. But the cost of the edge (x,y) is at least the cost of the edge (u,v), since edge (u,v) is a light edge crossing the cut. Construct a new tree T' from T by deleting the edge (x,y) in T and inserting the edge (u,v). Observe that the cost of the tree T' is at most the cost of the tree T since the cost of (x,y) is at least the cost of (u,v). Moreover $A \cup \{(u,v)\} \subset T'$ and $(x,y) \notin A$, hence edge (u,v) is safe for A.

■

The above theorem leads to the following corollary, where we fix a particular cut (i.e. the $cut(C, V \setminus C)$).

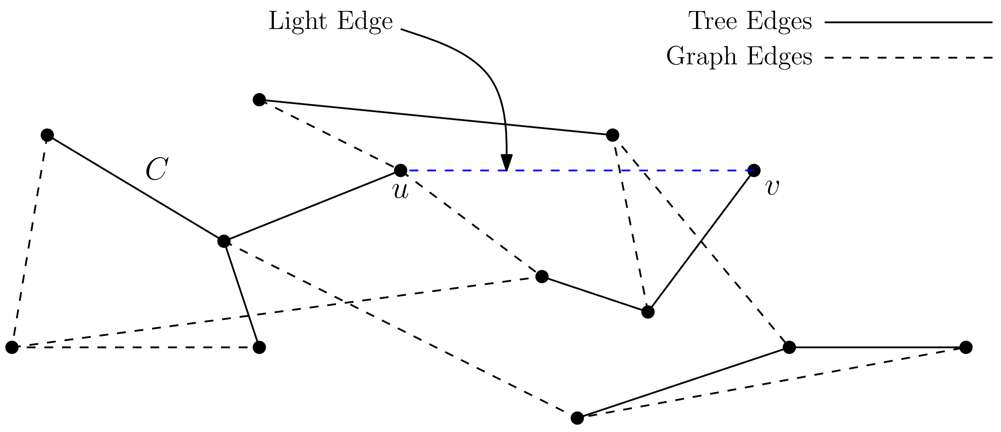


Figure 3.1: An example of Corollary 3.1.2. The edge (u,v) connects C to some other component of G_A and is a light edge; it is therefore safe to add to the MST.

Corollary 3.1.2 *Let $A \subset E$ be included in some MST. Consider the forest consisting of $G_A = (V, A)$, i.e., the graph with the same vertex set as G but restricted to the edges in A. Let $C = (V_C, E_C)$ be a connected component of G_A. Let (u,v) be a light edge connecting C to another connected component in G_A, then (u,v) is safe for A (See Figure 3.1).*

3.2 Kruskal's Algorithm for MST

Proposed by Kruskal in 1956, this algorithm follows directly from Corollary 3.1.2. Here are the main steps. To begin with the set A consists of only isolated vertices, and no edges (so, $|V|$ "connected" components in all).

1. Sort the edges of E in non-decreasing order with respect to their cost.
2. Examine the edges in order; if the edge joins two components then add that edge (a safe edge) to A.

To implement Step 2, we do the following. Let e_i be the edge under consideration, implying that all edges with a lesser cost than $e_i = (a, b)$ have already been considered. We need to check whether the endpoints a and b are within the same component or whether they join two different components. If the endpoints are within the same component, then we discard the edge e_i. Otherwise, since it is the next lightest edge overall, it must be the lightest edge between some pair of connected components, and so we know from Corollary 3.1.2 that it is safe to add to A. We will need to merge these two components to form a bigger component.

To accomplish all of this, we will need some data structure which supports the following operations:

- $\textsc{Make-Set}(v)$ - create a new set containing only the vertex v.
- $\textsc{Find}(v)$ - Find the set which presently contains the vertex v.
- $\textsc{Merge}(V_x, V_y)$ - Merge the two sets V_x and V_y together such that $\textsc{Find}$ will work correctly for all vertices in merged set.

We can implement this data structure as follows. For each vertex we keep track of which component it lies in using a label associated with the vertex. Initially each vertex belongs to its own component, which is done with $\textsc{Make-Set}$. During the algorithm the components will be merged, and the labels of the vertices will be updated. Assume that we need to merge the two components V_a and V_b corresponding to the end points a and b of the edge $e_i = (a, b)$. We use $\textsc{Find}(a)$ and $\textsc{Find}(b)$ to get the sets V_a and V_b respectively. We then call $\textsc{Merge}$ which will relabel all of the vertices in one of the components to have the same labels as the vertices of the other. The component which we relabel will be the one which is smaller in size. Given such a data structure, we can implement Kruskal's algorithm as in Algorithm 3.2.

Let us analyze the complexity of Kruskal's algorithm. Sorting the edges takes $O(|E| \log |E|)$ time. The test for an edge, whether it joins two connected components or not, can be done in constant time. (In all $O(E)$ time for all edges.) What remains is to analyze the complexity of merging the components which can be bounded by the total complexity of relabeling the vertices. Consider a particular vertex v, and let us estimate the maximum number of times this will be relabeled. Notice that the vertex gets relabeled only if it is in a smaller component and its component is merged with a larger one. Hence after merging, the size of the component containing v becomes at least double. Since the maximum size of a component is $|V|$, this implies that v can be relabeled at most $\log_2 |V|$ times. Therefore, the total complexity of the Step 2 of the algorithm is $O(|E| + |V| \log |V|)$ time. These results are summarized in the following theorem.

Theorem 3.2.1 (Kruskal) *A minimum (cost) spanning tree of an undirected connected graph $G = (V, E)$ can be computed in $O(|V| \log |V| + |E| \log |E|)$ time.*

Algorithm 3.2: Kruskal-MST

```
Input: Graph $G = (V, E)$, cost function $w$
Output: A minimum spanning tree of $G$
1  $A \leftarrow \emptyset$
2  foreach $v \in V$ do
3      MakeSet($v$)
4  end
5  sort the edges of $E$ in non-decreasing order w.r.t. $w$
6  foreach $e = \{a, b\} \in E$, where $e$ is taken in sorted order do
7      $V_a \leftarrow$ Find($a$)
8      $V_b \leftarrow$ Find($b$)
9      if $V_a \neq V_b$ then
10         $A \leftarrow A \cup \{e\}$
11         Merge($V_a, V_b$)
12     end
13 end
14 return $A$
```

3.3 Prim's MST algorithm

Prim's algorithm is very similar to Dijkstra's single source shortest path algorithm[26], and, in fact, their complexity analysis will be the same. Here the set A at any stage of the algorithm forms a tree, rather than a forest of connected components as in Kruskal's. Initially the set A consists of just one vertex. In each stage, a light edge is added to the tree connecting A to a vertex in $V \setminus A$.

The key to Prim's algorithm is in selecting that next light edge efficiently at each iteration. For each $v \in V \setminus A$, we keep track of the least cost edge which connects v to A, and the cost of this edge is used as the "key" value of v. These key values are then used to build a priority queue Q. See Figure 3.2 for an example of these sort of light edges.

In each step of the algorithm, the vertex v with the least priority is extracted out of Q. Suppose that corresponds to the edge $e = \{u, v\}$, where $u \in A$, then observe that e is a safe edge since it is the light edge for $cut(A, V \setminus A)$. We update $A := A \cup \{e\}$. Finally, after extracting v out of Q, we need to update Q. The details are explained in Algorithm 3.3.

The vertices that are in the set A at any stage of the algorithm are the vertices in $V \setminus Q$, i.e., the ones that are not in Q. $kev(v)$ is the weight of the light edge $\{v, \pi(v)\}$ connecting v to some vertex in the MST A. Notice that the *key* value for any vertex starts at infinity, when it is not adjacent to A via any edge, and then keeps decreasing.

Let us analyze the complexity of the algorithm. The main steps are the priority queue operations, namely *decrease-key* and *extract-min*. We perform $|V|$ extract-min operations in all, one for each vertex. We also perform $O(|E|)$ decrease-key operations, one for each edge. The following table shows the complexity of these operations depending on the type of priority queue you choose. These complexities are per operation, although the complexities of Fibonacci Heaps are *amortized* (kind of an average over the worst possible scenario! - more on that later).

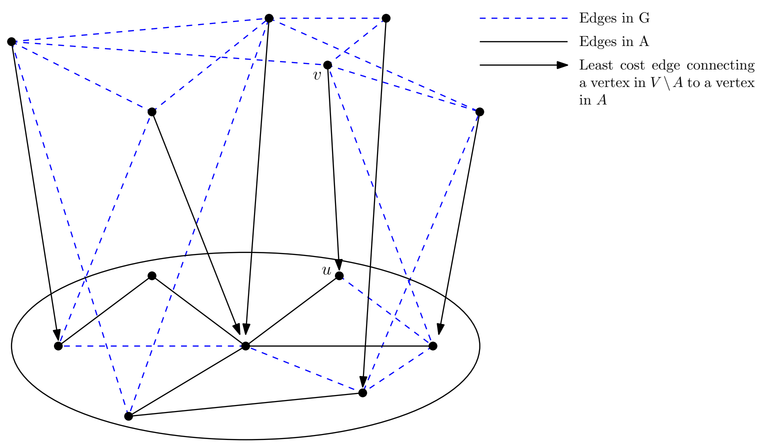

Edges in G
Edges in A
Least cost edge connecting a vertex in $V \setminus A$ to a vertex in A
v
u

Figure 3.2: Example of Prim's Algorithm showing the set A (encircled) and the least cost edges associated with each vertex in $V \setminus A$.

```
Algorithm 3.3: PRIM-MST
  Input: Graph $G = (V, E)$, cost function $w$, root vertex $r$
  Output: A minimum spanning tree of $G$
foreach $v \in V$ do
    $key(v) \leftarrow \infty$
    $\pi(v) \leftarrow$ nil /* $\pi$ keeps track of the parent of a vertex in the tree. */
end
$key(r) \leftarrow 0$
$Q \leftarrow V$ /* Priority queue consists of vertices with their key values */
while $Q \neq \emptyset$ do
    $u \leftarrow$ Extract-Min($Q$)
    foreach $v$ adjacent to $u$ do
        if $v \in Q$ and $w(u, v) < key(v)$ then
            $\pi(v) \leftarrow u$
            $key(v) \leftarrow w(u, v)$
        end
    end
end
```

	Binary Heaps	Fibonacci Heaps
Extract-min	$O(\log n)$	$O(\log n)$
Decrease-key	$O(\log n)$	$O(1)$

3.4 Randomized Algorithms for Minimum Spanning Trees

Here we discuss some results related to randomized algorithms for computing minimum spanning trees. These results are based on Section 10.3 of Raghavan and Motwani's book on Randomized

Algorithms [79] and T. Chan's simplified analysis from [20]. Assume that all edge weights are distinct and hence there is a unique MST in the given graph $G = (V, E)$. The randomized algorithm uses a few concepts which are discussed in the following subsections followed by the actual algorithm itself in Section 3.4.3. The first concept is an algorithm due to Boruvka from 1926[13, 14] which helps us to reduce the number of vertices in the graph. The second is about heavy and light edges with respect to a spanning tree.

3.4.1 Boruvka's Algorithm

Observe that for any vertex $v \in V$, the edge, say $\{v, w\}$, with the minimum weight incident to that vertex will be included in the MST, as per Corollary 3.1.2. This leads to a simple way to compute the MST as given in Algorithm 3.4.

```
Algorithm 3.4: Boruvka-MST
  Input: Graph $G$, cost function $w$ (all costs distinct)
  Output: A minimum spanning tree $T$ of $G$
1 $T \leftarrow \emptyset$
  // Each iteration of this loop is called a Phase
2 while $G$ contains more than a single vertex do
3     Mark the edges with the minimum weight incident to each vertex. Add these edges to $T$
4     Identify the connected components of the marked edges
5     Replace each connected component by a vertex
6     Eliminate all self loops. Eliminate all multiple edges between a pair of vertices, except
      the edge with the minimum weight
7 end
8 return $T$
```

A few observations about this algorithm:

- Let G' be the graph obtained from G after contracting the edges in a single phase of the algorithm. Then the MST of G is the union of the contracted edges from that phase and the edges in the MST of G'.
- In each contraction phase, the number of vertices in the graph is reduced by at least a half. Hence there will only be $O(\log |V|)$ phases in all.
- Each phase can be implemented in $O(|V| + |E|)$ time and so we obtain yet another MST finding algorithm. Total running time for this algorithm is $O(|E| \log |V|)$.

3.4.2 Heavy Edges

We've already seen the definition of a light edge. Now we examine edges which are not light.

Let F be any spanning tree of G (in particular, F may or may not be a minimum spanning tree). Consider any two vertices, say u and v, of G and there is a unique path $P(u, v)$ between them in F. Let $w_F(u, v)$ be the edge with the maximum weight along this path.

We say an edge $\{u, v\}$ is F-heavy if the weight of this edge is larger than the weight of each of the edges on the unique path between u and v in F. More formally, we define an edge $\{u, v\} \in E$

to be F-heavy if $w(u,v) > w_F(u,v)$, otherwise it is F-light. Observe that by this definition, all edges of F are F-light (every edge of a path in F is at most as heavy as the heaviest edge in that path of F).

Lemma 3.4.1 *Let F be a spanning tree of G which is not necessarily minimum. If an edge of G is F-heavy then it does not lie in the Minimum Spanning Tree of G.*

Proof. It is left for you to prove it formally. The proof proceeds by contradiction. Assume that the edge $e = \{u, v\} \in E$ is F-heavy, that T is an MST of G, and that $e \in T$ (so, we are talking about two trees here, T, which is known to be an MST, and F, which may be one as well). Consider the edges on the path $P(u,v)$ in F. Add all these edges to T and remove e from T, to obtain a graph G' (that is, $G' = T \cup P(u,v) \setminus \{e\}$, but since it may have some cycles, we cannot call it a tree). G' is still connected. Remove edges from $P(u,v)$ one-by-one from G' until G' is once again a tree. Observe that this new tree is still a spanning tree, but it must have weight lower than that of T, contradicting the minimality of T. ■

It is very important to note that because of the way F-light is defined, an edge which is F-light may or may not be in the MST. For example, if we construct F such that the heaviest edge of E is in F, that edge will be counted as F-light, even though it may not be present in any minimum spanning tree.

3.4.3 Randomized Algorithm

Algorithm 3.5 gives the main steps of the randomized algorithm we have been discussing. The analysis is based on Timothy Chan's paper[20].

```
Algorithm 3.5: Randomized-MST
  Input: Connected Graph $G = (V, E)$ with distinct edge weights
  Output: A minimum spanning tree $T$
1 Execute 3 phases of Boruvka's algorithm (reduces number of vertices). Let the resulting
  graph be $G_1 = (V_1, E_1)$, where $|V_1| \le |V|/8$ and $|E_1| \le |E|$. Let $C$ be the set of contracted
  edges. (Running time: $O(|V| + |E|)$)
2 Random Sampling: Choose each edge in $E_1$ with probability $p = 1/2$ to form the set $E_2$ and
  obtain the sampled graph $G_2 = (V_2 = V_1, E_2)$.
3 Compute Recursively the Minimum Spanning Tree of $G_2$, and let it be $F_2$. ($T(|V|/8, |E|/2)$)
4 Verification: Compute the set of $F_2$-light edges in $E_1$, and let this set be $E_3$. ($O(|V_1| + |E_1|)$
  time)
5 Final MST: Compute MST, $F_3$, of the graph $G_3 = (V_3 = V_1, E_3)$. ($T(|V|/8, |V|/4)$)
6 return MST of $G$ as $C \cup F_3$.
```

Theorem 3.4.2 *Algorithm 3.5 correctly computes the MST of G in $O(|V| + |E|)$ time.*

Proof. The correctness of the algorithm is straightforward. To estimate the complexity, the crux is in estimating the size of the set E_3, i.e., the size of the set of F_2-light edges in E_1. We will prove in the Sampling Lemma (Lemma 3.4.4) that for a random subset $R \subset E$, the expected number of edges that are light with respect to MST(R) is at most $(|E| \cdot |V_1|)/|R|$. In our case, the expected value of $|R| = |E_1|/2 \leq |E|/2$, and hence the expected number of F_2-light edges will be at most $2|V_1| \leq |V|/4$. Hence the running time of this algorithm is given by the recurrence

$$T(|V|, |E|) = O(|V| + |E|) + T(|V|/8, |E|/2) + T(|V|/8, |V|/4),$$

which magically solves to $O(|V| + |E|)$. ■

Before we describe the Sampling Lemma here are some technicalities. Consider that we are sampling the edges of the graph $G = (V, E)$, and the sampled edges form the subgraph R.

We use the notation R for both the set of edges as well as the sampled graph. Since we are sampling the edges, it is possible that the sampled graph R of the graph G is not connected, and hence there will not be any minimum spanning tree. To ensure connectedness, we will fix any spanning tree T_0 of G, consisting of $|V| - 1$ edges and we will consider the minimum spanning tree of $R \cup T_0$, denoted as $MST(R)$.

Lemma 3.4.3 (Observation about light edges) *An edge $e \in E$ is light with respect to $MST(R)$ if and only if $e \in MST(R \cup \{e\})$.*

Proof. If $e = (u, v)$ is light then there is some edge e' on the unique path between u and v in $MST(R)$ such that its weight, $w(e') = w_{MST(R)}(u, v)$, and hence e can be added to $MST(R)$ and e' can be removed to obtain MST of $R \cup \{e\}$. Therefore $e \in MST(R \cup \{e\})$.

Now suppose $e \in MST(R \cup \{e\})$. We need to show that e is light with respect to $MST(R)$. Since e is part of the MST, by definition it is light with respect to that MST. ■

Lemma 3.4.4 (Sampling Lemma) *For a graph $G = (V, E)$ and a random subset $R \subset E$, of edges, the expected number of edges that are light with respect to $MST(R)$ is at most $(|E| \cdot |V|)/|R|$.*

Proof. Pick a random edge $e \in E$ (this choice is independent of the edges in R). We will prove that e is light with respect to $MST(R)$ with probability at most $|V|/|R|$. From Lemma 3.4.3 we see that this is equivalent to finding the bound on the probability that $e \in MST(R \cup \{e\})$. Let $R' = R \cup \{e\}$. We will use a technique called *backward analysis*. First we analyze the probability on a fixed set R', and then we will show that the expression obtained is not dependent on the elements of R', but just the cardinality, and hence the probability holds unconditionally as well.

Instead of adding a random edge to R, we will think of deleting a random edge from R'. This is an easier proposition since we know the elements of R', having just fixed it. $MST(R')$ has $|V| - 1$ edges, and e is a random edge of R', hence the probability that e is an edge from $MST(R')$, given a fixed choice of R', is $(|V| - 1)/|R'| \leq |V|/|R|$. This bound is independent upon the choice of the set R', and holds unconditionally as well. ■

3.5 MST Verification

This section is contributed by Gregory Bint. If I give you any tree F derived from a graph $G = (V, E)$, can you identify whether F is a minimum spanning tree of G?

A trivial method for determining this would be to run a known-correct algorithm such as Kruskal's or Prim's on G and compare the output to F, however there are two main drawbacks to this approach:

1. It is too slow
2. The MST may not be unique, making a direct comparison difficult.

What we would like is to be able to calculate this in linear time with respect to the graph. The following lemma has been shown to be very useful in this respect, and it is used by virtually every MST verification algorithm.

Lemma 3.5.1 *Let F be a spanning tree of G, then F is a minimum spanning tree of G if and only if every edge in $E \setminus F$ is F-heavy.*

Proof. Let $P(u, v)$ be the unique tree path between u and v in F, and let $w_F(u, v)$ be the weight of the heaviest edge along that path. $w(e)$ or $w(u, v)$ is the weight of the edge e having endpoints u and v.

We first show that if F is a MST, then every edge in $E \setminus F$ is F-heavy. Let e be any edge in $E \setminus F$ with endpoints u and v and assume that e is F-light. Note that $P(u, v) \cup \{e\}$ is a cycle. Let $e' = \{x, y\}$ be the edge corresponding to $w_F(u, v)$. Since e is F-light, $w(e') > w(e)$, meaning we could replace e' by e in F to obtain a lighter tree overall, contradicting the minimality of F.

For the other half of the proof, we show that if every edge in $E \setminus F$ is F-heavy, then F is a MST of G. Again, we proceed by contradiction. Suppose that F is not a MST of G, then we should be able to lower the weight of the tree by replacing some edges in F with those from $E \setminus F$. But, for every $einE \setminus F$ with endpoints u and v, we have that $w(u, v) > w_F(u, v)$, so exchanging e for any other edge in $P(u, v)$ will increase the weight of F. ■

Given the above lemma, a natural idea for an algorithm would be to try to classify every edge in the graph, and then check if each non-tree edge is in fact F-heavy. This turns out to be something which *is* possible: given a graph $G = (V, E)$ and a tree F, we can partition the edges of G in two sets, the set of heavy edges and the set of light edges, with respect to F in $O(|V| + |E|)$ time.

We will see that many of the algorithms for doing so fairly complex, although there has been recent progress in simplifying it somewhat.

3.5.1 Overview of verification algorithms

Here we look at a brief history of the literature on MST Verification. As hinted at above, every single one of the following methods uses Lemma 3.5.1 as its underpinning. This problem can also be restated as the following:

Problem 3.5.2 (The Tree Path Maxima Problem) *Let F be a spanning tree of G, then we want to identify the cost of the heaviest edge along each tree path $P(u, v)$.*

Given an answer to Problem 3.5.2, we can perform a simple linear scan through the the edges of $G \setminus F$. For each edge $e \in G \setminus F$, we compare the cost of e with the tree path of its endpoints. If every edge e is heavier than its corresponding tree path maxima, then F is a MST of G. We look at this sort of translation of the problem in more detail in Section 3.5.6.

Here is a timeline of some results:

- In 1979, Tarjan introduced a method which uses path compression[93] of trees to achieve a near-linear time of $O(m\alpha(m,n))$ where α is the Inverse Ackermann function.
- In 1984, Komlós's provided an algorithm[62, 63] which showed that only a linear number of *comparisons* of the edge costs would be sufficient to solve the problem, however the algorithm itself has significantly more than linear overhead.
- In 1992, Dixon *et al.*[27] combine methods from Tarjan's 1979 algorithm and Komlós's 1984 algorithm to produce the first linear time MST verification algorithm. The problem is divided into one large problem and several small problems, with the larger being attacked with path compression, and the smaller with a lookup scheme which is bounded in size by Komlós's algorithm.
- In 1994, Karger *et al.* present an algorithm for computing the MST of a graph in *expected* linear time.[56, 59] While not a verification algorithm in itself, its output could be useful to help verify another tree, or to take the place of the other tree altogether (e.g., why bother verifying a potential MST in linear time when you can just *create* one!)
- In 1995, King produced another linear time MST verification method[58, 57] which is a great deal simpler than that of Dixon *et al.*. In this method, Boruvka's algorithm is used to reduce a general tree down to one which can be handled entirely by the full branching tree base case of Komlós's algorithm, which is simpler than his algorithm for general trees.
- In 2010, Hagerup simplifies King's method[42] even further and provides an implementation in the *D* programming language. Like King's method, Hagerup continues to use Komlós's *full branching tree* case, but eschews complex edge encoding schemes in favour of a richer logical data type.

We will walk through parts of Komlós's algorithm, Dixon *et al.*'s algorithm, King's algorithm, and finally Hagerup's algorithm as we piece together the tools needed for a reasonably simple approach to solving this problem.

3.5.2 Komlós's Algorithm

In 1984, Komlós[62, 63] gives an algorithm of sorts which can solve Problem 3.5.2 in $O(n+m)$ *comparisons*. He does not provide an implementable algorithm, however, and there are other factors of overhead in his method which would drive up the actual cost of a straight implementation. Nevertheless, this method of breaking down the problem is built upon by later papers, notably Dixon *et al.* in 1992, and King, which we cover in Section 3.5.4.

Komlós begins by considering two special cases of spanning trees. In each case, we consider F to be a directed tree with edges oriented away from the root. Additionally, we shift the edge costs down to their lower endpoint vertices, as this simplifies the conceptual model.

The first case occurs when the tree is a path. For the path, we construct a *symmetric order heap*, H, which a tree with both the binary search property on the ordering determined by the path, and the (maximum) heap property determined by the vertex costs. The root of H represents the heaviest vertex, and the heaviest vertex of any path from u to v is found at $LCA(u,v)$. Determining the LCA of two vertices in a tree can be accomplished in several ways and is covered elsewhere in these notes, but Komlós cites Harel[47] specifically.

The second case is somewhat more interesting and is concerned with processing *full branching trees*. Not to be confused with full *binary* trees, a full branching tree is defined as one where every leaf is at the same level, and every internal vertex has *at least* 2 children. Let F be our full branching tree with root r and all edges directed away from r.

We need to calculate the maximum cost edge of every path through F. Given a vertex y, let $A(y)$ be the set of all paths through F which contain y. That is $A(y) = \{P(x,z) | x \geqq y \geqq z\}$ where $u \geqq v$ denotes that u is a predecessor of v (or, u may equal v). Since F is directed away from the root, this means that u is at least as close to the root than v. Note that if F was not directed, then it might be possible for $x \geqq y \geqq z$ to hold even though $y \notin P(x,y)$. Given $A(y)$, let $A^*(y)$ be the set of all paths through y, but restricted to just the interval $[r,y]$; that is, just the subpath from the root down to y.

We process F one level at a time, starting from the root, finding the maximum weights of all paths in the sets $A^*(y)$. To do this, assume that we have calculated the maximum costs in all such paths up to level i, and that we are now trying to process some vertex y on level $i+1$. Let $\bar{y}$ be the parent of y. Since $\bar{y}$ resides on level i, we know the maximum cost of all paths in $A^*(\bar{y})$.

The key observation to make here is the following.

Property 3.5.3 *Consider two paths $P(x,\bar{y})$ and $P(x',\bar{y})$. If x is a predecessor of x' then the maximum cost in $P(x,\bar{y})$ is at least as large as the maximum cost in $P(x',\bar{y})$ (since, under these conditions, $P(x',\bar{y})$ is a subpath of $P(x,\bar{y})$).*

In $A^*(\bar{y})$, the shortest path is $P(\bar{y},\bar{y})$ while the longest is $P(r,\bar{y})$. By the above property, the maximum costs form a non-decreasing sequence with respect to the length of the path. That is, we can order the maximum costs by considering the path length. This observation about the ordering helps us while building $A^*(y)$, as we can use a binary search insertion of $f(y)$ to compare $f(y)$ against all path cost maximums in $A^*(\bar{y})$ simultaneously.

By now you should be asking "How are all these sets like $A(y)$ and $A^*(\bar{y})$ created, copied, and updated?" As far as Komlós's paper is concerned, the answer is "slowly". Essentially what Komlós shows us is that a linear number of *comparisons* are sufficient to determine maximum path costs, however finding those comparisons is left open.

The remainder of Komlós's paper details how these two primitive cases can be applied to any general tree, however this method is fairly complex, and as he states, results in too much overhead. No implementable algorithm is given in this paper.

3.5.3 Dixon *et al.*'s Technique

This technique has the distinction of being the first to achieve a linear running time, requiring $O(m)$ time on a graph with n vertices and m edges. The underlying process is fairly complex, however, and involves first preprocessing the graph into a suitable form.

The preprocessing itself is interesting as it shows a method of massaging a graph into a more attractive form for the problem at hand without affecting anything about the spanning tree that we wish to verify. The preprocessing involves the following steps.

For a given graph $G = (V,E)$ and spanning tree F, not necessarily minimum, we choose an arbitrary vertex r to be the root of F. Now consider any non-tree edge $\{v,w\}$ with cost $c(v,w)$ and lowest common ancestor u. If u is not one of v or w, then this implies that v and w are not *related*; that is, the v is neither ancestor nor descendant of w. In such a case, we replace the edge $\{v,w\}$ by $\{v,u\}$ and $\{u,w\}$, each with cost $c(v,w)$. F is unchanged by this process and, more importantly, the maximum weight along $P(v,w)$ is also unchanged, which preserves the current minimality of F. There are several linear time algorithms for finding lowest common ancestors in a tree (e.g., Harel & Tarjan, 1984[46] or Schieber & Vishkin, 1988[88]).

Taken over the entire graph, this will at most double the number of non-tree edges. When completed, every non-tree edge in F is a backedge.

In the second stage of preprocessing, we will mark several vertices. We can imagine these marks as subdividing the tree into edge-disjoint subtrees where a marked vertex represents a "root", and any marked descendants are ignored.

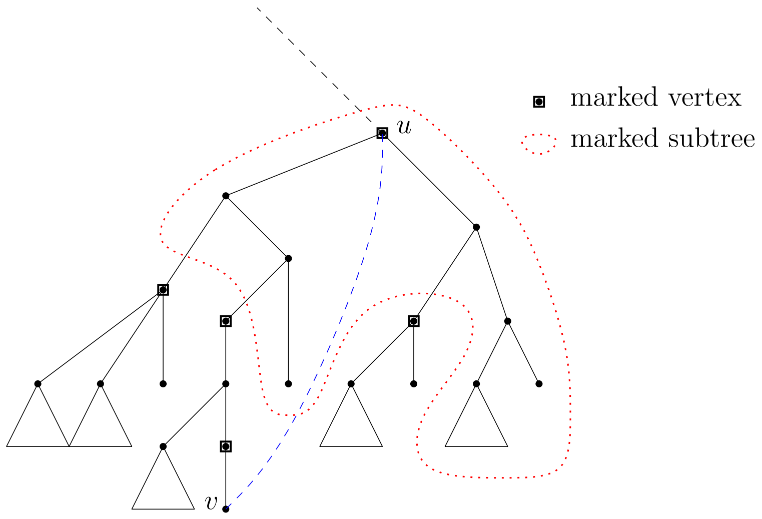


Figure 3.3: An example tree with marked vertices. The marked subtree defined by u is enclosed in red.

The choice of which vertices to mark is based on subtree size. Using a post-order traversal, for each vertex v we calculate $h = 1 + \sum\{s(w) | w \text{ is a child of } v\}$. Let g be a small integer, then if $h \le g$ we assign $s(v) := h$, otherwise $s(v) := 1$ and v becomes marked. We will look more at the specific choice of g later. Note the following important properties resulting from this process:

1. The number of marked vertices, and hence the number of subtrees, is at most $(n-1)/g + 1$.

2. Considering any subtree, if its root (marked vertex) is deleted, along with incident edges, we get a collection of disjoint trees, each with size at most g. We call each of these a *microtree*.

Following that process, r is also marked, although it will probably not have Property 2.

A final phase of edge replacements will ensure that all backedges either span two vertices belonging to the same microtree, or span between microtrees and marked vertices only (i.e., the edge $\{u,v\}$ in Figure 3.3 will be replaced).

To accomplish this, we first build the tree F' whose vertex set consists of all of the marked vertices of F, and where, for two vertices s and t in F', s is the parent of t (i.e, there is a tree edge between them) if s is the first marked vertex that we encounter when walking from t to r. We call T' the *macrotree*.

We can now eliminate the "long" edges, like $\{u,v\}$, by doing the following. Let $p(v)$ be the nearest marked vertex to v which is a proper ancestor of v. Note that if v is marked then $p(v) \neq v$.

We also assume that $p(r)$ is undefined (but it won't be needed anyway). We can calculate $p(v)$ for the entire tree using a depth-first search. For every non-tree edge $\{u, v\}$, assume w.l.o.g. that u is an ancestor of v and find $p(u)$ and $p(v)$. If $p(u) = p(v)$, then u and v are part of the same microtree (recall that the root of a microtree is *not* marked).

Otherwise, if $p(u) \neq p(v)$, then we know that there is at least one marked vertex between them. Let $r_1 = u$ if u is marked, or $r_1 = p(u)$ if u is not marked. Similarly, let $r_3 = v$ if v is marked, or $r_3 = p(v)$ if v is not marked. Let r_2 be the child of r_1 in F' (note: F', not F). We then replace $\{u, v\}$ by $\{u, r_2\}$, $\{r_2, r_3\}$, and $\{r_3, v\}$, skipping any edge that creates either a self-loop or which duplicates a tree edge. For edges which we did not skip, assign the cost $c(u, v)$. As in the first phase of edge replacements, assigning this cost preserves the current minimality of F.

With this preprocessing finished, we have now divided the problem into one large tree rooted at r with several microtrees around the periphery. The authors complete the process by using Tarjan's Path Compression on the large tree.

The microtrees are processed in a very different way. Essentially, the authors precalculate all possible minimum spanning trees on graphs containing at most g vertices. Leveraging Komlós's result, they show that for any such input, the corresponding decision tree for comparing edges and determining minimality is not too big. The choice of g is such that the total size of these precalculations is only $O(n)$, which places g in the neighbourhood of $O(\log \log n)$ [57].

3.5.4 King's Method

Presented by King[58, 57] in 1995, this method is not the first MST verification algorithm to achieve linear time (that falls to Dixon *et al.*[27]), however it is quite a bit simpler. King's method uses Boruvka's algorithm in a clever way to change any input tree into a full binary tree, which can then be entirely processed by the appropriate case presented by Komlós. This method requires linear time and space in the unit-cost RAM model with $\Theta(\log n)$ word size.

Boruvka Tree Property

The first step is to take our input tree F and convert it to a full binary tree. This is accomplished by running Boruvka's algorithm on the tree F (we usually would run Boruvka's on an entire graph, but not in this case). As Boruvka's runs on F, we can build a new tree B which represents the *execution* of the algorithm on F, rather than a modification of F itself.

Algorithm 3.6 details the construction of B. In the first step, a leaf is added to B for each vertex of F, so we already know that $|B| \geq |F|$. In fact, B will have at most twice as many vertices as F when we are finished. The algorithm proceeds by colouring the vertices and edges to represent subtrees within F, such that any vertices connected along a coloured (blue) path is considered part of the same subtree.

Refer to Figure 3.4 for an example of the algorithm's execution. Note the following important properties which ensure that B is a full branching tree.

1. In each step of Loop 1, an edge joins two blue trees into one.
2. In each phase of the while loop, every blue tree is combined by some edge with another blue tree. Thus, from every level of B, every vertex has a parent in the next level.

For every v in F there is a vertex $f(v)$ in B, and by construction we also have that for every path $F(x, y)$ there is a path $B(f(x), f(y))$. However, to show that there is any meaningful correspondence

Algorithm 3.6: FullBranchingTree

Input: A spanning tree $F = (V, E)$ with distinct edge weights

Output: A full branching tree B satisfying Lemma 3.5.4

```
Initialize $B$ as an empty tree
foreach vertex $v$ of $V$ do
    Colour $v$ blue, considering it as a singleton tree
    Add the leaf $f(v)$ to $B$
end
while there is more than one blue tree do
    // Loop ``1'', joins blue trees together
    foreach blue tree $a$ do
        Select a minimum cost edge $e$ incident to $a$ and colour it blue
    end
    // Loop ``2'', updates $B$
    foreach new blue tree $t$ do
        Add $f(t)$ to $B$
        Let $A$ be the set of trees joined into $t$ in Loop 1
        Add an edge $\{f(t), f(a)\}$ for each $a \in A$
        Set the cost of $\{f(t), f(a)\}$ to that of edge selected by $a$ in Loop 1 (i.e., $e$)
    end
end
return $B$
```

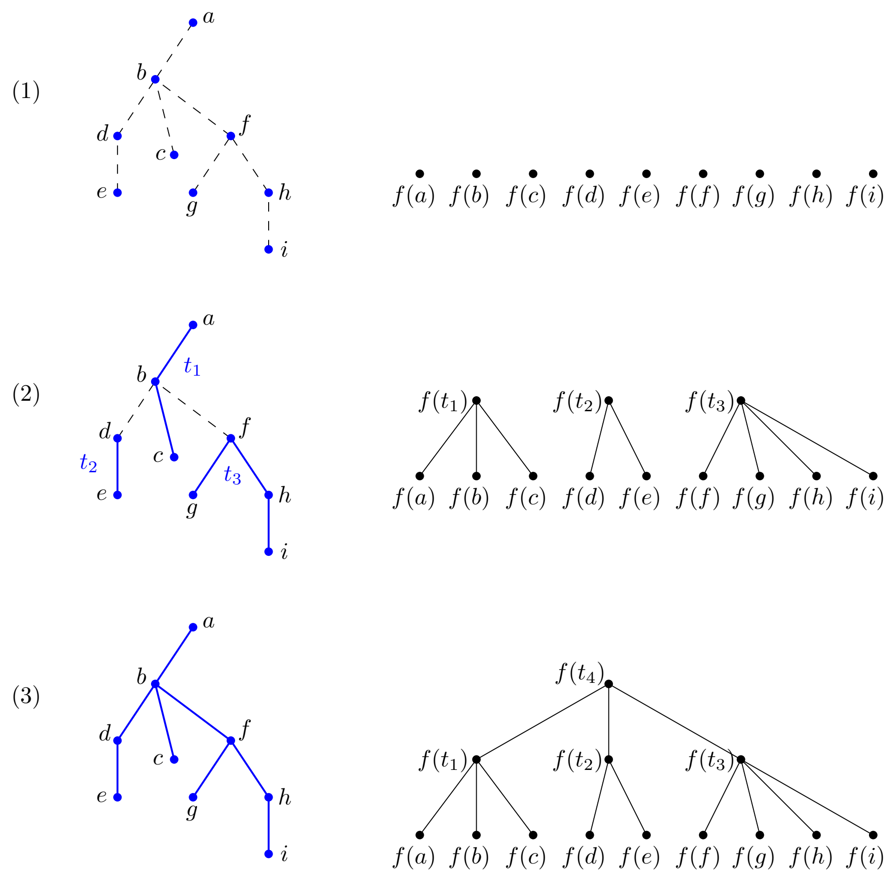


Figure 3.4: An example of the Full Binary Tree construction given by Algorithm 3.6. F is shown on the left and B on the right. Edge weights are not shown, so imagine that each tree chooses its minimum weight edge at each step.

---

between these paths beyond their existence, we need the following lemma, presented as Theorem 1 in King's paper.

Lemma 3.5.4 *Let F be any spanning tree and let B be the tree constructed by Algorithm 3.6. For any pair of vertices $x, y \in F$, the cost of the heaviest edge in $F(x, y)$ equals the cost of the heaviest edge in $B(f(x), f(y))$.*

Proof. Let the cost of an edge e be denoted by $w(e)$. For every edge $e \in B(f(x), f(y))$, we will show that there is an edge $e' \in F(x, y)$ such that $w(e') \geq w(e)$.

Suppose that $e = \{a, b\}$ such that a is the endpoint of e which is farthest from the root. As a is in B, $a = f(t)$ for some blue tree t, and t must contain either x or y, but not both. Similarly, $b = f(t')$ which is new blue tree consisting of $f(t)$ and others from the previous phase of the algorithm. Since $e \in B$, e was selected by t.

Let e' be the edge in $F(x, y)$ with exactly one endpoint in t. Since e' is adjacent to t, t would have considered e'. Since t ultimately chose e, it must be that $w(e') \geq w(e)$ since t chooses the edge with minimum cost.

To finish the proof we also need to show the following: The cost of the heaviest edge in $F(x, y)$ is the cost of the heaviest edge in $B(f(x), f(y))$. Let e be the heaviest edge in $F(x, y)$ (for simplicity, assume that there is a unique such edge). If e is ever selected by a blue tree which contains either x or y, then $B(f(x), f(y))$ contains an edge with the same weight.

Otherwise, assume that e is selected by some other blue tree t' not containing x or y. We know that e is on the path from x to y in F, so t' contained at least one intermediate vertex on that path. But since F is a tree, if it contains an intermediate vertex of $F(x, y)$, it must be incident to at least two edges of $F(x, y)$. By our assumption, e is the heaviest edge on this path, so t' would have selected the other edge, giving a contradiction. ■

The intuition with the last part of the above proof is that, since e is the heaviest edge along $F(x, y)$, any blue tree which includes part of that path, but which does not yet include x or y always has another edge to select which brings it "closer" to x or y.

King's algorithm now continues with B rather than F, which maintains the path maximum cost property for each path in F, implying that if Lemma 3.5.1 holds for B it will also hold for F.

The remainder of King's paper shows a bit-wise labeling scheme from for the vertices and edges of B which exploits Property 3.5.3 of the full binary tree case presented by Komlós. We will now jump to Hagerup's method to conclude our verification method, which he wrote specifically to simplify away from this labeling scheme, but which otherwise picks up at exactly this point of the algorithm.

3.5.5 Hagerup's Method

Hagerup presents an algorithm for solving the Tree Path Maxima problem (TPM) rather than MST Verification, *per se*, but as we have mentioned, a solution to TPM implies a solution to MST Verification. A sketch of such a translation is given in the next section.

The input to Hagerup's method assumes that we are given a tree on n vertices and a list of pairs $(u_1, v_1), \ldots, (u_m, v_m)$ such that in each pair u_i is a proper ancestor of v_i. At most, this list would describe the endpoints of every root to leaf path in B, and every subpath of such a root to leaf path. Any subset is also permissible. In practice, we choose a subset equivalent to the non-tree edges of G, the graph containing the spanning tree F we are trying to verify.

The basic algorithm involves collecting several types of information about each vertex. For every vertex u in B, we store the depth $d(u)$, and, if u is not the root r, we let $w(u)$ represent the cost of the edge from u to its parent. For each u we also build the following set:

$$D_u = \{d(u_i) | u_i \text{ is a proper ancestor of } u \text{ and } v_i \text{ is a descendant of } u\}$$

Simply put, D_u stores the set of depths corresponding to proper ancestors of u such that u is in the subpath represented by some pair (u_i, v_i) from the input.

We would also like to create the set M_u for each u, which stores a subset of the ancestors of u indicated by D_u. The choice of which ones are stored again exploits Property 3.5.3.

Consider any two successive ancestors d and d' of u which are indicated by D_u such that d is closer to the root, and d' is closer to u. Then $d \in M_u$ if the path maximum cost of the path $d \to u$ is greater than that of $d' \to u$. Put another way, we store only those ancestors of u where there is an actual increase in path maximum cost between it and the previous (closer) ancestor.

This can still work out to be a lot of entries, however, and a lot of copying between vertices, which breaks linear time. Fortunately, Hagerup was able to find an alternate, yet equivalent set representation which does satisfy our needs, and our desired running time, using the *set infix* operator. The details of this operator and its equivalence to M_u take a few pages to discuss and can be found in his paper.

3.5.6 Putting it all together

One way of applying all of the tools we have seen so far to build a complete MST Verification algorithm is as follows.

Taking a graph $G = (V, E)$ with spanning tree F, let $U = E \backslash F$ be the set of non-tree edges. We use King's method of using Boruvka's method to convert F to the full branching tree B. Translate U onto B so that $\forall e = \{x, y\} \in U$ we create $e' = \{f(x), f(y)\}$ and call the resulting graph G'. We next apply Dixon *et al.*'s first preprocessing step to G' to replace all cross edges with back edges. Let U' be the set of non-tree edges in G' after all of this.

The set U' corresponds to the pairs (u_i, v_i) that we need to input into Hagerup's algorithm. After that algorithm has run, MST Verification is completed by examining every non-tree edge in G, translating it to the equivalent one or two edges in U', querying B, and determining whether the non-tree edge is costlier than the tree path maximum.

The extra steps required to find U, translate it to B, and then find U' all take time linear in the number of edges.

3.6 Bibliographic Notes

Kruskal's algorithm, presented in Section 3.2 makes use of a data structure known as UNION-FIND or DISJOINT-SET. A near-linear time implementation was first described by Tarjan[92].

Boruvka's algorithm is quite old[13, 14], and not originally published in English. Neetil *et al.* published a translation from the original Czech in 2001 along with some comments.[80]

Komlós mentions that his method of using symmetric order heaps for processing paths is something of a well-known method by the time he covers it in his own paper. However, he was unable to find a reference to it in any other literature, which is why he took the time to write about it.

The way that King uses Boruvka's algorithm is first described by Tarjan in 1983[91].

3.7 Exercises

3.1 *Let S=(V,T) be a minimum cost spanning tree, where $|V| = n + 1$. Let $c_1 \leq c_2 \leq ... \leq c_n$ be the costs of the edges in T. Let S' be an arbitrary spanning tree with edge costs $d_1 \leq d_2 \leq .... \leq d_n$. Show that $c_i \leq d_i$, for $1 \leq i \leq n$.*

3.2 *Assume all edges in a graph G have distinct cost. Show that the edge with the maximum cost in any cycle in G cannot be in the Minimum Spanning Tree of G. Can you use this to design an algorithm for computing MST of G by deletion of edges, and what will be its complexity?*

3.3 *Recall that Dijkstra's SSSP algorithm was for directed (or undirected) graphs where the weights of the edges are positive and we need to compute shortest paths from the source vertex to all other vertices in the graph. What happens when some of the edges have negative weights. Try to consider the cases where the algorithm will fail and where the algorithm will still work.*

3.4 *Design an efficient algorithm to find a spanning tree of a connected, (positive) weighted, undirected graph $G = (V, E)$, such that the weight of the maximum-weight edge in the spanning tree is minimized (Justify your answer).*

3.5 *Let $G = (V, E)$ be a weighted directed graph, where the weight of each edge is a positive integer and is bounded by a number X. Show how shortest paths from a given source vertex s to all vertices of G can be computed in $O(X|V| + |E|)$ time (Justify your answer).*

3.6 *Prove that if all edge weights are distinct then the minimum spanning tree of a simple undirected graph is unique.*

3.7 *Provide a formal proof of Lemma 3.4.1.*

3.8 *Suppose all edge weights are positive integers in the range $1..|V|$ in a connected graph $G = (V, E)$. Devise an algorithm for computing Minimum Spanning Tree of G whose running time is better than that of Kruskal's or Prim's algorithm.*

3.9 *Consider a connected graph $G = (V, E)$ where each edge has a non-zero weight. Furthermore assume that all edge weights are distinct. Show that for each vertex $v \in V$, the edge incident to v with minimum weight belongs to a Minimum Spanning Tree.*

(Bonus Problem: Can you use this to devise an algorithm for MST - the above step identifies at least $|V|/2$ edges in MST - you can collapse these edges (by identifying the vertices and then recursively apply the same technique - the graph in the next step has at most half of the vertices that you started with - and so on!)

3.10 *Prove that the distance values extracted from the priority queue over the entire execution of Dijkstra's single source shortest path algorithm, in a directed connected graph with positive edge weights, is a NON-Decreasing sequence. Where is this fact used in the correctness of the algorithm?*

3.11 *Can you devise a faster algorithm for computing single source shortest path distances when all edge weights are 1? (Think of an algorithm that runs in $O(|V| + |E|)$ time on a graph $G = (V, E)$.)*

3.12 *Execute Dijkstra's SSSP algorithm on the following graph on 7 vertices and 18 edges starting at the source vertex s. The edges and their weights are listed in the following (the entry $(xy, 10)$ means the edge directed from the vertex x to the vertex y with edge weight 10):*

$(sb, 5)$, $(sa, 10)$, $(sf, 5)$, $(bf, 6)$, $(ba, 3)$, $(be, 5)$, $(bc, 5)$, $(fs, 2)$, $(fe, 4)$, $(ca, 3)$, $(ce, 2)$, $(cd, 5)$, $(df, 1)$, $(de, 1)$, $(ef, 1)$, $(ec, 1)$, $(af, 1)$, $(ae, 2)$.

3.13 *Recall that Dijkstra's SSSP algorithm only computes distances from source vertex to all the vertices. What modifications we should make to the algorithm so that it reports the shortest paths as well (in fact the collection of all these paths can be represented in a directed tree rooted at the source vertex).*

3.14 *Suppose in place of computing shortest path distance from a vertex to every other vertex, we are interested in finding the shortest path distances between every pair of vertices. Then one way to do this is to run Dijkstra's algorithm $|V|$ times, where each vertex in the graph $G = (V, E)$ is considered as a source vertex once. Can you devise an algorithm that is asymptotically faster than just running Dijkstra's algorithm $O(|V|)$ times?*

3.15 *Which of the following algorithms result in a minimum spanning tree? Justify your answer. Assume that the graph $G = (V, E)$ is connected.*

1. *Sort the edges with respect to decreasing weight.*
   Set $T := E$.
   For each edge e taken in the order of decreasing weight do, if $T - \{e\}$ is connected, then discard e from T.
   Set $MST(G) = T$.

2. *Set $T := \emptyset$.*
   For each edge e, taken in arbitrary order do, if $T \cup \{e\}$ has no cycles then
   $T := T \cup \{e\}$.
   Set $MST(G) = T$.

3. *Set $T := \emptyset$.*
   For each edge e, taken in arbitrary order do
   begin
   $T := T \cup \{e\}$.
   If T has a cycle c then let e' be a maximum weight edge on c.
   Set $T := T - \{e'\}$.
   end
   Set $MST(G) = T$.

3.16 *A spanning tree T of a undirected (positively) weighted graph G is called a minimum bottleneck spanning tree (MBST) if the edge with the maximum cost is minimum among all possible spanning trees. Show that a MST is always a MBST. What about the converse?*

3.17 *Design a linear time algorithm to compute MBST. (Note that an edge with medium weight can be found in linear time. Consider the set of edges whose weight is smaller than the weight of the 'median edge'. What happens if this graph is connected? disconnected?*

3.18 *Consider an undirected (positively) weighted graph $G = (V, E)$ with a MST T and a shortest path $\pi(s, t)$ between two vertices $s, t \in V$. Will T still be an MST and $\pi(s, t)$ be a shortest path if*
a) Weight of each edge is multiplied by a fixed constant $c > 0$.
b) Weight of each edge is incremented by a fixed constant $c > 0$.

3.19 *Let $G = (V, E)$ be a weighted simple connected graph, and assume that all edge weights are distinct. Define the weight of a spanning tree to be the sum total of the weights of edges in that tree. By definition, a minimum spanning tree T of G has the smallest sum total of the weight among all possible spanning trees of G. Suppose we are not interested in minimizing the sum total of the weights, but just the weight of the heaviest edge in a spanning tree. Call such a tree a* light spanning tree *(LST). First show that any MST of G is also a LST. Next show that a LST may not always be a MST. To compute LST, we can use an algorithm to compute MST and report that MST as a LST. You are asked to think of an alternate algorithm, running in $O(|V| + |E|)$ time, to find a LST. (Hint: Let e_m be the edge with the median weight among edges in $G = (V, E)$. Consider the subgraph G' formed by all edges in E, whose weight is at most the weight of e_m. Can you deduce something about LST from the connectivity of G'.)*

3.20 *Suppose you are given n-points in the plane. We can define a complete graph G on these points, where the weight of an edge $e = (u, v)$, is Euclidean distance between u and v. We need to partition these points into k non-empty clusters, for some $n > k > 0$. The property that this clustering should satisfy is that the minimum distance between any two clusters is maximized. (The distance between two clusters A and B is defined to be the minimum among the distances between pair of points, where one point is from cluster A and the other from cluster B.) Show that the connected components obtained after running Kruskal's algorithm till it finds all but the last $k - 1$ (most expensive) edges of MST of G produces an optimal clustering.*

Chapter 4

Network Flow

4.1 What is a Flow Network

A flow network consists of the following:

1. A simple finite directed graph $G = (V, E)$.
2. Two specified vertices, namely source s and target t.
3. For each edge $e \in E$, a non-negative number $c(e)$ called the capacity. If a pair of vertices u and v are not joined by an edge, then $c(u, v) = 0$.

Flow: A flow function f in G is a real-valued function

$$f : V \times V \to \Re$$

that satisfies the following three properties.

1. **Capacity Constraint**: For all $u, v \in V$, $f(u, v) \leq c(u, v)$.
2. **Skew Symmetry** (A tough constraint to see!): For all $u, v \in V$, $f(u, v) = -f(v, u)$. This is for notational purposes, and basically says that flow from a vertex u to vertex v is the negative of the flow in the reverse direction.
3. **Flow conservation**: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$. This uses the skew symmetry property, otherwise we have to sum up the flow values coming into a vertex and that should be equal to the sum of the outgoing flow values from that vertex. This is same as the Kirchoff law for current in an electrical circuit, i.e. no node can hold the current, or no node can hold the flow, or whatever comes in goes out. There is no reservoir at a node.

The value of the flow is defined to be the flow out of the source s or the flow into the target t, i.e.

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t).$$

The **Maximum Flow Problem** to find the flow of maximum value in a given flow network. See Figure 4.1 for an example of a network flow.

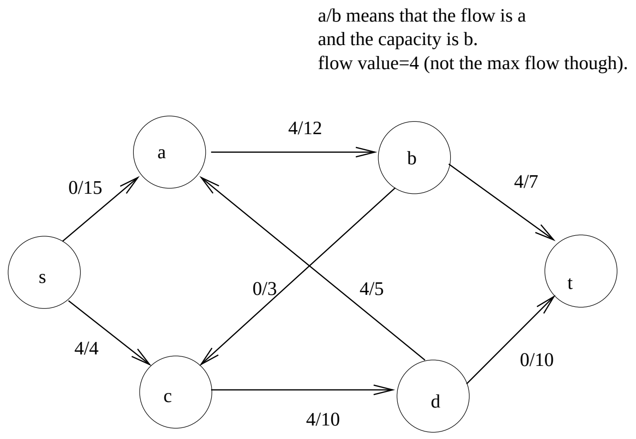

a/b means that the flow is a
and the capacity is b.
flow value=4 (not the max flow though).
a
b
c
d
s
t
0/15
4/12
4/7
0/3
4/5
4/4
4/10
0/10

Figure 4.1: An example of a network flow. Example is taken from Even [30]

4.2 Ford and Fulkerson's Algorithm

This is an iterative method for computing the flow.

```
Ford-Fulkerson-Method $(G, s, t)$
1. Initialize the flow $f$ to $0$.
2. While there exists an augmenting path $p$
    augment the flow $f$ along $p$.
3. Return $f$.
```

An augmenting path is a path from s to t along which additional flow can be sent. This path is found using the concept of residual networks. The residual network consists of those edges which can admit more flow. The residual capacity $c_f(u, v)$ of an edge (u, v) in a flow network is given by

$$c_f(u, v) = c(u, v) - f(u, v).$$

In our example $c_f(a, b) = 12-4 = 8$, $c_f(s, c) = 4-4 = 0$, $c_f(b, a) = c(b, a)-f(b, a) = 0-(-4) = 4$. Given a flow network G and the flow function f, the residual network $G_f = (V, E_f)$ consists of the same vertex set and the edges E_f are defined as follows:

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

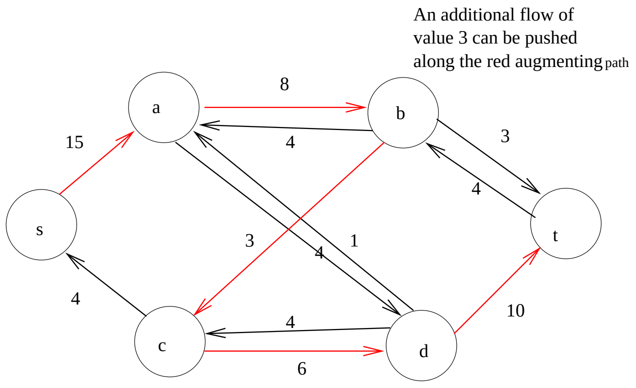
The residual network of our example is given in Figure 4.2.

As we can see that there is an augmenting path in this network (the red path), and the flow can be augmented along this path, by a value of 3. Hence we get a new flow network with the total flow value equals to $7 = 4 + 3$ given in Figure 4.3.

The new residual network that we obtain for the flow corresponding to the flow network in Figure 4.3 is given in Figure 4.4. Note that there exists an augmenting path that can further increases the flow value by 3.

The new flow graph is shown in Figure 4.5 and the corresponding residual network is shown in Figure 4.6.

This will continue for two more iterations and after that there is no path between s and t in the residual network. The corresponding figures are Figure 4.7 and Figure 4.8.



An additional flow of value 3 can be pushed along the red augmenting path
a
b
s
t
c
d
8
4
15
3
4
3
4
1
4
4
10
6

Figure 4.2: Residual network corresponding to the flow in Figure 4.1.

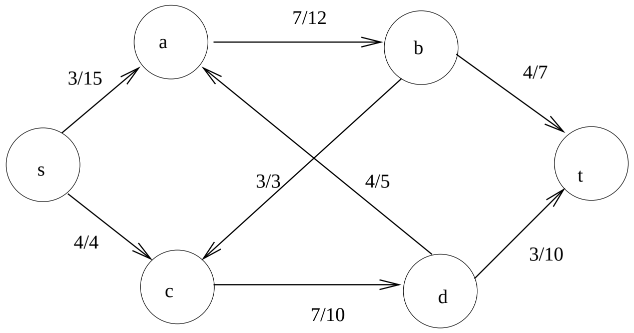


Figure 4.3: Flow network after augmenting the flow from Figures 4.1 and 4.2.

Now consider the residual network in Figure 4.8, where there are no paths joining s and t. As can be seen from the figure, there is a path from s to every vertex in the set $\{s, a, b\}$, and there are paths from vertices $\{c, d, t\}$ to t. This automatically partitions the set of vertices into two, call it a $s - t$ cut $\{S, T\}$, where $s \in S$ and $t \in T$ (in our example, $S = \{s, a, b\}$ and $T = \{c, d, t\}$). See Figure 4.9. Define the capacity of a cut as follows

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v).$$

In other words consider the edges crossing the cut, and sum up the capacities of the edges which go from a vertex in the set S to a vertex in the set T. Define the net flow across the cut to be

$$f(S, T) = \sum_{u \in S, v \in T} f(u, v).$$

In other words the net flow is the sum of the positive flow on edges going from S to T minus the sum of the positive flows on edges going from T to S (recall the skew symmetry property). Amazingly in our example $f(S, T) = c(S, T)$. Is it always true or just a luck! Before we get to this, a few observations.

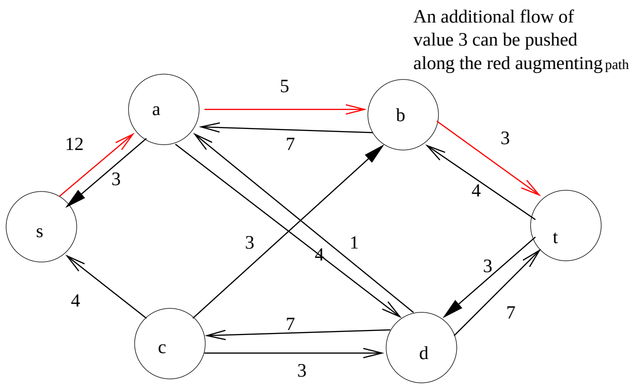


Figure 4.4: Residual network corresponding to the flow in Figure 4.3.

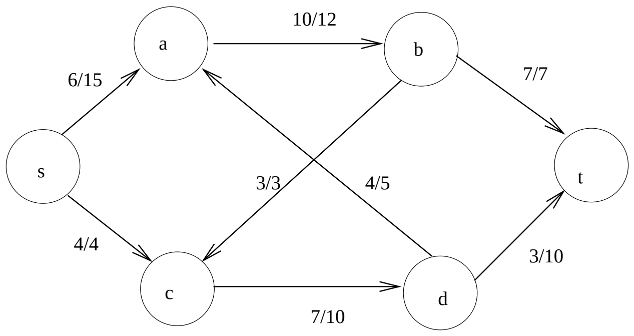


Figure 4.5: Flow network after augmenting the flow from Figures 4.3 and 4.4.

Observation 4.2.1 *For any $s-t$ cut S, T, and flow f*

$$|f| \leq c(S,T).$$

This follows from the definition of the flow across the cut. The flow $f(S,T)$ is defined to be the sum of the positive flows along the edges in the forward direction, i.e. the ones going from vertices in S to vertices in T minus the sum of the positive flows along the edges in the reverse direction. If we ignore the reverse direction, then clearly the flow along each edge in the forward direction is bounded by the capacity of the edge. Sum of these capacities is the capacity of the cut and hence the observation.

The following observation explains why the flow f' found using the augmenting paths in the residual graph G_f, can be augmented with the flow f in G, to obtain a new flow in G of a higher value $|f+f'| \geq |f|$.

Observation 4.2.2 *Let G be the flow network with flow f and G_f be the corresponding residual network and let f' be the flow in G_f. Then the flow sum $f+f'$ is a flow in G and its value is $|f+f'| = |f| + |f'|$.*

Proof: To prove that $f+f'$ is a flow in G, we need to prove that the three conditions are satisfied. I will prove here the capacity constraint, and others can be verified similarly. The capacity constraint

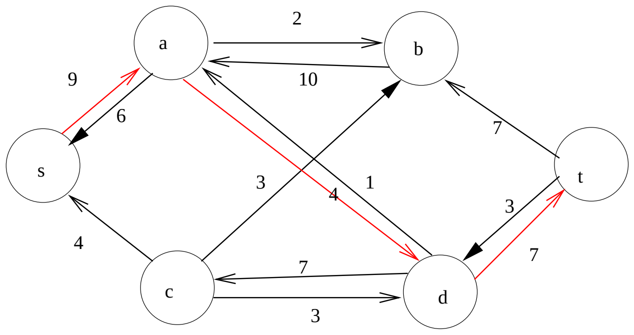


Figure 4.6: Residual network corresponding to the flow in Figure 4.5.

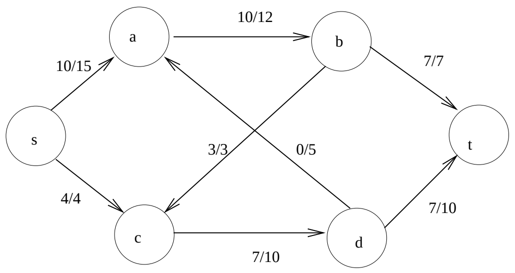


Figure 4.7: Resulting Flow network.

follows from

$$(f + f')(u, v) = f(u, v) + f'(u, v) \leq f(u, v) + (c(u, v) - f(u, v)) = c(u, v).$$

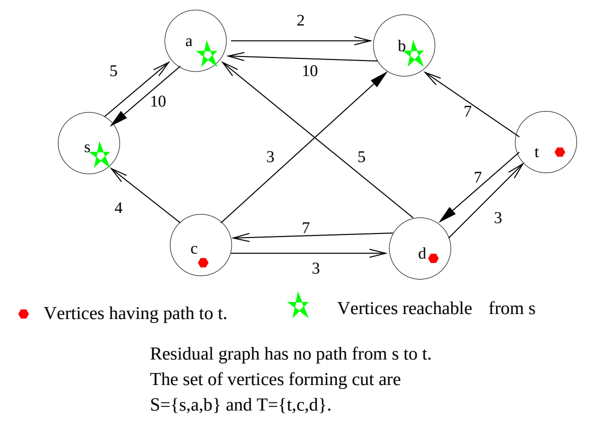
Observe that

$$|f + f'| = \sum_{v \in V}(f + f')(s, v) = \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) = |f| + |f'|. \square$$

Theorem 4.2.3 *Let f be a valid flow in the flow network $G = (V, E)$ from the source s to the target t, then the following statements are equivalent.*

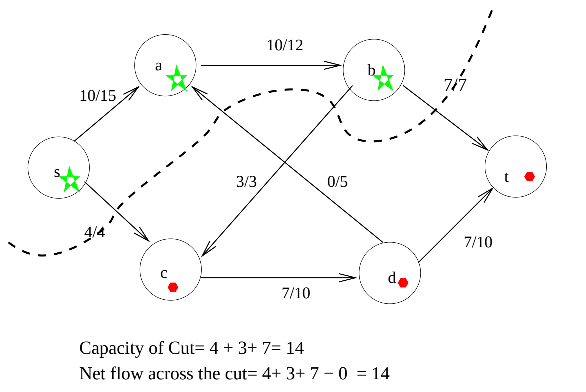
1. *Flow f is a maximum flow.*
2. *Residual network G_f does not contain an augmenting path.*
3. *There exists some cut $c(S, T)$ such that $|f| = c(S, T)$.*

This is the famous max flow min cut theorem.



a
b
s
t
c
d
2
10
5
10
7
3
5
7
3
4
7
3
● Vertices having path to t.
☆ Vertices reachable from s
Residual graph has no path from s to t.
The set of vertices forming cut are
S={s,a,b} and T={t,c,d}.

Figure 4.8: Residual network corresponding to the flow in Figure 4.7.



a
b
s
t
c
d
10/12
10/15
7/7
3/3
0/5
4/4
7/10
7/10
Capacity of Cut= 4 + 3+ 7= 14
Net flow across the cut= 4+ 3+ 7 − 0 = 14

Figure 4.9: An illustration of a cut.

Proof: Recall that to prove that the three statements are equivalent we need to show that $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$.
$1 \Rightarrow 2$: Let f be a maximum flow and, for contradiction, assume that there exists an augmenting path in G_f. Then we can increase the flow along the path using Observation 4.2.2 and contradicting that f is a maximum flow.
$2 \Rightarrow 3$: Define the set

$$S = \{v \in V | \text{there is a path from } s \text{ to } v \text{ in } G_f$$

and

$$T = V - S.$$

Also observe that $s \in S$ and $t \in T$, so it is a valid $s - t$ cut. Moreover for all edges (u, v) crossing

the cut, where $u \in S$ and $v \in T$, $f(u,v) = c(u,v)$, otherwise $(u,v) \in E_f$ and $v \in S$, which is not possible. Net flow across the cut (S,T) is $|f|$! Why? (Think about this yourself!) So we have shown a cut where 3 holds.
$3 \Rightarrow 1$: We know that the capacity of any cut is an upper bound to the value of the flow. If for a cut we obtain the equality, then we have attained the max-flow. (In other words the capacity of minimum cut is the value of the maximum flow!). □

This proves the correctness of the Ford-Fulkerson algorithm. The algorithm iteratively increases the value of the flow using augmenting paths and return the value of the flow, the maximum flow, when it is not able to find augmenting paths in the residual graph. How do we analyze the complexity of this algorithm?

First a special case where all capacities are integers. Observe that value of all flows computed during the algorithm are integers. In each iteration of the algorithm, the value of flow increases by at least 1. If f^* is a maximum flow, then the number of iterations in the algorithm are bounded by $|f^*|$. It is easy to see that each iteration requires $O(|E|)$ time; this involves computing residual graph (i.e. capacities on at most $2|E|$ edges), and computing a path between s and t (directed dfs or bfs). Hence the algorithm runs in $O(|f^*||E|)$ time - this is a strange complexity since the running time depends upon the value of the output! Is there a better way to analyze this algorithm!

4.3 Edmonds-Karp Algorithm

In this variation, the computation of the augmenting path is done using a BFS tree rooted at s. Augmenting path is a shortest path (in the unweighted residual graph) from s to t. It turns out that this variation leads to an algorithm that runs in $O(|V||E|^2)$ time. Here is the main lemma - let $\delta_f(s,v)$ denote the shortest path distance between s and v in the unweighted residual graph G_f, corresponding to the flow network G with flow function f.

Lemma 4.3.1 *Shortest path distance for all vertices $v \in V - \{s,t\}$ in G_f increases monotonically with each flow augmentation.*

Proof. By contradiction.
Caution: This is a little bit strange proof, and the proof in generic terms goes as follows. To prove the statement P, the contradictory proof assumes that $\neg P$ is true. Inside the proof we need to prove a claim C, which in turn is proved using the contradiction. Say $\neg C$ is true. The contradiction is arrived by showing that $\neg C$ is true only if P is true, but since $\neg P$ is assumed to be true, implying that C is true. Once we show that C is true, the contradiction to the original assumption is arrived at. (Wow! Pay attention)

Assume that for a vertex $v \in V - \{s,t\}$, the shortest path decreases after a flow augmentation. Let f be the flow before the augmentation and f' be the flow after the augmentation. Let v be the vertex with minimum $\delta_{f'}(s,v)$ whose distance was decreased by the augmentation (i.e. $\delta_{f'}(s,v) < \delta_f(s,v)$). Let u be the vertex just before v in the shortest path from s to v in $G_{f'}$, i.e. $(u,v) \in E_{f'}$. Then $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$. Moreover, $\delta_{f'}(s,u) \geq \delta_f(s,u)$ (by choice of v).

Now we will show that $(u,v) \notin E_f$, and as a consequence of that we will arrive to contradiction (somehow!).

First why $(u,v) \notin E_f$? Suppose $(u,v) \in E_f$, then $\delta_f(s,v) \leq \delta_f(s,u) + 1$ (triangle inequality - sum of two sides of the triangle is at least as big as the third side). But, $\delta_f(s,u)+1 \leq \delta_{f'}(s,u)+1 = \delta_{f'}(s,v)$. This implies that $\delta_f(s,v) \leq \delta_{f'}(s,v)$, contradicts our assumption!

Now consider the scenario that $(u,v) \notin E_f$ and $(u,v) \in E_{f'}$. The flow from v to u must have been increased in Edmonds - Karp algorithm and this edge must be on a shortest path. This implies that $\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 2$, and this contradicts the assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$. ∎

In each iteration of the augmenting path algorithm, at least one edge becomes critical, i.e. flow value becomes equal to its capacity. The critical edge disappears from the residual network. Of course the flow along this edge may be decreased in the future, and this edge may reappear again in the residual network, but this cannot happen more than $|V|/2$ times. Why?. Say (u,v) became critical, then $\delta_f(s,v) = \delta_f(s,u) + 1$. Flow along (u,v) is decreased only if (v,u) appears on an augmenting path, let f' be the flow then, $\delta_{f'}(u) = \delta_{f'}(v) + 1$. Since the shortest path distances are monotone, this implies that

$$\delta_{f'}(s,u) = \delta_{f'}(v) + 1 \geq \delta_f(s,v) + 1 = \delta_f(s,u) + 2.$$

Therefore the distance to u from the source has increased by at least 2 between two consecutive times that (u,v) became critical. The maximum distance is at most $|V|$ and hence an edge can become critical about $|V|/2$ times. There are $O(|E|)$ edges in all in the residual graph, and hence the number of augmentations (or iterations) are bounded by $O(|V||E|)$ times. Each augmentation can be implemented in $O(|E|)$ time, and hence flow between s and t in the graph $G = (V,E)$ can be computed in $O(|V||E|^2)$ time.

4.4 Applications of Network Flow

We can use the flow networks to compute Maximum Matching in a Bipartite Graphs. Recall that a Graph $G = (V = A \cup B, E)$ is bipartite, if the set of vertex V is partitioned into two sets A and B, such that all the edges in the graph are between vertices of A to vertices in B. A matching in a graph is a collection of edges such that no two edges in the matching are incident to the same vertex. A matching in G is called a maximum matching if the cardinality of the number of edges in it is maximum among all matchings in G. Note that there can be a number of maximum matching in a graph. Using flow networks we can compute easily maximum matching in G. Here is the simple method. We add two vertices, namely s and t, to the set of vertices in G. Vertex s is connected to all the vertices in the set A, and the capacity of all these edges is set to 1. The capacity of all the edges in the set E, i.e., the edges joining vertices in the set A to vertices in the set B, is set to 1. Lastly vertices in B are joined to t, and the capacity of these edges is set to 1. Let G' be the resulting graph. Compute the maximum $s - t$ flow in G'. Observe that the value of the flow is the size of the maximum matching. Why ?

Note that value of flow in each of the edge will be an integral value, since all capacities are integers (this is one of the exercises in [21]). Since the capacity of all the edges between vertices in A and B is 1, the value of flow on these edges will either be 0 or 1. This implies that no two edges in E incident on the same vertex will ever have nonzero flow. In other words the edges in E which have nonzero flow are the edges in a matching. Also maximum matching corresponds to a largest set of independent edges in G and each of these edges can admit a flow of value 1 and at the same time satisfy all the three conditions required for a flow network. Hence maximum matching in G corresponds to a valid flow in G'.

4.5 Exercises

4.1 *Construct a network flow example with 7 vertices and 11 directed edges, where each edge has a positive capacity and compute the maximum flow and minimum cut in this graph. You should show some of the steps in the algorithm. (Follow Edmonds-Karp shortest path heuristic).*

4.2 *Assume that we have a network flow graph $G = (V, E)$ with positive capacities on each of the edges and two specified vertices s and t. Suggest an efficient algorithm to find an edge in E, such that setting its capacity to zero (i.e. deleting this edge) will result in the largest decrease in the maximum flow in the resulting graph.*

4.3 *What is the total number of flow augmentations that are performed when in place of taking an arbitrary path in the residual network, we take the shortest path (i.e., the path with minimum number of links)? Please do not provide proof for this - just a couple of line of reasoning is sufficient.*

4.4 *Suppose we are given a flow network G, where edges have positive integer capacities, and $\mathcal{C} = \sum_{u,v \in V} c(u, v)$, where $c(u, v)$ is the capacity of the edge $e = (u, v) \in E$. Show the following*

1. *The value of the max flow is an integer.*
2. *There is an assignment of non-negative integer flow values on each edge of G, satisfying all the flow conservation conditions, so that G achieves max flow.*
3. *Show that the number of iterations required in the Ford-Fulkerson's algorithm (Residual network, find an augmenting path, augment the flow, repeat) is $O(\mathcal{C})$.*
4. *Show that in the worst case, Ford-Fulkerson's algorithm, as stated in Part 3 runs in exponential time.*
5. *Construct an example, where one can realize the worst case as stated in Part 4.*

4.5 *Let $G = (V, E)$ be the flow network. Let $C = \max_{(u,v) \in E} c(u, v)$ be the maximum capacity. Show the following:*

1. *Minimum cut of G has a capacity of at most $C|E|$.*
2. *For a given number $K > 0$, show how to find an augmenting path of capacity at least K in $O(|E|)$ time, provided that such a path exists.*
3. *Execute the following algorithm:*

```
(a) Initialize the flow $f = 0$;
(b) $K = 2^{\lfloor \log C \rfloor}$;
(c) While $K \geq 1$ do
    i. While there exists an augmenting path $p$ of capacity at least $K$ then augment flow $f$ along $p$.
    ii. $K := K/2$;
(d) Return $f$.
```

Show that the above algorithm computes Max-Flow.

4. *Show that the loop in Step 3c(i) is executed at most $O(|E|)$ times for each value of K.*

5. *Show that the algorithm runs in $O(|E|^2 \log C)$ time.*

4.6 *Let $G = (V, E)$ be a flow network. Recall that G is a complete graph, where some of the edges may have a capacity of zero. Suppose your task in the max flow problem is to increase the flow of a network as much as possible, but you are only allowed to increase the capacity of only one edge, whose capacity is strictly larger than zero. First show that there are networks where such an edge may not exist, i.e. increasing the capacity of a single edge (> 0 capacity) will not alter the value of the max-flow. Show that there are networks, where such an edge may exist. Try to design an algorithm which can detect whether flow can be increased.*

4.7 *A simple undirected graph $G = (V, E)$ is called k-edge connected if removal of any set of k-edges keeps G still connected. (e.g. cycles are 1-edge connected.) Show how to compute edge connectivity of G by invoking at most $|V|$ network flow computations.*

Chapter 5

Separators in a Planar Graph

This chapter is based on Kozen [64] and the famous paper of Lipton and Tarjan [67] on the planar separator theorem. Earlier we have seen that for a binary tree on n-nodes, there exists a node such that whose removal leaves no component having more than $2(n+1)/3$ nodes. This can be extended to outerplanar graphs, where we can remove a pair of vertices such that none of the components have more that $2(n+1)/3$ nodes. Usually this phenomenon is referred to as a balanced decomposition using small size separators. This is a 'key idea' in most of the divide and conquer type algorithms on these graphs. As can be seen that the depth of recursion will be $O(\log n)$ and since the size of the separator is small, the "merge" step will be economical as well. First we start with some preliminaries and then we will prove that in a planar graph there exists a separator of size $O(\sqrt{n})$.

5.1 Preliminaries

Definition 5.1.1 *A graph is called* planar *if the vertices and edges can be laid out (embedded) in the plane so that no two edges intersect except at their end points. An embedded planar graph is usually referred to as a* plane graph.

Definition 5.1.2 *In an embedded plane graph, we have vertices, edges and faces. The* dual *of a plane graph G is a planar graph G^* whose vertices correspond to faces of G and two vertices in G^* are joined together if the corresponding faces in G share an edge.*

Definition 5.1.3 *A plane graph G is* triangulated *if each of its face is a triangle, i.e., it is bounded be three edges. In other words, in the dual each vertex has degree three.*

Definition 5.1.4 *A set $S \subseteq V$ for a graph $G = (V, E)$ is called a* vertex separator, *if removal of vertices (and incident edges on these vertices) from G results in two disjoint sets of vertices $A, B \subseteq V$ with no edges between them. If the sizes of the sets A and B are a constant fraction of that of the size of V, then S is called as a* balanced separator.

Definition 5.1.5 *A planar graph $G = (V, E)$ consists of at most $|E| = 3|V| - 6$ edges. This follows from Euler's relation, i.e. $|V| - |E| + |F| = 2$. You may like to check the proof at*

`http:\\www.ics.uci.edu/~eppstein/junkyard/euler/`

Definition 5.1.6 *An* outerplanar graph *is a plane graph such that all its vertices lie on a single face. This face is usually referred to as the* outerface.

Definition 5.1.7 *The dual of a triangulated outerplanar graph is a binary tree.*

We have seen that a complete graph on five vertices, K_5, and a complete bipartite graph on six vertices, $K_{3,3}$, are nonplanar. It is easy to see that a tree is planar, and outerplanar graphs are planar. Both of these graphs admit small size separators. What we will prove in this chapter is that all planar graphs satisfy a similar property.

Theorem 5.1.8 *[Lipton and Tarjan [67]] Let $G = (V, E)$ be an embedded undirected triangulated planar graph, where $n = |V|$. There exists a partition of V into disjoint sets A, B, and S, such that*

1. $|A|, |B| \leq \frac{2n}{3}$
2. $|S| \leq 4\sqrt{n}$
3. *There is no edge in E that joins a vertex in A with a vertex in B.*
4. *Such a set S can be found in linear time.*

It will turn out that the way we prove this theorem, it will lead to a linear time algorithm (i.e. $O(|V| + |E|)$) for finding such a separator. Note that if the given graph is not embedded in the plane, then there is a linear time algorithm by Hopcroft and Tarjan that embeds it. In fact that algorithm also figures out in linear time whether the given graph is planar or not, and if it is planar it finds an embedding. Also if a plane graph is not triangulated, then it can be triangulated in linear time, by inserting required number of edges on each face. Other than this essentially we will use breadth first and the concept of fundamental cycles to prove this theorem.

5.2 Proof of the Planar Separator Theorem

Assume that the graph $G = (V, E)$ is undirected, connected, planar, triangulated and embedded. The first step in the proof/algorithm is to do a breadth-first search starting at an arbitrary vertex, say s, in G, and assign levels to vertices. Vertex s is at level 0, vertices adjacent to s are at level 1, vertices adjacent to level 1 vertices that have not been assigned any level are level 2 vertices, and so on. Let l be the last level, and pretend that there is a level $l+1$ which consists of no vertex (this is just required for the proof!). Let $L(t)$ denote the set of vertices that are in level t, $0 \leq t \leq l$. Recall that in BFS, no edge can span over two or more levels. All edges must connect vertices in the same level or consecutive levels. Observe that each of the level, $L(t)$, for $0 < t < l$, is a separator in its own right, although may not be of small size and may not lead to a balanced decomposition!

Number the vertices according to BFS ordering, where s gets number 1, followed by vertices in level 1, then vertices in level 2, and so on. Let t_1 be the middle level, that is the one which contains the vertex number $n/2$ in the BFS numbering. Consider the set $L(t_1)$. Note that $|\cup_{t<t_1} L(t)| < n/2$ and $|\cup_{t\leq t_1} L(t)| \geq n/2$. If $|L(t_1)| \leq 4\sqrt{n}$, then $S = L(t_1)$ and we are done. Note that in that case we can set the set A to be all the vertices in levels 0 up to the level $t_1 - 1$. Similarly the set B can be defined as all the vertices in levels $t_1 + 1$ to l. Clearly $|A| < n/2$ and $|B| < n/2$. In general, it is not necessary that $L(t_1)$ may satisfy the requirements on the size of the separator. Here is the lemma which will be very handy in that case.

Lemma 5.2.1 *There exists levels $t_0 \leq t_1$ and $t_2 > t_1$ such that, $t_2 - t_0 \leq \sqrt{n}$, $|L(t_0)| \leq \sqrt{n}$ and $|L(t_2)| \leq \sqrt{n}$.*

Proof: Note that $|L(0)| = 1$ and $|L(l+1)| = 0$. Let $t_0 \leq t_1$ be the largest number such that $|L(t_0)| \leq \sqrt{n}$. Let $t_2 > t_1$ be the smallest number such that $|L(t_2)| \leq \sqrt{n}$. Note that every level between t_0 and t_2 contains more than $\sqrt{n}$ vertices, therefore by pigeon hole principle they must be fewer than $\sqrt{n}$ levels between t_0 and t_2, otherwise G will have more than n vertices! Therefore, $t_2 - t_0 \leq \sqrt{n}$ $\square$

Define three sets C, D and E as follows: $C = \cup_{t<t_0} L(t)$, $D = \cup_{t_0<t<t_2} L(t)$ and $E = \cup_{t>t_2} L(t)$. If $|D| \leq 2/3n$, then we have the required separator, by setting $S = L(t_0) \cup L(t_2)$, A the largest of C, D or E and B the union of the other two.

What if $|D| > 2/3n$? Then both the sets C and E are small, have less than $1/3n$ vertices. We will find a $1/3 - 2/3$ separator S_D, of D, of size at most $2\sqrt{n}$. Let D be split into D' and D'' by S_D. Then S will include the vertices in $L(t_0), L(t_2)$, and the separator vertices S_D. Set $A = \max(C, E) \cup \min(D', D'')$ and $B = \min(C, E) \cup \max(D', D'')$. Observe that S, A, and B satisfy the required size criteria.

Next we will present some ideas regarding finding the separator S_D of D. First we remove all the vertices that are not in D, except the start vertex s. We connect s to all the vertices in level $t_0 + 1$. This can be done still preserving the planarity of D, since the original graph is planar. Now we construct a spanning tree T in D, such that its diameter is at most $2\sqrt{n}$. Start with vertices in level $L(t_2 - 1)$. For each vertex in this level, choose one of the vertex in the previous level $L(t_2 - 2)$, adjacent to it as its parent. Continue this process with vertices in levels $t_2 - 2$, $t_2 - 3$, $\cdots$, to obtain the tree T. Next we state two lemmas, that are relatively easy to prove, that will show the critical property relating the tree T, the plane graph D, its dual D^*, and the dual tree T'.

Lemma 5.2.2 *Let $G = (V, E)$ be a connected plane graph and G^* be its dual. For any $E' \subseteq E$, the subgraph (V, E') has a cycle if and only if the subgraph $(V^*, E - E')$ of G^* is disconnected.*

Lemma 5.2.3 *Let $G = (V, E)$ be a connected plane graph with dual $G^* = (V^*, E)$ and let $E' \subseteq E$. Then (V, E') is a spanning tree of G if and only if $(V^*, E - E')$ is a spanning tree of G^* (see Figure 5.1 for an illustration).*

Let E_T be the edges of the spanning tree T, constructed by following the parents in D as stated above. Recall that the diameter of T is at most $2\sqrt{n}$. Also D is triangulated. Consider the dual D^* of D, and consider the edges in $E - E_T$. They define a spanning tree T' in D^* (by Lemma 5.2.3). Also we can orient each edge in T' away from the root. Pick a face of D (say its outer face) and choose this as the root T'. It will turn out that the required separator S_D will be defined by an edge, $e = (u, v)$ in $e \in E - E_T$, and the path in the tree T between u and v. In other words, e defines a unique cycle, $c(e)$, in T. The cycle $c(e)$ is referred to as a fundamental cycle in literature. To compute/define $c(e)$ appropriately we first perform a DFS of T' and compute the following three quantities.

1. $I(e)$= number of vertices which are in the interior of the cycle $c(e)$.
2. $|c(e)|$= number of vertices on the cycle $c(e)$.
3. Linked list representation of $c(e)$.

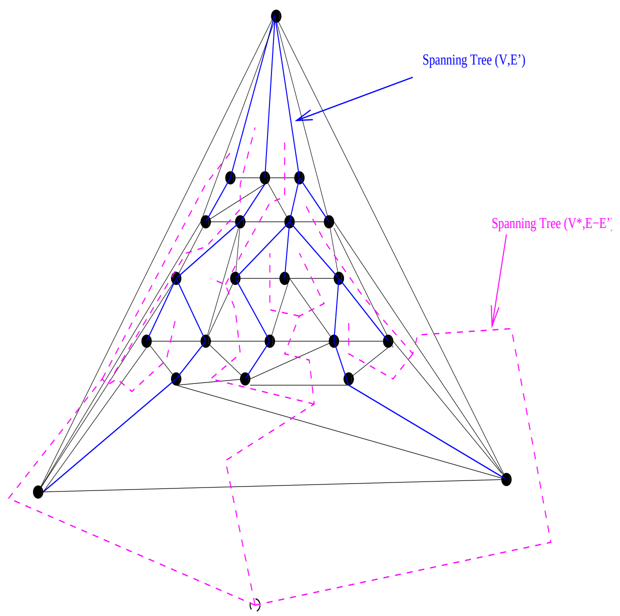


Figure 5.1: The edges of (V, E') are in bold-solid. The edges of $(V^*, E - E')$ are dashed-bold and the planar graph edges are in solid.

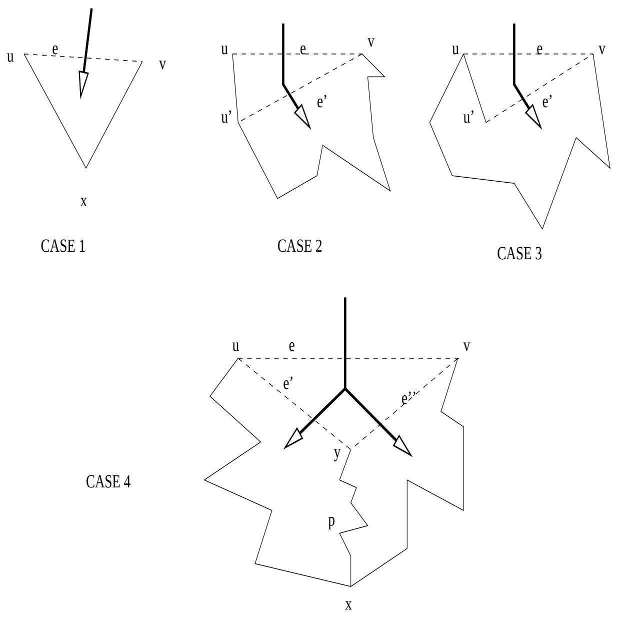


Figure 5.2: Bold edges represent the DFS traversal of faces of D corresponding to the tree T'. Dashed edges corresponds to edges in $E - E_T$ and solid edges corresponds to edges in E_T, i.e. a spanning tree of D.

For each step of DFS, one of the following four cases will occur (see Figure 5.2)

Case 1: DFS visits a leaf of T' (i.e. a triangular face of D). Then $I(e) = 0$, $|c(e)| = 3$, and $c(e) = \{x, u, v\}$.

Case 2: DFS visits a triangle corresponding to an edge $e = (u, v) \in E - E_T$, its degree is two and the other edge of the triangle is $e' = (u', v) \in E - E_T$ which was visited in the previous step. Moreover $u' \in c(e)$. Then $I(e) = I(e')$, $|c(e)| = |c(e')| + 1$, and $c(e) = uc(e')$.

Case 3: DFS visits a triangle corresponding to an edge $e = (u, v) \in E - E_T$, its degree is two and the other edge of the triangle is $e' = (u', v) \in E - E_T$ which was visited in the previous step. Moreover $u' \notin c(e)$. Then $I(e) = I(e') + 1$, $|c(e)| = |c(e')| - 1$, and $c(e)$ equals to $c(e')$ except that u' is removed from the front of the list.

Case 4: DFS visits a triangle corresponding to edge $e = (u, v) \in E - E_T$ and its degree is three. The other two edges $e' = (u, y) \in E - E_T$ and $e'' = (vy) \in E - E_T$ have been already visited by the DFS. Let p be the common path between the cycles $c(e')$ and $c(e'')$. One of the end points of p is x, and the other end point is y. Then $I(e) = I(e') + I(e'') + |p| - 1$, $|c(e)| = |c(e')| + |C(e'')| - 2|p| + 1$, and $c(e)$ consists of $c'xc''$, where c' is the cycle $c(e')$ with path p removed, and similarly c'' is the cycle $c(e'')$ with path p removed.

Lemma 5.2.4 *In the above setting of the graph D, there exists an edge $e \in E - E_T$ such that $I(e) \leq 2/3n$ and $n - (I(e) + |c(e)|) \leq 2/3n$.*

Proof. Let $e \in E - E_T$ be the first edge in the leaf to root path in T' such that $I(e) + |c(e)| \geq n/3$. Then $n - (I(e) + |c(e)|) \leq 2/3n$. We will prove that $I(e) \leq 2/3n$. The edge e corresponds to one of the four cases encountered in the DFS.

1. In Case 1, $I(e) = 0 \leq 2/3n$.

2. In Case 2, $I(e) + |c(e)| = I(e') + |c(e')| + 1$, and $I(e') + |c(e')| < n/3$, and hence $I(e) + |c(e)| \leq 2/3n$.

3. In Case 3, since $I(e) + |c(e)| = I(e') + |c(e')|$, e cannot be the first edge with this property.

4. In Case 4, $I(e') + |c(e')| < n/3$ and so is $I(e'') + |c(e'')| < n/3$. $I(e) + |c(e)| = I(e') + I(e'') + |p| - 1 + |c(e')| + |c(e'')| - 2|p| + 1 \leq 2/3n - |p| \leq 2/3n$.

∎

5.3 Generalizations of the Planar Separator Theorem

This section is contributed by Alexis Beingessner. In the previous section we saw that the planar separator theorem provides us with a procedure to separate the vertices of a planar graph $G = (V, E)$, $|V| = n$, into three sets A, B, S, where $|A|, |B| \leq 2n/3$, $|S| \leq 4\sqrt{n}$, and there exists no edge between A and B. In this section we will consider several generalizations of this theorem.

5.3.1 Weighted Separators

In our version of the planar separator theorem we considered all vertices to be equal. However, a common variant permits vertices to be weighted such that the sum of all weights is equal to 1 (any other set of non-negative weights can be trivially mapped to one like this). The only difference is that instead of bounding the sizes of the the sets A and B to be $\leq 2n/3$, we bound their weights to be $\leq 2/3$. The separating set S however is still bounded in terms of the number of vertices it contains, and may therefore have arbitrary weight.

To prove the weighted planar separator theorem, our proof from the previous section is sufficient. We need only change certain references to the sizes of sets to refer to the weight of the sets. The rest of the analysis largely follows unchanged.

From now on, we will assume that the planar separator theorem refers to the weighted variant of the planar separator theorem. If no weights are specified, we will assume that all vertices have equal weight, which coincides with our original definition.

5.3.2 r-Divisions

In this section we will use the planar separator theorem to construct a more general graph partitioning. The contents of this section are based on a paper by Frederickson [31].

We define a *region* to be a subset of the vertices of a graph $G = (V, E)$. An *interior vertex* of a region R is contained only in R, and adjacent only to other vertices in R. A *boundary vertex* is one that is shared between at least two regions. All vertices will be either boundary or interior. Given a parameter r, we will divide the graph into $\Theta(n/r)$ regions with $O(r)$ vertices each, and $O(\sqrt{r})$ boundary vertices each. Such a division will be called an *r-division.* Remark that the planar separator theorem provides an n-division by taking the two sets $A \cup S$ and $B \cup S$.

We begin with a potential naive algorithm. Start with a single region containing all of V. While any region R contains more than r vertices, apply the planar separator theorem on R to produce A, B, S. Now replace R with $R' = A \cup S$ and $R'' = B \cup S$.

Clearly this procedure produces regions with no more than r vertices, and since it reduces the size of a region by at most $2/3$ until this bound is satisfied, it follows that each region contains $\Theta(r)$ vertices. Consequently, there must be $\Theta(n/r)$ regions. However the number of boundary vertices is more complicated. Remark that initially we have a single region that is all interior vertices. Further, A and B consist entirely of interior vertices after constructing R' and R'', and S consists entirely of boundary vertices. Therefore we introduce at most $4\sqrt{n}$ boundary vertices at each recursive step.

To determine the number of boundary vertices, we define $b(v)$ for some vertex v to be one less than the number of regions it is contained in, and $B(n, r)$ to be the sum of $b(v)$ for all $v \in V$. Remark that $B(n, r)$ is strictly greater than the number of boundary vertices, as $b(v) \geq 1$ for all boundary vertices, by definition. Our algorithm gives us the following recurrence:

$$\begin{array}{rcll} B(n,r) & \leq & 4\sqrt{n} + B(\alpha n + O(\sqrt{n}), r) + B((1-\alpha)n + O(\sqrt{n}), r) & \text{for } n > r \\ B(n,r) & = & 0 & \text{for } n \leq r, \end{array}$$

where $1/3 \leq \alpha \leq 2/3$. This recurrence can be solved for $B(n, r) \leq 4n/\sqrt{r} - O(\sqrt{n})$. Therefore, the number of boundary vertices produced by this algorithm is $O(n/\sqrt{r})$. However this tells us nothing about the number of boundary vertices *per region.* Indeed, some regions may have many boundary vertices. To resolve this, we perform further processing on regions with more than $c\sqrt{r}$

boundary vertices, for some constant c. Given such a region R, we set all k boundary vertices of R to have weight $1/k$, and all interior vertices of R to have weight 0. We then apply the planar separator theorem to R and replace R as before. Since only the boundary vertices have weights, the planar separator theorem will split up the boundary vertices among the two resultant regions. Therefore, after enough iterations all regions will have few enough boundary vertices. Further, since the regions are still strictly shrinking, we cannot have violated the bounds on the size of the region. It remains to be proven that we have not violated the constraint on the maximum number of regions.

If a region has $i > c\sqrt{r}$ boundary vertices, then at most $di/(c\sqrt{r})$ splits will be performed, for some constants c and d. This will result in at most $di/(c\sqrt{r})$ new regions. If t_i is the number of regions with i boundary vertices, then the number of new regions will be at most

$$\sum_i (di/c\sqrt{r})t_i = O(n/r)$$

Therefore, our modified algorithm produces an r-division.

Remark that in our construction, our recursion tree has a depth of $O(\log(n/r))$. Further, each level has us do $O(n)$ work. Therefore, this algorithm runs in $O(n\log(n/r))$ time.

Further processing on the graph can provide our r-division with additional properties such as regions having a constant number of neighbors, and boundary vertices being shared between at most a constant number of regions.

5.3.3 Edge Separators

Up until now, we have only considered vertex separators. Edge separators are exactly the same as vertex separators, except that instead of removing vertices, we wish to remove edges. Specifically, given a graph $G = (V, E)$ we wish to find a cut-set $S \subseteq E$ that separates V into two disjoint subsets A, B. Every edge in S has one endpoint in A and one endpoint in B, and every edge in $E \setminus S$ has both of its endpoints in only A or B. In general we would like to ensure that A and B are approximately the same size, and S is small.

For graphs with low (e.g. constant) maximum degree, the results for edge separators are generally very similar to those for vertex separators. However on arbitrary planar graphs, edge separators perform much worse.

For instance, consider a graph $G = (V, E)$ in which every vertex has degree 1, except for some vertex v with degree $n-1$. This graph is a tree, and therefore planar. An excellent vertex separator for G would be $\{v\}$, as it would disconnect the entire graph, allowing us to pick any subsets of $V \setminus \{v\}$ we want for our separated sets. However an edge separator would have to remove a linear number of edges to get balanced sets.

From this example it is clear that not all results for vertex separators hold for edge-separators. In general, vertex separators are more powerful, as for every edge an edge-separator would need to remove, a vertex separator would need to remove at most one vertex, but potentially far fewer. Equivalently, if a vertex separator includes some vertex v, an edge separator would need to include every edge of v to achieve the same result. Consequently, results on edge separators often include factors based on the maximum or average degree of the graph [32].

We conclude our look at the planar separator theorem and its generalizations with a table of separator results. There are far too many results on separators with special requirements and for special classes of graphs to adequately report here. As a result, this table is by no means

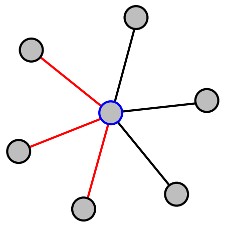


Figure 5.3: The optimal vertex separator (blue) for this graph is much smaller than a similarly balanced edge separator (red)

---

comprehensive. (Note that $\Delta(G) = \sum_{v \in V} deg(v)^2$.)

Separator	Graph	# of Sets	Set Sizes	Separator Size	Time	Reference
Vertex or Edge	Tree	2	$\leq 2n/3$	1	$O(n)$	[75]
Vertex	Planar	2	$\leq 2n/3$	$O(\sqrt{n})$	$O(n)$	[67]
Vertex	Planar	$\Theta(n/r)$	$O(r)$	$O(n/\sqrt{r})$	$O(n \log(n/r))$	[31]
Vertex	Genus g	-	$\leq \epsilon n$	$O(\sqrt{(g+1/\epsilon)n})$	$O(n+g)$	[2]
Vertex	Planar	-	$tw(G)$	$\leq 4\sqrt{2\sigma(G)/t}$	$O(n + T_{SSSP}(G))$	[3]
Edge	Planar	-	$tw(G)$	$\leq 4\sqrt{2\Delta(G)/t}$	$O(n + T_{SSSP}(G))$	[3]

5.4 Exercises

5.1 *Let T=(V,E) be a connected undirected tree such that each vertex has degree at most 3. Let n=|V|. Show that T has an edge whose removal disconnects T into two disjoint subtrees with no more than (2n+1)/3 vertices each. Give a linear time algorithm to find such and edge; prove its correctness.*

5.2 *Provide an algorithm running in $O(n \log k)$ time to partition the binary tree on n vertices into k ($k \leq n$) subtrees, so that each of the subtree is of size at most $(2/3)^k n$. Try to see whether you can improve the running time of this algorithm (this is not easy!).*

5.3 *Prove the weighted version of the planar-separator theorem. Let $G = (V, E)$ be an embedded undirected triangulated planar graph, where $n = |V|$. Each vertex $v \in V$ has a positive weight $w(v) \geq 0$ and $\sum_{v \in V} w(v) = 1$. There exists a partition of V into disjoint sets A, B, and S, such that*

1. *$w(A), w(B) \leq \frac{2}{3}$, where $w(A)$ is the sum total of weights of all the vertices in set A*
2. *$|S| \leq 4\sqrt{n}$*
3. *There is no edge in E that joins a vertex in A with a vertex in B.*
4. *Such a set S can be found in linear time.*

5.4 *Prove Lemma 5.2.2. Let $G = (V, E)$ be a connected planar graph and G^* be its dual. For any $E' \subseteq E$, the subgraph (V, E') has a cycle if and only if the subgraph $(V^*, E - E')$ of G^* is disconnected.*

5.5 *Let $T = (V, E)$ be a connected undirected tree such that all of its vertices have degree at most 3. Let $n = |V|$. Show that T has an edge whose removal disconnects T into two disjoint subtrees with no more than $\frac{2n+1}{3}$ vertices each.*

5.6 *Consider the following version of the (weighted) planar-separator theorem. Let $G = (V, E)$ be an embedded undirected triangulated planar graph, where $n = |V|$. Each vertex $v \in V$ has a positive weight $w(v) \geq 0$ and $\sum_{v \in V} w(v) = 1$. There exists a partition of V into disjoint sets A, B, and S, such that*

1. *$w(A), w(B) \leq \frac{2}{3}$, where $w(A)$ is the sum total of weights of all the vertices in set A*
2. *$|S| \leq 4\sqrt{n}$*
3. *There is no edge in E that joins a vertex in A with a vertex in B.*
4. *Such a set S can be found in linear time.*

Show what changes you need to make in the proof of (unweighted) Planar Separator Theorem to prove the above theorem.

5.7 *Prove the following: Let $G = (V, E)$ be a connected planar graph and G^* be its dual. For any $E' \subseteq E$, the subgraph (V, E') has a cycle if and only if the subgraph $(V^*, E - E')$ of G^* is disconnected.*

5.8 *Prove the following theorem on Geometric Separators. In 2-dimensions assume that you have n squares of arbitrary sizes. Squares are axis aligned. Moreover none of the points in the plane is inside more than k-squares. Prove that there exists either a vertical or a horizontal line which partitions the set of squares in such a way that at least $\lfloor \frac{n+1-k}{4} \rfloor$ of squares interiors lie to each side of the line. How fast you can find such a line?*

Chapter 6

Lowest Common Ancestor

Given a rooted binary tree T on n nodes, we are asked to preprocess it in $O(n)$ time so that the following type of queries can be answered in $O(1)$ time. Given any two nodes u and v of T, report their Lowest Common Ancestor $LCA(a, b)$, i.e., among all the common ancestors of nodes a and b, find the one which is furthest from the root of T. This subproblem arises in many graph applications. Orginal algorithm is due to Harel and Tarjan [1984]. Many years later, Schieber and Vishkin [1993] proposed a new algorithm for the same problem while studying parallel algorithms. Both of these algorithms are fairly complex and are considered to be far from being implementable. Recently, Bender and Farach-Colton [2000] proposed a fairly simple algorithm for the LCA problem, and thats what we present in this chapter.

It is well known that the following Range Minima Problem (RMQ) is related to the LCA problem. Given an array $A[1...n]$ consisting of n numbers, preprocess it so that given any two indices i and j, where $1 \leq i \leq j \leq n$, report the minimum element (or its index in A) in the subarray $A[i...j]$. Next we will show the reduction of the LCA problem to RMQ problem, and then provide a solution for the RMQ problem.

6.1 LCA $\rightarrow$ RMQ

Let T be the given binary rooted tree. Consider the depth first search traversal of T. Observe that the shallowest node encountered in the depth first traversal of T between u and v is the node corresponding to $LCA(u, v)$. (Recall that the main property of dfs traversal is that once it enters a subtree, then it completely visits all the nodes in the subtree - this sort of corresponds to a nice bracketing sequence.) Our aim is to find this node using the RMQs.

Corresponding to the dfs traversal of T, let E be the Euler tour. Recall that E stores the nodes of T in the same order as they are visited during the dfs traversal. The tour E consists of $2n - 1$ entries. Let the level of a node in T be its distance from the root. Corresponding to E, define a level array $L[1...2n - 1]$ which stores the level of the node $E[i]$ in $L[i]$. Furthermore, observe that a node may appear several times in Euler tour. For each node $x \in T$, we maintain an index $R(x)$ that stores the index of the first appearance of x in E. Given our notation, the nodes between $E[R(u), ..., R(v)]$ are nodes in Euler tour between the first visits of u and v. What is the shallowest node among the nodes in $E[R(u), ..., R(v)]$? For this we will look at the corresponding entries in the level array L. More precisely, we need to report what is the minimum element in the subarray $L[R(u)...R(v)]$; this returns us the index of the shallowest node (one with the smallest level) and denote this by $RMQ_L[R(u)...R(v)]$. Hence, $LCA(u,v) = E[RMQ_L[R(u)...R(v)]]$.

Lemma 6.1.1 *LCA problem on a rooted binary tree T of n nodes can be converted to the range minima query problem on an array L of size $2n-1$ elements. The reduction takes $O(n)$ time. Moreover, LCA queries can be answered within $O(1)$ time in addition to the time required to answer the range minima queries on L.*

Proof. Notice that the depth first traversal and the construction of Euler tour of T can be done in $O(n)$ time. Within the same time bounds we can maintain the level array as well as keep track of the first appearance of each node in Euler tour. Hence the conversion can be done in linear time. Given the query, $LCA(u,v)$, we need to find the representatives $R(u)$ and $R(v)$ in E, then need to answer the query $RMQ_L[R(u)...R(v)]$ followed by one more look up in the array E to report the node corresponding to $LCA(u,v)$. This computation only requires a few pointer manipulation and hence requires $O(1)$ time in addition to answering the range minima query. ∎

6.2 Range Minima Queries

Let A be the array of length n consisting of numbers. Our task is to preprocess A so that the range minima queries $RMQ(i,j)$, $1 \le i \le j \le n$, can be answered in $O(1)$ time.

6.2.1 A naive $O(n^2)$ algorithm

A simple way to achieve a constant query time is to precompute and store minima for each possible query. In all there are $O(n^2)$ possible queries of type $RMQ(i,j)$, where $1 \le i \le j \le n$, and for each of them we can compute and store the minima in the range $A[i,...,j]$. It is easy to see that this computation can be done in $O(n^2)$ time and then given a query it can be answered in $O(1)$ time.

6.2.2 An $O(n \log n)$ algorithm

In place of precomputing minima for each possible query, now we precompute minima's for only $O(n \log n)$ selected types of queries. For every i between 1 and n and for every j between 1 and $\log n$, we find minimum element in the subarray $A[i,...,i+2^j]$ (we are sloppy with boundary conditions here to keep it simple) and store it in a table in location $M[i,j]$. Next we show that using dynamic programming the table M can be computed in $O(n \log n)$ time. Minima in a subarray of size 2^j is computed by looking at the minima of two constituent blocks of size 2^{j-1}. Either $M[i,j] = M[i,j-1]$ or $M[i,j] = M[i+2^{j-1}-1, j-1]$.

How do we answer a range minimum query in $O(1)$ time? Let the query be $RMQ(i.j)$, where $1 \le i \le j \le n$. First compute $k = \lfloor \log_2 j - i \rfloor$. Now observe that 2^k is the largest interval, that is a power of 2, that fits in the range from i to j. Compute $RMQ(i,j)$ be finding out the minimum of two entries in the table, namely $M[i,k]$ and $M[j-2^k+1, k]$. Notice that these two table values have been precomputed and hence query can be answered in $O(1)$ time.

Lemma 6.2.1 *An array A consiting of n numbers can be preprocessed in $O(n \log n)$ time so that the range minima queries can be answered in $O(1)$ time.*

6.2.3 An $O(n)$ algorithm with ± 1 property

Consider the following special case of the array A where each element differs from its previous element either by a $+1$ or a -1 (this is especially true for the LCA problem as levels of consecutive nodes in Euler tour differs by 1). We will show that in this case A can be preprocessed in $O(n)$ time and RMQs can be answered in $O(1)$ time.

The strategy is pretty simple. First we partition array A into subarrays, where each subarray is of size $\frac{\log n}{2}$ (we are assuming that n is a nice power of 2, otherwise we have to use floors and ceilings and that will not add anything more in terms of understanding.) Within each subarray we find the minimum value and then store all these minimas in an array A'. Notice that the size of the array A' is $\frac{2n}{\log n}$ and hence it can be preprocessed in $O(n)$ time by using Lemma 6.2.1.

Consider a range minima query $RMQ(i, j)$ in array A, where $i \leq j$. It is answered as follows: Indices i and j may fall within the same subarray, therefore we need to preprocess each subarray for answering RMQs. If i and j fall in different subarrays then we compute the following three quantities:

1. Minimum value starting at index i up to the end of the subarray containing i.
2. Minimum value among the subarrays between the subarray containing i and j. This is computed using the preprocessing done for A' in constant time.
3. Minimum value from the beginning up to the index j within the subarry containing j.

Now our subproblem is reduced to solving the RMQ problem in subarrays of size $\frac{\log n}{2}$ with ± 1 property. The key observation here is that we do not have too many different kinds of these subarrays.

Claim 6.2.2 *Given two arrays of same size where each element in the first array is constant value more than the corresponding element in the second array, then the answer to RMQ queries (i.e. the index) is identical in both the arrays.*

Essentially the preprocessing and the RMQ queries work with relative order of elements in these arrays, and they do not need actual values of the elements. Hence for the two subarrays within the above claim, same preprocessing is sufficient to answer RMQ queries. We normalize each of the subarrays by first subtracting the initial value from each of the elements. Next we show that there are only $O(\sqrt{n})$ normal subarrays.

Claim 6.2.3 *There are at most $O(\sqrt{n})$ normalized subarrays. Each subarrays has length $\frac{\log n}{2}$, where the first element is a 0, and the elements in the array satisfy ± 1 property.*

Proof. Each normalized subarray can be specified by a ± 1 vector. Therefore, there are only $2^{\frac{1}{2}\log n - 1} = O(\sqrt{n})$ different types of subarrays of length $\frac{1}{2}\log n$. ■ We preprocess each of these

subarrays in $O(\log^2 n)$ time to answer RMQ queries in $O(1)$ time using the naive algorithm. The preprocessing requires in all $O(\sqrt{n}\log^2 n)$ time. We summarize the results in the following.

Lemma 6.2.4 *An array A consisting of n-numbers satisfying the ± 1 property can be preprocessed in $O(n)$ time so that the range minima queries can be answered in $O(1)$ time.*

Corollary 6.2.5 *A binary tree on n-nodes can be preprocessed in $O(n)$ time so that the lowest common ancestor queries can be answered in $O(1)$ time.*

6.3 RMQ → LCA

Next we show that an instance of the RMQ problem can be converted to an instance of the LCA problem. For a linear array A of size n, the tree T for the LCA problem consists of n nodes and given a RMQ query, we perform an equivalent LCA query on T, and whose answer in turn provides the answer for the original range minima query. This will imply that the general RMQ problem (i.e., even without the ± 1 property) can be answered in $O(1)$ time by performing an $O(n)$ time preprocessing. The key to this conversion is the concept of Cartesian Tree.

Let $A[1...n]$ be the input array on which we need to perform RMQ queries. Cartesian tree T for A is defined as follows. It is a rooted binary tree and the root of T stores the index of the smallest element in A. Deleting the minimum element from A splits it into two subarrays. Left and right children of the root are recursively defined Cartesion trees for left and right subarrays of A, respectively.

Claim 6.3.1 *Cartesian tree T for an array A of n-numbers can be constructed in $O(n)$ time.*

Proof. We scan the array A from left to right and incrementally build the Cartesian tree $T = T_n$ as follows. Suppose so far we have built the tree T_i with respect to elements $A[1..i]$ and we want to extend it for $A[1...i+1]$ to obtain T_{i+1}, where $i < n$. Main observation is that the node storing the index $i+1$ in T_{i+1} is on the rightmost path of T_{i+1}. We start at the rightmost node of T_i and follow the parent pointers till we find the location to insert $i+1$ in Cartesian tree. Note that each comparison will either add a node or removes one from the rightmost path. Since each node can only join the rightmost path once (if it leaves it then it can't be back to the rightmost path), therefore the total time in constructing T is $O(n)$. ■

Claim 6.3.2 *Let A be the array on n-numbers and T be the corresponding Cartesian tree storing the indices of elements in A in its node. Then $RMQ(i, j) = LCA(i, j)$.*

Proof. This follows from the recursive definition of Cartesian tree T. Let $k = LCA(i, j)$ in T. Observe that the node labeled k is the first node that separates i with j. In other words, the element $A[k]$ is the smallest element between $A[i]$ and $A[j]$, i.e. $RMQ[i, j] = k$. ■

6.4 Summary

In this chapter, we have shown that the lowest common ancestor query in a rooted binary tree on n-nodes can be answered by solving the range minima query in an array consisting of $2n-1$ numbers satisfying the ± 1 property. Moreover, the general RMQ problem in an array can be reduced to solving LCA queries on the corresponding Cartesian tree. All our preprocessing algorithms require linear time and the queries can be answered in constant time.

6.5 Exercises

6.1 *Prove that in the LCA algorithm of Bender and Farach-Colton, why does the reduction from the LCA problem to the range-minima query works, i.e., show that in place of finding the LCA of* *nodes u and v in the binary tree, why does it suffice to compute the smallest level number in the level array in an interval defined by the first occurrence of the node u an v in the level array.*

6.2 *This problem is to show that an arbitrary range minima query (RMQ) problem can be solved within the same complexity as the one with the ± 1 RMQ problem. Recall that the ± 1 RMQ problem for an array of size n required $O(n)$ time to preprocess and then the queries were answered in $O(1)$ time. The idea is to reduce an arbitrary RMQ problem to the LCA problem. This reduction uses Cartesian Tree. Let A be an array consiting of n numbers (need not satisfy the ± 1 property). The Cartesian Tree C for A is defined as follows: The root of C is the minimum element of A, and it stores the position of this element in the array. Removing the root element splits the array into left and right subarrays. The left and right children of the root are recursively constructed Cartesian trees of the left and right subarrays, respectively. Prove the following:*

1. *Cartesian tree C of an array A of size n can be computed in $O(n)$ time (use incremental construction).*

2. *Show that $RMQ_A(i,j) = LCA_C(i,j)$ (Recall that in C we store the indices i and j.)*

Chapter 7

Locality-Sensitive Hashing

Locality-Sensitive Hashing (LSH) is a method which is used for determining which items in a given set are similar. Rather than using the naive approach of comparing all pairs of items within a set, items are hashed into buckets, such that similar items will be more likely to hash into the same buckets. As a result, the number of comparisons needed will be reduced; only the items within any one bucket will be compared. Locality-sensitive hashing is often used when there exist an extremely large amount of data items that must be compared. In these cases, it may also be that the data items themselves will be too large, and as such will have their dimensionality reduced by a feature extraction technique beforehand.

The main application of LSH is to provide a method for efficient approximate nearest neighbor search through probabilistic dimension reduction of high-dimensional data. This dimensional reduction is done through feature extraction realized through hashing (eg. minhash signatures), for which different schemes are used depending upon the data. LSH is used in fields such as data mining, pattern recognition, computer vision, computational geometry, and data compression. It also has direct applications in spell checking, plagiarism detection, and chemical similarity.

In this chapter, we will give an illustration of the technique using the problem of finding similar documents, while concurrently showing supporting theory. Following this will be core theory including the definition of locality-sensitive hash families, and examples of families for different distance measures. Throughout the chapter, examples are also given for the application of families to illustrate their usage.

7.1 Similarity of Documents

We'll begin to introduce LSH by an illustration of the technique using the problem of finding similar documents. This problem appears, for example, on the web when attempting to find similar, or even duplicate web pages. A search engine would use this technique to allow these similar documents to either be grouped or only shown once on a results page, so that other possible search matches could also be prominently displayed.

Problem 7.1.1 *Given a collection of web-pages, find the near duplicate web-pages.*

Clearly the content of the page is what matters, so for a preprocessing step any HTML tags are stripped away and main textual content is kept. Typically multiple white spaces are also replaced by a singe space during this preprocessing. As a result we're left with a document containing a string of text characters. Keeping in mind that we want to compare similarity of documents opposed to exact equality, the text is split up into a set of smaller strings by a process called shingling. This allows the documents to be represented as sets, where fragments of documents can match others. Shingle length is an important factor in this, though will not be explained in depth. Generally though, the shingle length k should be large enough so that the probability of any given k-shingle appearing in any given document is relatively low. For our purposes, it is sufficient to know that choosing $k = 5$ works well for electronic mail, and $k = 9$ is suitable for large text documents.

Definition 7.1.2 k-shingle: *A k-shingle of a text document is defined to be any substring of length k which appears in the document.*

Example 7.1.3 *Let the document $D = \{adbdabadbcdab\}$, and $k = 2$. Then the possible k-shingles for D are: $\{ad, db, bd, da, ab, ba, bc, cd\}$.*

Using shingling, we have seen that a document can be represented as a set of k-shingles. However we can work instead with integers by using a hash function to map each k-shingle to an integer. This is also useful to reduce space requirements, as a regular 9-byte shingle can be converted to a 4-byte integer. Note however that even if we represent shingles this way, we still may be using 4 times more space than the original document. A solution to this is shown in the next section.

Example 7.1.4 *Let $k = 9$, and let $\mathcal{H}$ be a hash function that maps the set of characters to integers; $\mathcal{H} : |C|^9 \to \mathbb{Z}$. Let $|\mathbb{Z}| \leq 2^{32} - 1$.*

Now that we have documents mapped to sets of k-shingles, we can use a similarity measure called Jaccard Similarity to compare the two sets. The Jaccard Similarity is defined with respect to two sets S and T, and is the ratio of the size of the intersection of S and T to the size of their union. As a result of this, we can also redefine our problem statement.

Problem 7.1.5 *Given a collection of sets, find the pairs which have large intersections.*

Definition 7.1.6 Jaccard Similarity: *Let S and T be two sets. Define the Jaccard Similarity of the sets S and T as $\frac{|S \cap T|}{|S \cup T|}$. See Figure 7.1 for an illustration.*

7.2 Similarity-Preserving Summaries of Sets

As mentioned in the previous section, we'll now present a solution for the storage of sets of shingles using smaller representations called signatures. These signatures will also have the property that the similarity measure between any two will be approximately the same as the similarity between the two sets of shingles which they are derived from.

First, let's consider the natural representation of a set. We have a universe U from which elements of the set are drawn, which we'll assume are arbitrarily ordered. A set S with elements from U can be represented by a 0-1 vector of length $|U|$, where a 1 represents that the corresponding element from the universe is present in the set, and a 0 represents that element's absence from the set. Similarly for a collection of sets over the universe U, we can associate a *characteristic matrix*, where each column represents the vector corresponding to a set and each row corresponds to an element of U.

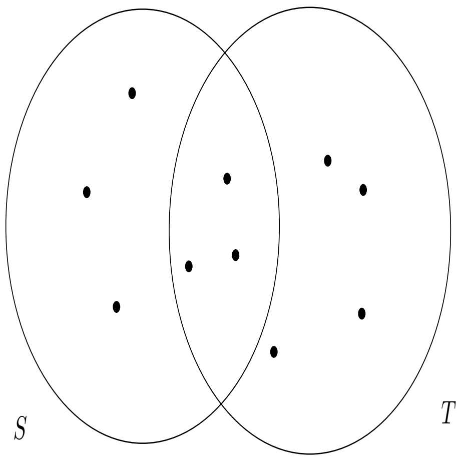


Figure 7.1: Jaccard similarity of two sets. The intersection $|S \cap T| = 3$ and the union $|S \cup T| = 10$. Therefore their Jaccard similarity is 3/10.

An example is given below in Table 7.1, where we have four sets (representing households in a neighborhood) and a universe U consisting of five elements (possible vacationing destinations). From this *characteristic matrix*, we can ascertain that set S_1 prefers cruise and safari, while S_2 just loves to visit resorts.

	S_1	S_2	S_3	S_4
Cruise	1	0	0	1
Ski	0	0	1	0
Resorts	0	1	0	1
Safari	1	0	1	1
Stay at Home	0	0	1	0

Table 7.1: A characteristic matrix for 4 sets with a universe consisting of 5 elements.

One effective way to compute the signature for a collection of sets is to use *minhashing*. For *minhashing*, the rows of the characteristic matrix are first randomly permuted. Then for each set (column in the characteristic matrix), its minhash value h is the number of the first row which is a 1. Using the previous example, suppose the permutation of rows results in Table 7.2 as shown below. The minhash values are then: $h(S_1) = 2$, $h(S_2) = 4$, $h(S_3) = 1$, and $h(S_4) = 2$.

In the following lemma we establish an important connection between the Jaccard similarity of two sets and the probability that their minhash values are the same after a random permutation of the rows of the characteristic matrix. In fact, we'll be able to shown that the Jaccard similarity is equal to the probability that these minhash values are the same.

Lemma 7.2.1 *For any two sets S_i and S_j, the probability that $h(S_i) = h(S_j)$ is equal to the Jaccard similarity of S_i and S_j.*

Proof. Focus on the columns representing the sets S_i and S_j in the characteristic matrix before the random permutation. For any row, the entries corresponding to these columns are either; (a)

	S_1	S_2	S_3	S_4
Ski	0	0	1	0
Safari	1	0	1	1
Stay at Home	0	0	1	0
Resorts	0	1	0	1
Cruise	1	0	0	1

Table 7.2: Characteristic matrix after the permutation of rows of Table 7.1.

both 0, (b) both 1, or (c) one is 0 and the other is 1. Let X be the number of rows of type (b) and let Y be the number of rows of type (c). Observe that the Jaccard similarity of S_i and S_j is $\frac{|X|}{|X|+|Y|}$.

Now, what is the probability that when we scan the rows from top to bottom, after the random permutation, we meet a type (b) row before a type (c) row? This is exactly $\frac{|X|}{|X|+|Y|}$, which is also precisely when $h(S_i) = h(S_j)$. ■

Next, we'll turn our attention to *minhash signatures*, which are smaller matrices created from repeatedly minhashing a characteristic matrix. Let M denote the characteristic matrix for a set system S with universe U. Pick a set of n random permutations of rows of the characteristic matrix. For each set in S compute its h-value with respect to each of the n-permutations. This results in a *signature matrix* — it consists of $|S|$ columns and n-rows, and the (i, j)-th entry corresponds to the signature of the j-th set with respect to the i-th hash function. Typically the signature matrix should be significantly smaller than M in size, as the size of universe U will be larger than n.

Example 7.2.2 *Shown below is the minhash signature created from Table 7.1 using* $n = 2$ *permutations (using functions* $h_1 = x + 1 \bmod 5$ *&* $h_2 = 3x + 1 \bmod 5$*).*

	S_1	S_2	S_3	S_4
h_1	1	3	0	1
h_2	0	2	0	0

Table 7.3: A signature matrix for 4 sets.

Computation of this signature matrix should be done in an efficient way, though following the explanation given above we'd be calculating n random permutations of a large characteristic matrix M; a very time-intensive task. Clearly we should attempt to avoid these permutations. To do this, we can use n random hash functions h to 'permute' the rows of M. We'll assume the mapping of $|U|$ elements onto themselves won't cause many collisions, however as there are certain to be some collisions $h(r)$ is not a 'perfect permutation'. So, given a row r of M, we can say that $h(r)$ denotes the new row number after a permutation. Shown below is an outline of the required steps to compute this signature matrix.

Step 1: Pick n random hash functions $h_1, \ldots, h_n$.

Step 2: Execute the following three steps for each row r of M.

1. Set Signature$(r, c) := \infty$, for $c = 1, \ldots, |S|$.
2. Apply each h-function to r for the new row numbers $h_1(r), \ldots, h_n(r)$ of r.
3. For $c = 1, \ldots, |S|$; if $M[r, c] = 1$ then Signature$(r, c) := \min(h_i(r), \text{Signature}(r, c))$.

Observe that for each non-zero entry of M, we perform $O(n)$ computations, and we do not need any additional memory except to store the description of n hash functions and the characteristic matrix M.

Although the signature matrix is fairly small compared to the characteristic matrix, its size could still be large. Let's look at the example from [85, Example 3.9], where there are signatures of length 250 for a set of one million documents. The total size of the signature matrix will be 1GB. Moreover, if documents are compared pairwise, the computation will be of the order of comparing a trillion documents!

The above example suggests that we should avoid the computation of pairwise similarity of documents as this requires $\binom{|S|}{2}$ comparison computations, which for a large number of documents any type of signatures will be prohibitively expensive. As an answer to this, we'll use Locality Sensitive Hashing (LSH) to quickly find any sets of documents which are likely to be similar, without using direct comparisons.

7.3 LSH for Minhash Signatures

The general idea, as expressed at the beginning of this chapter, is to hash documents in such a way that similar documents are more likely to be hashed into the same bucket as opposed to dissimilar documents. The documents will each be hashed multiple times, with the intent of altering the probability that similar items will become candidates while dissimilar items do not. The key point is that only the documents which fall into the same bucket are considered as potentially similar. If two documents do not map to the same bucket in any of the hashings then they are never considered as similar in this technique. Obviously, we want to minimize the number of dissimilar documents falling into the same bucket. At the same time, we hope that similar documents at least hash once into the same bucket. The good thing is that we have a methodology to achieve these objectives.

Let us assume that we have computed the minhash signature matrix for the set of documents of the previous section. We are going to partition the rows of this matrix into $b = n/r$ bands, where each band is made of r-consecutive rows. (See Figure 7.2. We'll assume that r divides n. For each band we define a hash function $h : \mathbb{R}^r \to \mathbb{Z}$, which takes a column vector of length r and maps it to an integer (i.e. a bucket). If we want we can even choose the same hash function for all the bands, but the buckets are kept distinct for each band. Now if two vectors of length r in any one of the bands hash to the same bucket, we declare that the corresponding documents are potentially similar. That's it — that's the technique. Let's do some analysis.

Lemma 7.3.1 *Let s be the Jaccard similarity of two documents. The probability that the minhash signature matrix agrees in all the rows of at least one band for these two documents is $1 - (1 - s^r)^b$. Note that this is also the probability that they will become a similar pair after the hashing and banding.*

Proof. Recall from Lemma 7.2.1, the probability that the minhash signature for these two documents are the same in any particular row of the signature matrix is s. Therefore, the probability

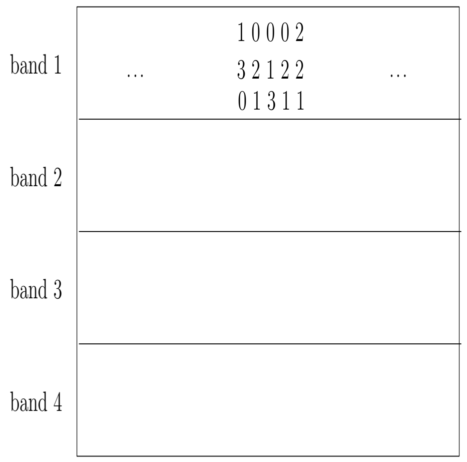


Figure 7.2: A signature matrix which has been separated into four bands of three rows each.

that the signatures agree in all the rows in one particular band is s^r. The probability that the signature will not agree in at least one of the rows in this band is $1 - s^r$.

Next let us calculate the probability that each band of the signature will not agree in at least one of its rows. This will be $(1 - s^r)^b$ as there are b bands. Therefore, the probability that the signatures will agree in at least one of the bands is $1 - (1 - s^r)^b$. ■

The function $1-(1-s^r)^b$ can be plotted with the x-axis representing values of s and the y-axis representing the output values of the function, with the values of b & r fixed. The curve created by this function has the form of an S-shaped curve. See Figure 7.3. In fact, the curve will have this shape regardless of the values of b and r. From inspecting this curve, we see that as the similarity s of documents approaches 1, the value of this function — the probability of the input elements being mapped to the same bucket — also increases towards 1. One important aspect of this curve however, is that the steepest slope occurs at the value of s which is around $t = (1/b)^{(1/r)}$. In other words, if the Jaccard similarity of the two documents is above the threshold t, then the probability that they will be found potentially similar using LSH is very high. What this technique has done is to give us some idea in terms of which documents are very likely to be similar. If required, we can now actually compare the documents (or their minhash signatures) to find out whether they are actually similar.

At an abstract level what we have done here is to use a family of functions (the minhash functions) and the banding technique to distinguish between pairs which are at a low distance (similar) to the pairs which are at a large distance (dissimilar). The steepness of the S curve suggests a threshold where this technique can be effective. In the next section, we will see that there are other families of functions that can be considered to separate the pairs which are at a low distance from the pairs which are at a higher distance.

7.4 Theory of Locality Sensitive Functions

In this section we will consider a family of functions $\mathcal{F}$. The families are typically hash functions, and we say that $f(x) = f(y)$ if the items x and y are hashed to the same bucket for a function

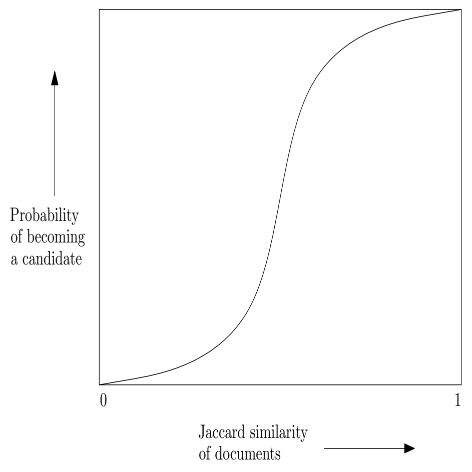


Figure 7.3: The S-curve.

$f \in \mathcal{F}$. For example, the minhash functions seen previously form a family of functions.

Recall that a distance measure satisfies certain natural properties: non-negativity, symmetry, and the triangle inequality. For example, the Euclidean distance between points in a space, edit distance between strings, Jaccard distance, and hamming distance all satisfy these properties.

Definition 7.4.1 *Let d be a distance measure, and let $d_1 < d_2$ be two distances in this measure. A family of functions $\mathcal{F}$ is said to be (d_1, d_2, p_1, p_2)-sensitive if for every $f \in \mathcal{F}$ the following two conditions hold;*

1. *If $d(x, y) \leq d_1$ then Probability$[f(x) = f(y)] \geq p_1$*
2. *If $d(x, y) \geq d_2$ then Probability$[f(x) = f(y)] \leq p_2$*

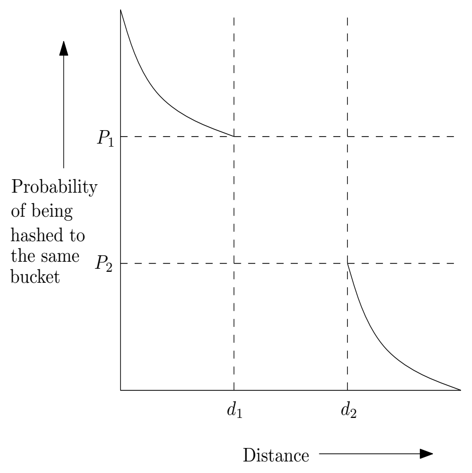


Figure 7.4: A smaller distance between items corresponds to a higher probability of similarity.

Basically, if the two items are close to each other with respect to the distance function, then the probability that they hash to the same bucket will be high. Conversely, if the two items are far from each other, then the probability that they hash to the same bucket will be low. See Figure 7.4 for an illustration.

For example, consider the Jaccard distance, which is defined as 1− Jaccard similarity of two sets. Let $0 \leq d_1 < d_2 \leq 1$. Note that the family of minhash-signatures is $(d_1, d_2, 1 - d_1, 1 - d_2)$-sensitive. Suppose that the Jaccard Similarity between two documents is at least s. Then their distance is at most $d_1 = 1 - s$. By Lemma 7.2.1 the probability that they will be hashed to the same value by minhash signatures is $s = 1 - d_1$. Suppose that the distance between two documents is $\geq d_2$. Hence their Jaccard similarity is at most $s = 1 - d_2$ and the probability that the minhash signatures map them to the same value is at most $s = 1 - d_2$.

Suppose we have a (d_1, d_2, p_1, p_2)-sensitive family $\mathcal{F}$. We can construct a new family $\mathcal{G}$ as follows. Each function $g \in \mathcal{G}$ is formed from a set of r independently chosen functions of $\mathcal{F}$, say $f_1, f_2, \ldots, f_r$ for some fixed value of r. Now, $g(x) = g(y)$ if and only if for all $i = 1, \ldots, r$, $f_i(x) = f_i(y)$. This is referred to as an *AND*-family.

Claim 7.4.2 *$\mathcal{G}$ is an $(d_1, d_2, {p_1}^r, {p_2}^r)$ family.*

Proof. Each function f_i is chosen independently, and this is the probability of all the r-events to occur simultaneously. ∎

We can also construct $\mathcal{G}$ as an *OR*-family. Each member g in $\mathcal{G}$ is constructed by taking b independently chosen members $f_1, f_2, \ldots, f_b$ from $\mathcal{F}$. We say that $g(x) = g(y)$ if and only if $f_i(x) = f_i(y)$ for at least one of the members in $\{f_1, f_2, \ldots, f_b\}$.

Claim 7.4.3 *$\mathcal{G}$ is an $(d_1, d_2, 1 - (1 - p_1)^b, 1 - (1 - p_2)^r)$ family.*

Proof. Each function f_i is chosen independently, and this is the probability of at least one of the b-events to occur. We can compute this by first finding the probability that none of the b events occur. ∎

Let us play with some values of b and r to see the effect of ANDs and ORs. We will also combine ORs and ANDs through function composition. We first construct an AND-family for a certain value of r, and then construct an OR-family on top of it for a certain value of b. Similarly we do this for an OR-family followed by an AND-family. Let us look at the amplification of the probabilities in Table 7.4 for different values of p.

p	p^5 ($\mathcal{F}_1$)	$1 - (1 - p)^5$ ($\mathcal{F}_2$)	$1 - (1 - p^5)^5$ ($\mathcal{F}_3$)	$(1 - (1 - p^5))^5$ ($\mathcal{F}_4$)
0.2	0.00032	0.67232	0.00159	0.13703
0.4	0.01024	0.92224	0.05016	0.66714
0.6	0.07776	0.98976	0.33285	0.94983
0.8	0.32768	0.99968	0.86263	0.99840
0.9	0.59049	0.99999	0.98848	0.99995

Table 7.4: Illustration of four families obtained for different values of p. $\mathcal{F}_1$ is the AND family for $r = 5$. $\mathcal{F}_2$ is OR family for $b = 5$. $\mathcal{F}_3$ is the AND-OR family for $r = 5$ and $b = 5$. $\mathcal{F}_4$ is the OR-AND family for $r = 5$ and $b = 5$.

Let's try to understand the columns of Table 7.4. Let $\mathcal{F}$ be a $(0.2, 0.6, 0.8, 0.4)$-sensitive minhash function family. This means if the distance between two documents x and y is ≤ 0.2, it is likely with probability ≥ 0.8 that they will hash to the same bucket. Similarly if the distance between x and y is ≥ 0.6, it is unlikely that they will hash to the same bucket (probability of hashing to the same bucket is ≤ 0.4). Let us focus our attention on the two rows corresponding to $p_2 = 0.4$ and $p_1 = 0.8$. For the AND-family we see that p^5 is substantially lower than p, but still $p_2^5 \to 0$ and p_1^5 is away from 0. For the OR-family $1-(1-p)^5 \geq p$, but still the value corresponding to p_1 tends towards 1 and the value corresponding to p_2 is away from 1. More interesting are the last two columns. Notice that the AND-OR family ($\mathcal{F}_3$) corresponds to the LSH for minhash signature family of Section 7.3. In this case the value corresponding to p_1 tends towards 1 and the value corresponding to p_2 is closer to 0. Notice that the value of this function with respect to the values of p forms an S-curve. Its fixed-point $p = 1-(1-p^5)^5$ is around $p \approx 0.7548$ [1]. This implies that for values of p significantly less than 0.75, AND-OR family amplifies it towards 0. Similarly, values of p larger than 0.75 are amplified to 1. Notice that this technique amplifies the probabilities in the right direction, provided we can apply the function several times (e.g. 25 times in our example), and are away from the fixed-point.

7.5 LSH Families

In the previous section we saw LSH families for the Jaccard distance measure. In this section we will construct LSH-families for various other distance measures, including Hamming and Euclidean distances. Note that not every distance measure may have a corresponding LSH-family.

7.5.1 Hamming Distance

Consider two d-dimensional vectors x and y. The Hamming distance $h(x, y)$ is defined as the number of coordinates in which x and y differ. Without loss of generality, we can assume that they are Boolean vectors — that is, each of their coordinates are either 1 or 0.

Example 7.5.1 *Let* $x = 110011$ *and* $y = 100111$. *Then* $h(x, y) = 2$, *as they differ in exactly two coordinates.*

Applying the requirements for a distance measure, we see that distances are non-negative, symmetric, and that the distance between two identical vectors is 0. They are also transitive, as the Hamming distance between any three vectors x, y, and z, satisfy the triangle inequality; $h(x, y) + h(y, z) \geq h(x, z)$. Therefore, we can use Hamming distance as a metric over the d-dimensional vectors.

Next, we construct a locality-sensitive family for the d-dimensional Boolean vectors. Let $f_i(x)$ denote the i-th coordinate (bit) of x. Then the probability that $f_i(x) = f_i(y)$ for a randomly chosen i will equal the number of coordinate agreements out of the total number of coordinates. Since the vectors x and y disagree in $h(x, y)$ positions out of d positions, then they agree in $d - h(x, y)$ positions. Hence $Pr[f_i(x) = f_i(y)] = 1 - h(x, y)/d$.

Claim 7.5.2 *For any* $d_1 < d_2$, $\mathcal{F} = \{f_1, f_2, \ldots, f_d\}$ *is a* $(d_1, d_2, 1 - d_1/d, 1 - d_2/d)$*-sensitive family of hash functions.*

[1] Thanks to WolframAlpha

Proof. Recall Definition 7.4.1. Let $p_1 = 1 - d_1/d$, and $p_2 = 1 - d_2/d$. A family of functions $\mathcal{F}$ is said to be (d_1, d_2, p_1, p_2)-sensitive if for every $f_i \in \mathcal{F}$ the following two conditions hold;

1. If $h(x, y) \le d_1$ then Probability$[f_i(x) = f_i(y)] \ge p_1$
2. If $h(x, y) \ge d_2$ then Probability$[f_i(x) = f_i(y)] \le p_2$

■

Note that the family $\mathcal{F}$ consists of only d-functions. This is in contrast to the minhash function family, where the size is not bounded.

7.5.2 Cosine Distance

As an alternative to Hamming distance, we can use Cosine distance to construct a locality-sensitive family of functions for d-dimensional vectors in a space. Consider two vectors in this space, x and y, along with a hyperplane through the origin with a normal vector v. These two vectors x and y have a cosine distance which is equal to the angle θ between them, and taken alone, this angle can be measured in the plane defined by the vectors. The hyperplane is randomly chosen via its normal vector v, and will then determine the sign of the dot products $v.x$ and $v.y$.

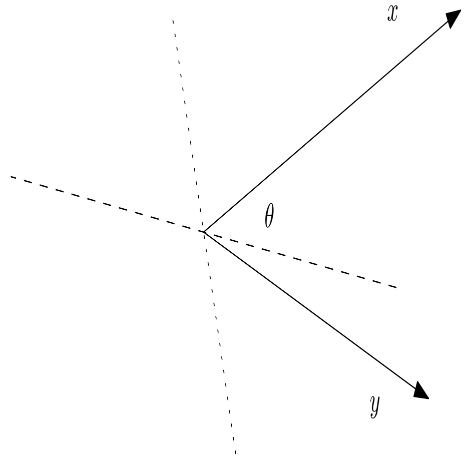


Figure 7.5: Two vectors x and y shown in a plane; two possible hyperplanes are also shown.

In Figure 7.5, the two vectors x and y are shown along with two possible hyperplanes. If the hyperplane is similar to the dashed line, then the dot products $v.x$ and $v.y$ will have different signs. However, if the hyperplane is similar to the dotted line, then $v.x$ and $v.y$ will have the same sign.

The probability that v will produce a hyperplane similar to the dotted line is $\theta/180$, as v would have to lie between x and y on the right (or on the left; $2\theta/360$). So, we create each hash function $f \in \mathcal{F}$ using a randomly chosen vector v, and can then say that $f(x) = f(y)$ if the dot products $v.x$ and $v.y$ have the same sign. Our resultant family $\mathcal{F}$ is a $(d_1, d_2, (180 - d_1)/180, (180 - d_2)/180)$-sensitive family of functions. Similar to other families, we can amplify probabilities by taking any combination of ANDs and ORs as before.

7.5.3 Euclidean Distance

To start with, let's consider points in a 2-dimensional space. Each hash function h will be represented by a line with random orientation in this space. We subdivide the line into intervals of an equal size a, with points landing in the same bucket if their orthogonal projection lies on the same interval. Using this method, if two points are close to each other then there are high chances that they will map to the same interval. Conversely, if the points are far away, it is unlikely they will fall into the same bucket (unless they are almost vertically aligned!).

Without loss of generality, we'll assume that the line is horizontal. Let p_i and p_j be two points in the plane. Let θ be the angle of the line passing through p_i and p_j with respect to the horizontal line. Observe that if the distance between the points is at most $a/2$ then the probability that the points map to the same interval is at least $1/2$. Moreover, if the distance between them is more than $2a$, then they will fall in the same interval with a probability of at most $1/3$. This is due to the fact that they have to be almost vertical; $|p_ip_j|\cos\theta \leq a$. For this to happen $\cos\theta \leq 1/2$, or the angle of the line p_ip_j forms with the horizontal needs to be between 60° and 90°. Observe that there is only 1/3rd chance that the angle between the horizontal and p_ip_j is in that range. See Figure 7.6 for an illustration.

Hence, the family with respect to the projection on a random line with intervals of size a is a $(a/2, 2a, 1/2, 1/3)$-sensitive family of hash functions. Again, we can easily amplify these probabilities by taking any combination of ANDs and ORs as before.

7.5.4 Fingerprint Matching

Fingerprint matching typically requires comparison of several features of the fingerprint pattern. These include ridge lines which form arches, loops, or circular patterns, along with minutia points & patterns which form ridges, and bifurcations. We can think of a fingerprint as a set of grid points, where matching any two fingerprints amounts to matching elements in corresponding grid cells.

Fingerprint matching falls into an application of Locality-Sensitive Hashing, and as a result of the data representation we are able to use its methods; mainly the family of minhash functions for set comparison. However in this section we will approach the problem in another manner, as the sets produced from fingerprints form a small grid of around 1000 points, and don't need to have more succinct signatures created.

Consider the probabilities of minutia appearances and matches between fingerprints. For this, let's say that the probability of a minutia existing in a random grid cell of any given fingerprint is 0.1. Also, assuming that we have two fingerprints of the same finger, let the probability of a minutia appearing in the same grid cell of both fingerprints given that one of them does have a minutia there be 0.9.

We'll define a locality-sensitive family of functions $\mathcal{F}$ as follows. Each function $f \in \mathcal{F}$ sends two fingerprints to the same bucket if they have minutia in each of three specific grid cells. Let us estimate what the chance is of ending up with two matching fingerprints. First, the probability that two arbitrary fingerprints will map to the same bucket with respect to function f is $0.1^6 = 0.000001$, which is a one in a million chance [2]. Assuming that we have two fingerprints which are from the same finger though, we can see that they are mapped to the same bucket by f with a probability of $0.1^3 \times 0.9^3 = 0.00729$. This is only about a 1 in 138 chance, though it can be amplified by using several hash functions (OR-family) and still not produce many false positives.

[2] Wonder why you are always the unlucky one :(

As an example, suppose we use 1000-way OR-functions. Then two fingerprints from different fingers will map to the same bucket with a probability of $1 - (1 - 0.000001)^{1000}) \approx 0.0001$. Along with this, two fingerprints from the same finger will map to the same bucket with a probability of $1-(1-0.00729)^{1000} \approx 0.9993$. This is significant, and hence finding the match in this case is highly likely. Also note that using this scheme, the chances of false negatives are less than 1 in 1000, and the chances of false positives are negligible.

For another example, let's now use 2000 functions which are partitioned into two groups of 1000 functions each. Assume also that we have constructed buckets for each of the two composite functions. Given a query fingerprint, we'll find the fingerprints which are potential matches using the above scheme in each group independently. Then we'll select only those fingerprints which are potential matches in both the groups (in the intersection). This produces a set of fingerprints which we'll actually compare to the query fingerprint. Note that in this scheme, actual comparisons of fingerprints happen with only a few fingerprints (those in the intersection). Here, the probability that we will detect a matching fingerprint is $(0.9993)^2 \approx 0.9986$. The number of false negatives are now about 1.4 in a 1000. The probability of false positives will be 0.0001^2, which is insignificant. Hence, we have to examine only 0.000001% of the database compared to 0.01% in the previous scheme.

7.5.5 Image Similarity

This section is contributed by Andrew Wylie. Image similarity can also be found using locality-sensitive hashing, as exemplified in earlier research on LSH. Often, images are processed using global descriptors such as color histograms to produce feature vectors which are then compared using different distance measures. One of the obvious applications of image similarity is in web search, as seen in use by TinEye, Google, and Bing. However most search engines do not use image similarity in search; instead they rely on information gathered from surrounding text on the web page along with any image meta-data. In this section, we will show how locality-sensitive hashing is used by Google in their VisualRank algorithm for visual image search, along with several locality-sensitive families which have been proposed for image similarity measures.

VisualRank

In the VisualRank algorithm, computer vision techniques are used along with locality-sensitive hashing. Consider a regular image search by a text query. An existing search technique may be relied on to retrieve the initial result candidates, which along with other images in the index are clustered according to their similarity (which may be precomputed). Centrality is then measured on the clustering, which will return the most canonical image(s) with respect to the query. The idea here is that agreement between users of the web about the image and its related concepts will result in those images being deemed more similar. VisualRank is defined iteratively by $VR = S^* \times VR$, where S^* is the image similarity matrix. As matrices are used, eigenvector centrality will be the measure applied, with repeated multiplication of VR and S^* producing the eigenvector we're looking for. Clearly, the image similarity measure is crucial to the performance of VisualRank since it determines the underlying graph structure. See Figure 7.7 for an illustration.

The main VisualRank system begins with local feature vectors being extracted from images using scale-invariant feature transform (SIFT). Local feature descriptors are used instead of the previously mentioned color histograms as they allow similarity to be considered between images with potential rotation, scale, and perspective transformations. Locality-sensitive hashing is then applied to these feature vectors using the p-stable distribution scheme. As seen in previous sections, amplification using AND/OR constructions are applied. As part of the applied scheme, a Gaussian distribution is used under the l_2 norm.

p-Stable Distributions

This locality-sensitive hashing scheme uses stable distributions to apply a sketching technique to the high dimensional feature vectors. However, instead of using the sketch to estimate the vector norm, it is used to assign hash values to each vector. These values map each vector to a point on the real line, which is the split into equal width segments to represent buckets. If two vectors v and u are close, they will have a small difference between l_p norms $||v-u||_p$, and they should collide with a high probability.

Definition 7.5.3 *A distribution $\mathcal{D}$ over $\mathcal{R}$ is called* p-stable*, if there exists $p \geq 0$ such that for any n real numbers $v_1, \dots, v_n$ and independent identically distributed variables $X_1, \dots, X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where X is a random variable with distribution $\mathcal{D}$.*

Example 7.5.4 *Cauchy Distributions are 1-stable, and Gaussian Distributions are 2-stable.*

Note how the sum $\sum_i v_i X_i$ appears if we consider that v_i's are taken from a vector v. This operation then corresponds exactly to the dot product between vectors v and a, where a has entries taken from the stable distribution. From our definition then, we see that this random variable has the same distribution as the l_p norm when the latter is multiplied by a random variable from the same distribution. This is exactly the method of the sketching technique, where multiple $v.a$'s are computed to estimate $||v||_p$.

However as mentioned beforehand, in this case we'll use the $v.a$'s to assign a hash value to each v instead of estimating the l_p norm. Also, as the dot product $v.a$ projects each vector onto the real line, we'll separate the line into equal width segments of size r. We define each hash function of our family by

$$h_{a,b}(v) = \lfloor \frac{v.a + b}{r} \rfloor,$$

where a is a d-dimensional random vector with entries chosen independently from our stable distribution, and b is a real number chosen uniformly from the range $[0, r]$. Let $f_p(t)$ denote the probability density function of absolute value of our stable distribution, and let $c = ||v-u||_p$ for two vectors v, u. Since we have a random vector a from our stable distribution, $v.a - u.a$ is distributed as cX where X is a random variable from our stable distribution. Therefore our probability function is;

$$p(c) = Pr_{a,b}[h_{a,b}(v) = h_{a,b}(u)] = \int_0^r \frac{1}{c} f_p(\frac{t}{c})(1 - \frac{t}{r})dt.$$

For a fixed r the probability of collision decreases monotonically with $c = ||v-u||_p$. Thus, our family of hash functions are (r_1, r_2, p_1, p_2)-sensitive for $p_1 = p(1)$, $p_2 = p(c)$, and $r_2/r_1 = c$. Note that for each c we attempt to choose a finite r to make p as small as possible.

χ^2 Distance

In the context of image retrieval & similarity search, the Euclidean metric (l_2) does not always provide as accurate or relevant results when compared to χ^2 distance. This is especially so when using global descriptors such as color histograms. In this section, we will show an adaptation of the p-stable distribution scheme for χ^2 distance.

Definition 7.5.5 *Given any two vectors v and u with d positive components, the χ^2 distance between them is*

$$\chi^2(v,u) = \sqrt{\sum_{i=1}^{d} \frac{(v_i - u_i)^2}{v_i + u_i}}.$$

To begin with, all points are mapped into a space of smaller dimension where they will be clustered in a way which preserves local sensitivity. This is done by projecting all points onto a line l_a using a random vector a whose entries are chosen independently from a Gaussian distribution. The line is then uniformly partitioned into intervals of size W with respect to χ^2 distance. Our hash functions are then defined in a similar manner to the previous section — as the dot product of the feature vector v with a random vector a with entries independently chosen from a Gaussian distribution;

$$h_{a,b}(v) = \lfloor yw(v.a) + b \rfloor.$$

In this case, b is a real number taken randomly from the uniform distribution $\mathcal{U}([0,1])$. The function yw is defined as

$$yw(x) = \frac{\sqrt{\frac{8x}{W^2} + 1} - 1}{2}$$

as a result of the non-uniform mapping from l_2 to χ^2 distance; points mapping to the same bucket in the χ^2 scheme may be mapped to different buckets in the l_2 scheme.

Following from the reasoning in p-stable distributions, we know that $v.a - u.a$ is distributed as cX, where $c = ||v - u||_p$, and X is a random variable drawn from the distribution. Also similar to the previous section, $f_p(t)$ denotes the probability density function of the absolute value of the 2-stable Gaussian distribution. Therefore our probability function is;

$$p(c) = Pr_{a,b}[h_{a,b}(v) = h_{a,b}(u)] = \int_0^{(n+1)W^2} \frac{1}{c} f_p(\frac{t}{c}) \left(1 - \frac{t}{(n+1)W^2}\right) dt.$$

We then define $c' = X^2(v,u)$. Since v, u have positive entries, the two distances will vary similarly when taking the dot product. Therefore, p decreases monotonically with respect to c, and p also decreases monotonically with respect to c'. If we set $p_1 = p(r_1)$ and $p_2 = p(r_2)$ (with $r_1 < r_2$), then this family of functions is (r_1, r_2, p_1, p_2)-sensitive.

7.6 Bibliographic Notes

This section is contributed by Andrew Wylie. Concepts relating to locality-sensitive hashing first started to form around 1994, when a method for finding similar files in a large file system was proposed [70]. This research introduced the idea behind shingling through ‘fingerprinting’, where several anchor strings were chosen to begin a substring used in the similarity search. Along with this, checksums were also used to represent these fingerprints. Shingling itself was then formalized a few years later, along with k-shingles of a document (using words as tokens), and resemblance (similarity) of two documents was defined [19]. The authors also introduced a method for computing a sketch (signature) of a document which could be used in comparisons; hence, preserving similarity.

After this research came a couple of papers which proposed similarity search through the use of hashing [37, 49]. Approximate nearest-neighbor search was first reduced to the problem of point location in equal balls, where locality-sensitive hashing was introduced for sublinear time queries. This algorithm was then improved in [49] with respect to query time. Along with this, the distance measures for hamming distance and the resemblance measure given in [19] were used as schemes. It's worth noting that the approximate search is deemed accurate enough for practical purposes, where the actual closest neighbor can still be found by checking all approximate near-neighbors [5].

Concurrent with this research, shingling was formally introduced along with min-wise independent families of permutations [18]. These concepts were developed as a result of the authors' work on the AltaVista web index algorithm which was used for detecting similar documents. Minwise-independent families also underpin the theory behind minhashing, as the resemblance of document-sets is shown to be equal to the probability that the min-wise permutation of two sets using random permutation functions are equal.

Next in the development of locality-sensitive hashing was a scheme based on p-stable distributions ($p \in (0, 2]$) under the l_p norm [24]. This scheme both improved on the l_2 norm distance running time found in [49], and works in Euclidean space without embeddings. These findings are then improved upon by [5], resulting in a near-optimal algorithm.

Locality-sensitive hashing is used in for video identification and retrieval. In this case, feature vectors are usually constructed from video frames using certain color histograms. In [55], the authors address two weaknesses of using LSH for this purpose. Responding to a non-uniform distribution of points in the space, they focus on using a hierarchical hash table to produce a more uniform distribution. In addition to this, they also attempt to partition dimensions more carefully in order to produce a more even hashing. A new scheme for video retrieval is then proposed in [48], where a color histogram is used which better handles the adverse effects of brightness & color variations. This is used in conjunction with an additional uniform distance shrinking projection which is applied to the produced feature vectors.

Locality-sensitive hashing is also used in image search. Similar to applications in video (for a single frame), images are often processed using color histograms to produce feature vectors which can be compared. This method was used in [37], with the histograms compared using the l_1 norm. Following this, techniques using chi^2 distance [40], p-stable distributions [53], and kernelized locality-sensitive hashing [66] have been proposed. Kernelized locality-sensitive hashing has also been used as a basis for search on text images in [77].

Along with these applications, LSH has recently been proposed for use in a wide range of other areas. One of these is the creation of hash values in P2P networks [25]. This would allow for a conceptual search, as data with similar ontologies would be located near each other. Here, data is defined by its extracted concept vectors, which are then hashed into buckets based on the cosine distance measure. Another, in [52], kernelized LSH is applied to an utterance model in order to identify speakers. In this case the hamming distance metric is used. Other areas include use for species diversity estimation by allowing ease of grouping similar DNA sequences [86], and incremental clustering for affinity groups in Hadoop in order to store related data closer together [54].

Recently, [94] also proposed a new scheme based on entropy for LSH. The argument is that an improvement can be made for [24] such that the distribution of mapped values will be approximately uniform. Note how this approach attempts the same result as [55] using different methods.

7.7 Exercises

7.1 *Given two documents $D_1 = \{$ "His brow trembled"$\}$ and $D_2 = \{$ "The brown emblem"$\}$, compute all k-shingles for each document with $k = 4$.*

7.2 *Compute the Jaccard Similarity of the two sets of k-shingles for D_1 and D_2.*

7.3 *Compute the signature matrix (minhash signature) of a given permutation of characteristic matrix in Table 7.5, using $n = 3$ different hash functions.*

Element	S_1	S_2	S_3	S_4
a	1	0	1	1
b	1	0	1	0
c	0	1	0	1
d	1	0	1	0
e	0	0	1	0

Table 7.5: A characteristic matrix.

7.4 *Explain the reasoning behind the proof of Lemma 7.2.1.*

7.5 *Explain the overall procedure used to calculate document similarity, where locality-sensitive hashing is used in the process, and why locality-sensitive hashing would be favorable to use.*

7.6 *Show that the minhash function family is $(d_1, d_2, 1 - d_1, 1 - d_2)$-sensitive.*

7.7 *Calculate hamming distance for the vectors $v = [5, 1, 3, 2, 4]$ and $u = [1, 1, 2, 2, 4]$.*

7.8 *Discuss the similarity between Jaccard distance and Hamming distance.*

7.9 *Show that the AND amplification construction is (d_1, d_2, p_1^r, p_2^r)-sensitive.*

7.10 *When applying amplification constructions to a locality-sensitive family of functions, which order of composition is 'better', and why? Explain when you would want to use different orders of construction.*

7.11 *Show that the Jaccard Distance which is defined as $1-$ the Jaccard Similarity between the two sets is a metric.*

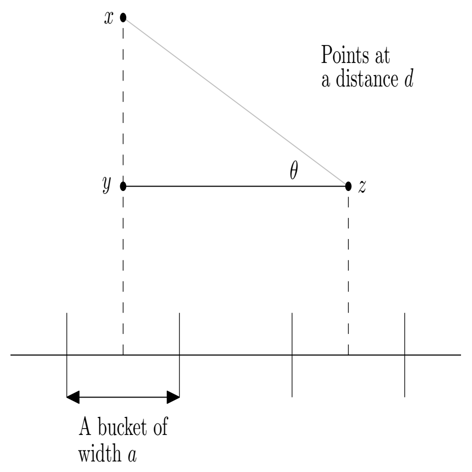


Figure 7.6: With a distance $d > 2a$, points x and z are in separate intervals unless θ is large enough.

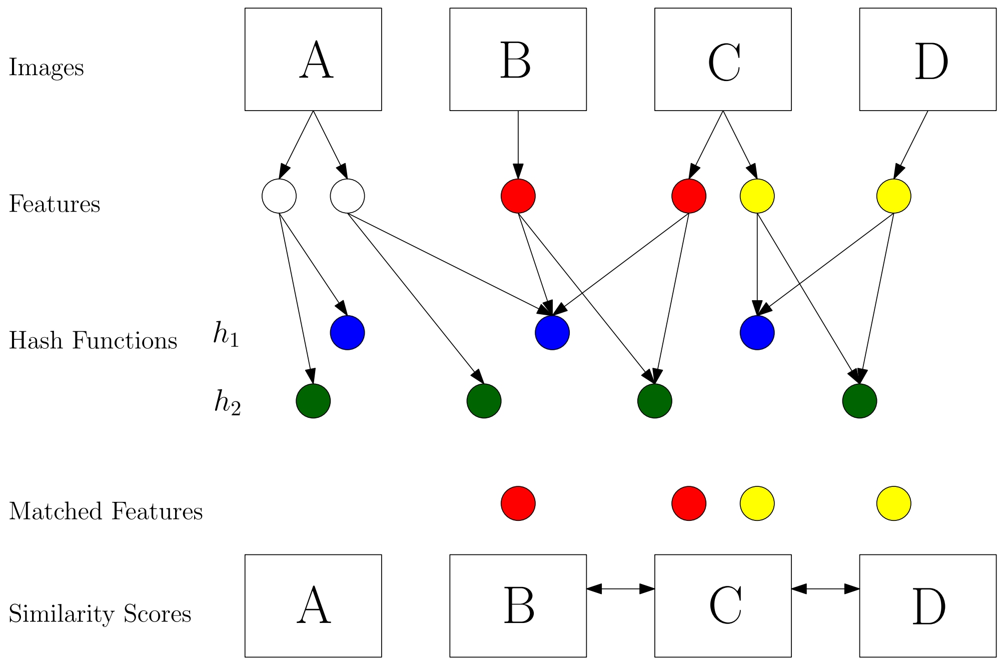


Figure 7.7: Images are considered similar if their local features are hashed into the same bin in multiple hash tables. In this example, the blue and green circles are buckets where features are hashed; B is similar to C, and C is similar to D.

Chapter 8

Dimensionality Reduction

The material of this chapter is from Dasgupta and Gupta [23], Dubhashi & Panconesi [28], Matousek [72, 71], Bourgain [15], and the lecture notes of Harold Racke on a course on metric embeddings offered at IIIT Chicago a while ago. This chapter is under construction. I plan to cover the following.

8.1 Preliminaries: Metric spaces and embeddings

We will define metric spaces, introduce the concept of metric embedding, introduce isometric and nonisometric embeddings via the concept of distortion - contraction and expansion. We show that any n-point metric space (X, d) can be embedded isometrically into n-dimensional L_∞. We also show that there are metric spaces which cannot be embedded isometrically always in the plane endowed with Euclidean distance.

8.2 Embeddings into L_∞-normed spaces

This section covers a theorem due to Matousek, see [72, 71], using the proof technique of Bourgain [15]. We show that $< X, d > \overset{O(\log^2 n)}{\longrightarrow} L_\infty^{O(\log^2 n)}$.

8.3 Embeddings into L_p-normed spaces

This section covers results of Bourgain [15]. We show that $< X, d > \overset{O(\log n)}{\longrightarrow} L_p^{O(\log^2 n)}$.

8.4 Johnson and Lindenstrauss Theorem

In this section we prove a celebrated theorem of Johnson and Lindenstrauss from 1984.

Theorem 8.4.1 *Let V be a set of n points in d-dimensions. A mapping $f : \mathbb{R}^d \to \mathbb{R}^k$ can be computed, in randomized polynomial time, so that for all pairs of points $u, v \in V$,*

$$(1-\epsilon)||u-v||^2 \leq ||f(u) - f(v)||^2 \leq (1+\epsilon)||u-v||^2,$$

where $0 < \epsilon < 1$ and $n, d,$ and $k \geq 4(\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3})^{-1} \ln n$ are positive integers.

Intuitively, the theorem says that points in d-dimensional space can be mapped to a k dimensional space, where k is significantly less than d, and the interpoint Euclidean distance between a pair of points in V is preserved up to a factor of $(1 \pm \epsilon)$. The proof is based on showing that the squared length of a random vector is concentrated around its mean when the vector is projected onto a random k-dimensional subspace, i.e., with probability $O(1/n^2)$, the length is within $(1 \pm \epsilon)$ of the mean. In place of estimating the length of a unit vector after projecting it onto a random k-dimensional space, we will estimate the length of a random unit vector when we project it to a fixed k-dimensional space (e.g. in the space spanned by the first k-coordinates).

Recall Standardized Normal Distributions $\mathcal{N}(0,1)$.

Lemma 8.4.2 *Let $X_1, \ldots, X_d$ be d-independent standardized normal $\mathcal{N}(0,1)$ random variables. Let $X = (X_1, X_2, \ldots, X_d)$ and define $Y = \frac{1}{||X||}X$. Then, Y is a random point drawn uniformly from the surface of the d-dimensional unit sphere.*

Proof. Since the distribution for each X_i is given by $f(x_i) = \frac{1}{\sqrt{2\pi}}e^{-x_i^2/2}$, the distribution of X, due to Independence, is $\Pi_{i=1}^d[\frac{1}{\sqrt{2\pi}}e^{-x_i^2/2}] = \frac{1}{2\pi^{d/2}}e^{-\frac{1}{2}\sum_{i=1}^d x_i^2}$. Observe that the density function of X only depends upon the distance of $(X_1, \ldots, X_d)$ from the origin, and thus it is spherically symmetric. Since Y is obtained by dividing this by $||X||$, Y is a random point on the unit sphere. ∎

From now on let $Y = (Y_1, \ldots, Y_d)$ be a random unit vector as stated in Lemma 8.4.2. Define a vector $Z \in \mathbb{R}^K$, which is obtained by projecting Y onto its first k-coordinates (i.e. ignoring all other coordinates, except the first k). Let L represent the squared length of Z ($L = ||Z||^2$).

Lemma 8.4.3 *$E[L] = k/d$.*

Proof.

$$E[L] = E[||Z^2||] = \frac{1}{||X||^2}E[X_1^2 + \cdots + X_k^2] = \frac{1}{||X||^2}\sum_{1=1}^{k} E[X_i^2] = k\frac{E[X_1^2]}{||X||^2}.$$

Observe that

$$||Y||^2 = 1 = \frac{1}{||X||^2}\sum_{i=1}^{d} X_i^2.$$

This implies that

$$d\frac{E[X_1^2]}{||X||^2} = 1.$$

Hence

$$E[L] = k\frac{E[X_1^2]}{||X||^2} = \frac{k}{d}.$$

∎

Next we state the main lemma, first without proving it, and show how Theorem 8.4.1 follows from it.

Lemma 8.4.4 *Let $k < d$. Then*

1. *If* $\beta < 1$*, then* $P(L \leq \beta\frac{k}{d}) \leq e^{\frac{k}{2}(1-\beta+\ln\beta)}$.

2. *If* $\beta > 1$*, then* $P(L \geq \beta\frac{k}{d}) \leq e^{\frac{k}{2}(1-\beta+\ln\beta)}$.

Proof. (Proof of Theorem 8.4.1) The case of $k \geq d$ is trivial. Therefore, we assume that $k < d$. Let S be a random k-dimensional subspace of $\mathbb{R}^d$ (for example we choose randomly k coordinates). Project each point $v \in V$ and let its projection in $\mathbb{R}^k$ be v'. Let $Y := \frac{v-w}{||v-w||}$ be a unit vector, and let $Z := \frac{v'-w'}{||v-w||}$ be its random projection. Let $L = ||Z||^2$. Note that Z behaves as required in Lemma 8.4.3. Hence $E[L] = k/d$. For $0 < \epsilon < 1$, set $\beta = 1 - \epsilon$. Note that

$$
\begin{aligned}
P(||v'-w'||^2 \leq (1-\epsilon)||v-w||^2 k/d) &= P(L \leq (1-\epsilon)k/d) && (8.1)\\
&\leq e^{\frac{k}{2}(1-(1-\epsilon)+\ln(1-\epsilon))} && (8.2)\\
&\leq e^{\frac{k}{2}(\epsilon-(\epsilon+\frac{\epsilon^2}{2}))} && (8.3)\\
&= e^{-\frac{k\epsilon^2}{4}} && (8.4)\\
&\leq e^{-2\ln n} && (8.5)\\
&= \frac{1}{n^2} && (8.6)
\end{aligned}
$$

Note that for Equation 8.3 we used that $\ln(1-\epsilon) \leq -\epsilon - \epsilon^2/2$, for $0 < \epsilon < 1$. For Equation 8.5 (and 8.11), we use the fact that $k \geq 4(\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3})^{-1}\ln n$.

$$
\begin{aligned}
P(||v'-w'||^2 \geq (1+\epsilon)||v-w||^2 k/d) &= P(L \geq (1+\epsilon)k/d) && (8.7)\\
&\leq e^{\frac{k}{2}(1-(1+\epsilon)+\ln(1+\epsilon))} && (8.8)\\
&\leq e^{\frac{k}{2}(-\epsilon+(\epsilon-\frac{\epsilon^2}{2}+\frac{\epsilon^3}{3}))} && (8.9)\\
&= e^{-\frac{k(\epsilon^2/2-\epsilon^3/3)}{2}} && (8.10)\\
&\leq e^{-2\ln n} && (8.11)\\
&= \frac{1}{n^2} && (8.12)
\end{aligned}
$$

Set the map of each $v \in V$ as $f(v) = \sqrt{\frac{d}{k}}v'$. Consider a pair of points $u, v \in V$, and by above calculations, the probability that $(1-\epsilon)||u-v||^2 \leq ||f(u)-f(v)||^2 \leq (1+\epsilon)||u-v||^2$ does not hold is at most $2/n^2$. Thus, for any pair of points, the probability this condition is not true is at most $\binom{n}{2}\frac{2}{n} = 1 - \frac{1}{n}$. Therefore, the mapping f has the desired distortion property. We can boost the success probability further by repeating the experiment several times. Observe that as the mapping is fairly straighforward and can be easily seen to be obtained in polynomial time. ∎

Before we prove Lemma 8.4.4, we need the following.

Lemma 8.4.5 *Let* X *be* $\mathcal{N}(0,1)$*. Then* $E[e^{sX^2}] = \frac{1}{\sqrt{1-2s}}$*, for* $s < 1/2$.

Proof. Note that X is given by the probability distribution $f(x) = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}$. Moreover, expected value of a function, say $H(X)$, of a random variable X is given by $E(H(X)) = \int_{-\infty}^{+\infty} H(x)f(x)dx$.

Thus,

$$\begin{aligned}
E[e^{sX^2}] &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{sx^2} e^{-\frac{x^2}{2}} dx && (8.13)\\
&= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-(1-2s)x^2/2} dx && (8.14)\\
&\text{set } y = x\sqrt{1-2s} && (8.15)\\
&= \frac{1}{\sqrt{1-2s}} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-y^2/2} dy && (8.16)\\
&= \frac{1}{\sqrt{1-2s}} \text{ (since } \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} e^{-y^2/2} dy = 1) && (8.17)
\end{aligned}$$

■

Proof.(Proof of Lemma 8.4.4) First, we want to prove that for $\beta < 1$, $P(L \le \beta \frac{k}{d}) \le e^{\frac{k}{2}(1-\beta+\ln\beta)}$.

Equivalently, we want to show that

$$P(d(X_1^2 + \cdots + X_k^2) \le k\beta(X_1^2 + \cdots + X_d^2)) \le e^{\frac{k}{2}(1-\beta+\ln\beta)}.$$

Note that $P(d(X_1^2 + \cdots + X_k^2) \le k\beta(X_1^2 + \cdots + X_d^2))$

$$\begin{aligned}
&= P(k\beta(X_1^2 + \cdots + X_d^2) - d(X_1^2 + \cdots + X_k^2) \ge 0) && (8.18)\\
&= P(e^{t(k\beta(X_1^2+\cdots+X_d^2)-d(X_1^2+\cdots+X_k^2)} \ge e^{t.0} = 1) \text{ (for } t > 0) && (8.19)\\
&\le E[e^{t(k\beta(X_1^2+\cdots+X_d^2)-d(X_1^2+\cdots+X_k^2)}] \text{ (Using Markov's inequality)} && (8.20)
\end{aligned}$$

Recall that if A and B are independent r.v. than $E[AB] = E[A]E[B]$. Moreover, $E[A^2] = E[A]^2$. Also, each of these X_i's have identical distribution, and they are independent r.v.'s. After rearranging and replacing all X_i's by X_1, we obtain

$$\begin{aligned}
&= E[e^{tk\beta X_1^2}]^{d-k} E[e^{t(k\beta-d)X_1^2}]^k && (8.21)\\
&= \left(\frac{1}{\sqrt{1-2tk\beta}}\right)^{d-k} \left(\frac{1}{\sqrt{1-2t(k\beta-d)}}\right)^k \text{ (Using Lemma 8.4.5)} && (8.22)
\end{aligned}$$

The above inequality holds, provided $0 < t < 1/2k\beta$. Now we want to minimize the quantity in Inequality 8.22. This is equivalent to maximizing

$$q(t) = (1-2tk\beta)^{d-k}(1-2t(k\beta-d))^k.$$

We obtain, using derivatives of a product etc., that maximum occurs when $t = \frac{1-\beta}{2\beta(d-k\beta)}$. The value of $q(t = \frac{1-\beta}{2\beta(d-k\beta)}) = \left(\frac{d-k}{d-k\beta}\right)^{d-k} \left(\frac{1}{\beta}\right)^k$. Substituting this back in Inequality 8.22, as $1/\sqrt{q(t)}$, we obtain that $P(L \le \beta\frac{k}{d}) \le \beta^{\frac{k}{2}} \left(1 + \frac{(1-\beta)k}{(d-k)}\right)^{(d-k)/2}$. With simple manipulation and using the fact that $\ln(1+x) \leq x$, we obtain that $\beta^{\frac{k}{2}}\left(1+\frac{(1-\beta)k}{(d-k)}\right)^{(d-k)/2} \leq e^{\frac{k}{2}(1-\beta+\ln\beta)}$, as desired. One can show this by taking natural log on both the sides.

Next, using the identical method as above, we show that for $\beta > 1$, $P(L \geq \beta\frac{k}{d}) \leq e^{\frac{k}{2}(1-\beta+\ln\beta)}$. This amounts to showing that

$$P(d(X_1^2+\cdots+X_k^2) \geq k\beta(X_1^2+\cdots+X_d^2)) \leq \left(\frac{1}{\sqrt{1+2tk\beta}}\right)^{d-k}\left(\frac{1}{\sqrt{1+2t(k\beta-d)}}\right)^{k} \tag{8.23}$$

$$= \frac{1}{\sqrt{q(-t)}} \tag{8.24}$$

Now the rest of the proof requires finding the value that maximizes $q(-t)$. This is maximized at $t = -\frac{1-\beta}{2\beta(d-k\beta)}$. Rest of the proof is same. ∎

8.5 Applications of Metric Embeddings

8.6 Exercises

8.1 *Show that for any embedding f of a metric space (X, d) to another metric space (X', d'), the distortion of f is at least* 1.

Chapter 9

Second moment method with applications

Main part of this chapter is a copy of the article published in ALGOSENSORS 2015 titled Plane and Planarity Thresholds for Random Geometric Graphs by Ahmad Biniaz, Evangelos Kranakis, Anil Maheshwari and Michiel Smid [8]. The material for Cliques in $G(n, 1/2)$ is adapted from Nikal Bansal's class notes as well as from [76]. Over time this will be refined to give a broader overview of the second moment method and some more applications and exercises will be added.

9.1 Preliminaries

Recall basic probability definitions from Chapter 1. The *variance* of a r.v. X is defined to be $V[X] = E[(X-\mu)^2]$, where $\mu = E[X]$. Now consider $E[(X-\mu)^2] = E[X^2 - 2\mu X + \mu^2] = E[X^2] - 2\mu E[X] + E[\mu^2] = E[X^2] - E[X]^2$. Thus, $V[X] = E[(X-\mu)^2] = E[X^2] - E[X]^2$. Moreover, for two independent random variables, X and Y, $Var[X+Y] = Var[X] + Var[Y]$. Markov's inequality states that for a non-negative discrete r.v. X and $t > 0$ a constant, $P(X \geq t) \leq E[X]/t$. Furthermore, Chebyshev's inequality states that $Pr(|X-\mu| \geq t\sqrt{Var[X]}) \leq \frac{1}{t^2}$.

Let us toss a fair coin n times and estimate what is the probability of getting $\frac{3}{4}n$ heads? Let X_i be a 0-1 r.v. indicating the outcome of the i-th toss, where $X_i = 1$ ($X_i = 0$) indicates that the outcome is a head (tail). Let $X = \sum_{i=1}^{n} X_i$ and X represents the total number of heads in n-tosses. Observe that $E[X_i] = 1/2$ and $V[X_i] = E[X_i{}^2] - E[X_i]^2 = 1/2 * 1^2 + 1/2 * 0^2 - (1/2)^2 = 1/4$. Thus $E[X] = n/2$ and since X_i's are independent, $V[X] = n/4$. To estimate $Pr[X \geq \frac{3}{4}n]$ using Markov's inequality, set $t = 3/2$ and it results in $Pr[X \geq 3/2 * n/2] \leq 2/3$. To estimate it using Chebyshev's inequality, $Pr(|X-\mu| \geq t\sqrt{Var[X]}) \leq \frac{1}{t^2}$, we need to set $t = \sqrt{n/4}$ to obtain $Pr(|X - n/2| \geq n/4) \leq 4/n$.

We will use the following result for the second moment method.

Theorem 9.1.1 *If $X \geq 0$ is a r.v. taking integer values, $Pr[X = 0] \leq \frac{Var[X]}{E[X]^2}$. Moreover, if $Var[X] = o(E[X]^2)$ for large values of n, $Pr[X = 0] = o(1)$.*

Proof. We will use Chebyshev's inequality. Observe that $Pr[X = 0] \leq Pr[|X - E[X]| \geq E[X]]$. By substituting $t = \frac{E[X]}{\sqrt{Var[X]}}$ in Chebyshev's inequality $Pr(|X - E[X]| \geq t\sqrt{Var[X]}) \leq \frac{1}{t^2}$, we obtain

$Pr[X=0] \leq \frac{Var[X]}{E[X]^2}$. If $Var[X] = o(E[X]^2)$ for large values of n, $Pr[X=0] = \lim_{n\to\infty} \frac{Var[X]}{E[X]^2} \to 0$, or in other words $Pr[X=0] = o(1)$. ■

We will see applications of this theorem in this chapter.

9.2 Cliques in a random graph

Consider a random graph $G(n, p = 1/2)$. Note that $G(n,p)$ is a graph on n vertices where each edge between a pair of vertices occurs (independently of other edges) with probability p. We will provide an estimate on the size of the largest clique in $G(n,1/2)$ using the second moment method. In particular we will prove that

Theorem 9.2.1 *Let $\omega(G)$ denote the size of the largest clique in a graph G. Given an $\epsilon > 0$, (a) $Pr[\omega(G(n,1/2)) > (2+\epsilon)\log n] = o(1)$ and (b) $Pr[(2-\epsilon)\log n \leq \omega(G(n,1/2)) \leq (2+\epsilon)\log n] = 1 - o(1)$.*

To prove the first part of this theorem, we will estimate how many subsets of k vertices in $G(n,1/2)$ will form a clique. Fix a subset S of k vertices and let X_S be a 0-1 indicator r.v. indicating whether S is a clique or not. If S is a clique then $X_S = 1$, otherwise $X_S = 0$. Since each edge in S is chosen independent of other edges, $Pr[X_S = 1] = (\frac{1}{2})^{\binom{k}{2}}$ and $E[X_S] = Pr[X_S = 1]$. Let X denote the total number of cliques of size k in $G(n,1/2)$ and let V be the set of vertices in $G(n,1/2)$. Observe that

$$E[X] = \sum_{S\subset V:|S|=k} E[X_S] = \binom{n}{k}\left(\frac{1}{2}\right)^{\binom{k}{2}} \leq \left(\frac{\sqrt{2}n}{2^{k/2}}\right)^k.$$

By setting $k = (2+\epsilon)\log n$ as in Theorem, we observe that $\frac{\sqrt{2}n}{2^{k/2}} \leq \sqrt{2}n^{-\epsilon/2}$. For large values of n, $n^{-\epsilon/2} < 1$. Also $(n^{-\epsilon/2})^k$ is $o(1)$ (as n increases, $k = (2+\epsilon)\log n$ also increases). Thus by Markov's inequality $Pr[X \geq 1] \leq E[X] = o(1)$, i.e. the probability that there is a clique on k vertices in $G(n,1/2)$ is very small for large values of n.

Next we show the second part of Theorem 9.2.1 using the Second Moment Method. Let X_S and X be as before and let $k = (2-\epsilon)\log n$. Using this value of k, it is easy to see that for large values of n, $E[X] = \binom{n}{k}\left(\frac{1}{2}\right)^{\binom{k}{2}} \to \infty$. Therefore, we will show that $Var[X] = o(E[X]^2)$ and then use Theorem 9.1.1 to conclude that the probability that there is no clique is $o(1)$ and hence the probability that there is a clique is $1 - o(1)$. Let us first compute $Var[X] = E[X^2] - E[X]^2$. Note that

$$E[X^2] = E[\sum_{S\subset V:|S|=k}\sum_{T\subset V:|T|=k} X_S X_T] = \sum_{S\subset V:|S|=k}\sum_{T\subset V:|T|=k} E[X_S X_T].$$

Moreover, $E[X]^2 = E[X]E[X] = \sum_{S\subset V:|S|=k} E[X_S] \sum_{T\subset V:|T|=k} E[X_T]$. Now consider two sets of k vertices S and T. If they have fewer than two vertices in common than whether S is a clique or not has no influence on whether T is a clique or not. Therefore, for $|S\cap T| < 2$, $E[X_S X_T] = E[X_S]E[X_T]$. Thus we need to consider

$$Var[X] = \sum_{S\subset V:|S|=k}\sum_{T\subset V:|T|=k} E[X_S X_T] - E[X_S]E[X_T] \qquad (9.1)$$

only for those pairs of S and T such that $|S \cap T| \geq 2$, since for the other values $E[X_S X_T]$ cancels $E[X_S]E[X_T]$. Thus, Equation 9.1 can be rewritten as

$$Var[X] = \sum_{S \subset V:|S|=k} \sum_{l=2}^{k} \sum_{T \subset V:|T|=k,|S \cap T|=l} E[X_S X_T] - E[X_S]E[X_T]. \tag{9.2}$$

Since, $E[X_S]E[X_T]$ is non-negative, we can obtain the following inequality from Equation 9.2.

$$Var[X] \leq \sum_{S \subset V:|S|=k} \sum_{l=2}^{k} \sum_{T \subset V:|T|=k,|S \cap T|=l} E[X_S X_T]. \tag{9.3}$$

Now observe that there are $\binom{n}{k}$ possibilities for choosing vertices for forming the set S of size k, there are $\binom{k}{l}$ possibilities for selecting l vertices that are common between S and T and there are $\binom{n-k}{k-l}$ possibilities for choosing the remaining vertices in $T \setminus S$. Hence,

$$Var[X] \leq \sum_{l=2}^{k} \binom{n}{k}\binom{k}{l}\binom{n-k}{k-l} E[X_S X_T]. \tag{9.4}$$

Let us compute $E[X_S X_T]$. Since X_S and X_T are 0-1 r.v., $E[X_S X_T]$ is same as the probability that S is a clique and T is a clique, or in other words, all pairs of vertices in S are connected and all pairs of vertices in T are connected. In all there are k vertices in S and $k-l$ vertices in T. The total number of edges in $S \cup T$ is $2\binom{k}{2} - \binom{l}{2}$. Thus,

$$E[X_S X_T] = \frac{1}{2}^{2\binom{k}{2}-\binom{l}{2}} \tag{9.5}$$

Substituting expression for $E[X_S X_T]$ in Equation 9.4, we obtain

$$Var[X] \leq \sum_{l=2}^{k} \binom{n}{k}\binom{k}{l}\binom{n-k}{k-l} 2^{\binom{l}{2}-2\binom{k}{2}}. \tag{9.6}$$

Next we show that $Var[X]/E[X]^2$ is $o(1)$. Note that $E[X] = \binom{n}{k}2^{-\binom{k}{2}}$. Define

$$f(l) = \frac{\binom{n}{k}\binom{k}{l}\binom{n-k}{k-l}2^{\binom{l}{2}-2*\binom{k}{2}}}{(\binom{n}{k}2^{-\binom{k}{2}})^2}.$$

This can be rewritten as

$$f(l) = \frac{\binom{k}{l}\binom{n-k}{k-l}2^{\binom{l}{2}}}{\binom{n}{k}}.$$

Thus,

$$\frac{Var[X]}{E[X]^2} \leq \sum_{l=2}^{k} f(l). \tag{9.7}$$

Consider

$$\begin{aligned} f(2) &= \frac{\binom{k}{2}\binom{n-k}{k-2}2^{\binom{2}{2}}}{\binom{n}{k}} \\ &= \frac{k(k-1)\binom{n-k}{k-2}}{\binom{n}{k}} \end{aligned}$$

Since $k = (2+\epsilon)\log n << n$, it can be shown that $f(2) = o(1)$. Similarly, $f(k) = o(1)$. It seems that (but I don't know how to show this in a simple way) that $f(l) = o(1)$ for $l = 2, \cdots, k$. This concludes the proof of Theorem 9.2.1.

9.3 Thresholds for Random Geometric Graphs

Wireless networks are usually modelled as disk graphs in the plane. Given a set P of points in the plane and a positive parameter r, the *disk graph* is the geometric graph with vertex set P which has a straight-line edge between two points $p, q \in P$ if and only if $|pq| \leq r$, where $|pq|$ denotes the Euclidean distance between p and q. If $r = 1$, then the disk graph is referred to as *unit disk graph*. A *random geometric graph*, denoted by $G(n, r)$, is a geometric graph formed by choosing n points independently and uniformly at random in a unit square; two points are connected by a straight-line edge if and only if they are at Euclidean distance at most r, where $r = r(n)$ is a function of n and $r \to 0$ as $n \to \infty$.

We say that two line segments in the plane *cross* each other if they have a point in common that is interior to both edges. Two line segments are *non-crossing* if they do not cross. Note that two non-crossing line segments may share an endpoint. A geometric graph is said to be *plane* if its edges do not cross, and *non-plane*, otherwise. A graph is *planar* if and only if it does not contain K_5 (the complete graph on 5 vertices) or $K_{3,3}$ (the complete bipartite graph on six vertices partitioned into two parts each of size 3) as a minor. A *non-planar graph* is a graph which is not planar.

A graph property $\mathcal{P}$ is *increasing* if a graph G satisfies $\mathcal{P}$, then by adding edges to G, the property $\mathcal{P}$ remains valid in G. Similarly, $\mathcal{P}$ is *decreasing* if a graph G satisfies $\mathcal{P}$, then by removing edges from G, the property $\mathcal{P}$ remains valid in G. $\mathcal{P}$ is called a *monotone* property if $\mathcal{P}$ is either increasing or decreasing. Connectivity and "having a clique of size k" are increasing monotone properties, while planarity and "being plane" are decreasing monotone properties in $G(n, r)$, where the value of r increases.

By [39] any monotone property of a random geometric graphs has a threshold function. The thresholds in random geometric graphs are expressed by the distance r. In the sequel, the term w.h.p. (with high probability) is to be interpreted to mean that the probability tends to 1 as $n \to \infty$. For an increasing property $\mathcal{P}$, the threshold is a function $t(n)$ such that if $r = o(t(n))$ then w.h.p. $\mathcal{P}$ does not hold in $G(n, r)$, and if $r = \omega(t(n))$ then w.h.p. $\mathcal{P}$ holds in $G(n, r)$. Symmetrically, for a decreasing property $\mathcal{P}$, the threshold is a function $t(n)$ such that if $r = o(t(n))$ then w.h.p. $\mathcal{P}$ holds in $G(n, r)$, and if $r = \omega(t(n))$ then w.h.p. $\mathcal{P}$ does not hold in $G(n, r)$. Note that a threshold function may not be unique. It is well known that $\sqrt{\ln n/n}$ is a connectivity threshold for $G(n, r)$; see [41, 81, 82]. Here we investigate thresholds in random geometric graphs for having a connected subgraph of constant size, being plane, and being planar.

Related Work: Random graphs were first defined and formally studied by Gilbert in [35] and Erdös and Rényi [29]. It seems that the concept of a random geometric graph was first formally suggested by Gilbert in [36] and for that reason is also known as Gilbert's disk model. These classes of graphs are known to have numerous applications as a model for studying communication primitives (broadcasting, routing, etc.) and topology control (connectivity, coverage, etc.) in idealized wireless sensor networks as well as extensive utility in theoretical computer science and many fields of the mathematical sciences.

An instance of Erdös-Rényi graph [29] is obtained by taking n vertices and connecting any two with probability p, independently of all other pairs; the graph derived by this scheme is denoted by $G_{n,p}$. In $G_{n,p}$ the threshold is expressed by the edge existence probability p, while in $G(n,r)$ the threshold is expressed in terms of r. In both random graphs and random geometric graphs, property thresholds are of great interest [9, 17, 33, 39, 73]. Note that edge crossing configurations in $G(n,r)$ have a geometric nature, and as such, have no analogues in the context of the Erdös-Rényi model for random graphs. However, planarity, and having a clique of specific size are of interest in both $G_{n,p}$ and $G(n,r)$.

Bollobás and Thomason [10] showed that any monotone property in random graphs has a threshold function. See also a result of Friedgut and Kalai [33], and a result of Bourgain and Kalai [16]. In the Erdös-Rényi random graph $G_{n,p}$, the connectivity threshold is $p = \log n/n$ and the threshold for having a giant component is $p = 1/n$; see [4]. The planarity threshold for $G_{n,p}$ is $p = 1/n$; see [9, 89].

A general reference on random geometric graphs is [84]. There is extensive literature on various aspects of random geometric graphs of which we mention the related work on coverage by [44, 51] and a review on percolation, connectivity, coverage and colouring by [7]. As in random graphs, any monotone property in geometric random graphs has a threshold function [17, 39, 65, 73].

Random geometric graphs have a connectivity threshold of $\sqrt{\ln n/n}$; see [41, 81, 82]. Gupta and Kumar [41] provided a connectivity threshold for points that are uniformly distributed in a disk. By a result of Penrose [83], in $G(n,r)$, any threshold function for having no isolated vertex (a vertex of degree zero) is also a connectivity threshold function. Panchapakesan and Manjunath [81] showed that $\sqrt{\ln n/n}$ is a threshold for being an isolated vertex in $G(n,r)$. This implies that $\sqrt{\ln n/n}$ is a connectivity threshold for $G(n,r)$. For $k \geq 2$, the details on the k-connectivity threshold in random geometric graphs can be found in [83, 84]. Connectivity of random geometric graphs for points on a line is studied by Godehardt and Jaworski [38]. Appel and Russo [6] considered the connectivity under the L_∞-norm.

Organization: In Section 9.3.1 we show that for a constant k, the distance threshold for having a connected subgraph on k points is $n^{\frac{-k}{2k-2}}$. We show that the same threshold is valid for the existence of a clique of size k. Based on that, we prove the following thresholds for a random geometric graph to be plane or planar. In Section 9.3.2, we prove that $n^{-2/3}$ is a distance threshold for a random geometric graph to be plane. In Section 9.3.3, we prove that $n^{-5/8}$ is a distance threshold for a random geometric graph to be planar.

9.3.1 The threshold for having a connected subgraph on k points

In this section, we look for the distance threshold for "existence of connected subgraphs of constant size"; this is an increasing property. For a given constant k, we show that $n^{\frac{-k}{2k-2}}$ is the threshold function for the existence of a connected subgraph on k points in $G(n,r)$. Specifically, we show that if $r = o(n^{\frac{-k}{2k-2}})$, then w.h.p. $G(n,r)$ has no connected subgraph on k points, and if $r = \omega(n^{\frac{-k}{2k-2}})$, then w.h.p. $G(n,r)$ has a connected subgraph on k points. We also show that the same threshold function holds for the existence of a clique of size k.

Theorem 9.3.1 *Let $k \geq 2$ be an integer constant. Then, $n^{\frac{-k}{2k-2}}$ is a distance threshold function for $G(n,r)$ to have a connected subgraph on k points.*

Proof. Let $P_1, \ldots, P_{\binom{n}{k}}$ be an enumeration of all subsets of k points in $G(n,r)$. Let $DG[P_i]$ be the subgraph of $G(n,r)$ that is induced by P_i. Let X_i be the random variable such that

$$X_i = \begin{cases} 1 & \text{if } DG[P_i] \text{ is connected,} \\ 0 & \text{otherwise.} \end{cases}$$

Let the random variable X count the number of sets P_i for which $DG[P_i]$ is connected. It is clear that

$$X = \sum_{i=1}^{\binom{n}{k}} X_i. \tag{9.8}$$

Observe that $\mathrm{E}[X_i] = \Pr[X_i = 1]$. Since the random variables X_i have identical distributions, we have

$$\mathrm{E}[X] = \binom{n}{k} \mathrm{E}[X_1]. \tag{9.9}$$

We obtain an upper bound and a lower bound for $\Pr[X_i = 1]$. First, partition the unit square into squares of side equal to r. Let $\{s_1, \ldots, s_{1/r^2}\}$ be the resulting set of squares. For a square s_t, let S_t be the $kr \times kr$ square which has s_t on its left bottom corner; see Figure 9.1(a). S_t contains at most k^2 squares each of side length r (each S_t on the boundary of the unit square contains less than k^2 squares). Let $A_{i,t}$ be the event that all points in P_i are contained in S_t. Observe that if $DG[P_i]$ is connected then P_i lies in S_t for some $t \in \{1, \ldots, 1/r^2\}$. Therefore,

$$\text{if } DG[P_i] \text{ is connected, then } (A_{i,1} \vee A_{i,2} \vee \cdots \vee A_{i,1/r^2}),$$

and hence we have

$$\Pr[X_i = 1] \leq \sum_{t=1}^{1/r^2} \Pr[A_{i,t}] \leq \sum_{t=1}^{1/r^2} (k^2 r^2)^k = k^{2k} r^{2k-2}. \tag{9.10}$$

Now, partition the unit square into squares with diagonal length equal to r. Each such square has side length equal to $r/\sqrt{2}$. Let $\{s_1, \ldots, s_{2/r^2}\}$ be the resulting set of squares. Let $B_{i,t}$ be the event that all points of P_i are in s_t. Observe that if all points of P_i are in the same square, then $DG[P_i]$ is a complete graph and hence connected. Therefore,

$$\text{if } (B_{i,1} \vee B_{i,2} \vee \cdots \vee B_{i,2/r^2}), \text{ then } DG[P_i] \text{ is connected,}$$

and hence we have

$$\Pr[X_i = 1] \geq \sum_{t=1}^{2/r^2} \Pr[B_{i,t}] = \sum_{t=1}^{2/r^2} \left(\frac{r^2}{2}\right)^k = \frac{1}{2^{k-1}} r^{2k-2}. \tag{9.11}$$

Since $k \geq 2$ is a constant, Inequalities (9.10) and (9.11) and Equation (9.9) imply that

$$\mathrm{E}[X_i] = \Theta(r^{2k-2}), \tag{9.12}$$
$$\mathrm{E}[X] = \Theta(n^k r^{2k-2}). \tag{9.13}$$

If $n \to \infty$ and $r = o(n^{\frac{-k}{2k-2}})$ we conclude that the following inequalities are valid

$$\begin{aligned}\Pr[X \geq 1] &\leq \mathrm{E}[X] \text{ (by Markov's Inequality)}\\ &= \Theta(n^k r^{2k-2}) \text{ (by (9.13))}\\ &= o(1). \end{aligned} \tag{9.14}$$

Therefore, w.h.p. $G(n,r)$ has no connected subgraph on k points.

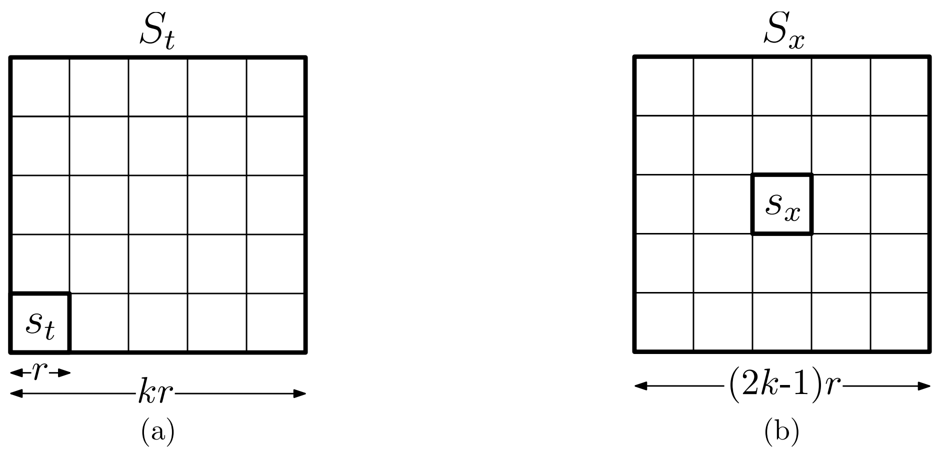


Figure 9.1: (a) The square S_t has s_t on its left bottom corner. (b) The square S_x which is centered at s_x.

In the rest of the proof, we assume that $r = \omega(n^{\frac{-k}{2k-2}})$. In order to show that w.h.p. $G(n,r)$ has at least one connected subgraph on k vertices, we show, using the second moment method [4], that $\Pr[X = 0] \to 0$ as $n \to \infty$. Recall from Chebyshev's inequality that

$$\Pr[X = 0] \leq \frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2}. \tag{9.15}$$

Therefore, in order to show that $\Pr[X = 0] \to 0$, it suffices to show that

$$\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2} \to 0. \tag{9.16}$$

In view of Identity (9.8) we have

$$\mathrm{Var}(X) = \sum_{1 \leq i,j \leq \binom{n}{k}} \mathrm{Cov}(X_i, X_j), \tag{9.17}$$

where $\mathrm{Cov}(X_i, X_j) = \mathrm{E}[X_i X_j] - \mathrm{E}[X_i]\mathrm{E}[X_j] \leq \mathrm{E}[X_i X_j]$. If $|P_i \cap P_j| = 0$ then $DG[P_i]$ and $DG[P_j]$ are disjoint. Thus, the random variables X_i and X_j are independent, and hence $\mathrm{Cov}(X_i, X_j) = 0$.

It is enough to consider the cases when P_i and P_j are not disjoint. Assume $|P_i \cap P_j| = w$, where $w \in \{1, \ldots, k\}$. Thus, in view of Equation (9.17), we have

$$\begin{aligned} \mathrm{Var}(X) &= \sum_{w=1}^{k} \sum_{|P_i \cap P_j| = w} \mathrm{Cov}(X_i, X_j) \\ &\le \sum_{w=1}^{k} \sum_{|P_i \cap P_j| = w} \mathrm{E}[X_i X_j]. \end{aligned} \tag{9.18}$$

The computation of $\mathrm{E}[X_i, X_j]$ involves some geometric considerations which are being discussed in detail below. Since X_i and X_j are 0-1 random variables, $X_i X_j$ is a 0-1 random variable and

$$X_i X_j = \begin{cases} 1 & \text{if both } DG[P_i] \text{ and } DG[P_j] \text{ are connected,} \\ 0 & \text{otherwise.} \end{cases}$$

By the definition of the expected value we have

$$\begin{aligned} \mathrm{E}[X_i X_j] &= \Pr[X_j = 1 | X_i = 1] \Pr[X_i = 1] \\ &= \Pr[X_j = 1 | X_i = 1] \mathrm{E}[X_i]. \end{aligned} \tag{9.19}$$

By (9.12), $\mathrm{E}[X_i] = \Theta(r^{2k-2})$. It remains to compute $\Pr[X_j = 1 | X_i = 1]$, i.e., the probability that $DG[P_j]$ is connected given that $DG[P_i]$ is connected. Consider the k-tuples P_i and P_j under the condition that $DG[P_i]$ is connected. Let x be a point in $P_i \cap P_j$. Partition the unit square into squares of side length equal to r. Let s_x be the square containing x. Let S_x be the $(2k-1)r \times (2k-1)r$ square centered at s_x. S_x contains at most $(2k-1)^2$ squares each of side length r (if S_x is on the boundary of the unit square then it contains less than $(2k-1)^2$ squares); see Figure 9.1(b). The area of S_x is at most $(2kr)^2$, and hence the probability that a specific point of P_j is in S_t is at most $4k^2r^2$. Since P_i and P_j share w points, in order for $DG[P_j]$ to be connected, the remaining $k - w$ points of P_j must lie in S_x. Thus, the probability that $DG[P_j]$ is connected given that $DG[P_i]$ is connected is at most $(4k^2r^2)^{k-w} \le c_w r^{2k-2w}$, for some constant $c_w > 0$. Thus, $\Pr[X_j = 1 | X_i = 1] \le c_w r^{2k-2w}$. In view of Equation (9.19), we have

$$\mathrm{E}[X_i X_j] \le c'_w \cdot r^{2k-2w} \cdot r^{2k-2} = c'_w r^{4k-2w-2}, \tag{9.20}$$

for some constant $c'_w > 0$.

Since P_i and P_j are k-tuples which share w points, $|P_i \cup P_j| = 2k - w$. There are $\binom{n}{2k-w}$ ways to choose $2k - w$ points for $P_i \cup P_j$. Since we choose w points for $P_i \cap P_j$, $k - w$ points for P_i alone, and $k - w$ points for P_j alone, there are $\binom{2k-w}{w, k-w, k-w}$ ways to split the $2k - w$ chosen points into P_i and P_j. Based on this and Inequality (9.20), Inequality (9.18) turns out to

$$\begin{aligned}\mathrm{Var}(X) &\le \sum_{w=1}^{k} \sum_{|P_i \cap P_j|=w} \mathrm{E}[X_i X_j] \\ &\le \sum_{w=1}^{k} \binom{n}{2k-w} \binom{2k-w}{w, k-w, k-w} c'_w r^{4k-2w-2} \\ &\le \sum_{w=1}^{k} c''_w n^{2k-w} r^{4k-2w-2}.\end{aligned}$$

for some constants $c''_w > 0$. Consider (9.16) and note that by (9.13), $\mathrm{E}[X]^2 \ge c'' n^{2k} r^{4k-4}$, for some constant $c'' > 0$. Thus,

$$\begin{aligned}\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2} &\le \sum_{w=1}^{k} \frac{c''_w n^{2k-w} r^{4k-2w-2}}{c'' n^{2k} r^{4k-4}} = \sum_{w=1}^{k} \frac{c''_w}{c''} \cdot \frac{1}{n^w r^{2w-2}} \\ &= \frac{c''_1}{c''} \cdot \frac{1}{n^1 r^0} + \frac{c''_2}{c''} \cdot \frac{1}{n^2 r^2} + \cdots + \frac{c''_k}{c''} \cdot \frac{1}{n^k r^{2k-2}} \qquad (9.21)\end{aligned}$$

Since $r = \omega(n^{\frac{-k}{2k-2}})$, all terms in (9.21) tend to zero. This proves the convergence in (9.16). Thus, $\Pr[X = 0] \to 0$ as $n \to \infty$. This implies that if $r = \omega(n^{\frac{-k}{2k-2}})$, then $G(n, r)$ has a connected subgraph on k vertices with high probability. ■ ■

In the following theorem we show that if $k = O(1)$, then $n^{\frac{-k}{2k-2}}$ is also a threshold for $G(n, r)$ to have a clique of size k; this is an increasing property.

Theorem 9.3.2 *Let $k \ge 2$ be an integer constant. Then, $n^{\frac{-k}{2k-2}}$ is a distance threshold function for $G(n, r)$ to have a clique of size k.*

Proof. By Theorem 9.3.1, if $r = o(n^{\frac{-k}{2k-2}})$, then w.h.p. $G(n, r)$ has no connected subgraph on k vertices, and hence it has no clique of size k. This proves the first statement. We prove the second statement by adjusting the proof of Theorem 9.3.1, which is based on the second moment method. Assume $r = \omega(n^{\frac{-k}{2k-2}})$. Let $P_1, \ldots, P_{\binom{n}{k}}$ be an enumeration of all subsets of k points. Let X_i be equal to 1 if $DG[P_i]$ is a clique, and 0 otherwise. Let $X = \sum X_i$.

Partition the unit square into a set $\{s_1, \ldots, s_{1/r^2}\}$ of squares of side length r. Let S_t be the $2r \times 2r$ square which has s_t on its left bottom corner. If $DG[P_i]$ is a clique then P_i lies in S_t for some $t \in \{1, \ldots, 1/r^2\}$. Therefore,

$$\Pr[X_i = 1] \le 4^k r^{2k-2}.$$

Now, partition the unit square into a set $\{s_1, \ldots, s_{2/r^2}\}$ of squares with diagonal length r. If all points of P_i fall in the square s_t, then $DG[P_i]$ is a clique. Thus,

$$\Pr[X_i = 1] \ge \frac{1}{2^{k-1}} r^{2k-2}.$$

Since $k \geq 2$ is a constant, we have

$$\mathrm{E}[X_i] = \Theta(r^{2k-2}),$$
$$\mathrm{E}[X] = \Theta(n^k r^{2k-2}).$$

In view of Chebyshev's inequality we need to show that $\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2}$ tends to 0 as n goes to infinity. We bound $\mathrm{Var}(X)$ from above by Inequality (9.18). Consider the k-tuples P_i and P_j under the condition that $DG[P_i]$ is a clique. Let $|P_i \cap P_j| = w$, and let x be a point in $P_i \cap P_j$. Partition the unit square into squares of side length r. Let s_x be the square containing x. Let S_x be the $3r \times 3r$ square centered at s_x. In order for $DG[P_j]$ to be a clique, the remaining $k - w$ points of P_j must lie in S_x. Thus,

$$\mathrm{E}[X_i X_j] \leq c'_w r^{4k-2w-2},$$

for some constant $c'_w > 0$. By a similar argument as in the proof of Theorem 9.3.1, we can show that for some constants $c'', c''_w > 0$ the followings inequalities are valid:

$$\mathrm{Var}(X) \leq \sum_{w=1}^{k} c''_w n^{2k-w} r^{4k-2w-2},$$
$$\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2} \leq \sum_{w=1}^{k} \frac{c''_w}{c''} \cdot \frac{1}{n^w r^{2w-2}}.$$

Since $r = \omega(n^{\frac{-k}{2k-2}})$, the last inequality tends to 0 as n goes to infinity. This completes the proof for the second statement. ■ ■ As a direct consequence of Theorem 9.3.2, we have the following

corollary.

Corollary 9.3.3 n^{-1} *is a threshold for* $G(n,r)$ *to have an edge, and* $n^{-\frac{3}{4}}$ *is a threshold for* $G(n,r)$ *to have a triangle.*

9.3.2 The threshold for $G(n,r)$ to be plane

In this section we investigate the threshold for a random geometric graph to be plane; this is a decreasing property. Recall that $G(n,r)$ is plane if no two of its edges cross. As a warm-up exercise we first prove a simple result which is based on the connectivity threshold for random geometric graphs, which is known to be $\sqrt{\ln n/n}$.

Theorem 9.3.4 *If* $r \geq \sqrt{\frac{c \ln n}{n}}$, *with* $c \geq 36$, *then w.h.p.* $G(n,r)$ *is not plane.*

Proof. In order to prove that w.h.p. $G(n,r)$ is not plane, we show that w.h.p. it has a pair of crossing edges. Partition the unit square into squares each with diagonal length r. Then subdivide each such square into nine sub-squares as depicted in Figure 9.2. There are $\frac{18}{r^2}$ sub-squares, each of side length $\frac{r}{3\sqrt{2}}$. The probability that no point lies in a specific sub-square is $(1 - \frac{r^2}{18})^n$. Thus, the probability that there exists an empty sub-square is at most

$$\frac{18}{r^2}\left(1 - \frac{r^2}{18}\right)^n \leq n\left(1 - \frac{c \ln n}{18n}\right)^n \leq n^{1-c/18} \leq \frac{1}{n},$$

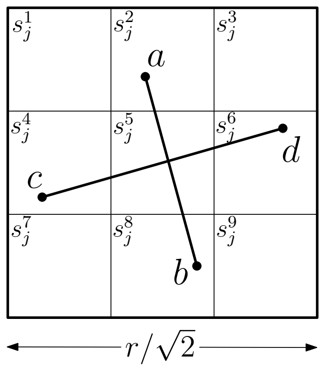


Figure 9.2: An square of diameter r which is partitioned into nine sub-squares.

when $c \geq 36$. Therefore, with probability at least $1 - \frac{1}{n}$ all sub-squares contain points. By choosing four points a, b, c, and d as depicted in Figure 9.2, it is easy to see that the edges (a, b) and (c, d) cross. Thus, w.h.p. $G(n, r)$ has a pair of crossing edges, and hence w.h.p. it is not plane. ■ ■

In fact, Theorem 9.3.4 ensures that w.h.p. there exists a pair of crossing edges in each of the squares. This implies that there are $\Omega\left(\frac{n}{\ln n}\right)$ disjoint pair of crossing edges, while for $G(n, r)$ to be not plane we need to show the existence of at least one pair of crossing edges. Thus, the value of r provided by the connectivity threshold seems rather weak. By a different approach, in the rest of this section we show that $n^{-\frac{2}{3}}$ is the correct threshold.

Lemma 9.3.5 *Let (a, b) and (c, d) be two crossing edges in $G(n, r)$, and let Q be the convex quadrilateral formed by a, b, c, and d. Then, two adjacent sides of Q are edges of $G(n, r)$.*

Proof. Refer to Figure 9.3. At least one of the angles of Q, say $\angle cad$, is bigger than or equal to $\pi/2$. It follows that in the triangle $\triangle cad$ the side cd is the longest, i.e., $|cd| \geq \max\{|ac|, |ad|\}$. Since $|cd| \leq r$, both $|ac|$ and $|ad|$ are at most r. Thus, ac and ad—which are adjacent—are edges of $G(n, r)$. ■ ■

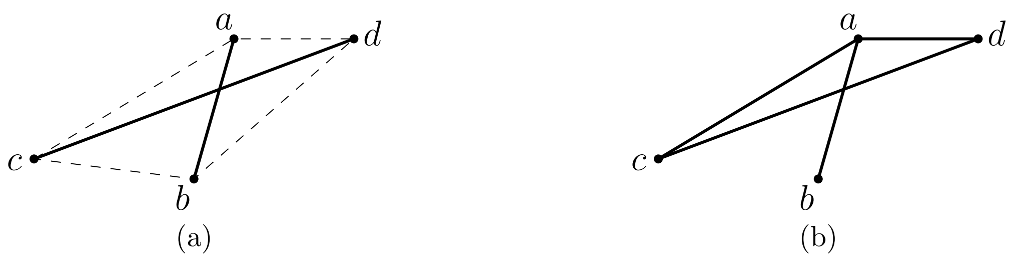


Figure 9.3: (a) Illustration of Lemma 9.3.5. (b) Crossing edges (a, b) and (c, d) form an anchor.

In the proof of Lemma 9.3.5, a is connected to b, c, and d. So the distance between a to each of b, c, and d is at most r. Thus, we have the following corollary.

Corollary 9.3.6 *The endpoints of every two crossing edges in $G(n, r)$ are at distance at most $2r$ from each other. Moreover, there exists an endpoint which is within distance r from other endpoints.*

Based on the proof of Lemma 9.3.5, we define an *anchor* as a set $\{a, b, c, d\}$ of four points in $G(n, r)$ such that three of them form a triangle, say $\triangle cad$, and the fourth vertex, b, is connected to a by an edge which crosses cd; see Figure 9.3(b). We call a as the *crown* of the anchor. The crown is within distance r from the other three points. Note that bc and bd may or may not be edges of $G(n, r)$. In view of Lemma 9.3.5, two crossing edges in $G(n, r)$ form an anchor. Conversely, every anchor in $G(n, r)$ introduces a pair of crossing edges.

Observation 9.3.7 *$G(n, r)$ is plane if and only if it has no anchor.*

Theorem 9.3.8 *$n^{-\frac{2}{3}}$ is a threshold for $G(n, r)$ to be plane.*

Proof. In order to show that $G(n, r)$ is plane, by Observation 9.3.7, it is enough to show that it has no anchors. Every anchor has four points and it is connected. By Theorem 9.3.1, if $r = o(n^{-\frac{2}{3}})$, then w.h.p. $G(n, r)$ has no connected subgraph on 4 points, and hence it has no anchors. This proves the first statement.

We prove the second statement by adjusting the proof of Theorem 9.3.1 for $k = 4$. Assume $r = \omega(n^{-\frac{2}{3}})$. Let $P_1, \dots, P_{\binom{n}{4}}$ be an enumeration of all subsets of 4 points. Let X_i be equal to 1 if $DG[P_i]$ contains an anchor, and 0 otherwise. Let $X = \sum X_i$. In view of Chebyshev's inequality we need to show that $\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2}$ tends to 0 as n goes to infinity.

Partition the unit square into a set $\{s_1, \dots, s_{2/r^2}\}$ of squares with diagonal length r. Then, subdivide each square s_j, into nine sub-squares $s_j^1, \dots, s_j^9$ as depicted in Figure 9.2. If each of $s_j^1, s_j^3, s_j^7, s_j^9$ or each of $s_j^2, s_j^4, s_j^6, s_j^8$ contains a point of P_i, then $DG[P_i]$ is a convex clique of size four and hence it contains an anchor. Thus,

$$\Pr[X_i = 1] \geq \frac{r^6}{2^3} \cdot \frac{2}{9^4}.$$

This implies that $\mathrm{E}[X_i] = \Omega(r^6)$, and hence $\mathrm{E}[X] = \Omega(n^4 r^6)$. Therefore,

$$\mathrm{E}[X]^2 \geq c'' n^8 r^{12},$$

for some constant $c'' > 0$. By a similar argument as in the proof of Theorem 9.3.1 we bound the variance of X from above by

$$\mathrm{Var}(X) \leq c_1'' n^7 r^{12} + c_2'' n^6 r^{10} + c_3'' n^5 r^8 + c_4'' n^4 r^6.$$

Since $r = \omega(n^{-\frac{2}{3}})$, $\frac{\mathrm{Var}(X)}{\mathrm{E}[X]^2}$ tends to 0 as n goes to infinity. That is, w.h.p. $G(n, r)$ has an anchor. By Observation 9.3.7, w.h.p. $G(n, r)$ is not plane. ■ ■

As a direct consequence of the proof of Theorem 9.3.8, we have the following:

Corollary 9.3.9 *With high probability if a random geometric graph is not plane, then it has a clique of size four.*

Note that every anchor introduces a crossing and each crossing introduces an anchor. Since, every anchor is a connected graph and has four points, by (9.13) we have the following corollary.

Corollary 9.3.10 *The expected number of crossings in $G(n, r)$ is $\Theta(n^4 r^6)$.*

9.3.3 The threshold for $G(n, r)$ to be planar

In this section we investigate the threshold for the planarity of a random geometric graph; this is a decreasing property. By Kuratowski's theorem, a finite graph is planar if and only if it does not contain a subgraph that is a subdivision of K_5 or of $K_{3,3}$. Note that any plane random geometric graph is planar too; observe that the reverse statement may not be true. Thus, the threshold for planarity seems to be larger than the threshold of being plane. By a similar argument as in the proof of Theorem 9.3.4 we can show that if $r \geq \sqrt{c \ln n/n}$, then w.h.p. each square with diagonal length r contains K_5, and hence $G(n, r)$ is not planar.

Theorem 9.3.11 *$n^{-\frac{5}{8}}$ is a threshold for $G(n, r)$ to be planar.*

Proof. By Theorem 9.3.2, if $r = \omega(n^{-\frac{5}{8}})$, then w.h.p. $G(n, r)$ has a clique of size 5. Thus, w.h.p. $G(n, r)$ contains K_5 and hence it is not planar. This proves the second statement of the theorem.

If $r = o(n^{-\frac{5}{8}})$, then by Theorem 9.3.1, w.h.p. $G(n, r)$ has no connected subgraph on 5 points, and hence it has no K_5. Similarly, if $r = o(n^{-\frac{3}{5}})$, then w.h.p. $G(n, r)$ has no connected subgraph on 6 points, and hence it has no $K_{3,3}$. Since $n^{-\frac{5}{8}} < n^{-\frac{3}{5}}$, it follows that if $r = o(n^{-\frac{5}{8}})$, then w.h.p. $G(n, r)$ has neither K_5 nor $K_{3,3}$ as a subgraph.

Note that, in order to prove that $G(n, r)$ is planar, we have to show that it does not contain any subdivision of either K_5 or $K_{3,3}$. Any subdivision of either K_5 or $K_{3,3}$ contains a connected subgraph on $k \geq 5$ vertices. Since $n^{-5/8} < n^{-k/(2k-2)}$ for all $k \geq 5$, in view of Theorem 9.3.1, we conclude that if $r = o(n^{-\frac{5}{8}})$, then w.h.p. $G(n, r)$ has no subdivision of K_5 and $K_{3,3}$, and hence $G(n, r)$ is planar. This proves the first statement of the theorem. ■ ■

As a direct consequence of the proof of Theorem 9.3.11, we have the following:

Corollary 9.3.12 *With high probability if a random geometric graph does not contain a clique of size five, then it is planar.*

9.3.4 Conclusions

We presented thresholds for random geometric graphs to have a connected subgraph of constant size, to be plane, and to be planar. A natural open problem is to extend Theorem 9.3.1 for connected subgraphs of k vertices where k is not necessarily a constant, and for connected subgraphs of k vertices which have diameter δ.

9.4 Exercises

9.1 *Show that for two independent random variables, X and Y, $Var[X + Y] = Var[X] + Var[Y]$.*

Chapter 10

Stable Matchings

This entire chapter is contributed by Michael St. Jules.

10.1 Stable Marriages

The stable marriage problem is the following: given n men and n women, each man with a list ranking of the women in order of preference and each woman with a preference list of the men, find a *stable matching* of the men and women, i.e. n man-woman pairs that is *stable*.

Definition 10.1.1 *An **instance** of the stable marriage problem is a set of M of men and a set W of women, each of size $n \in \mathbb{N}$ and for each $m \in M$, an ordered list $L(m)$ that is a permutation of the women in W and for each $w \in W$, an ordered list $L(w)$ that is a permutation of the men in M.*

*We will use $m, m', m'', m_1, m_2, m_3, \ldots, m_n$ to denote men, and similarly for women with w. The lists $L(m)$ and $L(w)$ are m's and w's **preference lists** (or simply **lists**), respectively. We say m **prefers** w **to** w', or w is a **better partner** for m than w' or w' is a **worse partner** for m than w, if w appears before w' in $L(m)$. We do the same for w, m and m'.*

The following is an instance:

$$\begin{array}{ll} m_1\ [w_1\ w_3\ w_2] & w_1\ [m_2\ m_3\ m_1] \\ m_2\ [w_3\ w_1\ w_2] & w_2\ [m_3\ m_1\ m_2] \\ m_3\ [w_3\ w_1\ w_2] & w_3\ [m_1\ m_2\ m_3] \end{array}$$

In practice, the instance would be represented as two $n \times n$ matrices, one with the men's lists filling out the rows and the other with the women's lists filling out the rows.

Definition 10.1.2 *A **matching** for an instance of the stable marriage problem is a set $\mathcal{M} \subseteq M \times W$ such that each $m \in M$ and each $w \in W$ appear in exactly one ordered pair in the set $\mathcal{M}$.*

Hence each man is paired with exactly one woman and each woman, with exactly one man. In fact, $\mathcal{M} : M \to W$ defines a bijection, with $\mathcal{M}(m)$ as the partner of $m \in M$ in the matching and $\mathcal{M}^{-1}(w)$ being that of $w \in W$.

Definition 10.1.3 *A pair $(m, w) \in M \times W$ is a **blocking pair** for the matching $\mathcal{M}$ if m and w both prefer each other to their own partners in $\mathcal{M}$.*

Definition 10.1.4 *A matching $\mathcal{M}$ is **stable** or a **solution**, if there is no blocking pair in $M \times W$ for $\mathcal{M}$.*

The motivation for such a definition is that given a blocking pair (m, w), m and w would *leave* their partners for each other.

Definition 10.1.5 *$(m, w) \in M \times W$ is a **possible pairing** if $(m, w) \in \mathcal{M}$ for some stable matching $\mathcal{M}$. Furthermore, m is called a **possible partner** for w, and similarly w for m.*

Definition 10.1.6 *A stable matching $\mathcal{M}$ is **man-optimal (man-pessimal)** if each $m \in M$ is paired with the $w \in W$ he prefers most (least, respectively) of all possible partners. We let $best(m) = w$ ($worst(m) = w$, resp.), to define a function $best : M \to W$ ($worst : M \to W$, resp.).*

Define **woman-optimal**, **woman-pessimal**, $best : W \to M$ and $worst : W \to M$ similarly.

We will prove that the following algorithm, originally from Gale and Shapley [34], always computes a stable matching (so that one always exists), and in $O(n^2)$ time. We shall also see that the stable matching returned is man-optimal and woman-pessimal, so that it does not depend on the order in which men are chosen to propose. The following more general version of the algorithm is adapted from Kleinberg and Tardos [60], and its analysis is from Gale and Shapley's original paper and some proofs adapted from Irving [50].

Stable marriage algorithm

```
While there remains a man $m \in M$ who has no proposal to another held, do
    $w$ := the next women on $m$'s list to whom $m$ has not proposed
    $m$ proposes to $w$
    if $w$ holds no proposal, then
        $w$ holds $m$'s proposal
    else $m'$ := the man from whom $w$ holds a proposal
        if $w$ prefers $m'$ to $m$, then $w$ rejects $m$
        else
            $w$ rejects $m'$
            $w$ holds $m$'s proposal
return the set $\mathcal{M} := \{(m, w) \in M \times W |\ w$ holds a proposal from $m\}$
```

Let's consider running the algorithm on the following instance:

$$\begin{array}{ll} m_1\ [w_1\ w_3\ w_2] & w_1\ [m_2\ m_3\ m_1] \\ m_2\ [w_3\ w_1\ w_2] & w_2\ [m_3\ m_1\ m_2] \\ m_3\ [w_3\ w_1\ w_2] & w_3\ [m_1\ m_2\ m_3] \end{array}$$

m_1 proposes to w_1, who accepts: (m_1, w_1)
m_2 proposes to w_3, who accepts: $(m_1, w_1), (m_2, w_3)$
m_3 proposes to w_3, who rejects m_3: $(m_1, w_1), (m_2, w_3)$

m_3 proposes to w_1, who accepts m_3 and rejects m_1: $(m_3, w_1), (m_2, w_3)$
m_1 proposes to w_3, who rejects m_1: $(m_3, w_1), (m_2, w_3)$
m_1 proposes to w_2, who accepts: $(m_3, w_1), (m_2, w_3), (m_1, w_2)$
Then $\mathcal{M} := \{(m_3, w_1), (m_2, w_3), (m_1, w_2)\}$ is a stable matching.

Lemma 10.1.7 *The stable marriage algorithm runs in $O(n^2)$-time.*

Proof. We assume that preference checking by women and finding the next person on a preference list by men can be achieved in $O(1)$ time. Hence, given the women's preference lists, we can scan them and form a ranking matrix for the women, where the entry ij corresponds to w_i's ranking of m_j, i.e. m_j's position on $L(w_i)$. Similarly, given a ranking matrix for men, we can construct the men's preference lists by copying the ranking matrix, pairing each entry with the corresponding woman ranked, and then sorting the women on each row with respect to rank. This preprocessing takes $O(n^2)$ time.

Observe that each man proposes at most once to each woman, so at most n^2 proposals are made, and each iteration of the while loop corresponds to a single proposal and takes time $O(1)$. Hence $O(n^2)$ time in all. ■

Note also that $O(n^2)$ is *linear* in the input size.

Lemma 10.1.8 *After the moment $w \in W$ is first proposed to, w always holds a proposal.*

Proof. Once w holds a proposal, she only rejects it for a better one, and there is no 'unproposing'. ■

In fact, each man does worse with each proposal made, while women are never worse off after a proposal than before.

Lemma 10.1.9 *The set $\mathcal{M}$ returned at the end of the stable marriage algorithm is a matching.*

Proof. Suppose, by way of contradiction, that $\mathcal{M}$ is not a matching. First, since women can hold only a single proposal, each $m \in M$ and $w \in W$ appears at most once in $\mathcal{M}$, so it follows that some $m \in M$ or some $w \in W$ is unpaired. These two conditions are equivalent, again since women can only hold a single proposal, and also because $|M| = |W| = n$. Hence, we have both an unpaired man $m \in M$ and an unpaired woman $w \in W$. Since m is unpaired, he had no proposal held at the end of the procedure and so must have proposed to w before termination. Then, by Lemma 10.1.8, w could not have been unpaired. ■

Lemma 10.1.10 *If $w \in W$ rejects $m \in M$ in the stable marriage algorithm, then m and w cannot be partners in any stable matching.*

Proof. Suppose m's rejection by w is the first to occur such that (m, w) is a pairing in a stable matching $\mathcal{M}$. w rejected m because she already held or later received a proposal from a better partner, say m'. Then, so that (m', w) is not a blocking pair for $\mathcal{M}$, contradicting stability, it must be that m' prefers his own partner, w', in $\mathcal{M}$ to w. However, since m' prefers w' to w, m' must have proposed to and been rejected by w' before w, contrary to m's rejection by w being the first such. ■

Corollary 10.1.11 *If in the stable marriage algorithm, $m \in M$ proposed to $w \in W$, then, in any stable matching*

(i) m cannot have a better partner than w;

(ii) w cannot have a worse partner than m.

Proof. m proposed to w since he was rejected by all the women he preferred to w and hence could not be paired with (by Lemma 10.1.10), so (i) follows.

Suppose, by way of contradiction, that $(m', w) \in M \times W$ is a pair in some stable matching $\mathcal{M}$, where w prefers m to m'. Then m is paired with some $w' \in W$ and by (i), m prefers w to w'. Hence (m, w) is a blocking pair for $\mathcal{M}$, contradicting stability. ∎

Theorem 10.1.12 *The stable marriage algorithm computes a man-optimal and woman-pessimal stable matching in $O(n^2)$-time.*

Proof. The $O(n^2)$ upper bound follows from Lemma 10.1.7. The set $\mathcal{M}$ returned is a matching by Lemma 10.1.9. $\mathcal{M}$ is stable, since if $(m, w) \in M \times W$ were a blocking pair, then m prefers w to $\mathcal{M}(m)$, contradicting Corollary 10.1.11(i). That $\mathcal{M}$ is man-optimal and woman-pessimal also follows from Corollary 10.1.11. ∎

Corollary 10.1.13 *A stable matching $\mathcal{M}$ is man-optimal if and only if it is woman-pessimal.*

Corollary 10.1.14 *There always exists a stable matching, i.e. $\mathcal{M} := \{(m, best(m)) \in M \times W | m \in M\} = \{(worst(w), w) \in M \times W | w \in W\}$.*

10.2 Hospitals/Residents (College Admissions)

The hospitals/residents problem is a generalization of the stable marriage problem in which residents and hospitals are considered instead of men and women, allowing multiple residents to be matched to a hospital up to its quota. The problem was presented as the college admissions problem and solved by Gale and Shapley in the same paper as the stable marriages problem, with essentially the same algorithm, the chief difference being that colleges (hospitals) can hold multiple applications, up to their quota. In this setting, we do not require that every applicant is assigned to a college or that every college meets its quota, and we also allow the possibility that a college may prefer not to accept a particular applicant even when its quota is not met or that an applicant may prefer not to apply to a particular college even if it would mean not being accepted anywhere.

In the applicant-optimal procedure, when a college receives a new application while its currently held applicants would meet its quota, it rejects the worst application from the new one and the ones it already held.

In the college-optimal procedure, colleges make offers to applicants which have not rejected them up until they have enough offers held to fill their quotas.

Definition 10.2.1 *An **instance** of the college admission problem consists of a set A of n applicants, a set C of m colleges with a quota $q_c \geq 1$ for each $c \in C$ and the (possibly incomplete) preference lists for all $a \in A, c \in C$.*

Definition 10.2.2 *An* ***assignment*** *is a set $\mathcal{A} \subseteq A \times C$ such that each applicant $a \in A$ appears in at most one pair, and each college $c \in C$ appears in at most q_c pairs.*

Definition 10.2.3 *A* ***blocking pair*** *$(a, c) \in A \times C$ for the assignment $\mathcal{A}$ is a pair such that a and c are in each others' preference lists and one of the following holds:*

(i) c's quota is met, a prefers c to his current college and c prefers a to at least one of its applicants;

(ii) c's quota is not met and a prefers c to his current college.

Definition 10.2.4 *An assignment $\mathcal{A}$ is* ***stable*** *if there is no blocking pair for it in $A \times C$.*

It's clear that if we had a blocking pair (a, c), then in case (i), c would like to reject its worst student in order to accept a, and in case (ii), c would simply accept a because it has the room to do so.

Lemma 10.1.10 generalizes to both the applicant- and college-optimal algorithms, and it follows from it that the assignment is stable and, in fact, applicant- or college-optimal, respectively.

For the applicant-optimal procedure, we can represent the list of applicants to a college using a heap data structure with keys as the applicant ranking for that college, since we only need access to the worst applicant on the list when a college must reject one. Then this lookup is $O(1)$ and insertion and deletion are $O(\log(q_c))$. For simplicity's sake, we assume that $\forall a \in A, c \in C, a \in L(c) \Rightarrow c \in L(a)$ to avoid having each college rank every applicant, even those which it would never accept. Otherwise, when $a \in A$ applies to $c \in C$, we must check if $a \in L(c)$ and we'd like to do this in constant time (in particular, without having to traverse $L(c)$). For $q :=$ the largest quota greater than or equal to n (or 2 if none, since we only need to reject applications if a college's quota is met), the running time will be $O(\log(q)(n + \Sigma_{a \in A}|L(a)|)) \subseteq O(\log(q)nm)$.

For the college-optimal procedure, there is no need to store each college's list of applicants; we need only know how many offers each college has made. Here, we assume that $\forall a \in A, c \in C, c \in L(a) \Rightarrow a \in L(c)$ to avoid having each applicant rank every college, even those to which they would never apply. The college-optimal algorithm's running time will be $O(m + \Sigma_{c \in C}|L(c)|) \subseteq O(n + m + \Sigma_{a \in A}|L(a)| + \Sigma_{c \in C}|L(c)|) \subseteq O(nm)$, where the second of these three is linear.

10.3 Stable Roommates

(This section is work in progress!) The stable roommates problem also generalizes stable marriages: instead of separate sets of men and women, there is only a single set of n individuals, each with a list of the other $n - 1$ individuals sorted by preference. Irving [50] gave an $O(n^2)$-time algorithm to find a stable matching if one exists, and reporting that none exists, otherwise.

Definition 10.3.1 *An* ***instance*** *of the stable roommates problem is a set of X of individuals, with size $n \in \mathbb{N}$, and for each $x \in X$, a preference list $L(x)$, that is a permutation of the remaining individuals in X. We may assume $X = \{1, 2, \dots, n\}$. Again, we say x* ***prefers*** *y* ***to*** *z, if y appears before z in $L(x)$. Preference lists (not necessarily the original ones) together are called a (preference)* ***table****, and a table T' is a* ***subtable*** *of a table T, if $x \in X$ is in $y \in X$'s list in T' implies the same in T and x comes before z on y's list in T' implies the same in T. A* ***table for the instance*** *is a subtable of the original table. Since the instance is understood to be fixed, we will usually just refer to a table for the instance just as table.*

Definition 10.3.2 *A **matching** for an instance of the stable roommates problem is a bijection $\mathcal{M} : X \to X$ such that $\mathcal{M}(x) \neq x$ for all $x \in X$ and $\mathcal{M}(x) = y$ iff $\mathcal{M}(y) = x$ (or $\mathcal{M}(\mathcal{M}(x)) = x$ for all $x \in X$, i.e. $\mathcal{M}$ is its own inverse).*

Note that $\mathcal{M} = \{(x, \mathcal{M}(x)) \in X \times X | x \in X\}$.

It follows from this definition that n must be even for a matching to exist, since we do not allow $x \in X$ to be alone ($\mathcal{M}$'s domain is X) or matched with himself ($\mathcal{M}(x) \neq x$).

Equivalently, we can define a matching as a set of unordered pairs in $\mathcal{P}(X)$ (the power set of X) where each individual appears in exactly one pair. In this case, any matching must have size $n/2$.

Blocking pair and **stable matching/solution** and **possible pairing** (**possible partner**) are defined again as in Definitions 10.1.3, 10.1.4 and 10.1.5.

Any stable matching for a stable marriage instance will be a solution to the stable roommates instance constructed by adding to the end of each man's preference list the remaining men and doing the same for the women. Like the stable marriage problem, then, a stable roommates instance may have multiple solutions. Unlike the stable marriage problem, however, not every instance of the stable roommates problem has a solution, e.g.

1 [2 3 4]
2 [3 1 4]
3 [1 2 4]
4 [1 2 3]

There are three matchings, but each has a blocking pair:

{(1,2), (3,4)} is blocked by (2,3); {(1,3), (2,4)}, by (1,2); and {(1,4), (2,3)}, by (1,3).

Note that by the symmetry of the instance, 4's list can actually be an arbitrary permutation of {1, 2, 3}, and the same will still hold.

Irving's algorithm is split up into two phases, the first of which is similar to the stable marriage algorithm, but we now allow anyone to propose or hold proposals (and even both). In pseudocode:

Phase 1

```
while there remains $x \in X$ who has no proposal held and has not proposed to everyone, do
    $next$ := the next on $x$'s list to whom $x$ has not yet proposed
    $x$ proposes to $next$
    if $next$ holds no proposal, then
        $next$ holds $x$'s proposal
    else $y$ := the individual from whom $next$ holds a proposal
        if $next$ prefers $y$ to $x$, then $next$ rejects $proposer$
        else
            $next$ rejects $y$
            $next$ holds $x$'s proposal
```

In words, we let those who have no proposal held to propose to the individual they prefer most of those to whom they've not already proposed, if one exists, but if there is no such individual left, then we stop the procedure.

We assume again that we have a ranking matrix for the individuals as we described in Lemma 10.1.7, and if not, we can construct one in $O(n)$ time. Then, we can use a linked list to implement the set of individuals who have no proposal held, so that insertion and deletion are $O(1)$, resulting in $O(n)$ preprocessing time to copy X. Alternatively, we can traverse X letting *proposer* be each individual in turn with a nested while-loop inside that updates *proposer* to y if y is rejected by *next*, so that rejected individuals have priority in proposing. In either case, the operations inside the loop(s) are $O(1)$ all together and each iteration corresponds to a single proposal (except possibly the last, if there is no stable marriage), of which most $(n-1)^2$, so the procedure takes $O(n^2)$ time.

Results corresponding to Lemma 10.1.8, Lemma 10.1.10, Corollary 10.1.11, which were adapted from Irving's original paper, hold again in this setting with nearly identical proofs (so they are omitted here):

Lemma 10.3.3 *After the moment $x \in X$ is first proposed to, x always holds a proposal.*

Lemma 10.3.4 *If $y \in X$ rejects $x \in X$ in Phase 1, then x and y cannot be partners in any stable matching.*

Corollary 10.3.5 *If during Phase 1, $x \in X$ proposed to $y \in X$, then, in any stable matching*

(i) x cannot have a better partner than y;

(ii) y cannot have a worse partner than x.

Corollary 10.3.6 *If after Phase 1, $x \in X$ has no proposal held and has proposed to everyone on his list, then no stable matching exists.*

Proof. x was therefore rejected by everyone, so by the Lemma 10.3.4, cannot be partners with anyone in a stable matching. ■

Then we would exit reporting so, as we should, but after we find that an individual's list becomes empty after the following reductions to the original table:

Corollary 10.3.7 *If, after Phase 1, $y \in X$ holds a proposal from $x \in X$, then y's preference list can be reduced (without eliminating possible partners) by deleting from it*

(i) all those to whom y prefers x (i.e. all those after x);

(ii) all those who hold a proposal from a person whom they prefer to y (including those who have rejected y, i.e. all those before x);

In the resulting table,

(iii) y is first on x's list; and x, last on y's;

(iv) in general, b appears on a's list if and only if a appears on b's.

Proof.(i) and (ii) follow from Corollary 3.4.1.(ii); and (iii), from (i) and the parenthesized part of (ii). For (iv), suppose a holds a proposal from c; and b, from d. Then a is on b's list $\iff$ b does not prefer d to a (by (i)), and a does not prefer c to b (by (ii)) $\iff$ a is on b's list (by (ii) and (i)). ■

We will refer to this table as the Phase 1 table. In fact, for a particular instance, the Phase 1 table is always the same. When no list is empty, the resulting table is an example of a *stable table*:

Definition 10.3.8 *A table T for an instance is **stable** if, in T:*

(i) x is first in y's list, abbreviated $f_T(x) = y$, iff y is last in x's list, abbreviated $l_T(y) = x$;

(ii) for $x, y \in X$, x is not in y's list iff y is not in x's iff x prefers $l_T(x)$ to y or y prefers $l_T(y)$ to x (in the original table);

(iii) no list is empty.

(i) and (iii) imply that f_T and l_T are inverse functions $X \to X$, and so must both be bijective, i.e. injective and surjective. Note that once all of the $f_T(x)$ are determined (or all of the $l_T(x)$ are), then the whole table is determined by (ii). It should be clear that any table can be made so that (i) and (ii) hold by reducing the lists, although a list may become empty in doing so. In fact, any table for which (i) and (ii) hold is a subtable of the Phase 1 table, and the reductions to obtain the Phase 1 table are exactly those necessary to establish (i) and (ii):

Lemma 10.3.9 *Definition 10.3.8 (i) and (ii) hold for the Phase 1 table, and so if no list is empty, the table is stable.*

Furthermore,

Lemma 10.3.10 *If each list in a stable table T contains exactly one person, then they specify a stable matching.*

Proof. By Definition 10.3.8(i), the lists specify a matching. Suppose that $(x, y) \in X \times X$ but x and y are not paired in the matching. Then x and y are not in each others' lists in T. Then, by Definition 10.3.8(ii), x prefers the last (i.e. sole) person on his list to y, or y, the last (i.e. sole) person on his list, to x, so that (x, y) cannot be a blocking pair. The matching is therefore stable. ■

Then, using Lemma 10.3.9, the following holds:

Corollary 10.3.11 *If each list in the Phase 1 table contains exactly one person, then they specify a stable matching.*

Before we include Phase 2 of the algorithm in pseudocode, we describe and justify it.

We prove that at each step until termination, Definitions 10.3.8(i) and (ii) hold. The base case is established by Lemma 10.3.9. Hence, if the reduced lists contain exactly one person each, then we are done by Lemma 10.3.10 (Corollary 10.3.11). Otherwise, at least one is empty, in which case, we will see that there is no stable matching so that we end the procedure, or at least one has more than one person in it. In this last case, we will search for a *rotation* (an "all-or-nothing cycle" in Irving's original paper) and *eliminate* it: we reduce the preference lists again with respect to this rotation in such a way that Definitions 10.3.8(i) and (ii) continue to hold. If any list becomes empty, we exit reporting that no stable matching exists. Otherwise, the inductive step holds.

Definition 10.3.12 *A **rotation** in a stable table is a cyclic sequence $(a_1, b_1), (a_2.b_2), \ldots, (a_r, b_r) \in X \times X$ of pairs of individuals such that, where $a_{r+1} = a_1$ and $b_{r+1} = b_1$:*

(i) the a_i's are distinct;

(ii) b_i is first on a_i's list ($f_T(a_i) = b_i$) for each i; equivalently, a_i is last on b_i's list for each i ($l_T(b_i) = a_i$);

(iii) b_{i+1} is the second on a_i's list for each i.

First, note that a_i is in b_{i+1}'s list for each i.

Then, a rotation can be depicted in the following way with respect to the reduced preference lists in the stable table:

$$\begin{array}{rlrl}
a_1 & [\, b_1 \quad b_2 \ldots] & b_1[& \ldots \quad a_r \ldots \quad a_1] \\
a_2 & [\, b_2 \quad b_3 \ldots] & b_2[& \ldots \quad a_1 \ldots \quad a_2] \\
a_3 & [\, b_3 \quad b_4 \ldots] & b_3[& \ldots \quad a_2 \ldots \quad a_3] \\
\vdots & & \vdots & \\
a_i & [\, b_i \quad b_{i+1} \ldots] & b_i[& \ldots a_{i-1} \ldots \quad a_i] \\
\vdots & & \vdots & \\
a_{r-1} & [\, b_{r-1} \quad b_r \ldots] & b_{r-1}[& \ldots a_{r-2} \ldots a_{r-1}] \\
a_r & [\, b_r \quad b_1 \ldots] & b_i[& \ldots a_{r-1} \ldots \quad a_r]
\end{array}$$

We observe that a rotation is completely determined once one a_i or b_i is known: once a_i is known, (iii) gives us b_{i+1} and (ii), a_{i+1}; once b_i is known, (ii) gives us a_i and we repeat the previous argument. It then follows that since the a_i are all distinct, so must be the b_i.

Also, note that the length of the rotation is $r \geq 2$, since otherwise $b_1 = b_2$, so that $b_1 = b_2$ is both first and second on a_1's list, a contradiction.

Lemma 10.3.13 *A stable table in which a's list has length at least 2 contains a rotation.*

Proof. To find a rotation, we let $p_1 \in X$ be an individual whose list has length at least 2, and define, inductively:

q_{i+1} = the second in p_i's reduced list

p_{i+1} = the last in q_{i+1}'s reduced list (so q_{i+1} is first in p_{i+1}'s)

until this sequence repeats some p_s, so that $p_{s+r} = p_s$. It follows that each p_i will have at least 2 individuals on his reduced list, for if p_i has only q_i on his list and i is the least such that this occurs, then q_i must have only p_i on his list by Definition 10.3.8(i), because q_i is both the first and the last on p_i's list. By uniqueness, it follows that $p_i = p_{i-1}$, contradicting i being the least. Hence, we can, in fact, choose q_{i+1} at each step.

Then, we let $a_i = p_{s+i-1}$ for $i = 1, \ldots, r$ and $b_i = q_{s+i-1}$, for $i = 1, \ldots, r$ to give us a rotation, and we call $p_1, \ldots, p_{s-1}$ the **tail** for the rotation. ■

Tails are not generally unique for a rotation, since instead of starting our search from p_1, starting from any other p_j would have also lead to the same rotation. Also, tails may be empty, as is the case when $p_1 = a_1$.

Then, having found a rotation, we *eliminate* the rotation: we force each b_i to reject a_i and have each a_i propose to b_{i+1}, reducing the preference lists again to satisfy Definition 10.3.8 (i) and (ii) and proving the inductive step in our reductions. Hence $x \in X$ and $y \in X$ are removed from each other's lists in the rotation elimination if (and only if) one of them is b_i for some i and the other succeeds a_{i-1} in b_i's list. The result for the cycle is that, for each i:

a_i [b_i $b_{i+1} \ldots$] becomes a_i [$b_{i+1} \ldots$], and

b_i [$\ldots a_{i-1} \ldots a_i$] becomes b_i [$\ldots a_{i-1}$].

Additionally, we remove b_i from the lists of the successors of a_{i-1} in b_i's list, so that the removals remain symmetric. The proof for the following lemma is incomplete.

Lemma 10.3.14 *If there exists a rotation $(a_1, b_1), \ldots, (a_r, b_r)$ in a stable table T and the table resulting from the elimination of this rotation, T', has no empty list, then T' is a stable subtable of T':*

(i) $f_{T'}(a_i) = b_{i+1}$ *for each i;*

(ii) $l_{T'}(b_i) = a_{i-1}$ *for each i;*

(iii) $f_{T'}(x) = f_T(x)$ *for $x \in X, x \neq a_i$ for each i, and* $l_{T'}(x) = l_T(x)$ *for $x \in X, x \neq b_i$ for each i;*

(iv) if $p_1, \ldots, p_{s-1}$ is a tail (as described in Lemma 10.3.13 for the rotation and the 2nd on p_j's list changes after the rotation elimination, then either p_j is matched in T' or $j = s - 1$ (i.e. the last on the tail), but not both.

Proof. Stability of T' and (i)-(iii) follow from the way we eliminate rotations.

For (iv), suppose p_j's 2nd value changed from $q_{j+1} \in X$ after the rotation elimination. Then this occurred either because $f_T(p_j)$ was removed from p_j's list and vice-versa (so $f_{T'}(p_j) = q_{j+1}$ or q_{j+1} was also removed) or p_j and q_{j+1} were removed from each other's lists, so that one of them is b_i for some i and the other succeeded a_{i-1} in b_i's list before the rotation was eliminated.

If $f_T(p_j)$ was removed from p_j's, then by (i) and (iii), it follows that $p_j = a_i$ for some i, contrary to p_j being in the tail. Hence, $p_j = b_i$ or $q_{j+1} = b_i$ for some i.

If $p_j = b_i$, then p_j prefers a_{i-1} to q_{j+1}, where both were on his list but q_{j+1} was 2nd, so that a_{i-1} was first, i.e. $f_T(p_j) = a_{i-1}$, and since $l_{T'}(p_j) = l_{T'}(b_i) = a_{i-1}$, a_{i-1} is the only individual remaining on p_j's list and vice-versa by Definition 10.3.8(i), so that $p_j = b_i$ and a_{i-1} are matched.

Otherwise $q_{j+1} = b_i$, i.e. the second on p_j's list was b_i and so p_{j+1} was last on b_i's list, i.e. $p_{j+1} = a_i$. Then p_j was not among the a_k but p_{j+1} was, so that p_{j+1} must have been the first, i.e. $p_{j+1} = a_1$ and so $j = s - 1$ (and $i = 1$).

Why not both? This is unclear!

■

The following lemma, whose proof we leave as an exercise, justifies rotation elimination.

Lemma 10.3.15 *Let $(a_1, b_1), \ldots, (a_r, b_r)$ be a rotation in a stable table T. Then,*

(i) in any stable matching contained in T, either a_i and b_i are partners for all values of i or for no value of i;

(ii) if there is such a stable matching in T in which a_j and b_j are partners, then modifying it so that a_i and b_{i+1} are instead partners for each i also gives a stable matching.

Then, the following two results follow immediately, by induction on the Phase 2 steps outlined earlier:

Corollary 10.3.16 *If the original problem instance admits a stable matching, then there is a stable matching contained in any of the tables in the sequence of reductions.*

Corollary 10.3.17 *If one or more among the lists in one of the tables in the sequence is empty, then the original problem instance admits no stable matching.*

With that above two corollaries, we've proven the correctness of the procedure, whose remainder we now include:

Phase 2

```
reduce the preference lists as described in Corollary 10.3.7
if any list becomes empty during the reductions, then
    exit the procedure reporting that there is no stable matching
if, after the reductions, each list has a unique individual in it, then
    return $\mathcal{M} := \{(x, y) \in X \times X \mid y$ is in $x$'s reduced list$\}$
while there remains an individual in $X$ with at least 2 in his reduced list, do
    find a rotation using the tail of the previous one (if any)
    eliminate the rotation
    if any list becomes empty during the reductions, then
        exit the procedure reporting that there is no stable matching
return $\mathcal{M} := \{(x, y) \in X \times X \mid y$ is in $x$'s reduced list$\}$
```

The only issue that remains to implement Phase 2 efficiently.

In the first reduction after Phase 1, we keep track of the first individual in X whose reduced list has size at least 2, called $first\_unmatched$.

When Phase 2 starts, to find rotations, we first store an array of length n with entry i indicating whether we've 'visited' $i \in X$ in our search for a rotation. We of course initialize each entry to 'unvisited'. Then, at each step in which we search for a rotation, we store a linked list of visited individuals, the p_i, for $i = 1, \ldots, s + r - 1$, and mark them as visited in the array until we find one that's already been visited, starting with $p_1 = first\_unmatched$ for the first rotation. We eliminate the rotation we've found and then 'unvisit' all of the a_i and remove them from our list.

If there is no tail from the previous rotation, we start at $first\_unmatched$ (which may have to be updated beforehand).

If there is a tail, we start our search from p_{s-1}, the last individual in the tail, to avoid repeating the sequence $p_1, \ldots p_{s-1}$. This is justified by Lemma 10.3.14, since we have two possibilities. The first is that our new search starting from p_1 will lead back to p_{s-1} as the 2nd individuals on the p_j's lists up to but not necessarily including $j = s - 1$ (by (iv)) as well as the last individuals on the q_js' lists (up until the rotation) remained the same (by (iii)). The second possibility is that once p_j is repeated for some $j > s - 1$ in our new search, it will follow that each p_k, for $k \geq j$, will have at least 2 on his list by induction (in the same way we guaranteed each p_i had at least two on his list for each i to prove Lemma 10.3.13, so that those p_k are still unmatched and hence the 2nd on their lists remained the same after eliminating the rotation (by (iv)).

Furthermore, reducing the preference lists can be done implicitly by storing the indices of the first, second and last person on an individual's list that should remain in the reduced list and updating these as necessary. We store arrays for these, where $f[x]$, $s[x]$ and $l[x]$ are the corresponding values for x. Then, with the ranking matrix, each removal can then be done in constant time, since it just means comparing the rank of an individual to be removed with the indices of the first and last and updating these appropriately. We only update the value of $s[x]$ during the search for a cycle, and the new value can be found by traversing x's list from $s[x]$ until we find some p who prefers x to $l[p]$, i.e. p and x are in each others' reduced lists, and we update $s[x] := p$.

Theorem 10.3.18 *The running time of Irving's algorithm is $O(n^2)$.*

Proof. Phase 1 takes $O(n^2)$ time. Since each removal takes constant time and there are at most $n(n-2)+1$ removals (n for each individual, $n-2$ to reduce an individual's list to a single individual and the $+1$ in the case that one list becomes empty and we stop there), so all removals together in all reductions require $O(n^2)$ time. Checking whether a stable matching is found and constructing it also takes $O(n^2)$ time, as we only scan the list X and pair each $x \in X$ with $f[x] = l[x]$.

$first\_unmatched$ is updated at most n times and $s[x]$, at most $n-1$ times for each x, corresponding to traversing the lists from left to right at most once, so together, this is again $O(n^2)$.

All that remains is to show that the time spent finding rotations is also $O(n^2)$:

If m rotations are found and eliminated during the entire procedure, we let t_i and r_i denote the lengths of the tail and the rotation for the ith rotation. For $i = 1, \dots m$, finding the ith rotation took at most $c + d(r_i + t_i - t_{i-1}) + er_i$ operations, for some constants, $c, d, e > 0$, since $r_i + t_i$ is the number of individuals in the list of p_j, but the first t_{i-1} were already visited in the search for the $i-1$th, so we do not revisit them, and we only visit $r_i + t_i - t_{i-1}$ new individuals at this step (where $t_0 = 0$, hence $d(r_i + t_i - t_{i-1})$. Also, after the search, we unvisit the a_j of the cycle, hence er_i.

Note also that since the elimination of the ith rotation means at least $2r_i$ removals from the preference lists, and $r_i \geq 2$ for each i, so that

$$4m \leq 2\Sigma_{i=1}^{m} r_i \leq n^2$$

Then, taking the sum over all the rotations, we get at most

$$\begin{aligned}\Sigma_{i=1}^{m}[c + d(r_i + t_i - t_{i-1}) + er_i] &= cm + (d+e)\Sigma_{i=1}^{m} r_i + d(t_m - t_0) \\ &\leq \frac{c}{2}n^2/4 + \frac{d+e}{2}n^2 + dn,\end{aligned}$$

which is $O(n^2)$. ■

10.4 Exercises

10.1 *Prove Lemma 10.3.14. Hint: for (ii), first consider the case where $a_i = b_j$ for some i, j and use (i).*

10.2 Geometric Stable Roommates. *Consider the stable roommate problem where the individuals are elements of the real numbers,* $\mathbb{R}$ *and* x *prefers* y *to* z *iff* $|x-y| < |x-z|$, *i.e. the norm replaces the preference lists, and assume there are no ties. Describe an algorithm to find a stable matching in* $\Theta(nlog(n))$*-time.*
(Hint: use dynamic binary search trees.)
What about the vector space $\mathbb{R}^d$ *for* $d \in \mathbb{Z}^+$, *where* $x = (x_1, x_2, \ldots, x_d)$ *and* $|x| = \sqrt{(x_1)^2 + \cdots + (x_d)^2}$*?*

10.3 Median Matching. *Given a stable roommates instance with a solution, let* $\mathcal{S}$ *be a subset of all the stable matchings such that* $|\mathcal{S}|$ *is odd. Show that* $\mathcal{M}$, *defined by* $\mathcal{M}(x) :=$ *the* x*'s median partner in all matchings in* $\mathcal{S}$, *is a stable matching. Hint: first consider the case where* $\mathcal{S}$ *contains only 3 stable matchings.*

Chapter 11

Additional Exercises

11.1 Random Problems

1. Construct the string matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaabaab$.

2. Assume that the decision version of the 3-SAT, Vertex Cover and Clique problems are NP-complete. An independent set of an undirected graph $G = (V, E)$ is a subset I of V such that no two vertices in I are connected by an edge in E. The decision version of the independent set problem (G, k), $k \geq 0$, is to determine whether G has an independent set of size at least k. Prove that the Independent Set problem is NP-Complete.

3. Let $G = (V, E)$ be an undirected connected simple graph. A matching is a set of edges of G such that no two edges in the set are incident on the same vertex. A matching is **maximal** if it is not a proper subset of any other matching. A matching is **maximum** if the number of edges in the matching is largest. A vertex cover of $G = (V, E)$ is a set of vertices $V' \subseteq C$ such that if $(u, v) \in E$, then either $u \in V'$ or $v \in V'$ or both in V'. The size of the vertex cover is the cardinality of the set V'.

   (a) Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover of G.

   (b) Consider a maximal matching M in $G = (V, E)$. Let

   $$T = \{v \in V : \text{some edge in M is incident on v}\}.$$

   What can you say about the subgraph of G induced by the vertices of G that are not in T. Conclude that $2|M|$ is the size of a vertex cover for G.

   (c) Present an $O(|E|)$ time (greedy) algorithm to compute maximal matching.

   (d) Conclude that the above algorithm for computing maximal matching is a 2-approximation algorithm for maximum matching.

   Consider the Proposal Algorithm that we discussed in the class. Its outlined below:
   Input: A list of n men and n women, where each man has an ordered list of n women (a permutation) whom he will like to marry. Similarly, each woman has an ordered list (a permutation) of n men whom she will like to marry.

Output: A set of n pairs forming a stable perfect matching. Each pair consists of a man and a woman.

```
Proposal-Algorithm (M,W)
1. While there exists an unmarried man m do
2.     m proposes to the most preferred woman w on his list
         whom he has not proposed so far.
3.     if w is not married then w marries m (end if)
4.     if w is married to m' but prefers m over m' then
5.        w divorces m' and marries m
       (end if)
   (endwhile)
```

(a) Prove or Disprove: The output of this algorithm is always the same and is independent of in which order the men are picked up in Step 1. In other words the above algorithm produces the same set of stable matchings and is independent of the choice of men m in Line 1.

(b) Show that this algorithm is favorable to men compared to women. One possible way to show this is to construct example(s) where women do much better than men when the same algorithm is run by interchanging men with women.

(c) List all the data structures (including the operations) that you will use to implement the Proposal Algorithm. (e.g. Stacks/Arrays/...).

(d) State the complexity of the Proposal Algorithm using $O()$ notation (Remember to use your data structures and their operations from Problem 3). (Recall that there are at most $n * n$ iterations in this algorithm, since a man never proposes to the same woman twice and in each iteration there is a proposal made)

(e) Can you devise a proposal algorithm that is unbiased?

4. Given a binary tree with all the relevant pointers (child/parent), describe a simple algorithm, running in linear time, that can compute and store the size of the subtrees at each node in the tree. (Size of a subtree at a node v is the total number of nodes, including itself, in the subtree rooted at v.) Justify why your algorithm is correct and analyze it and show that it runs in linear time.

5. Outline a search algorithm, running in $O(\log n)$ time, to report the i-th smallest number in a set consisting of n elements. The set is represented as a red-black tree where in addition to the usual information that we store at a node (pointer to left child, right child, parent, key, colour) we also store the size of the subtree at that node. The parameter i is supplied to the search algorithm at the run-time and assume that the red-black tree with the additional information about the size of the subtrees has been precomputed.

6. Assume that we have n-integers in the range 1000 to 9999. In the radix-sort method we sorted them by first sorting them using the (stable) counting sort by the Least-Significant digit and then the next least significant digit and so on. Why does this algorithm works? Sketch the main idea in the proof. Why does this algorithm fails when we first sort them using most-significant bit and then the second most significant bit and so on?

7. Assume that we are given n intervals (possibly overlapping) on a line. Each interval is specified by its left and right end points. Devise an efficient algorithm that can find a point on the line which is contained in the maximum number of intervals. Your algorithm should run in $O(n^2)$ time. (Bonus Problem: Devise an algorithm that runs in $O(n \log n)$ time? Another Bonus Problem: Show that this problem has $\Omega(n \log n)$ as its lower bound.)

8. Devise an $O(n \log k)$ time algorithm to merge k-sorted lists into a single sorted list, where n is the total number of elements in all input lists.

9. Suppose you are given n-points in the plane. We can define a complete graph on these points, where the weight of an edge $e = (u, v)$, is Euclidean distance between u and v. We need to partition these points into k non-empty clusters, for some $n > k > 0$. The property that this clustering should satisfy is that the minimum distance between any two clusters is maximized. (The distance between two clusters A and B is defined to be the minimum among the distances between pair of points, where one point is from cluster A and the other from cluster B.) Show that the connected components obtained after running Kruskal's algorithm till it finds all but the last $k-1$ (most expensive) edges of MST produces an optimal clustering.

10. Although the 3CNF-SAT is NP-Complete, show that in polynomial time we can determine whether a boolean formula given in disjunctive normal form is satisfiable; the formula consists of n variables and k clauses. A formula is in Disjunctive normal form, if clauses are joined by ORs and literals within a clause are joined by ANDs (DNF is OR of ANDs and CNF is AND of ORs!). You need to provide an algorithm and show that its correct and its running time is polynomial in n and k.

11. Although the 3CNF-SAT is NP-Complete, show that 2CNF-SAT is solvable in polynomial time. (This is same as exercise 34.4-7 in Book and contains Hints!).

12. Given an integer $m \times n$ matrix A and an integer m-vector b, the 0-1 integer programming problem asks whether there is an integer n-vector x with elements in the set {0,1} such that $Ax \leq b$. Prove that 0-1 integer programming problem is NP-Complete by providing a reduction from 3CNF-SAT or Subset-Sum problem.

13. Construct an instance of the subset-sum problem corresponding to the following 3CNF-SAT:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Provide a satisfying assignment to 3CNF-SAT and show that it provides a valid solution for the subset-sum problem.

14. For the same 3CNF-SAT in Problem 4, illustrate the reduction to the clique problem. Construct an equivalent graph and show that for a satisfying assignment you have an appropriate size clique and vice verse (Use colours to illustrate the clique!)

15. Consider the problem of CNF-Satisfiability (we are not putting any restriction on number of literals in each clause), where each variable only occurs at most twice (positive and negative form of a variable are not counted separately). Show that satisfiability can be determined in polynomial time in this case.

Hint: If a variable only occurs in its positive (or negative) form, then we can just satisfy those clauses and remove them from consideration. The problem starts when we have both version of the variable (one clause containing it in positive form and the other in the negative form - but then the following resolution rule applies: if we have two clauses of the form $(x \vee w \vee y \vee z) \wedge (\neg x \vee u \vee y \vee z)$, then this is satisfiable if and only if the clause $(y \vee z \vee u \vee w)$ is satisfiable. In other words we can remove the variable x from consideration!)

16. You have invited 500 guests for your graduation party in a huge hall in Chateau Laurier. The guests needs to be seated, where each table has 20 seats. Unfortunately, the guests are not all friendly with each other; for sure you do not want two of your guests to sit on the same table if they are not friendly. Suppose you know the complete friendship matrix F (a 0-1 matrix indicating whether a pair of guests (i, j) are friendly or not). Can you devise a decision algorithm to decide whether it is possible to hold this party with the restriction of 20 per table and no two enemies land up on the same table! What about in general, where n is the number of guests and k is the number of guests per table and we can assume k divides n? Is this problem NP-Complete?

17. Consider two sets A and B, each having n integers in the range from 0 to cn, where $c > 1$ is a constant. Define the Cartesian sum of A and B as the set C given by $C = \{x + y : x \in A \text{ and } y \in B\}$. Note that the integers in C are in the range 0 to $2cn$. We want to find all the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Provide an $O(n \log n)$ algorithm for this problem.

18. Given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. For any vertex cover $V' \subseteq V$, define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight. Provide an Integer Linear Programming formulation for this problem. Then provide a relaxation of the integer program. Show that using the rounding techniques we can obtain a 2-approximation algorithm for this problem. Provide a formal proof that your solution is a 2-approximation.

19. Given a simple graph $G = (V, E)$, we define a cut to be a partition of the vertex set V into two non-empty sets A and B, where $A \cup B = V$ and $A \cap B = \emptyset$. An edge $(a, b) \in E$ is said to cross the cut if $a \in A$ and $b \in B$. The size of the cut corresponding to the partition (A, B) is defined to be the number of edges crossing the cut. The maximum cut problem is to find a partition of V such that the size of the cut is maximized. Consider the following algorithm:
Step 1: Find any partition of V.
Step 2: For every vertex $v \in V$, if v would have more edges crossing the cut if placed in the opposite partition, then move v to the opposite partition.
Prove the following
    (a) Prove that the above algorithm runs in polynomial time. What is the running time?
    (b) Prove that the size of the cut produced by the above algorithm is at least half of the size of the maximum cut. (In other words its an 1/2-approximation algorithm.)

20. Although the 3CNF-SAT is $\mathcal{NP}$-Complete, show that in polynomial time we can determine whether a boolean formula given in disjunctive normal form is satisfiable. The formula consists of n variables and k clauses. (A formula is in Disjunctive normal form, if clauses are joined by ORs and literals within a clause are joined by ANDs). You need to provide an

algorithm whose running time is polynomial in n and k.

Show what is wrong with the following argument: Given a 3CNF-SAT, we can use distributive law to construct an equivalent formula in Disjunctive Normal Form. Here is an example: $(x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2}) = (x_1 \wedge \bar{x_1}) \vee (x_1 \wedge \bar{x_2}) \vee (x_2 \wedge \bar{x_1}) \vee (x_2 \wedge \bar{x_2}) \vee (\bar{x_3} \wedge \bar{x_1}) \vee (\bar{x_3} \wedge \bar{x_2})$. We have just now shown that DNF is in $\mathcal{P}$. This implies that 3CNF-SAT is in $\mathcal{P}$ and $\mathcal{P} = \mathcal{NP}$.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] L. Aleksandrov and H. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM J. Discret. Math.*, 9(1):129–150, February 1996.

[3] Lyudmil Aleksandrov, Hristo Djidjev, Hua Guo, and Anil Maheshwari. Partitioning planar graphs with costs and weights. *J. Exp. Algorithmics*, 11, February 2007.

[4] N. Alon and J. H. Spencer. *The probabilistic method.* John Wiley & Sons, 3rd edition, 2007.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pages 459–468, 2006.

[6] Martin J. B. Appel and Ralph P. Russo. The connectivity of a graph on uniform points on $[0,1]^d$. *Statistics & Probability Letters*, 60(4):351–357, 2002.

[7] P. Balister, A. Sarkar, and B. Bollobás. Percolation, connectivity, coverage and colouring of random geometric graphs. In *Handbook of Large-Scale Random Networks*, pages 117–142. Springer, 2008.

[8] Ahmad Biniaz, Evangelos Kranakis, Anil Maheshwari, and Michiel Smid. Plane and planarity thresholds for random geometric graphs. In *Proc. ALGOSENSORS 2015 (Patras, Greece)*, Lecture Notes in Computer Science, Berlin, Germany, 2015. Springer.

[9] B. Bollobás. *Random graphs.* Cambridge University Press, 2001.

[10] Béla Bollobás and Andrew Thomason. Threshold functions. *Combinatorica*, 7(1):35–38, 1987.

[11] Bla Bollobs. *Modern Graph Theory.* Graduate texts in mathematics. Springer, Heidelberg, corrected edition, 1998.

[12] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* Elsevier, New York, 1976.

[13] Otakar Borvka. O jistém problému minimálním. *Prce mor. prodovd. spol. v Brn III*, 3:37–58, 1926.

[14] Otakar Borvka. Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnickỳ obzor*, 15:153–154, 1926.

[15] J. Bourgain. On lipschitz embedding of finite metric spaces in hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.

[16] Jean Bourgain and Gil Kalai. Threshold intervals under group symmetries. *Convex Geometric Analysis MSRI Publications Volume 34*, pages 59–63, 1998.

[17] Milan Bradonjić and Will Perkins. On sharp thresholds in random geometric graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, pages 500–514, 2014.

[18] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:327–336, 1998.

[19] A.Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29, 1997.

[20] Timothy M. Chan. Backwards analysis of the Karger-Klein-Tarjan algorithm for minimum spanning. *Inf. Process. Lett.*, 67(6):303–304, 1998.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[23] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Struct. Algorithms*, 22(1):60–65, 2003.

[24] Mayur Datar and Piotr Indyk. Locality-sensitive hashing scheme based on p-stable distributions. In *In SCG 04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM Press, 2004.

[25] L.B. de Paula, R.S. Villaca, and M.F. Magalhaes. A locality sensitive hashing approach for conceptual classification. In *Semantic Computing (ICSC), 2010 IEEE Fourth International Conference on*, pages 408–413, 2010.

[26] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[27] B. Dixon, M. Rauch, and R. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.

[28] D.P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms.* Cambridge University Press, 2009.

[29] P. Erdös and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

[30] Shimon Even. *Graph Algorithms.* W. H. Freeman & Co., New York, NY, USA, 1979.

[31] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[32] Greg N. Frederickson. Optimal algorithms for tree partitioning. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 168–177, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[33] E. Friedgut and G. Kalai. Every monotone graph property has a sharp threshold. *Proceedings of the American Mathematical Society*, 124(10):2993–3002, 1996.

[34] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1972.

[35] E. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, pages 1141–1144, 1959.

[36] E. Gilbert. Random plane networks. *Journal of the Society for Industrial & Applied Mathematics*, 9(4):533–543, 1961.

[37] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. pages 518–529, 1997.

[38] Erhard Godehardt and Jerzy Jaworski. On the connectivity of a random interval graph. *Random Struct. Algorithms*, 9(1-2):137–161, 1996.

[39] A. Goel, S. Rai, and B. Krishnamachari. Sharp thresholds for monotone properties in random geometric graphs. In *Proceedings of STOC*, pages 580–586. ACM, 2004.

[40] D. Gorisse, M. Cord, and F. Precioso. Locality-sensitive hashing for chi2 distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(2):402–409, 2012.

[41] Piyush Gupta and P. R. Kumar. Critical power for asymptotic connectivity in wireless networks. In *Stochastic Analysis, Control, Optimization and Applications*, pages 547–566, 1998.

[42] Torben Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In Christophe Paul and Michel Habib, editors, *Graph-Theoretic Concepts in Computer Science*, volume 5911 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 2010.

[43] Torben Hagerup and Christine Rüb. A guided tour of Chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

[44] P. Hall. On the coverage of k-dimensional space by k-dimensional spheres. *The Annals of Probability*, 13(3):991–1002, 1985.

[45] Frank Harary. *Graph theory.* Addison-Wesley, 1991.

[46] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[47] Dov Harel. A linear time algorithm for the lowest common ancestors problem. In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pages 308–319, 1980.

[48] S. Hu. Efficient video retrieval by locality sensitive hashing. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 2, pages 449–452, 2005.

[49] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[50] Robert W. Irving. An efficient algorithm for the "stable roommates" problem. *J. Algorithms*, 6(4):577–595, 1985.

[51] S. Janson. Random coverings in several dimensions. *Acta Mathematica*, 156(1):83–118, 1986.

[52] Woojay Jeon and Yan-Ming Cheng. Efficient speaker search over large populations using kernelized locality-sensitive hashing. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4261–4264, 2012.

[53] Yushi Jing and S. Baluja. Visualrank: Applying pagerank to large-scale image search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(11):1877–1890, 2008.

[54] K.A. Kala and K. Chitharanjan. Locality sensitive hashing based incremental clustering for creating affinity groups in hadoop; hdfs - an infrastructure extension. In *Circuits, Power and Computing Technologies (ICCPCT), 2013 International Conference on*, pages 1243–1249, 2013.

[55] Zixiang Kang, Wei Tsang Ooi, and Qibin Sun. Hierarchical, non-uniform locality sensitive hashing and its application to video identification. In *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, volume 1, pages 743–746 Vol.1, 2004.

[56] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, March 1995.

[57] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[58] Valerie King. A simpler minimum spanning tree verification algorithm. In SelimG. Akl, Frank Dehne, Jrg-Rdiger Sack, and Nicola Santoro, editors, *Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 440–448. Springer Berlin Heidelberg, 1995.

[59] Philip N. Klein and Robert E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 9–15, New York, NY, USA, 1994. ACM.

[60] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[61] Donald E. Knuth. *The art of computer programming, volume 1-3*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[62] J. Komlos. Linear verification for spanning trees. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 201–206, 1984.

[63] J. Komls. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[64] D. Kozen. *The design and analysis of algorithms*. Springer, 1992.

[65] Bhaskar Krishnamachari, Stephen B. Wicker, Rámon Béjar, and Marc Pearlman. Critical density thresholds in distributed wireless networks. In *Communications, information and network security*, 2002.

[66] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(6):1092–1104, 2012.

[67] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[68] L. Lovász and M.D. Plummer. *Matching Theory.* Akadémiai Kiadó, Budapest, 1986. Also published as Vol. 121 of the North-Holland Mathematics Studies, North-Holland Publishing, Amsterdam.

[69] A. Maheshwari and M. Smid. *Introduction to Theory of Computation.* Free Online, 2012.

[70] Udi Manber. Finding similar files in a large file system. In *in Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, 1994.

[71] Jiri Matousek. *Lectures on Discrete Geometry.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[72] Ji Matouek. On the distortion required for embedding finite metric spaces into normed spaces. *Israel Journal of Mathematics*, 93(1):333–344, 1996.

[73] Gregory L. Mccolm. Threshold functions for random graphs on a line segment. *Combinatorics, Probability and Computing*, 13:373–387, 2001.

[74] P.L. Meyer. *Introductory probability and statistical applications.* Addison-Wesley, Boston, MA, USA, 1970.

[75] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *J. ACM*, 44(1):1–29, January 1997.

[76] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* Cambridge University Press, New York, NY, USA, 2005.

[77] Tanmoy Mondal, Nicolas Ragot, Jean-Yves Ramel, and Umapada Pal. A fast word retrieval technique based on kernelized locality sensitive hashing. In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 1195–1199, 2013.

[78] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[79] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms.* Cambridge University Press, New York, NY, USA, 1995.

[80] Jaroslav Neetil, Eva Milkov, and Helena Neetilov. Otakar borvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(13):3 – 36, 2001. ¡ce:title¿Czech and Slovak 2¡/ce:title¿.

[81] Padma Panchapakesan and D Manjunath. On the transmission range in dense ad hoc radio networks. In *Proceedings of IEEE Signal Processing Communication (SPCOM)*, 2001.

[82] Mathew D. Penrose. The longest edge of the random minimal spanning tree. *The annals of applied probability*, pages 340–361, 1997.

[83] Mathew D. Penrose. On k-connectivity for a geometric random graph. *Random Struct. Algorithms*, 15(2):145–164, 1999.

[84] Mathew D. Penrose. *Random geometric graphs*, volume 5. Oxford University Press Oxford, 2003.

[85] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets.* Cambridge University Press, New York, NY, USA, 2011.

[86] Z. Rasheed, H. Rangwala, and D. Barbara. Lsh-div: Species diversity estimation using locality sensitive hashing. In *Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on*, pages 1–6, 2012.

[87] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* McGraw-Hill Higher Education, 5th edition, 2002.

[88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. In JohnH. Reif, editor, *VLSI Algorithms and Architectures*, volume 319 of *Lecture Notes in Computer Science*, pages 111–123. Springer New York, 1988.

[89] J. H. Spencer. *Ten lectures on the probabilistic method*, volume 52. SIAM, 1987.

[90] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[91] Robert Tarjan. *Data Structures and Network Algorithms*, volume 44, chapter 6. Minimum Spanning Trees, pages 71–83. SIAM, 1983.

[92] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[93] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, October 1979.

[94] Qiang Wang, Zhiyuan Guo, Gang Liu, and Jun Guo. Entropy based locality sensitive hashing. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1045–1048, 2012.