COMP5703 (DAA): Introduction to Algorithms

These notes extensively use material from Lars Arge, David Mount, CLRS book, Knuth's Art of Computer Programming - vol. 1 and are always under construction! Any bad comments and good suggestions can be sent at anil@scs.carleton.ca

Anil Maheshwari

December 2, 2008

Contents

1 General Stuff and Asymptotics **5**
- 1.1 Introduction . . . 5
- 1.2 Model of Computation . . . 6
- 1.3 Asymptotics . . . 6
  - 1.3.1 O-notation . . . 7
  - 1.3.2 Ω-notation (big-Omega) . . . 8
  - 1.3.3 Θ-notation (Big-Theta) . . . 8
- 1.4 How to analyze Recurrences? . . . 9
- 1.5 Strassen's Matrix Multiplication . . . 10
- 1.6 Matrix Multiplication . . . 10
  - 1.6.1 Strassen's Algorithm . . . 11

2 Introduction to Graphs and their Representation **13**
- 2.1 Introduction and Definitions . . . 13
- 2.2 How to represent graphs in a computer? . . . 15
- 2.3 Graph traversal . . . 15
  - 2.3.1 Breadth-first search (BFS) . . . 16
- 2.4 **Topological sort and DFS** . . . 17
  - 2.4.1 Topological Sort . . . 17
  - 2.4.2 Depth First Search . . . 18
  - 2.4.3 Computation of $low(v)$ . . . 19
  - 2.4.4 **Biconnectivity** . . . 20
  - 2.4.5 Equivalence Relation . . . 20
  - 2.4.6 Biconnectivity . . . 21

3 Minimum Spanning Trees **25**
- 3.1 Minimum Spanning Trees . . . 25
- 3.2 Kruskal's Algorithm for MST . . . 26
- 3.3 Prim's MST algorithm . . . 27
- 3.4 Randomized Algorithms for Minimum Spanning Trees . . . 28
  - 3.4.1 Boruvka's Algorithm . . . 28
  - 3.4.2 Heavy edges and MST Verification . . . 29
  - 3.4.3 Randomized Algorithm . . . 29

4 Network Flow **31**
- 4.1 What is a Flow Network . . . 31
- 4.2 Ford and Fulkerson's Algorithm . . . 32
- 4.3 Edmonds-Karp Algorithm . . . 36
- 4.4 Applications of Network Flow . . . 37

5 Lowest Common Ancestor **39**
5.1 LCA $\rightarrow$ RMQ 39
5.2 Range Minima Queries 40
5.2.1 A naive $O(n^2)$ algorithm 40
5.2.2 An $O(n \log n)$ algorithm 40
5.2.3 An $O(n)$ algorithm with ± 1 property 40
5.3 RMQ $\rightarrow$ LCA 41
5.4 Summary 42

6 Separators in a Planar Graph **43**
6.1 Preliminaries 43
6.2 Proof of the Planar Separator Theorem 44

7 Complexity Theory **49**
7.1 The running time of algorithms 49
7.2 The complexity class **P** 50
7.2.1 Some examples 50
7.3 The complexity class **NP** 52
7.3.1 **P** is contained in **NP** 56
7.3.2 Deciding **NP**-languages in exponential time 57
7.3.3 Summary 58
7.4 Non-deterministic algorithms 58
7.5 NP-complete languages 60
7.5.1 Two examples of reductions 61
7.5.2 Definition of NP-completeness 64
7.5.3 An **NP**-complete domino game 66
7.5.4 Examples of **NP**-complete languages 70
7.6 Exercises 75

8 Problems **79**

Chapter 1

General Stuff and Asymptotics

1.1 Introduction

- Class[1] is about *designing* and *analyzing algorithms*

  [1]: Keep Asking que
  - handwriting!- espec
  slow me down and to
  understand Indian E

  - *Algorithm*:
    - Mis-spelled *logarithm!*.
    - The first most popular algorithm is the Euclid's algorithm for computing 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,...)

  Revise 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 we need 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
    * Arithmetics (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 limits for functions describing the running time of algorithms. We will describe the formal definitions from [3] 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?

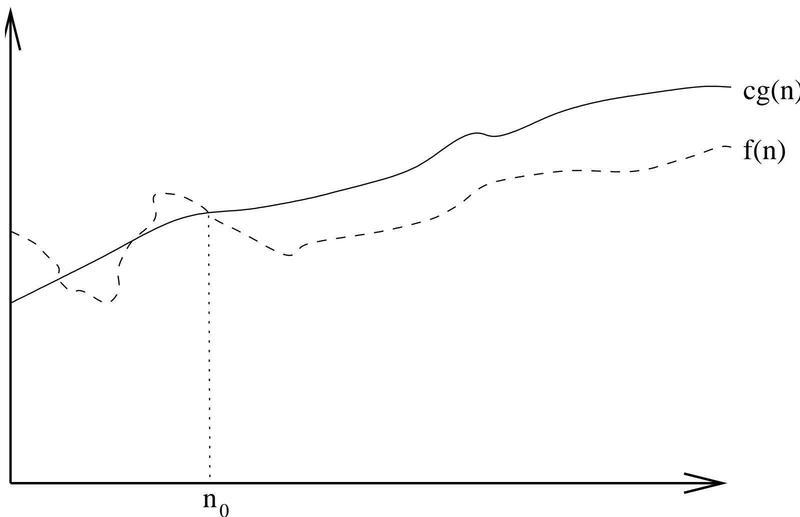
$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{(n^3 + 3n^2 + 2n)/6}{n^3} = 1/6,$$

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.



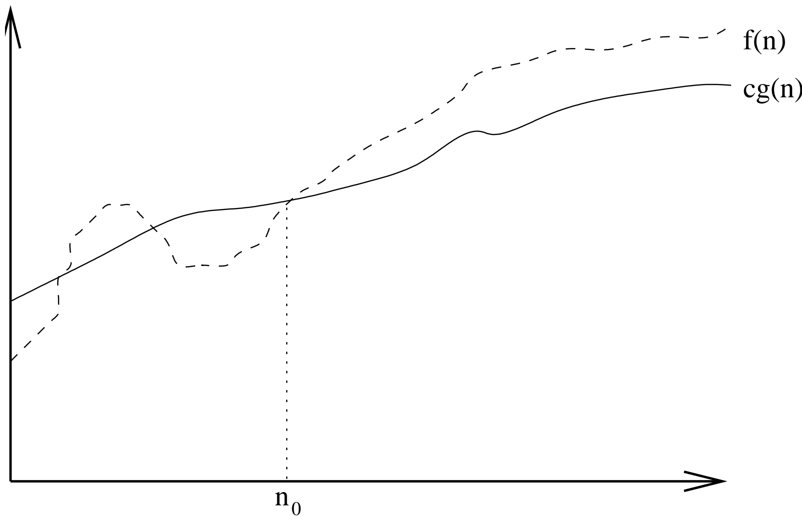
- Examples:
  - $1/3n^2 - 3n \in O(n^2)$ because $1/3n^2 - 3n \leq cn^2$ if $c \geq 1/3 - 3/n$ which holds for $c = 1/3$ and $n > 1$.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in O(n^k)$, where $k \geq d$ is a constant. What are c and n_0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

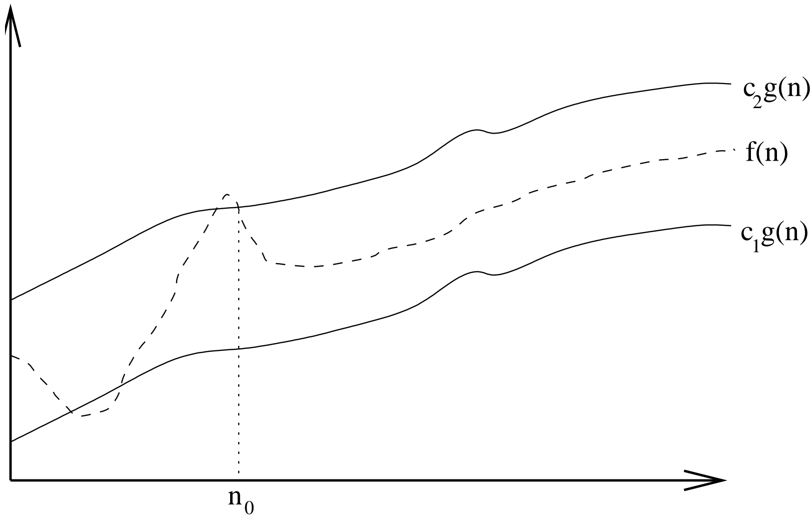


- Examples:
  - $1/3n^2 - 3n = \Omega(n^2)$ because $1/3n^2 - 3n \geq cn^2$ if $c \leq 1/3 - 3/n$ which is true if $c = 1/6$ and $n > 18$.
  - Let $p(n) = \sum_{i=0}^{d} a_i n^i$ be a polynomial of degree d and assume that $a_d > 0$. Then $p(n) \in \Omega(n^k)$, where $k \leq d$ is a constant. What are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

1.3.3 Θ-notation (Big-Theta)

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

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



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

- 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 (of course I would have made some errors on what I have written below). We need to sort of visualize it using the 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

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

ves the
vill get

Now apply the various cases -2-

1.5 Strassen's Matrix Multiplication

1.6 Matrix Multiplication

To illustrate the analysis of recurrences as well as the divide and conquer method, we consider Strassen's matrix multiplication method [?, 9]. Let A, B and C be three $n \times n$ matrices, where $Z = X \cdot Y$. There are n rows, n columns and n^2 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 $T(n) = O(n^{\log_2^7})$.

1.6.1 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 -3-

[3]: For sure I am no supposed to rememb seven S's.

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

  - 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 $O(n^{2.3\cdots})$). 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. -4- 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 [8] for details).

[4]: Is it really intere Field? Probability? Uniform Distribution Random? Why the h am doing this course first class is so bad?

Theorem 1.6.1 *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 $\{0,1\}^n$, probablility that $Pr[ABr = Cr] \leq 1/2$.*

Chapter 2

Introduction to Graphs and their Representation

2.1 Introduction and Definitions

Graphs were discovered by Euler (Königsberg bridge problem[5]), Kirchoff (electrical networks[6]) and Cayley (enumeration of organic chemical isomers[7]) in different contexts. Graphs are combinatorial structures used in computer science. Lists, Trees, Directed Acyclic Graphs, Flow Charts, Control Flow Graphs, and Planar Graphs 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 [Graph Theory], Berge [Graphs and Hypergraphs], Bollobas [Graph Theory], Bondy and Murty [Graph Theory], Lovasz [Matching Theory]. We need to get used to some of the basic definitions.

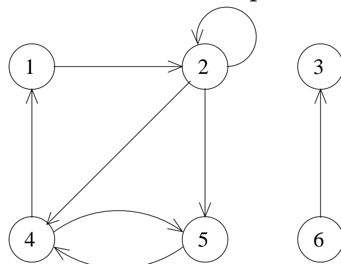
[5]: Leonhard Euler, 1707-1783

[6]: Gustav Kirchoff, 1824-1887: At every an electrical circuit t of all currents should equal to zero, i.e., th charge cannot accum a node - or what com must go out!

[7]: Arthur Cayley, 1821-1895

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

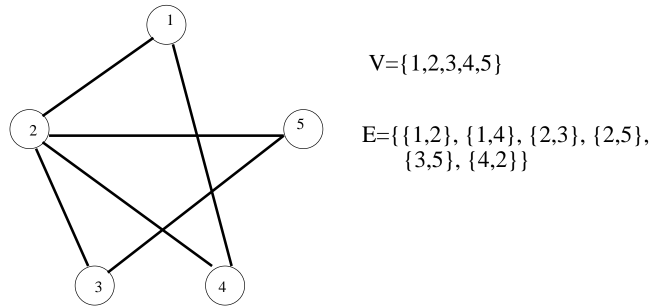
- *Directed graphs*: E is a set of ordered pairs of vertices (u, v) where $u, v \in V$



V = {1, 2, 3, 4, 5, 6}

E = {(1,2), (2,2), (2,4), (2,5)
(4,1), (4,5), (5,4), (6,3)

- *Undirected graph*: E is a set of unordered pairs of vertices $\{u, v\}$ where $u, v \in V$



Incidence An edge $\{u, v\}$ is *incident* to u and v. (In our example, the edge {1,4} is incident to the vertex 1 and to the vertex 4.

Degree of vertex in undirected graph is the number of edges incident to it. (In our example degree of vertex 1 is 2, degree of vertex 2 is 4.

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 $u = v$

In our undirected graph example, a path from the vertex $u = 1$ to the vertex $v = 5$ consists of vertices $< u = 1, 4, 2, 5 = v >$

Connected An undirected graph is connected if every pair of vertices are joined by a path. (In our example the graph is connected.)

Component The connected components are the equivalence classes of the vertices under the "reachability" relation. (In our example there is only 1 connected component.)

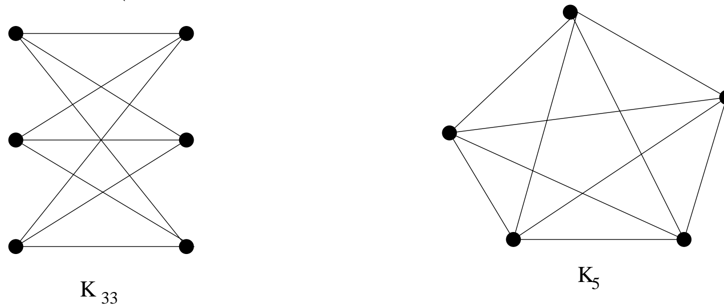
Strongly Connected A directed graph is *strongly connected* if every pair of vertices are reachable from each other. (In our example the graph is not strongly connected since there is no way to reach from vertex 3 to 4 or from the vertex 1 to vertex 6.)

Strongly connected components The *strongly connected components* are the equivalence classes of the vertices under the "mutual reachability" relation. (In our example the strongly connected components will be {{1,2,5,4},{3},{6}}. Its upto you to say that a singleton vertex is strongly connected or not!.)

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). The graph in our example is simple. 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).



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 $K_{3,3}$.[8] 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.

[8]: This is one of th fundamental theorem 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 ijth 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 ijth entry in the matrix A. Moreover, this is independent of the sizes of V and E. But 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. This is because we can either search in the list of v or the list of w, and hence its advantageous to search in the smaller list.

2.3 Graph traversal

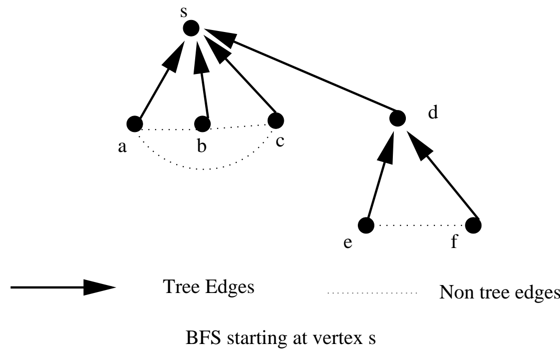
Once we have a graph, represented inside the computer, what do we do with it? When we goto a new (small!) city what is the best way to explore all the bars? What is the best way to search for a particular web page just following the hyperlinks (assume no search engine and assume that we can go from any 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:
  - Start at a source vertex and visit,
  - All vertices at distance 1,
  - Followed by all vertices at distance 2,
  - Followed by all vertices at distance 3,

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



BFS starting at vertex s

Here is the BFS algorithm.

```
procedure BFS(v)
Input: A simple connected undirected graph G=(V,E) and the start vertex v
       represented as adjacency lists.
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); // Let x be in the front of Q.
      Dequeue(Q); // Remove the element in the front of 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.

2.4 Topological sort and DFS

2.4.1 Topological Sort

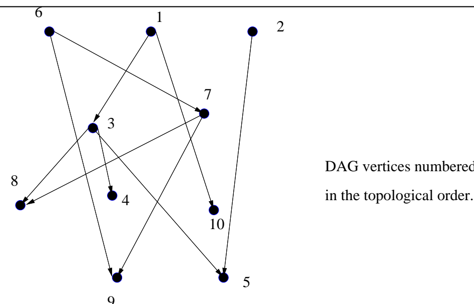
Let $G = (V, E)$ be a directed acyclic graph (DAG) on the vertex set V and directed edge set E. A *topological sort* of a DAG is a linear ordering of all its vertices such that if G contains a directed edge (u, v), then u appears before v in the ordering. One can think of this process as assigning a number $f : V \to \{1, \cdots, |V|\}$ to each vertex such that for every directed edge (u, v), $f(u) < f(v)$. We will sketch two algorithms, first a slower and then the 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 using adjacency matrix representation.

Step 1: Start from any vertex and follow edges backwards until a vertex v is found having no incoming edges.

Step 2: Make v the next vertex in the total order.

Step 3: Delete the vertex v and all of its outgoing edges and goto Step 1.



DAG vertices numbered in the topological order.

Observe that in Step 1 we will find a vertex v since DAGs have no cycles and they are finite graphs. Moreover a vertex is assigned a number when it has no incoming edges. This should be sufficient to prove the correctness.

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 linear $O(|V| + |E|)$ time?

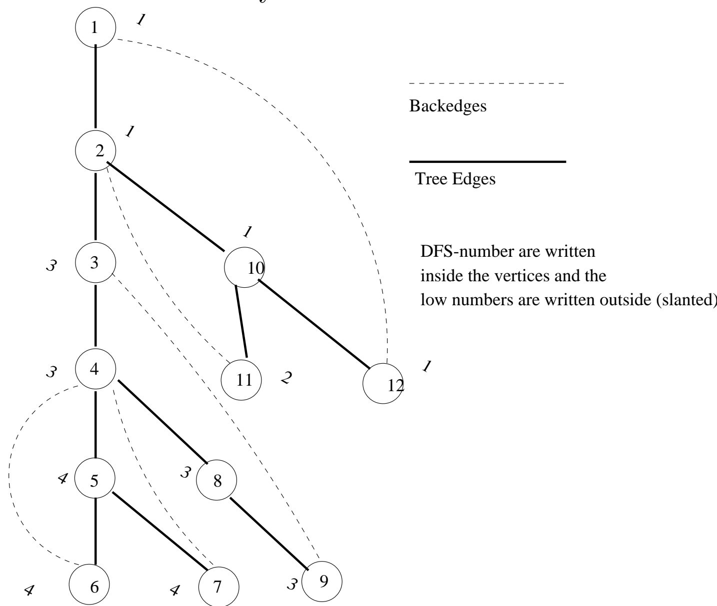
2.4.2 Depth First Search

DFS is another way of exploring the 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 $G = (V, E)$. We will discuss DFS in G. (Similar ideas generalize to disconnected as well as directed graphs). Informally DFS on G can be described as follows:

- Select a vertex v of G which is initially unvisited.
- Make v visited.
- 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 $T \cap B = \emptyset$. Here is the formal description of the algorithm (taken from Aho, Hopcroft, Ullman [1]). Each vertex in G will be assigned a DFS-number, i.e., the order in which they are first visited in the DFS.



Algorithm: DFS on $G = (V, E)$ 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.
begin

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)

end

---

procedure SEARCH(v)
begin

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 */

end

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 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 descendent 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 $low(v)$

Here 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 the graph is biconnected, as well as computing its biconnected components. (This will the topic of the next section.)
What is $low(v)$?
First relabel the vertices by their DFS-number. 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 descendent of v and w is an ancestor of v in the DFS tree$\})$

Here is the intuition. Consider the subtree T_v of the DFS-tree T for the vertex v. What is the vertex closest to the root of T which we can reach just using the back edges from T_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 ancestor vertices of v, which are joined by the back edges from the vertices in T_v.
In order to compute $low(v)$, we need to compute three values and we will sketch the modification in the DFS algorithm to compute them. The three values 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.

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

Here is the revised SEARCH(v) procedure that computes the low values as well.

---

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) $low(v) := \min(low(v), low(w))$ /*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

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

2.4.4 Biconnectivity

2.4.5 Equivalence Relation

Before we talk about biconnectivity, we need to recall the concept and definitions of 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 $[a] = [b]$, where $a, b \in A$.

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

Example of an Equivalence Relation: Let R be the 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, denoted by [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 [5].

2.4.6 Biconnectivity

Most of the material in this section is from Kozen [4] 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.

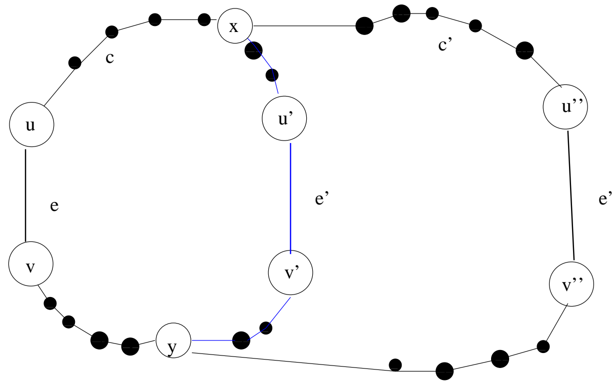
Biconnected A connected graph is biconnected if any pair of distinct vertices lie on a simple cycle (one with no repeated vertices). Equivalently, for every distinct triple of vertices v, a, and b, there exists a path between a and b not containing v. Observe that G is biconnected if and only if it has 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 2.4.1 *The relation $\equiv$ defined above is an equivalence relation.*

Proof [4]: 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 in itself lie 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''. 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 $e \equiv e''$. □

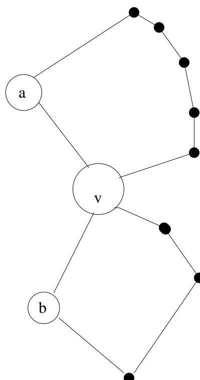
Now we have an equivalence relation. What are the equivalence classes of this relation with respect to the edge set of G.

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. These lemmas are extremely useful (especially in computer science) since they give a complete characterization of the object/structure under consideration, and often lead to an algorithm.

Lemma 2.4.2 *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. 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.

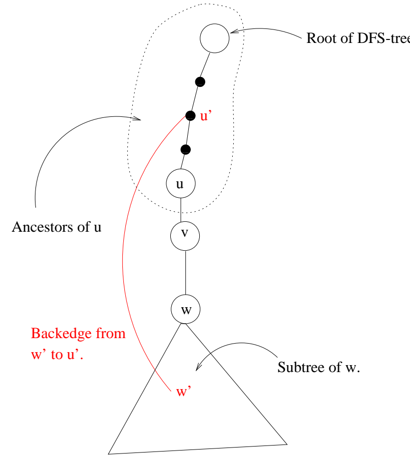


Now suppose that v is contained in two distinct biconnected components, and is adjacent to vertices a and b in these components, respectively. Then $(va) \not\equiv (vb)$. Then all paths between a and b goes through v, and hence removing v disconnects G. So v is an articulation vertex. □

Lemma 2.4.3 *Let (uv) and (vw) be two adjacent tree edges in a DFS of G. Then $(uv) \equiv (vw)$ if and only if there exists a back edge from some descendent of w to some ancestor of u.*

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

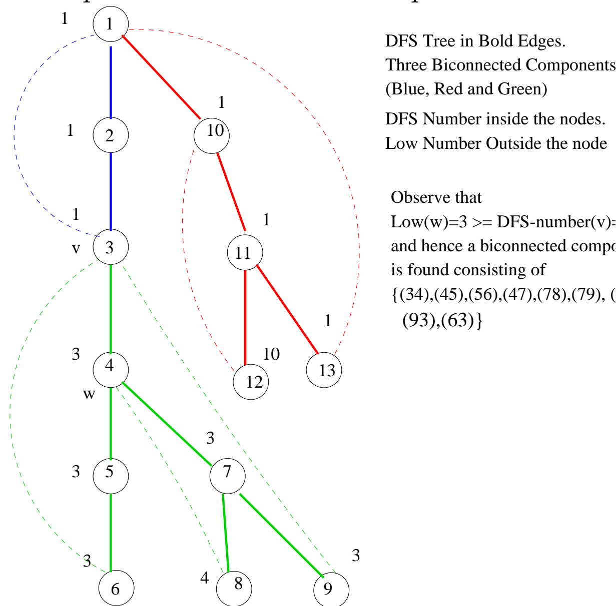
If there exists a backedge from some descendent w' of w to some ancestor u' of u, then $(uv) \equiv (vw)$, since there is a simple cycle consisting of the tree path between u' and w' and the backedge $w'u'$.



Suppose $(uv) \equiv (vw)$. By definition, there is a simple cycle consisting 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 u to the root of the DFS tree.) $\Box$

Lemma 2.4.4 *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 descendent of w (including w) and a proper ancestor of v.

Part (a) is easy to prove and part (b) follows from the previous lemma! Here is the modification to the DFS procedure to compute the biconnected components.



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) in the STACK if it is already not there. After discovering a pair (v, w) such that w is a child of v and $low(w) \geq DFS - number[v]$, then 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 (A problem for the next assignment).

Chapter 3

Minimum Spanning Trees

3.1 Minimum Spanning Trees

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

As in the CLRS book [3], we will describe the two algorithms for MSTs (i.e. Kruskal's and Prim's), both of them are greedy algorithms and are based on the following generic algorithm. The algorithm maintains a subset of edges A, which is a subset of some MST of G.

Algorithm: Generic-MST(G,w)
$A := \emptyset$
while $A \neq MST$ do
find a **safe edge** $\{(u, v)\}$ for A and $A := A \cup \{(u, v)\}$.
return A.

What is a safe edge?
But first a few definitions.

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

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

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

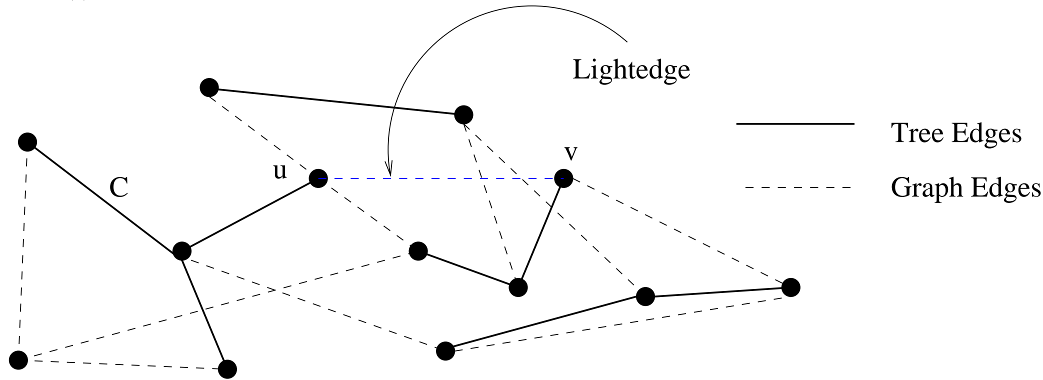
Light edge Edge with the minimum cost crossing the cut is called a light edge.

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

Proof: Assume that T is a MST that includes A. If T as well 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 $P_T(uv)$, 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 $(x, y) \in P_T(uv)$ that crosses the $cut(S, T - S)$. 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 edges 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 $cut(C, V - C)$).



Corollary 3.1.2 *Let $A \subset E$ be included in some MST. Consider the forest consisting $G_A = (V, A)$, i.e. the vertices in 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.*

3.2 Kruskal's Algorithm for MST

Proposed by Kruskal in 1956 and the algorithm follows directly from the corollary. Here are the main steps. To begin with the set A consists of only isolated vertices (in all $|V|$ components).

1. Sort the edges of E into an increasing order with respect to their cost.

2. Pick the edges in order, and if the edge joins two components then add that edge (a safe edge) to set A.

To implement Step 2, we do the following. Let e_i be the edge under consideration, and all the edges whose cost is smaller 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 they join two different components. If the endpoints are within the same component, then we discard the edge e_i, otherwise we need to merge the two components to form a bigger component. For each vertex we keep track of which component it lies in using a label associated with the vertex. Initially each vertex is in its own component, and 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 C_a and C_b corresponding to the end points a and b of the edge $e_i = (a, b)$, and then relabel the vertices. We update the labels of only one of the components, the component which is smaller in size. Here are the details of Step 2 with respect to edge $e_i = (a, b)$.

1. If label of vertex a is the same as the label of vertex b, then discard the edge e_i, since a and b are in the same component, otherwise e_i is a safe edge and continue with the next step.

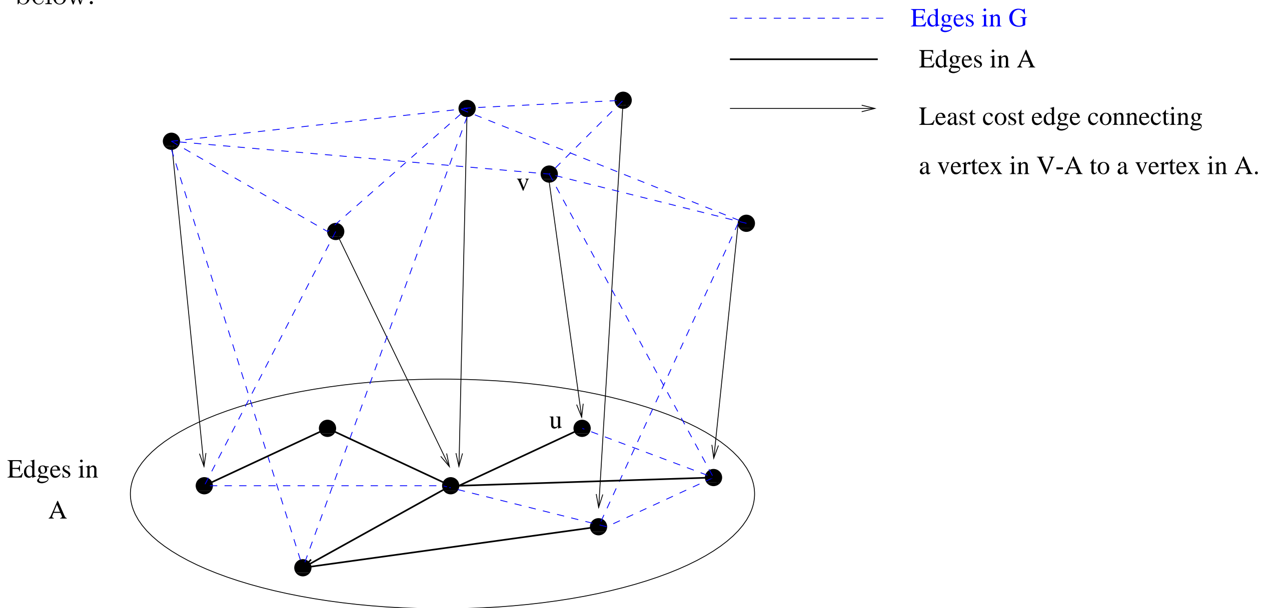
2. Assume that $|C_a| \leq |C_b|$. Then update the labels of vertices in C_a.

Let us analyze the complexity of this 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. The complexity of merging the components 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 becomes at least twice. Since the maximum size of a component is $|V|$, this implies that v can be relabeled at most $\log_2 |V|$ times, since each time it is relabeled the size of its component at least doubles. So the total complexity of the Step 2 of the algorithm is $O(|E| + |V| \log |V|)$ time, and the result is summarized in the following theorem.

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

3.3 Prim's MST algorithm

Prim's algorithm is very similar to Dijkstra's single source shortest path algorithm (in fact the complexity analysis will be the same). Here the set A at any stage of the algorithm forms a tree. Initially the set A consists of just a vertex (call it the root or the seed vertex!), and then in each stage a light edge (a stem!) is added to the tree connecting A to a vertex in $V - A$. The key to the efficiency to Prim's algorithm is in selecting the "stem" edge efficiently. For each vertex $v \in V - A$, we keep track of what is the least cost edge that connects v to A, the cost of this edge is the "key" value associated to v. The key values are stored in a priority queue Q; in each step the vertex v with the least priority is extracted out of Q, and suppose that corresponds to the edge (u, v), where $u \in A$. Observe that this edge is a safe edge since is the light edge for $cut(A, V - A)$. Then update $A := A \cup \{(u, v)\}$. Also after extracting v out of Q, we need to update Q. The details are explained below.



MST-PRIM(G, w, r) /* r is the root; w the cost function */
for each $v \in V$ do { $key[v] := \infty$; $\pi[v]$:=nil }
/* Π keeps track of the parent of a vertex in the tree. */
$key[r] := 0$;
$Q := V$ /* Priority queue consists of vertices with their key values */

While $Q \neq \emptyset$ do
Begin

$u :=$ Extract-Min(Q);
for each vertex $v \in Adjacency(u)$ do
if $v \in Q$ and $w(u, v) < key[v]$ then $\{\pi[v] := u; key[v] := w(u, v)\}$
End.

The vertices that are in the set A at any stage of the algorithm are the vertices in $V - Q$, 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 and then keeps decreasing.

Let us analyze the complexity of the algorithm. The main steps are the priority queue operations, namely the decrease-key and the extract-min operation. In all we perform $|V|$ extract-min operation, one for each vertex. Also we perform $O(|E|)$ decrease key operation, once for each edge. Here is the table describing the complexity of these operations depending upon which implementation of the priority queue is used.

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

These complexities are per operation. The complexities of Fibonacci Heaps are amortized (kind of an average over the worst possible scenario! - more on that later).

3.4 Randomized Algorithms for Minimum Spanning Trees

Here we discuss some of the recent results, in the last decade, 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 [8] and T. Chan's simplified analysis from [2]. 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 three concepts which are discussed in the subsequent subsection followed by the actual algorithm. First concept is the algorithm due to Boruvka from 1926 which helps us in reducing the number of vertices in the graph in linear time. The second concept is about heavy and light edges with respect to a spanning tree, and the last concept is about MST verification.

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. A simple way to compute the MST will be as follows:

Boruvka's Algorithm for Computing MST
Input : A connected graph $G = (V, E)$ and the weights of edges are all distinct.
Output: Minimum Spanning Tree T.

1. $T := \emptyset$.
2. Mark the edges with the minimum weight incident to each vertex. Add these edges to T.
3. Identify the connected components of the marked edges.
4. Replace each connected component by a vertex.

5. Eliminate all self loops. Eliminate all multiple edges between a pair of vertices, except the edge with the minimum weight.

6. Repeat Steps 2-5 (called a phase) till we are left with a single vertex.

We make a few observations. Let G' be the graph obtained from G after contracting the edges in a phase in the algorithm. Then the MST of G is the union of the contracted edges and the edges in the MST of G'. Also in each contraction phase of the above algorithm the number of vertices in the graph reduce by at least a half. Hence there will be in all $O(\log |V|)$ phases. Also observe that each phase can be implemented in $O(|V| + |E|)$ time and hence we obtain another algorithm that achieves the running time of $O(|E| \log |V|)$.

3.4.2 Heavy edges and MST Verification

Let F be any spanning tree (may not be the minimum!) of G. 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 on this path. Define an edge $(u, v) \in E$ to be F-heavy if $w(uv) > w_F(u, v)$, otherwise it is F-light. 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. Observe that all edges of F are F-light. Here is an important Lemma.

Lemma 3.4.1 *Let F be a spanning tree (need not be the minimum) of G. If an edge of G is F-heavy then it does not lie in the Minimum Spanning Tree of G. (Note that if an edge is F-light it may or may not be in the MST).*

Proof: It is left for you to prove it formally. The proof proceeds by contradiction. Assume that the edge $(uv) \in E$ is F-heavy and it participates in the MST, T, of G. Consider the edges on the path $P(u, v)$ in F. Add all these edges to T and remove the edge (uv) from T, to obtain a graph G'. G' is still connected. Remove one by one the edges added to G' till G' remains connected. Observe that the new tree achieved by this process is still a spanning tree and has weight lower than that of T, this contradicts the minimality of T. □

Given a spanning tree F of G, how do we verify that it is MST of G. If all the edges in $E \setminus F$ are F-heavy then F is the MST of G. How quickly can we test whether an edge is heavy or light? It turns out that 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. The algorithms are fairly complex and there are series of papers on these. In fact finding a simple algorithm for this problem is still open. With some effort one can obtain an algorithm that runs in time close to linear by using some range minima techniques, but the complexity will involve an extra factor corresponding to inverse-Ackerman functions.

3.4.3 Randomized Algorithm

Here are the main steps of the algorithm. The analysis is based on Timothy Chan's paper [2].

Input: A connected graph $G = (V, E)$ with distinct edge weights.
Output: Minimum Spanning Tree T.

1. Reduce the number of vertices by executing three phases of Boruvka's algorithm. Let the resulting graph be $G_1 = (V_1, E_1)$, where $|V_1| \leq |V|/8$ and $|E_1| \leq |E|$. 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 the MST of G as $C \cup F_3$.

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

Proof: Correctness 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 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 $|E||V_1|/|R|$. In our case 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 $O(|V| + |E|)$. $\square$

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. We will fix a 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 $MST(R)$. This will ensure the connectedness of the minimum spanning tree of R.

Observation 3.4.3 *Observe that an edge $e \in E$ is light with respect to $MST(R)$ if and only if $e \in MST(R \cup \{e\})$.*

Proof: It is straightforward. 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. $\square$

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 $|E||V|/|R|$.*

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 the Observation 3.4.3 this is same as finding what is the bound on the probability that $e \in MST(R \cup \{e\})$. Let $R' = R \cup \{e\}$. We will use the standard analysis technique, called as the backward analysis. First we will analyze the probability by fixing the set R', and then we will show that the expression obtained is independent of the choice of the elements, but just the cardinality of the sets, 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 helps since we know the elements of the set R'.) Note that $MST(R')$ has $|V| - 1$ edges, and e is a random edge of R', and 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. $\square$

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.

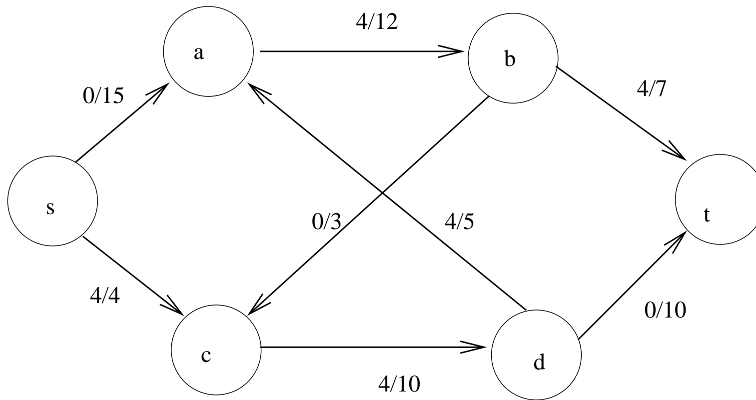
3. **Flow conservation**: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = 0$. This uses the skew symmetry property, otherwise we have to sum up the flow values coming into a vertex and that should be equal to the sum of the outgoing flow values from that vertex. This is same as the Kirchoff law 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.
Here is an example from Shimon Even's book.

a/b means that the flow is a
and the capacity is b.
flow value=4 (not the max flow though).



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.

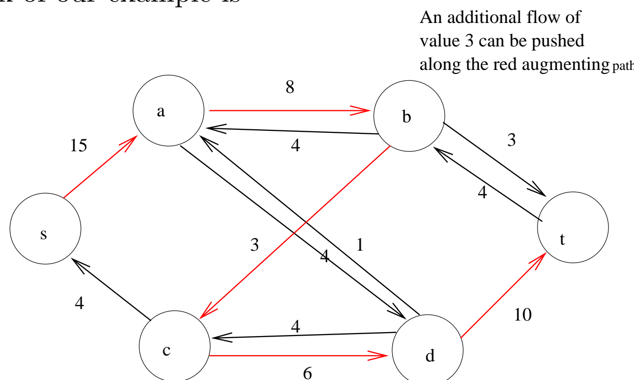
An augmenting path is a path from s to t along which additional flow can be sent. This path is found using the concept of residual networks. The residual network consists of those edges which can admit more flow. The residual capacity $c_f(u, v)$ of an edge (u, v) in a flow network is given by

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

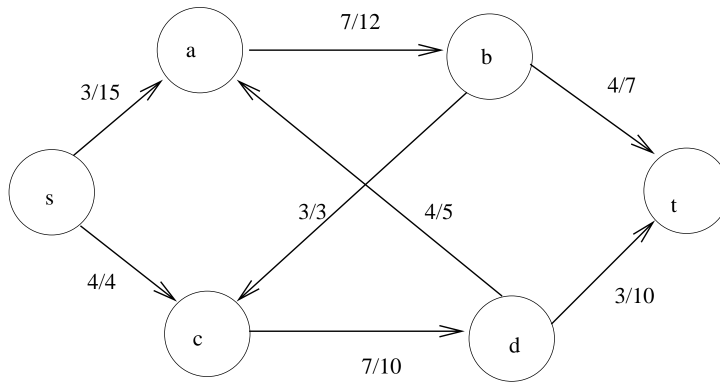
In our example $c_f(a, b) = 12-4 = 8$, $c_f(s, c) = 4-4 = 0$, $c_f(b, a) = c(b, a)-f(b, a) = 0-(-4) = 4$. Given a flow network G and the flow function f, the residual network $G_f = (V, E_f)$ consists of the same vertex set and the edges E_f are defined as follows:

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

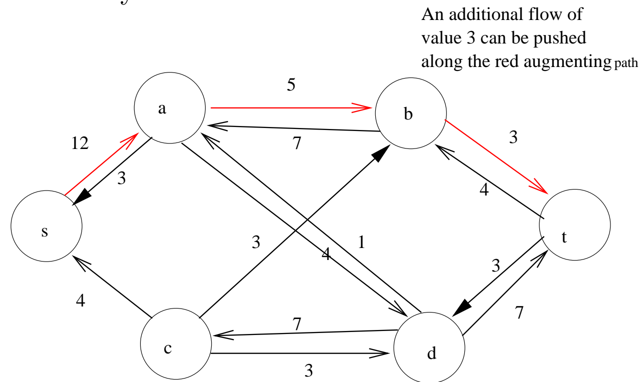
The residual network of our example is



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 $7 = 4 + 3$.

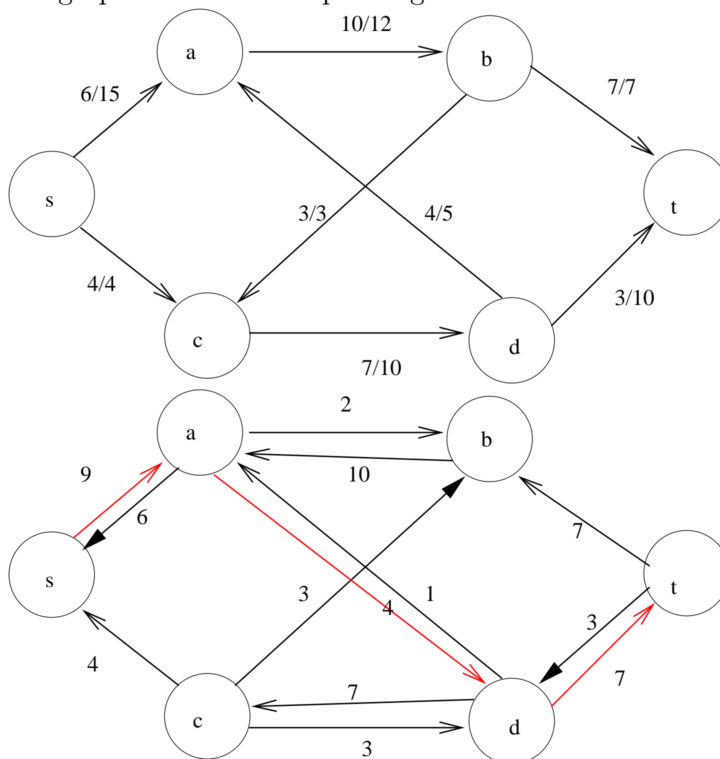


The new residual network that we get is the following, and there is an augmenting path that further increases the flow value by 3.

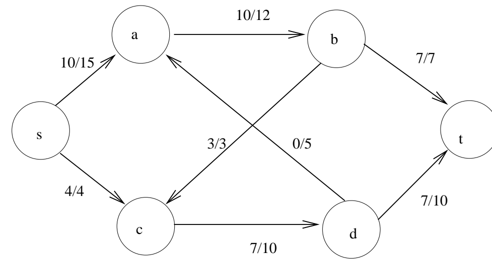


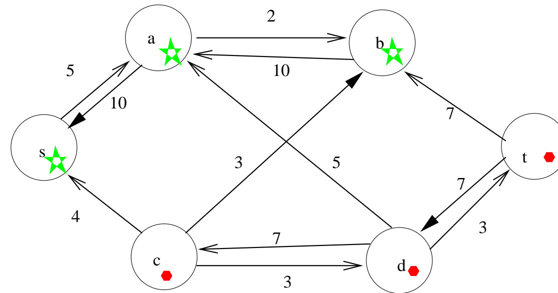
.

Here is the new flow graph and the corresponding residual network.



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





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}.

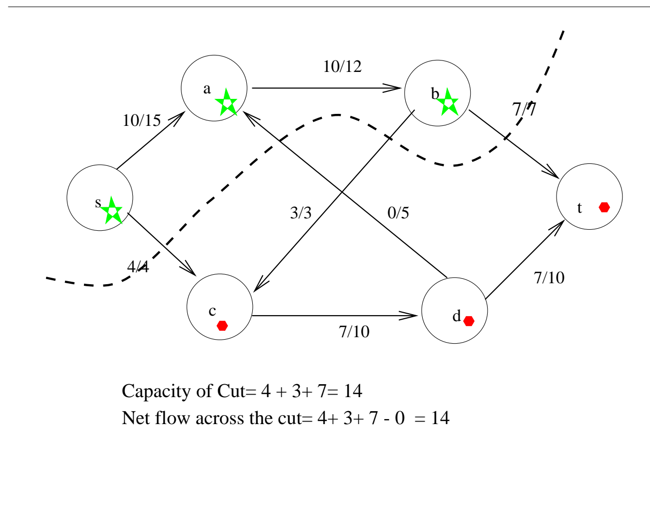
Now consider the residual network in the last step, where there are no paths joining s and t. As can be seen from the figure, there is a path from s to every vertex in the set $\{s, a, b\}$, and there are paths from vertices $\{c, d, t\}$ to t. This automatically partitions the set of vertices into two,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



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 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.

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 [3]). 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'.

Chapter 5

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.

5.1 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 5.1.1 *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. ■

5.2 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 $RMQ(i, j)$, $1 \leq i \leq j \leq n$, can be answered in $O(1)$ time.

5.2.1 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 $1 \leq i \leq j \leq n$, 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.

5.2.2 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 $1 \leq i \leq j \leq n$. 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 5.2.1 *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.*

5.2.3 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 5.2.1.

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 5.2.2 *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 5.2.3 *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 5.2.4 *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 5.2.5 *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.*

5.3 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 5.3.1 *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 5.3.2 *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. $RMQ[i,j] = k$. ∎

5.4 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.

Chapter 6

Separators in a Planar Graph

This chapter is based on Kozen [4] and the famous paper of Lipton and Tarjan on planar separator theorem. Earlier we have seen that in a binary tree on n-nodes, there exists a vertex such that whose removal leaves no component having more than $2(n+1)/3$ nodes. This can be extended quite easily 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 $O(\sqrt{n})$.

6.1 Preliminaries

Definition 6.1.1 *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.*

Definition 6.1.2 *In an embedded planar graph, we have vertices, edges and faces. The* dual *of a planar graph G is the 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 6.1.3 *A planar 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 6.1.4 *A set $S \subseteq V$ for a graph $G = (V, E)$ is called a* 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 6.1.5 *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 6.1.6 *A* outerplanar graph *is a planar graph such that all vertices lie on a single face (usually referred to as the outerface).*

Definition 6.1.7 *The dual of a triangulated outerplanar graph is a binary tree.*

We have seen that complete graph on five vertices, K_5, and complete bipartite graph on six vertices, $K_{3,3}$, is nonplanar. It is easy to see that a tree is planar, and as well as outerplanar graphs are planar. Both of these graphs admit small size separators to get a balanced decomposition. What we will prove in this chapter is that in fact planar graphs have the similar property. This is the theorem we will like to prove:

Theorem 6.1.8 *[Lipton and Tarjan] 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 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 figure out in linear time whether the given graph is planar or not, and if it is planar it finds an embedding of it in the plane. Also if the planar graph is not triangulated, then it can be triangulated in linear time, by inserting required number of edges. Other than this essentially we will use breadth first and the concept of fundamental cycles to prove this theorem.

6.2 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 cross over two or more levels. All edges must connect vertices in the same level or consecutive level. 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!

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. 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 6.2.1 *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 $t_2 - t_0 \leq \sqrt{n}$ □

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.

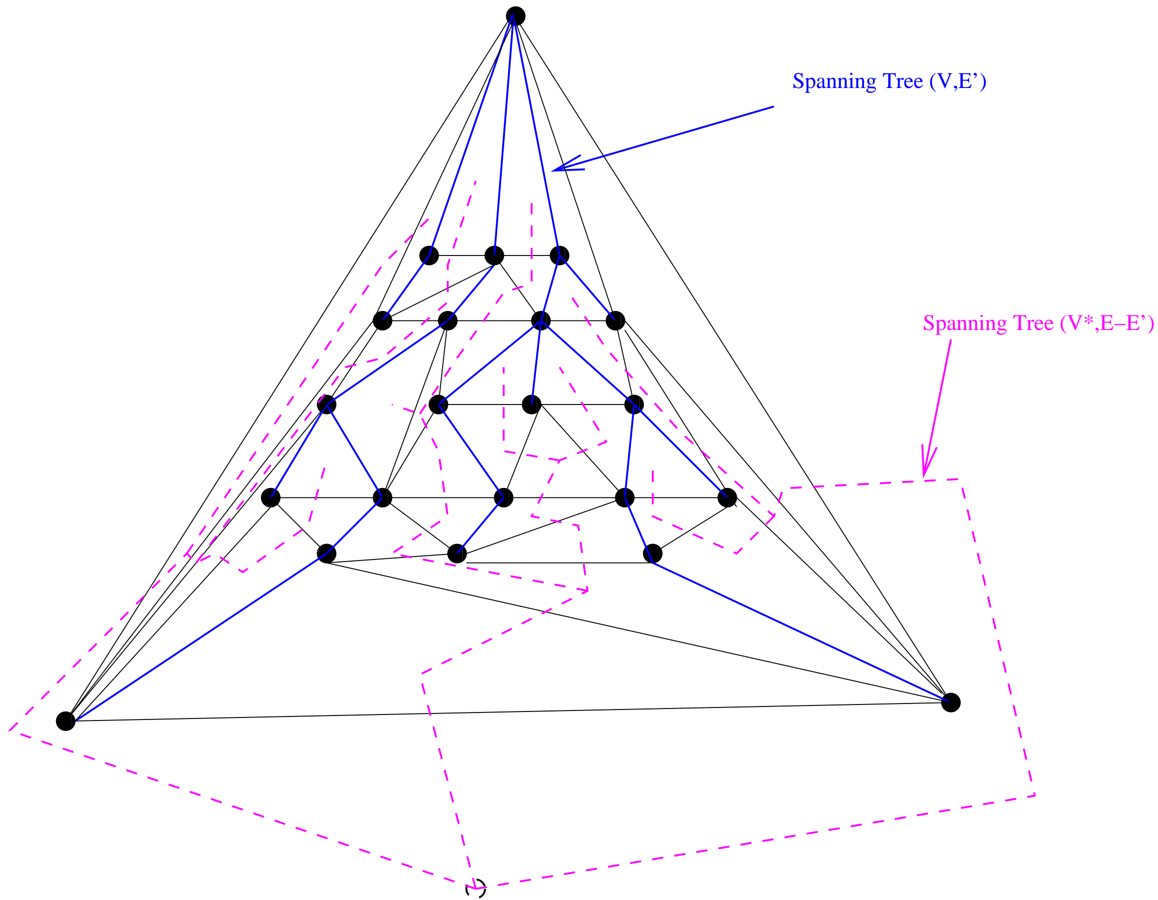


Figure 6.1: The edges of (V, E') are in bold-solid. The edges of $(V^*, E - E')$ are dashed-bold and the planar graph edges are in solid.

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 of 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 $A = \max(C, D) \cup \min(D', D'')$ and $B = \min(C, D) \cup \max(D', D'')$. 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 easy to prove, that will show the critical property relating the tree T, the planar graph D, its dual D^*, and the dual tree T'.

Lemma 6.2.2 *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.*

Lemma 6.2.3 *Let $G = (V, E)$ be a connected planar 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 in G^* (see Figure 6.1 for an illustration).*

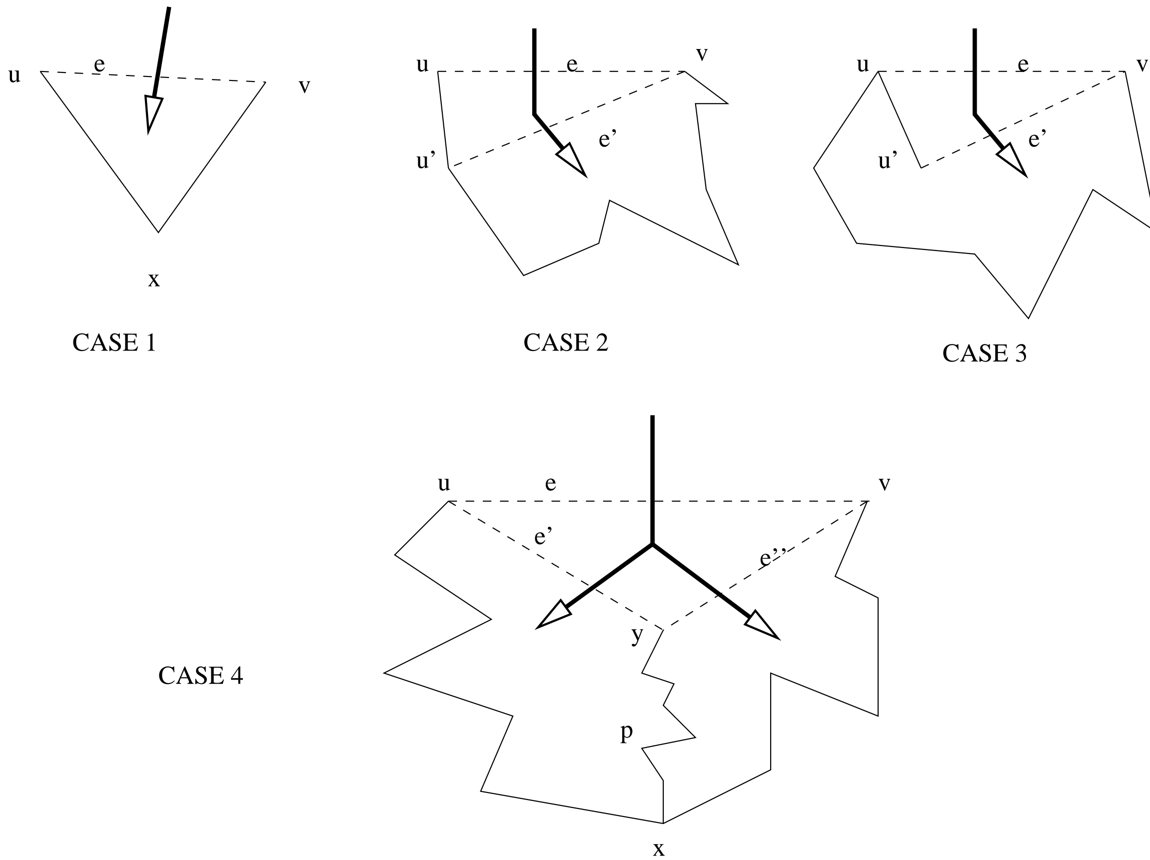


Figure 6.2: Bold edges represent the DFS traversal of faces of D corresponding to the tree T'. Dashed edges corresponds to edges in $E - E_T$ and solid edges corresponds to edges in E_T, i.e. a spanning tree of D.

Let E_T be the edges in the spanning tree T in D. 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^*. 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. 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 $c(e)$.

For each step of DFS, one of the following four cases will occur (see Figure 6.2)

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 6.2.4 *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 $I(e'') + |c(e'')| < n/3$. $I(e) + |c(e)| = I(e') + I(e'') + |p| - 1 + |c(e')| + |c(e'')| - 2|p| + 1 \leq 2/3n - |p| \leq 2/3n$.

□

Chapter 7

Complexity Theory

This Chapter is from Professor Michiel Smid and part of it is used in the course COMP 2805 (Theory of Computation). Some of the terminology is from that course. In case some of these terms are not clear, then please refer to our class notes of the course COMP 2805.

In the previous chapters (in the courses notes of COMP 2805), we have considered the problem of what can be computed by Turing machines (i.e., computers) and what cannot be computed. We did not, however, take the efficiency of the computations into account. In this chapter, we introduce a classification of decidable languages A, based on the running time of the "best" algorithm that decides A. That is, given a decidable language A, we are interested in the "fastest" algorithm that, for any given string w, decides whether or not $w \in A$.

7.1 The running time of algorithms

Let M be a Turing machine, and let w be an input string for M. We define the *running time* $t_M(w)$ of M on input w as

$$t_M(w) := \text{ the number of computation steps made by } M \text{ on input } w.$$

As usual, we denote by $|w|$, the number of symbols in the string w. We denote the set of non-negative integers by $\mathbb{N}_0$.

Definition 7.1.1 Let Σ be an alphabet, let $T : \mathbb{N}_0 \to \mathbb{N}_0$ be a function, let $A \subseteq \Sigma^*$ be a decidable language, and let $F : \Sigma^* \to \Sigma^*$ be a computable function.

- We say that the Turing machine M decides the language A *in time* T, if

  $$t_M(w) \leq T(|w|)$$

  for all strings w in Σ^*.

- We say that the Turing machine M computes the function F *in time* T, if

  $$t_M(w) \leq T(|w|)$$

  for all strings $w \in \Sigma^*$.

In other words, the "running time function" T is a function of the *length of the input*, which we usually denote by n. For any n, the value of $T(n)$ is an upper bound on the running time of the Turing machine M, on *any* input string of length n.

To give an example, consider the Turing machine of Section **??** that decides, using one tape, the language consisting of all palindromes. The tape head of this Turing machine moves from the

left to the right, then back to the left, then to the right again, back to the left, etc. Each time it reaches the leftmost or rightmost symbol, it deletes this symbol. The running time of this Turing machine, on any input string of length n, is

$$O(1+2+3+\ldots+n) = O(n^2).$$

On the other hand, the running time of the Turing machine of Section **??**, which also decides the palindromes, but using two tapes instead of just one, is $O(n)$.

In Section **??**, we mentioned that all computation models listed there are equivalent, in the sense that if a language can be decided in one model, it can be decided in any of the other models. We just saw, however, that the language consisting of all palindromes allows a faster algorithm on a two-tape Turing machine than on one-tape Turing machines. (Even though we did not prove this, it is true that $\Omega(n^2)$ is a lower bound on the running time to decide palindromes on a one-tape Turing machine.) The following theorem can be proved.

Theorem 7.1.2 *Let A be a language (resp. let F be a function) that can be decided (resp. computed) in time T by an algorithm of type M. Then there is an algorithm of type N that decides A (resp. computes F) in time T', where*

M	N	T'
k-tape Turing machine	*one-tape Turing machine*	$O(T^2)$
one-tape Turing machine	*Java program*	$O(T^2)$
Java program	*k-tape Turing machine*	$O(T^4)$

7.2 The complexity class P

Definition 7.2.1 We say that algorithm M decides the language A (resp. computes the function F) in *polynomial time*, if there exists an integer $k \geq 1$, such that the running time of M is $O(n^k)$, for any input string of length n.

It follows from Theorem 7.1.2 that this notion of "polynomial time" does not depend on the model of computation:

Theorem 7.2.2 *Consider the models of computation "Java program", "k-tape Turing machine", and "one-tape Turing machine". If a language can be decided (resp. a function can be computed) in polynomial time in one of these models, then it can be decided (resp. computed) in polynomial time in all of these models.*

Because of this theorem, we can define the following two complexity classes:

$$\mathbf{P} := \{A : \text{ the language } A \text{ is decidable in polynomial time}\},$$

and

$$\mathbf{FP} := \{F : \text{ the function } F \text{ is computable in polynomial time}\}.$$

7.2.1 Some examples

Palindromes

Let *Pal* be the language

$$Pal := \{w \in \{a,b\}^* : \ w \text{ is a palindrome}\}.$$

We have seen that there exists a one-tape Turing machine that decides *Pal* in $O(n^2)$ time. Therefore, $Pal \in \mathbf{P}$.

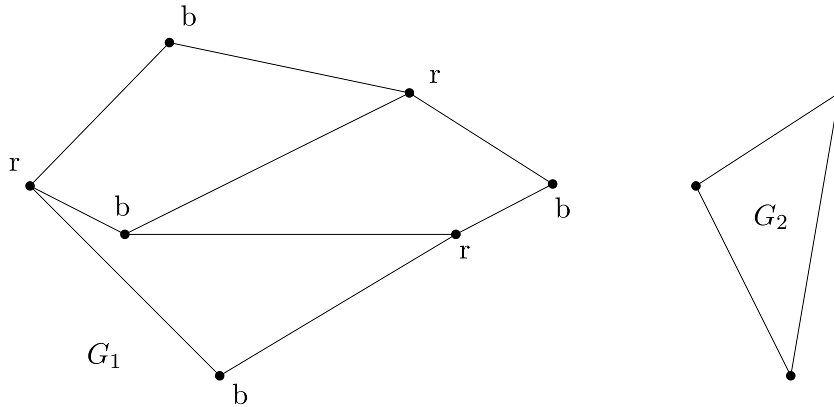


Figure 7.1: *The graph G_1 is 2-colorable;* r *stands for red;* b *stands for blue. The graph G_2 is not 2-colorable.*

Some functions in FP

The following functions are in the class **FP**:

- $F_1 : \mathbb{N}_0 \to \mathbb{N}_0$ defined by $F_1(x) := x + 1$,
- $F_2 : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $F_2(x, y) := x + y$,
- $F_3 : \mathbb{N}_0^2 \to \mathbb{N}_0$ defined by $F_3(x, y) := xy$.

Context-free languages

We have shown in Section **??** that every context-free language is decidable. The algorithm presented there, however, does not run in polynomial time. In COMP 3804, the following result will be shown (using a technique called *dynamic programming*):

Theorem 7.2.3 *Let Σ be an alphabet, and let $A \subseteq \Sigma^*$ be a context-free language. Then $A \in$* **P**.

Observe that, obviously, every language in **P** is decidable.

The 2-coloring problem

Let G be a graph with vertex set V and edge set E. We say that G is *2-colorable*, if it is possible to give each vertex of V a color such that

1. for each edge $(u, v) \in E$, the vertices u and v have different colors, and
2. only two colors are used to color all vertices.

See Figure 7.1 for an example. We define the following language:

$$2Color := \{\langle G \rangle : \text{ the graph } G \text{ is 2-colorable}\},$$

where $\langle G \rangle$ denotes the binary string that encodes the graph G.

We claim that $2Color \in \mathbf{P}$. In order to show this, we have to construct an algorithm that decides in polynomial time, whether or not any given graph is 2-colorable.

Let G be an arbitrary graph with vertex set $V = \{1, 2, \ldots, m\}$. The edge set of G is given by an *adjacency matrix*. This matrix, which we denote by E, is a two-dimensional array with m rows and m columns. For all i and j with $1 \leq i \leq m$ and $1 \leq j \leq m$, we have

$$E(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

The length of the input G, i.e., the number of bits needed to specify G, is equal to $m^2 =: n$. We will present an algorithm that decides, in $O(n)$ time, whether or not the graph G is 2-colorable.

The algorithm uses the colors red and blue. It gives the first vertex the color red. Then, the algorithm considers all vertices that are connected by an edge to the first vertex, and colors them blue. Now the algorithm is done with the first vertex; it marks this first vertex.

Next, the algorithm chooses a vertex i that already has a color, but that has not been marked. Then it considers all vertices j that are connected by an edge to i. If j has the same color as i, then the input graph G is not 2-colorable. Otherwise, if vertex j does not have a color yet, the algorithm gives j the color that is different from i's color. After having done this for all neighbors j of i, the algorithm is done with vertex i, so it marks i.

It may happen that there is no vertex i that already has a color but that has not been marked. (In other words, each vertex i that is not marked does not have a color yet.) In this case, the algorithm chooses an arbitrary vertex i having this property, and colors it red. (This vertex i is the first vertex in its connected component that gets a color.)

This procedure is repeated until all vertices of G have been marked.

We now give a formal description of this algorithm. Vertex i has been *marked*, if

1. i has a color,
2. all vertices that are connected by an edge to i have a color, and
3. the algorithm has verified that each vertex that is connected by an edge to i has a color different from i's color.

The algorithm uses two arrays $f(1 \ldots m)$ and $a(1 \ldots m)$, and a variable M. The value of $f(i)$ is equal to the color (red or blue) of vertex i; if i does not have a color yet, then $f(i) = 0$. The value of $a(i)$ is equal to

$$a(i) = \begin{cases} 1 & \text{if vertex } i \text{ has been marked,} \\ 0 & \text{otherwise.} \end{cases}$$

The value of M is equal to the number of marked vertices. The algorithm is presented in Figure 7.2. You are encouraged to convince yourself of the correctness of this algorithm. That is, you should convince yourself that this algorithm returns YES if the graph G is 2-colorable, whereas it returns NO otherwise.

What is the running time of this algorithm? First we count the number of iterations of the outer while-loop. In one iteration, either M increases by one, or a vertex i, for which $a(i) = 0$, gets the color red. In the latter case, the variable M is increased during the next iteration of the outer while-loop. Since, during the entire outer while-loop, the value of M is increased from zero to m, it follows that there are at most $2m$ iterations of the outer while-loop. (In fact, the number of iterations is equal to m plus the number of connected components of G minus one.)

One iteration of the outer while-loop takes $O(m)$ time. Hence, the total running time of the algorithm is $O(m^2)$, which is $O(n)$. Therefore, we have shown that $2\mathit{Color} \in \mathbf{P}$.

7.3 The complexity class NP

Before we define the class **NP**, we consider some examples.

```
Algorithm 2Color
for i := 1 to m do f(i) := 0; a(i) := 0 endfor;
f(1) := red; M := 0;
while M ≠ m
do (* Find the minimum index i for which vertex i has not
      been marked, but has a color already *)
   bool := false; i := 1;
   while bool = false and i ≤ m
   do if a(i) = 0 and f(i) ≠ 0 then bool := true else i := i + 1 endif;
   endwhile;
   (* If bool = true, then i is the smallest index such that
      a(i) = 0 and f(i) ≠ 0.
      If bool = false, then for all i, the following holds: if a(i) = 0, then
      f(i) = 0; because M < m, there is at least one such i. *)
   if bool = true
   then for j := 1 to m
        do if E(i, j) = 1
           then if f(i) = f(j)
                then return NO and terminate
                else if f(j) = 0
                     then if f(i) = red
                          then f(j) := blue
                          else f(j) := red
                          endif
                     endif
                endif
           endif
        endfor;
        a(i) := 1; M := M + 1;
   else i := 1;
        while a(i) ≠ 0 do i := i + 1 endwhile;
        (* an unvisited connected component starts at vertex i *)
        f(i) := red
   endif
endwhile;
return YES
```

Figure 7.2: *An algorithm that decides whether or not a graph G is 2-colorable.*

Example 7.3.1 Let G be a graph with vertex set V and edge set E, and let $k \geq 1$ be an integer. We say that G is *k-colorable*, if it is possible to give each vertex of V a color such that

1. for each edge $(u, v) \in E$, the vertices u and v have different colors, and
2. at most k different colors are used to color all vertices.

We define the following language:

$$k\mathit{Color} := \{\langle G \rangle : \text{ the graph } G \text{ is } k\text{-colorable}\}.$$

We have seen that for $k = 2$, this problem is in the class **P**. For $k \geq 3$, it is not known whether there exists an algorithm that decides, in polynomial time, whether or not any given graph is k-colorable. In other words, for $k \geq 3$, it is not known whether or not $k\mathit{Color}$ is in the class **P**.

Example 7.3.2 Let G be a graph with vertex set $V = \{1, 2, \ldots, m\}$ and edge set E. A *Hamilton cycle* is a cycle in G that visits each vertex exactly once. Formally, it is a sequence $v_1, v_2, \ldots, v_m$ of vertices such that

1. $\{v_1, v_2, \ldots, v_m\} = V$, and

2. $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v_m, v_1)\} \subseteq E$.

We define the following language:

$$HC := \{\langle G \rangle : \text{ the graph } G \text{ contains a Hamilton cycle}\}.$$

It is not known whether or not HC is in the class **P**.

Example 7.3.3 The *sum of subset* language is defined as follows:

$$SOS := \{\langle a_1, a_2, \ldots, a_m, b \rangle : \quad m, a_1, a_2, \ldots, a_m, b \in \mathbb{N}_0 \text{ and } \exists I \subseteq \{1, 2, \ldots, m\}, \textstyle\sum_{i \in I} a_i = b\}.$$

Also in this case, no polynomial-time algorithm is known that decides the language SOS. That is, it is not known whether or not SOS is in the class **P**.

Example 7.3.4 An integer $x \geq 2$ is a prime number, if there are no $a, b \in \mathbb{N}$ such that $a \neq x$, $b \neq x$, and $x = ab$. Hence, the language of all non-primes that are greater than or equal to two, is

$$NPrim := \{\langle x \rangle : \; x \geq 2 \text{ and } x \text{ is not a prime number}\}.$$

It is not obvious at all, whether or not $NPrim$ is in the class **P**. In fact, it was shown only in 2002 that $NPrim$ is in the class **P**.

Observation 7.3.5 *The four languages above have the following in common: If someone gives us a "solution" for any given input, then we can easily, i.e., in polynomial time, verify whether or not this "solution" is a correct solution. Moreover, for any input to each of these four problems, there exists a "solution" whose length is polynomial in the length of the input.*

Let us again consider the language $k\,Color$. Let G be a graph with vertex set $V = \{1, 2, \ldots, m\}$ and edge set E, and let k be a positive integer. We want to decide whether or not G is k-colorable. A "solution" is a coloring of the nodes using at most k different colors. That is, a solution is a sequence $f_1, f_2, \ldots, f_m$. (Interpret this as: vertex i receives color f_i, $1 \leq i \leq m$). This sequence is a correct solution if and only if

1. $f_i \in \{1, 2, \ldots, k\}$, for all i with $1 \leq i \leq m$, and
2. for all i with $1 \leq i \leq m$, and for all j with $1 \leq j \leq m$, if $(i, j) \in E$, then $f_i \neq f_j$.

If someone gives us this solution (i.e., the sequence $f_1, f_2, \ldots, f_m$), then we can verify in polynomial time whether or not these two conditions are satisfied. The length of this solution is $O(m \log k)$: for each i, we need about $\log k$ bits to represent f_i. Hence, the length of the solution is polynomial in the length of the input, i.e., it is polynomial in the number of bits needed to represent the graph G and the number k.

For the Hamilton cycle problem, a solution consists of a sequence $v_1, v_2, \ldots, v_m$ of vertices. This sequence is a correct solution if and only if

1. $\{v_1, v_2, \ldots, v_m\} = \{1, 2, \ldots, m\}$ and
2. $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v_m, v_1)\} \subseteq E$.

These two conditions can be verified in polynomial time. Moreover, the length of the solution is polynomial in the length of the input graph.

Consider the sum of subset problem. A solution is a sequence $c_1, c_2, \ldots, c_m$. It is a correct solution if and only if

1. $c_i \in \{0,1\}$, for all i with $1 \leq i \leq m$, and
2. $\sum_{i=1}^{m} c_i a_i = b$.

Hence, the set $I \subseteq \{1, 2, \ldots, m\}$ in the definition of *SOS* is the set of indices i for which $c_i = 1$. Again, these two conditions can be verified in polynomial time, and the length of the solution is polynomial in the length of the input.

Finally, let us consider the language *NPrim*. Let $x \geq 2$ be an integer. The integers a and b form a "solution" for x if and only if

1. $2 \leq a < x$,
2. $2 \leq b < x$, and
3. $x = ab$.

Clearly, these three conditions can be verified in polynomial time. Moreover, the length of this solution, i.e., the total number of bits in the binary representations of a and b, is polynomial in the number of bits in the binary representation of x.

Languages having the property that the correctness of a proposed "solution" can be verified in polynomial time, form the class **NP**:

Definition 7.3.6 A language A belongs to the class **NP**, if there exist a polynomial p and a language $B \in \mathbf{P}$, such that for every string w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B.$$

In words, a language A is in the class **NP**, if for every string w, $w \in A$ if and only if the following two conditions are satisfied:

1. There is a "solution" s, whose length $|s|$ is polynomial in the length of w (i.e., $|s| \leq p(|w|)$, where p is a polynomial).
2. In polynomial time, we can verify whether or not s is a correct "solution" for w (i.e., $\langle w, s \rangle \in B$ and $B \in \mathbf{P}$).

Hence, the language B can be regarded to be the "verification language":

$$B = \{\langle w, s \rangle : \text{ } s \text{ is a correct "solution" for } w\}.$$

We have given already informal proofs of the fact that the languages *kColor*, *HC*, *SOS*, and *NPrim* are all contained in the class **NP**. Below, we formally prove that $NPrim \in \mathbf{NP}$. To prove this claim, we have to specify the polynomial p and the language $B \in \mathbf{P}$. First, we observe that

$$NPrim = \{\langle x \rangle : \begin{array}{l} \text{there exist } a \text{ and } b \text{ in } \mathbb{N} \text{ such that} \\ 2 \leq a < x, 2 \leq b < x \text{ and } x = ab \ \}. \end{array} \tag{7.1}$$

We define the polynomial p by $p(n) := n + 2$, and the language B as

$$B := \{\langle x, a, b \rangle : x \geq 2, 2 \leq a < x, 2 \leq b < x \text{ and } x = ab\}.$$

It is obvious that $B \in \mathbf{P}$: For any three positive integers x, a, and b, we can verify in polynomial time whether or not $\langle x, a, b \rangle \in B$. In order to do this, we only have to verify whether or not $x \geq 2$, $2 \leq a < x$, $2 \leq b < x$, and $x = ab$. If all these four conditions are satisfied, then $\langle x, a, b \rangle \in B$. If at least one of them is not satisfied, then $\langle x, a, b \rangle \notin B$.

It remains to show that for all $x \in \mathbb{N}$:

$$\langle x \rangle \in NPrim \Longleftrightarrow \exists a, b : |\langle a, b \rangle| \leq |\langle x \rangle| + 2 \text{ and } \langle x, a, b \rangle \in B. \tag{7.2}$$

(Remember that $|\langle x \rangle|$ denotes the number of bits in the binary representation of x; $|\langle a, b \rangle|$ denotes the total number of bits of a and b, i.e., $|\langle a, b \rangle| = |\langle a \rangle| + |\langle b \rangle|$.)

Let $x \in NPrim$. It follows from (7.1) that there exist a and b in $\mathbb{N}$, such that $2 \leq a < x$, $2 \leq b < x$, and $x = ab$. Since $x = ab \geq 2 \cdot 2 = 4 \geq 2$, it follows that $\langle x, a, b \rangle \in B$. Hence, it remains to show that

$$|\langle a, b \rangle| \leq |\langle x \rangle| + 2.$$

The binary representation of x contains $\lfloor \log x \rfloor + 1$ bits, i.e., $|\langle x \rangle| = \lfloor \log x \rfloor + 1$. We have

$$\begin{aligned} |\langle a, b \rangle| &= |\langle a \rangle| + |\langle b \rangle| \\ &= (\lfloor \log a \rfloor + 1) + (\lfloor \log b \rfloor + 1) \\ &\leq \log a + \log b + 2 \\ &= \log ab + 2 \\ &= \log x + 2 \\ &\leq \lfloor \log x \rfloor + 3 \\ &= |\langle x \rangle| + 2. \end{aligned}$$

This proves one direction of (7.2).

To prove the other direction, we assume that there are positive integers a and b, such that $|\langle a, b \rangle| \leq |\langle x \rangle| + 2$ and $\langle x, a, b \rangle \in B$. Then it follows immediately from (7.1) and the definition of the language B, that $x \in NPrim$. Hence, we have proved the other direction of (7.2). This completes the proof of the claim that

$$NPrim \in \mathbf{NP}.$$

7.3.1 P is contained in NP

Intuitively, it is clear that $\mathbf{P} \subseteq \mathbf{NP}$, because a language is

- in **P**, if for every string w, it is possible to *compute* the "solution" s in polynomial time,
- in **NP**, if for every string w and for any given "solution" s, it is possible to *verify* in polynomial time whether or not s is a correct solution for w (hence, we do not need to compute the solution s ourselves, we only have to verify it).

We give a formal proof of this:

Theorem 7.3.7 $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Let $A \in \mathbf{P}$. We will prove that $A \in \mathbf{NP}$. Define the polynomial p by $p(n) := 0$ for all $n \in \mathbb{N}_0$, and define

$$B := \{\langle w, \epsilon \rangle : w \in A\}.$$

Since $A \in \mathbf{P}$, the language B is also contained in **P**. It is easy to see that

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) = 0 \text{ and } \langle w, s \rangle \in B.$$

This completes the proof. ∎

```
Algorithm NonPrime
(* decides whether or not ⟨x⟩ ∈ NPrim *)
if x = 0 or x = 1 or x = 2
then return NO and terminate
else a := 2;
     while a < x
     do if x mod a = 0
          then return YES and terminate
          else a := a + 1
          endif
     endwhile;
     return NO
endif
```

Figure 7.3: *An algorithm that decides whether or not a number x is contained in the language NPrim.*

7.3.2 Deciding NP-languages in exponential time

Let us look again at the definition of the class **NP**. Let A be a language in this class. Then there exist a polynomial p and a language $B \in \mathbf{P}$, such that for all strings w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B. \tag{7.3}$$

How do we decide whether or not any given string w belongs to the language A? If we can find a string s that satisfies the right-hand side in (7.3), then we know that $w \in A$. On the other hand, if there is no such string s, then $w \notin A$. How much time do we need to decide whether or not such a string s exists?

For example, let A be the language *NPrim*, and let $x \in \mathbb{N}$. The algorithm in Figure 7.3 decides whether or not $\langle x \rangle \in NPrim$.

It is clear that this algorithm is correct. Let n be the length of the binary representation of x, i.e., $n = \lfloor \log x \rfloor + 1$. If $x > 2$ and x is a prime number, then the while-loop makes $x - 2$ iterations. Therefore, since $n - 1 = \lfloor \log x \rfloor \leq \log x$, the running time of this algorithm is at least

$$x - 2 \geq 2^{n-1} - 2,$$

i.e., it is at least *exponential* in the length of the input.

We now prove that every language in **NP** can be decided in exponential time. Let A be an arbitrary language in **NP**. Let p be the polynomial, and let $B \in \mathbf{P}$ be the language such that for all strings w,

$$w \in A \Longleftrightarrow \exists s : |s| \leq p(|w|) \text{ and } \langle w, s \rangle \in B. \tag{7.4}$$

The following algorithm decides, for any given string w, whether or not $w \in A$. It does so by looking at *all* possible strings s for which $|s| \leq p(|w|)$:

```
for all s with |s| ≤ p(|w|)
do if ⟨w, s⟩ ∈ B
   then return YES and terminate
   endif
endfor;
return NO
```

The correctness of the algorithm follows from (7.4). What is the running time? We assume that w and s are represented as binary strings. Let n be the length of the input, i.e., $n = |w|$.

How many binary strings s are there whose length is at most $p(|w|)$? Any such s can be described by a sequence of length $p(|w|) = p(n)$, consisting of the symbols "0", "1", and the blank symbol. Hence, there are at most $3^{p(n)}$ many binary strings s with $|s| \leq p(n)$. Therefore, the for-loop makes at most $3^{p(n)}$ iterations.

Since $B \in \mathbf{P}$, there is an algorithm and a polynomial q, such that this algorithm, when given any input string z, decides in $q(|z|)$ time, whether or not $z \in B$. This input z has the form $\langle w, s \rangle$, and we have

$$|z| = |w| + |s| \leq |w| + p(|w|) = n + p(n).$$

It follows that the total running time of our algorithm that decides whether or not $w \in A$, is bounded from above by

$$\begin{aligned} 3^{p(n)} \cdot q(n + p(n)) &\leq 2^{2p(n)} \cdot q(n + p(n)) \\ &\leq 2^{2p(n)} \cdot 2^{q(n+p(n))} \\ &= 2^{p'(n)}, \end{aligned}$$

where p' is the polynomial that is defined by $p'(n) := 2p(n) + q(n + p(n))$.

If we define the class **EXP** as

$$\mathbf{EXP} := \{A : \text{there exists a polynomial } p \text{, such that } A \text{ can be decided in time } 2^{p(n)}\},$$

then we have proved the following theorem.

Theorem 7.3.8 **NP** $\subseteq$ **EXP**.

7.3.3 Summary

- **P** $\subseteq$ **NP**. It is not known whether **P** is a *proper* subclass of **NP**, or whether **P** $=$ **NP**. This is one of the most important open problems in computer science. If you can solve this problem, then you will get one million dollars; not from us, but from the Clay Mathematics Institute, see

  `http://www.claymath.org/prizeproblems/index.htm`

  Most people believe that **P** is a proper subclass of **NP**.

- **NP** $\subseteq$ **EXP**, i.e., each language in **NP** can be decided in exponential time. It is not known whether **NP** is a proper subclass of **EXP**, or whether **NP** $=$ **EXP**.

- It follows from **P** $\subseteq$ **NP** and **NP** $\subseteq$ **EXP**, that **P** $\subseteq$ **EXP**. It can be shown that **P** is a proper subset of **EXP**, i.e., there exist languages that can be decided in exponential time, but that cannot be decided in polynomial time.

- **P** is the class of those languages that can be decided *efficiently*, i.e., in polynomial time. Sets that are not in **P**, are not efficiently decidable.

7.4 Non-deterministic algorithms

The abbreviation **NP** stands for Non-deterministic Polynomial time. The algorithms that we have considered so far are *deterministic*, which means that at any time during the computation, the next computation step is uniquely determined. In a *non-deterministic* algorithm, there are one or more possibilities for being the next computation step, and the algorithm chooses one of them.

To give an example, we consider the language *SOS*, see Example 7.3.3. Let m, a_1, a_2, $\ldots, a_m$, and b be elements of $\mathbb{N}_0$. Then

$$\langle a_1, a_2, \ldots, a_m, b \rangle \in SOS \iff \text{there exist } c_1, c_2, \ldots, c_m \in \{0,1\}, \text{ such that } \sum_{i=1}^{m} c_i a_i = b.$$

The following non-deterministic algorithm decides the language *SOS*:

Algorithm SOS$(m, a_1, a_2, \ldots, a_m, b)$:
$s := 0$;
for $i := 1$ **to** m
do $s := s \mid s := s + a_i$
endfor;
if $s = b$
then return YES
else return NO
endif

The line

$$s := s \mid s := s + a_i$$

means that either the instruction "$s := s$" or the instruction "$s := s + a_i$" is executed.

Let us assume that $\langle a_1, a_2, \ldots, a_m, b \rangle \in SOS$. Then there are $c_1, c_2, \ldots, c_m \in \{0,1\}$ such that $\sum_{i=1}^{m} c_i a_i = b$. Assume our algorithm does the following, for each i with $1 \leq i \leq m$: In the i-th iteration,

- if $c_i = 0$, then it executes the instruction "$s := s$",
- if $c_i = 1$, then it executes the instruction "$s := s + a_i$".

Then after the for-loop, we have $s = b$, and the algorithm returns YES; hence, the algorithm has correctly found out that $\langle a_1, a_2, \ldots, a_m, b \rangle \in SOS$. In other words, in this case, *there exists at least one accepting computation.*

On the other hand, if $\langle a_1, a_2, \ldots, a_m, b \rangle \notin SOS$, then the algorithm always returns NO, no matter which of the two instructions is executed in each iteration of the for-loop. In this case, there is *no accepting computation.*

Definition 7.4.1 Let M be a non-deterministic algorithm. We say that M *accepts* a string w, if there exists at least one computation that, on input w, returns YES.

Definition 7.4.2 We say that a non-deterministic algorithm M *decides* a language A *in time* T, if for every string w, the following holds: $w \in A$ if and only if there exists at least one computation that, on input w, returns YES and that takes at most $T(|w|)$ time.

The non-deterministic algorithm that we have seen above decides the language *SOS* in linear time: Let $\langle a_1, a_2, \ldots, a_m, b \rangle \in SOS$, and let n be the length of this input. Then

$$n = |\langle a_1 \rangle| + |\langle a_2 \rangle| + \ldots + |\langle a_m \rangle| + |\langle b \rangle| \geq m.$$

For this input, there is a computation that returns YES and that takes $O(m) = O(n)$ time.

As in Section 7.2, we define the notion of "polynomial time" for non-deterministic algorithms. The following theorem relates this notion to the class **NP** that we defined in Definition 7.3.6.

Theorem 7.4.3 *A language A is in the class* **NP** *if and only if there exists a non-deterministic Turing machine (or Java program) that decides A in polynomial time.*

7.5 NP-complete languages

Languages in the class **P** are considered *easy*, i.e., they can be decided in polynomial time. People believe (but cannot prove) that **P** is a proper subclass of **NP**. If this is true, then there are languages in **NP** that are *hard*, i.e., cannot be decided in polynomial time.

Intuition tells us that if $\mathbf{P} \neq \mathbf{NP}$, then the *hardest* languages in **NP** are not contained in **P**. These languages are called **NP**-*complete*. In this section, we will give a formal definition of this concept.

If we want to talk about the "hardest" languages in **NP**, then we have to be able to compare two languages according to their "difficulty". The idea is as follows: We say that a language B is "at least as hard" as a language A, if the following holds: If B can be decided in polynomial time, then A can also be decided in polynomial time.

Definition 7.5.1 Let $A \subseteq \{0,1\}^*$ and $B \subseteq \{0,1\}^*$ be languages. We say that $A \leq_P B$, if there exists a function

$$f : \{0,1\}^* \rightarrow \{0,1\}^*$$

such that

1. $f \in \mathbf{FP}$ and
2. for all strings w in $\{0,1\}^*$,
$$w \in A \Longleftrightarrow f(w) \in B.$$

If $A \leq_P B$, then we also say that "B is at least as hard as A", or "A is polynomial-time reducible to B".

We first show that this formal definition is in accordance with the intuitive definition given above.

Theorem 7.5.2 *Let A and B be languages such that $B \in \mathbf{P}$ and $A \leq_P B$. Then $A \in \mathbf{P}$.*

Proof. Let $f : \{0,1\}^* \rightarrow \{0,1\}^*$ be the function in **FP** for which

$$w \in A \Longleftrightarrow f(w) \in B. \tag{7.5}$$

The following algorithm decides whether or not any given binary string w is in A:

$u := f(w)$;
if $u \in B$
then return YES
else return NO
endif

The correctness of this algorithm follows immediately from (7.5). So it remains to show that the running time is polynomial in the length of the input string w.

Since $f \in \mathbf{FP}$, there exists a polynomial p such that the function f can be computed in time p. Similarly, since $B \in \mathbf{P}$, there exists a polynomial q, such that the language B can be decided in time q.

Let n be the length of the input string w, i.e., $n = |w|$. Then the length of the string u is less than or equal to $p(|w|) = p(n)$. (Why?) Therefore, the running time of our algorithm is bounded from above by

$$p(|w|) + q(|u|) \leq p(n) + q(p(n)).$$

Since the function p', defined by $p'(n) := p(n) + q(p(n))$, is a polynomial, this proves that $A \in \mathbf{P}$. ∎

The following theorem states that the relation $\leq_P$ is reflexive and transitive. We leave the proof as an exercise.

Theorem 7.5.3 *Let A, B, and C be languages. Then*

1. *$A \leq_P A$, and*
2. *if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.*

We next show that the languages in **P** are the *easiest* languages in **NP**:

Theorem 7.5.4 *Let A be a language in* **P**, *and let B be an arbitrary language such that $B \neq \emptyset$ and $B \neq \{0,1\}^*$. Then $A \leq_P B$.*

Proof. We choose two strings u and v in $\{0,1\}^*$, such that $u \in B$ and $v \notin B$. (Observe that this is possible.) Define the function $f : \{0,1\}^* \to \{0,1\}^*$ by

$$f(w) := \begin{cases} u & \text{if } w \in A, \\ v & \text{if } w \notin A. \end{cases}$$

Then it is clear that for any binary string w,

$$w \in A \Longleftrightarrow f(w) \in B.$$

Since $A \in \mathbf{P}$, the function f can be computed in polynomial time, i.e., $f \in \mathbf{FP}$. ∎

7.5.1 Two examples of reductions

Sum of subsets and knapsacks

We start with a simple reduction. Consider the two languages

$$SOS := \{\langle a_1, \ldots, a_m, b\rangle : \begin{array}{l} m, a_1, \ldots, a_m, b \in \mathbb{N}_0 \text{ and there exist} \\ c_1, \ldots, c_m \in \{0,1\}, \text{ such that } \sum_{i=1}^m c_i a_i = b\} \end{array}$$

and

$$\begin{aligned} KS := \; & \{\langle w_1, \ldots, w_m, k_1, \ldots, k_m, W, K\rangle : \\ & m, w_1, \ldots, w_m, k_1, \ldots, k_m, W, K \in \mathbb{N}_0 \\ & \text{and there exist } c_1, \ldots, c_m \in \{0,1\}, \\ & \text{such that } \textstyle\sum_{i=1}^m c_i w_i \leq W \text{ and } \sum_{i=1}^m c_i k_i \geq K\}. \end{aligned}$$

The notation KS stands for *knapsack*: We have m pieces of food. The i-th piece has weight w_i and contains k_i calories. We want to decide whether or not we can fill our knapsack with a subset of the pieces of food such that the total weight is at most W, and the total amount of calories is at least K.

Theorem 7.5.5 *$SOS \leq_P KS$.*

Proof. Let us first see what we have to show. According to Definition 7.5.1, we need a function $f \in \mathbf{FP}$, that maps input strings for *SOS* to input strings for *KS*, in such a way that

$$\langle a_1, \ldots, a_m, b \rangle \in SOS \Longleftrightarrow f(\langle a_1, \ldots, a_m, b \rangle) \in KS.$$

In order for $f(\langle a_1, \ldots, a_m, b \rangle)$ to be an input string for *KS*, this function value has to be of the form

$$f(\langle a_1, \ldots, a_m, b \rangle) = \langle w_1, \ldots, w_m, k_1, \ldots, k_m, W, K \rangle.$$

We define

$$f(\langle a_1, \ldots, a_m, b \rangle) := \langle a_1, \ldots, a_m, a_1, \ldots, a_m, b, b \rangle.$$

It is clear that $f \in \mathbf{FP}$. We have

$$\begin{aligned}
& \langle a_1, \ldots, a_m, b \rangle \in SOS \\
\Longleftrightarrow \quad & \text{there exist } c_1, \ldots, c_m \in \{0,1\} \text{ such that } \textstyle\sum_{i=1}^m c_i a_i = b \\
\Longleftrightarrow \quad & \text{there exist } c_1, \ldots, c_m \in \{0,1\} \text{ such that } \textstyle\sum_{i=1}^m c_i a_i \leq b \text{ and } \textstyle\sum_{i=1}^m c_i a_i \geq b \\
\Longleftrightarrow \quad & \langle a_1, \ldots, a_m, a_1, \ldots, a_m, b, b \rangle \in KS \\
\Longleftrightarrow \quad & f(\langle a_1, \ldots, a_m, b \rangle) \in KS.
\end{aligned}$$

∎

Cliques and Boolean formulas

We will define two languages $A = 3SAT$ and $B = Clique$ that have, at first sight, nothing to do with each other. Then we show that, nevertheless, $A \leq_P B$.

Let G be a graph with vertex set V and edge set E. A subset V' of V is called a *clique*, if each pair of distinct vertices in V' is connected by an edge in E. We define the following language:

$$Clique := \{\langle G, k \rangle : k \in \mathbb{N} \text{ and } G \text{ has a clique with } k \text{ vertices}\}.$$

We encourage you to prove the following claim:

Theorem 7.5.6 *Clique* $\in$ **NP**.

Next we consider *Boolean formulas* φ, with variables $x_1, x_2, \ldots, x_m$, having the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{7.6}$$

where each C_i, $1 \leq i \leq k$, is of the form

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i.$$

Each ℓ_a^i is either a variable or the negation of a variable. An example of such a formula is

$$\varphi = (x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4).$$

A formula φ of the form (7.6) is said to be *satisfiable*, if there exists a truth-value in $\{0,1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. Our example formula is satisfiable: If we take $x_1 = 0$ and $x_2 = 1$, and give x_3 and x_4 an arbitrary value, then

$$\varphi = (0 \vee 1 \vee 0) \wedge (x_3 \vee 1 \vee x_4) \wedge (1 \vee \neg x_3 \vee \neg x_4) = 1.$$

We define the following language:

$$3SAT := \{\langle \varphi \rangle : \varphi \text{ is of the form (7.6) and is satisfiable}\}.$$

Again, we encourage you to prove the following claim:

Theorem 7.5.7 *$3SAT \in$ **NP**.*

Observe that the elements of *Clique* (which are pairs consisting of a graph and a positive integer) are completely different from the elements of $3SAT$ (which are Boolean formulas). We will show that $3SAT \leq_P$ *Clique*. Recall that this means the following: If the language *Clique* can be decided in polynomial time, then the language $3SAT$ can also be decided in polynomial time. In other words, any polynomial-time algorithm that decides *Clique* can be converted to a polynomial-time algorithm that decides $3SAT$.

Theorem 7.5.8 *$3SAT \leq_P$ Clique.*

Proof. We have to show that there exists a function $f \in$ **FP**, that maps input strings for $3SAT$ to input strings for *Clique*, such that for each Boolean formula φ that is of the form (7.6),

$$\langle \varphi \rangle \in 3SAT \Longleftrightarrow f(\langle \varphi \rangle) \in \textit{Clique}.$$

The function f maps the binary string encoding an arbitrary Boolean formula φ to a binary string encoding a pair (G, k), where G is a graph and k is a positive integer. We have to define this function f in such a way that

$$\varphi \text{ is satisfiable} \Longleftrightarrow G \text{ has a clique with } k \text{ vertices.}$$

Let

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

be an arbitrary Boolean formula in the variables $x_1, x_2, \ldots, x_m$, where each C_i, $1 \leq i \leq k$, is of the form

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i.$$

Remember that each ℓ_a^i is either a variable or the negation of a variable.

The formula φ is mapped to the pair (G, k), where the vertex set V and the edge set E of the graph G are defined as follows:

- $V = \{v_1^1, v_2^1, v_3^1, \ldots, v_1^k, v_2^k, v_3^k\}$. The idea is that each vertex v_a^i corresponds to one term ℓ_a^i.
- The pair (v_a^i, v_b^j) of vertices form an edge in E if and only if
  - $i \neq j$ and
  - ℓ_a^i is not the negation of ℓ_b^j.

To give an example, let φ be the Boolean formula

$$\varphi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3), \tag{7.7}$$

i.e., $k = 3$, $C_1 = x_1 \vee \neg x_2 \vee \neg x_3$, $C_2 = \neg x_1 \vee x_2 \vee x_3$, and $C_3 = x_1 \vee x_2 \vee x_3$. The graph G that corresponds to this formula is given in Figure 7.4.

It is not difficult to see that the function f can be computed in polynomial time. So it remains to prove that

$$\varphi \text{ is satisfiable} \Longleftrightarrow G \text{ has a clique with } k \text{ vertices.} \tag{7.8}$$

To prove this, we first assume that the formula

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

is satisfiable. Then there exists a truth-value in $\{0, 1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. Hence, for each i with $1 \leq i \leq k$, there is at least one term ℓ_a^i in

$$C_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$$

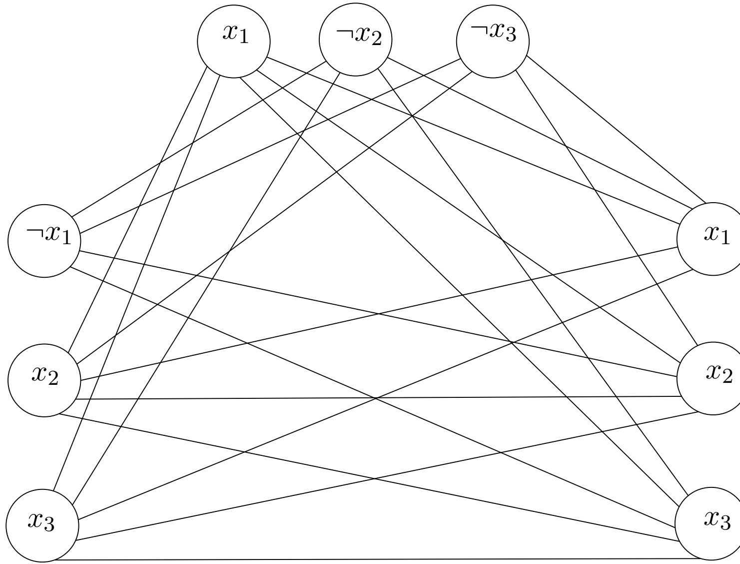


Figure 7.4: *The formula φ in (7.7) is mapped to this graph. The vertices on the top represent C_1; the vertices on the left represent C_2; the vertices on the right represent C_3.*

---

that is true (i.e., has value 1).

Let V' be the set of vertices obtained by choosing for each i, $1 \leq i \leq k$, exactly one vertex v_a^i such that ℓ_a^i has value 1.

It is clear that V' contains exactly k vertices. We claim that this set is a clique in G. To prove this claim, let v_a^i and v_b^j be two distinct vertices in V'. It follows from the definition of V' that $i \neq j$ and $\ell_a^i = \ell_b^j = 1$. Hence, ℓ_a^i is not the negation of ℓ_b^j. But this means that the vertices v_a^i and v_b^j are connected by an edge in G.

This proves one direction of (7.8). To prove the other direction, we assume that the graph G contains a clique V' with k vertices.

The vertices of G consist of k groups, where each group contains exactly three vertices. Since vertices within the same group are not connected by edges, the clique V' contains exactly one vertex from each group. Hence, for each i with $1 \leq i \leq k$, there is exactly one a, such that $v_a^i \in V'$. Consider the corresponding term ℓ_a^i. We know that this term is either a variable or the negation of a variable, i.e., ℓ_a^i is either of the form x_j or of the form $\neg x_j$. If $\ell_a^i = x_j$, then we give x_j the truth-value 1. Otherwise, we have $\ell_a^i = \neg x_j$, in which case we give x_j the truth-value 0. Since V' is a clique, each variable gets at most one truth-value. If a variable has no truth-value yet, then we give it an arbitrary truth-value.

If we substitute these truth-values into φ, then the entire formula has value 1. Hence, φ is satisfiable. ∎

In order to get a better understanding of this proof, you should verify the proof for the formula φ in (7.7) and the graph G in Figure 7.4.

7.5.2 Definition of NP-completeness

Reductions, as defined in Definition 7.5.1, allow us to compare two language according to their difficulty. A language B in **NP** is called **NP**-complete, if B belongs to the *most difficult* languages in **NP**; in other words, B is at least as hard as *any* other language in **NP**.

Definition 7.5.9 Let $B \subseteq \{0,1\}^*$ be a language. We say that B is **NP**-*complete*, if

1. $B \in \mathbf{NP}$ and

2. $A \leq_P B$, for *every* language A in **NP**.

Theorem 7.5.10 *Let B be an* **NP***-complete language. Then*

$$B \in \mathbf{P} \Longleftrightarrow \mathbf{P} = \mathbf{NP}.$$

Proof. Intuitively, this theorem should be true: If the language B is in **P**, then B is an easy language. On the other hand, since B is **NP**-complete, it belongs to the most difficult languages in **NP**. Hence, the most difficult language in **NP** is easy. But then all languages in **NP** must be easy, i.e., $\mathbf{P} = \mathbf{NP}$.

We give a formal proof. Let us first assume that $B \in \mathbf{P}$. We already know that $\mathbf{P} \subseteq \mathbf{NP}$. Hence, it remains to show that $\mathbf{NP} \subseteq \mathbf{P}$. Let A be an arbitrary language in **NP**. Since B is **NP**-complete, we have $A \leq_P B$. Then, by Theorem 7.5.2, we have $A \in \mathbf{P}$.

To prove the converse, assume that $\mathbf{P} = \mathbf{NP}$. Since $B \in \mathbf{NP}$, it follows immediately that $B \in \mathbf{P}$. ∎

Theorem 7.5.11 *Let B and C be languages, such that $C \in$* **NP** *and $B \leq_P C$. If B is* **NP***-complete, then C is also* **NP***-complete.*

Proof. First, we give an intuitive explanation of the claim: By assumption, B belongs to the most difficult languages in **NP**, and C is at least as hard as B. Since $C \in \mathbf{NP}$, it follows that C belongs to the most difficult languages in **NP**. Hence, C is **NP**-complete.

To give a formal proof, we have to show that $A \leq_P C$, for all languages A in **NP**. Let A be an arbitrary language in **NP**. Since B is **NP**-complete, we have $A \leq_P B$. Since $B \leq_P C$, it follows from Theorem 7.5.3, that $A \leq_P C$. Therefore, C is **NP**-complete. ∎

Theorem 7.5.11 can be used to prove the **NP**-completeness of languages: Let C be a language, and assume that we want to prove that C is **NP**-complete. We can do this in the following way:

1. We first prove that $C \in \mathbf{NP}$.

2. Then we find a language B that looks "similar" to C, and for which we already know that it is **NP**-complete.

3. Finally, we prove that $B \leq_P C$.

4. Then, Theorem 7.5.11 tells us that C is **NP**-complete.

Of course, this leads to the question "How do we know that the language B is **NP**-complete?" In order to apply Theorem 7.5.11, we need a "first" **NP**-complete language; the **NP**-completeness of this language must be proven using Definition 7.5.9.

Observe that it is not clear at all that there exist **NP**-complete languages! For example, consider the language $3SAT$. If we want to use Definition 7.5.9 to show that this language is **NP**-complete, then we have to show that

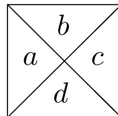
- $3SAT \in \mathbf{NP}$. We know from Theorem 7.5.7 that this is true.

- $A \leq_P 3SAT$, for *every* language $A \in \mathbf{NP}$. Hence, we have to show this for languages A such as $kColor$, HC, SOS, *NPrim*, KS, *Clique*, and for *infinitely* many other languages.

In 1973, Cook has exactly done this: He showed that the language $3SAT$ is **NP**-complete. Since his proof is rather technical, we will prove the **NP**-completeness of another language.

7.5.3 An NP-complete domino game

We are given a finite collection of *tile types*. For each such type, there are arbitrarily many tiles of this type. A *tile* is a square that is partitioned into four triangles. Each of these triangles contains a symbol that belongs to a finite alphabet Σ. Hence, a tile looks as follows:



We are also given a square *frame*, consisting of cells. Each cell has the same size as a tile, and contains a symbol of Σ.

The problem is to decide whether or not this *domino game* has a solution. That is, can we completely fill the frame with tiles such that

- for any two neighboring tiles s and s', the two triangles of s and s' that touch each other contain the same symbol, and
- each triangle that touches the frame contains the same symbol as the cell of the frame that is touched by this triangle.

There is one final restriction: The orientation of the tiles is fixed, they cannot be rotated.

Let us give a formal definition of this problem. We assume that the symbols belong to the finite alphabet $\Sigma = \{0,1\}^m$, i.e., each symbol is encoded as a bit-string of length m. Then, a tile type can be encoded as a tuple of four bit-strings, i.e., as an element of Σ^4. A frame consisting of t rows and t columns can be encoded as a string in Σ^{4t}.

We denote the language of all solvable domino games by *Domino*:

$$\begin{aligned} \textit{Domino} \;:=\; & \{\langle m,k,t,R,T_1,\ldots,T_k\rangle : \\ & m \geq 1, k \geq 1, t \geq 1, R \in \Sigma^{4t}, T_i \in \Sigma^4, 1 \leq i \leq k, \\ & \text{frame } R \text{ can be filled using tiles of types} \\ & T_1,\ldots,T_k.\} \end{aligned}$$

We will prove the following theorem.

Theorem 7.5.12 *The language Domino is* **NP***-complete.*

Proof. It is clear that *Domino* $\in$ **NP**: A solution consists of a $t \times t$ matrix, in which the (i,j)-entry indicates the type of the tile that occupies position (i,j) in the frame. The number of bits needed to specify such a solution is polynomial in the length of the input. Moreover, we can verify in polynomial time whether or not any given "solution" is correct.

It remains to show that

$$A \leq_P \textit{Domino}, \text{ for every language } A \text{ in } \mathbf{NP}.$$

Let A be an arbitrary language in **NP**. Then there exist a polynomial p and a non-deterministic Turing machine M, that decides the language A in time p. We may assume that this Turing machine has only one tape.

On input $w = a_1 a_2 \ldots a_n$, the Turing machine M starts in the start state z_0, with its tape head on the cell containing the symbol a_1. We may assume that during the entire computation, the tape head never moves to the left of this initial cell. Hence, the entire computation "takes place" in and to the right of the initial cell. We know that

$w \in A \iff$ on input w, there exists an accepting computation that makes at most $p(n)$ computation steps.

At the end of such an accepting computation, the tape only contains the symbol 1, which we may assume to be in the initial cell, and M is in the final state z_1. In this case, we may assume that the accepting computation makes exactly $p(n)$ computation steps. (If this is not the case, then we extend the computation using the instruction $z_1 1 \rightarrow z_1 1 N$.)

We need one more technical detail: We may assume that $za \rightarrow z'bR$ and $za' \rightarrow z''b'L$ are not both instructions of M. Hence, the state of the Turing machine uniquely determines the direction in which the tape head moves.

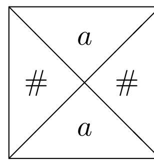
We have to define a domino game, that depends on the input string w and the Turing machine M, such that

$$w \in A \iff \text{this domino game is solvable.}$$

The idea is to encode an accepting computation of the Turing machine M as a solution of the domino game. In order to do this, we use a frame in which each row corresponds to one computation step. This frame consists of $p(n)$ rows. Since an accepting computation makes exactly $p(n)$ computation steps, and since the tape head never moves to the left of the initial cell, this tape head can visit only $p(n)$ cells. Therefore, our frame will have $p(n)$ columns.

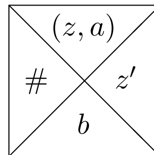
The domino game will use the following tile types:

1. For each symbol a in the alphabet of the Turing machine M:

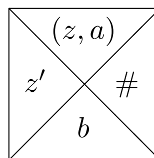
   Intuition: Before and after the computation step, the tape head is not on this cell.

2. For each instruction $za \rightarrow z'bR$ of the Turing machine M:

   Intuition: Before the computation step, the tape head is on this cell; the tape head makes one step to the right.

3. For each instruction $za \rightarrow z'bL$ of the Turing machine M:

   Intuition: Before the computation step, the tape head is on this cell; the tape head makes one step to the left.

4. For each instruction $za \rightarrow z'bN$ of the Turing machine M:

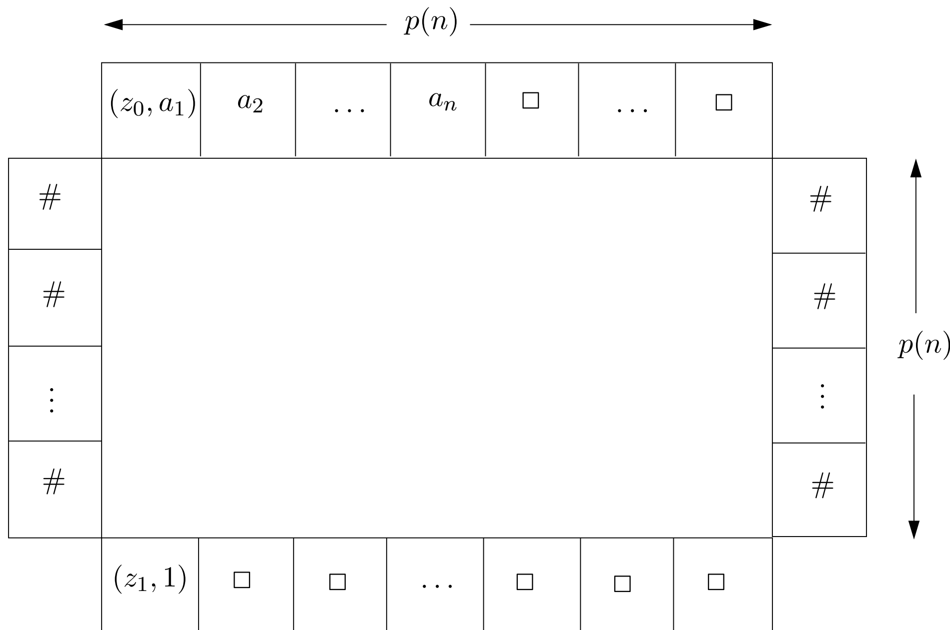
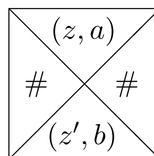
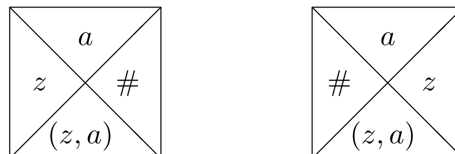


Figure 7.5: *The $p(n) \times p(n)$ frame of the domino game.*



Intuition: Before and after the computation step, the tape head is on this cell.

5. For each state z and for each symbol a in the alphabet of the Turing machine M, there are two tile types:



Intuition: The leftmost tile indicates that the tape head enters this cell from the left; the righmost tile indicates that the tape head enters this cell from the right.

This specifies all tile types. The $p(n)\times p(n)$ frame is given in Figure 7.5. The top row corresponds to the start of the computation, whereas the bottom row corresponds to the end of the computation. The left and right columns correspond to the part of the tape in which the tape head can move.

The encodings of these tile types and the frame can be computed in polynomial time.

It can be shown that, for any input string w, any accepting computation of length $p(n)$ of the Turing machine M can be encoded as a solution of this domino game. Conversely, any solution of this domino game can be "translated" to an accepting computation of length $p(n)$ of M, on input string w. Hence, the following holds.

$$\begin{aligned} w \in A \iff & \text{ there exists an accepting computation that makes} \\ & \; p(n) \text{ computation steps} \\ \iff & \text{ the domino game is solvable.} \end{aligned}$$

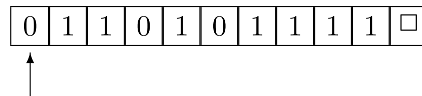
Therefore, we have $A \leq_P$ *Domino*. Hence, the language *Domino* is **NP**-complete. ∎

An example of a domino game

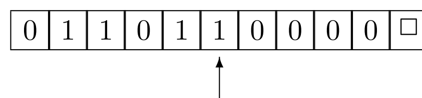
We have defined the domino game corresponding to a Turing machine that solves a decision problem. Of course, we can also do this for Turing machines that compute functions. In this section, we will exactly do this for a Turing machine that computes the successor function $x \rightarrow x+1$.

We will design a Turing machine with one tape, that gets as input the binary representation of a natural number x, and that computes the binary representation of $x+1$.

Start of the computation: The tape contains a 0 followed by the binary representation of the integer $x \in \mathbb{N}_0$. The tape head is on the leftmost bit (which is 0), and the Turing machine is in the start state z_0. Here is an example, where $x = 431$:



End of the computation: The tape contains the binary representation of the number $x+1$. The tape head is on the rightmost 1, and the Turing machine is in the final state z_1. For our example, the tape looks as follows:



Our Turing machine will use the following states:

z_0 : start state; tape head moves to the right
z_1 : final state
z_2 : tape head moves to the left; on its way to the left, it has not read 0

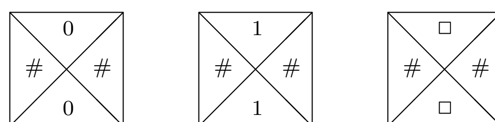
The Turing machine has the following instructions:

$$z_0 0 \rightarrow z_0 0R \qquad z_2 1 \rightarrow z_2 0L$$
$$z_0 1 \rightarrow z_0 1R \qquad z_2 0 \rightarrow z_1 1N$$
$$z_0 \square \rightarrow z_2 \square L$$

In Figure 7.6, you can see the sequence of states and tape contents of this Turing machine on input $x = 11$.

We now construct the domino game that corresponds to the computation of this Turing machine on input $x = 11$. Following the general construction in Section 7.5.3, we obtain the following tile types:

1. The three symbols of the alphabet yield three tile types:

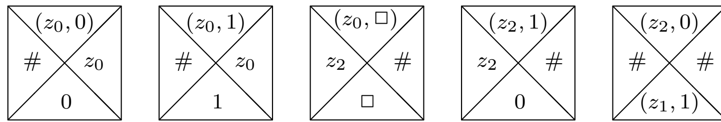


$(z_0, 0)$	1	0	1	1	$\square$
0	$(z_0, 1)$	0	1	1	$\square$
0	1	$(z_0, 0)$	1	1	$\square$
0	1	0	$(z_0, 1)$	1	$\square$
0	1	0	1	$(z_0, 1)$	$\square$
0	1	0	1	1	$(z_0, \square)$
0	1	0	1	$(z_2, 1)$	$\square$
0	1	0	$(z_2, 1)$	0	$\square$
0	1	$(z_2, 0)$	0	0	$\square$
0	1	$(z_1, 1)$	0	0	$\square$

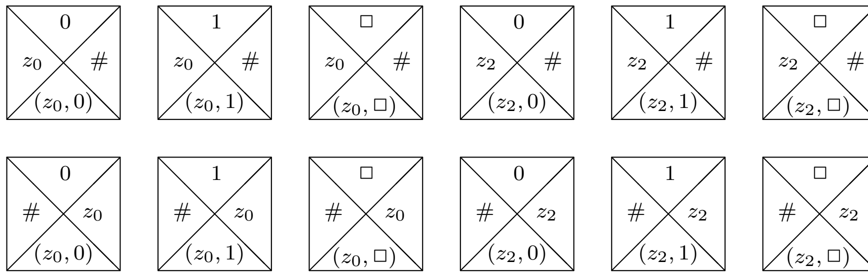
Figure 7.6: *The computation of the Turing machine on input $x = 11$. The pair (state,symbol) indicates the position of the tape head.*

---

2. The five instructions of the Turing machine yield five tile types:



3. The states z_0 and z_2, and the three symbols of the alphabet yield twelve tile types:



The computation of the Turing machine on input $x = 11$ consists of nine computation steps. During this computation, the tape head visits exactly six cells. Therefore, the frame of the domino game has nine rows and six columns. This frame is given in Figure 7.7. In Figure 7.8, you find the solution of the domino game. Observe that this solution is nothing but an equivalent way of writing the computation of Figure 7.6. Hence, the computation of the Turing machine corresponds to a solution of the domino game; in fact, the converse also holds.

7.5.4 Examples of NP-complete languages

In Section 7.5.3, we have shown that *Domino* is **NP**-complete. Using this result, we will apply Theorem 7.5.11 to prove the **NP**-completeness of some other languages.

Satisfiability

We consider Boolean formulas φ, in the variables $x_1, x_2, \ldots, x_m$, having the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{7.9}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{k_i}^i.$$

	$(z_0, 0)$	1	0	1	1	$\square$	
#							#
#							#
#							#
#							#
#							#
#							#
#							#
#							#
#							#
	0	1	$(z_1, 1)$	0	0	$\square$	

Figure 7.7: *The frame of the domino game for input* $x = 11$.

Each ℓ_j^i is either a variable or the negation of a variable. Such a formula φ is said to be *satisfiable*, if there exists a truth-value in $\{0, 1\}$ for each of the variables $x_1, x_2, \ldots, x_m$, such that the entire formula φ is true. We define the following language:

$$SAT := \{\langle\varphi\rangle : \varphi \text{ is of the form (7.9) and is satisfiable}\}.$$

We will prove that SAT is **NP**-complete.

It is clear that $SAT \in \mathbf{NP}$. If we can show that

$$\textit{Domino} \leq_P SAT,$$

Figure 7.8: *The solution of the domino game for input* $x = 11$.

---

then it follows from Theorem 7.5.11 that SAT is **NP**-complete. (In Theorem 7.5.11, take $B :=$ *Domino* and $C := SAT$.)

Hence, we need a function $f \in \mathbf{FP}$, that maps input strings for *Domino* to input strings for SAT, in such a way that for every domino game D, the following holds:

$$\text{domino game } D \text{ is solvable} \Longleftrightarrow \begin{array}{l}\text{the formula encoded by the} \\ \text{string } f(\langle D \rangle) \text{ is satisfiable.}\end{array} \tag{7.10}$$

Let us consider an arbitrary domino game D. Let k be the number of tile types, and let the frame have t rows and t columns. We denote the tile types by $T_1, T_2, \ldots, T_k$.

We map this domino game D to a Boolean formula φ, such that (7.10) holds. The formula φ will have variables

$$x_{ij\ell}, 1 \leq i \leq t, 1 \leq j \leq t, 1 \leq \ell \leq k.$$

These variables can be interpretated as follows:

$$x_{ij\ell} = 1 \Longleftrightarrow \text{there is a tile of type } T_\ell \text{ at position } (i,j) \text{ of the frame.}$$

We define:

- For all i and j with $1 \leq i \leq t$ and $1 \leq j \leq t$:

  $$C^1_{ij} := x_{ij1} \vee x_{ij2} \vee \ldots \vee x_{ijk}.$$

  This formula expresses the condition that there is at least one tile at position (i,j).

- For all i, j, ℓ and ℓ' with $1 \leq i \leq t$, $1 \leq j \leq t$, and $1 \leq \ell < \ell' \leq k$:

  $$C^2_{ij\ell\ell'} := \neg x_{ij\ell} \vee \neg x_{ij\ell'}.$$

  This formula expresses the condition that there is at most one tile at position (i,j).

- For all i, j, ℓ and ℓ' with $1 \leq i \leq t$, $1 \leq j < t$, $1 \leq \ell \leq k$ and $1 \leq \ell' \leq k$, such that $i < t$ and the right symbol on a tile of type T_ℓ is not equal to the left symbol on a tile of type $T_{\ell'}$:

  $$C^3_{ij\ell\ell'} := \neg x_{ij\ell} \vee \neg x_{i,j+1,\ell'}.$$

  This formula expresses the condition that neighboring tiles in the same row "fit" together. There are symmetric formulas for neighboring tiles in the same column.

- For all j and ℓ with $1 \leq j \leq t$ and $1 \leq \ell \leq k$, such that the top symbol on a tile of type T_ℓ is not equal to the symbol at position j of the upper boundary of the frame:

  $$C^4_{j\ell} := \neg x_{1j\ell}.$$

  This formula expresses the condition that tiles that touch the upper boundary of the frame "fit" there. There are symmetric formulas for the lower, left, and right boundaries of the frame.

The formula φ is the conjunction of all these formulas C^1_{ij}, $C^2_{ij\ell\ell'}$, $C^3_{ij\ell\ell'}$, and $C^4_{j\ell}$. The complete formula φ consists of

$$O(t^2k + t^2k^2 + t^2k^2 + tk) = O(t^2k^2)$$

terms, i.e., its length is polynomial in the length of the domino game. This implies that φ can be constructed in polynomial time. Hence, the function f that maps the domino game D to the Boolean formula φ, is in the class **FP**. It is not difficult to see that (7.10) holds for this function f. Therefore, we have proved the following result.

Theorem 7.5.13 *The language SAT is* **NP***-complete.*

In Section 7.5.1, we have defined the language $3SAT$.

Theorem 7.5.14 *The language* $3SAT$ *is* **NP***-complete.*

Proof. It is clear that $3SAT \in \mathbf{NP}$. If we can show that

$$SAT \leq_P 3SAT,$$

then the claim follows from Theorem 7.5.11. Let

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

be an input for SAT, in the variables $x_1, x_2, \ldots, x_m$. We map φ, in polynomial time, to an input φ' for $3SAT$, such that

$$\varphi \text{ is satisfiable} \Longleftrightarrow \varphi' \text{ is satisfiable.} \tag{7.11}$$

For each i with $1 \leq i \leq k$, we do the following. Consider

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{k_i}^i.$$

- If $k_i = 1$, then we define
$$C_i' := \ell_1^i \vee \ell_1^i \vee \ell_1^i.$$
- If $k_i = 2$, then we define
$$C_i' := \ell_1^i \vee \ell_2^i \vee \ell_2^i.$$
- If $k_i = 3$, then we define
$$C_i' := C_i.$$
- If $k_i \geq 4$, then we define
$$\begin{aligned} C_i' \quad := \quad & (\ell_1^i \vee \ell_2^i \vee z_1^i) \wedge (\neg z_1^i \vee \ell_3^i \vee z_2^i) \wedge (\neg z_2^i \vee \ell_4^i \vee z_3^i) \wedge \ldots \\ & \wedge (\neg z_{k_i-3}^i \vee \ell_{k_i-1}^i \vee \ell_{k_i}^i), \end{aligned}$$
where $z_1^i, \ldots, z_{k_i-3}^i$ are new variables.

Let

$$\varphi' := C_1' \wedge C_2' \wedge \ldots \wedge C_k'.$$

Then φ' is an input for $3SAT$, and (7.11) holds. ∎

Theorems 7.5.6, 7.5.8, 7.5.11, and 7.5.14 imply:

Theorem 7.5.15 *The language Clique is* **NP***-complete.*

The traveling salesperson problem

We are given two positive integers k and m, a set of m cities, and an integer $m \times m$ matrix M, where

$$M(i, j) = \text{the cost of driving from city } i \text{ to city } j,$$

for all $i, j \in \{1, 2, \ldots, m\}$. We want to decide whether or not there is a tour through all cities whose total cost is less than or equal to k. This problem is **NP**-complete.

Bin packing

We are given three positive integers m, k, and ℓ, a set of m objects having volumes $a_1, a_2, \ldots, a_m$, and k bins. Each bin has volume ℓ. We want to decide whether or not all objects fit within these bins. This problem is **NP**-complete.

Here is another interpretation of this problem: We are given m jobs that need time $a_1, a_2, \ldots, a_m$ to complete. We are also given k processors, and an integer ℓ. We want to decide whether or not it is possible to divide the jobs over the k processors, such that no processor needs more than ℓ time.

Time tables

We are given a set of courses, class rooms, and professors. We want to decide whether or not there exists a time table such that all courses are being taught, no two courses are taught at the same time in the same class room, no professor teaches two courses at the same time, and conditions such as "Prof. L. Azy does not teach before 1pm" are satisfied. This problem is **NP**-complete.

Motion planning

We are given two positive integers k and ℓ, a set of k polyhedra, and two points s and t in $\mathbb{Q}^3$. We want to decide whether or not there exists a path between s and t, that does not intersect any of the polyhedra, and whose length is less than or equal to ℓ. This problem is **NP**-complete.

Map labeling

We are given a map with m cities, where each city is represented by a point. For each city, we are given a rectangle that is large enough to contain the name of the city. We want to decide whether or not these rectangles can be placed on the map, such that

- no two rectangles overlap,
- For each i with $1 \leq i \leq m$, the point that represents city i is a corner of its rectangle.

This problem is **NP**-complete.

This list of **NP**-complete problems can be extended almost arbitrarily: For thousands of problems, it is known that they are **NP**-complete. For all of these, it is *not known*, whether or not they can be solved efficiently (i.e., in polynomial time). Collections of **NP**-complete problems can be found in the book

- M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of* **NP***-Completeness.* W.H. Freeman, New York, 1979,

and on the web page

`http://www.nada.kth.se/~viggo/wwwcompendium/`

7.6 Exercises

7.1 Prove that the function $F : \mathbb{N} \to \mathbb{N}$, defined by $F(x) := 2^x$, is not in **FP**.

7.2 Prove Theorem 7.5.3.

7.3 Prove that the language *Clique* is in the class **NP**.

7.4 Prove that the language $3SAT$ is in the class **NP**.

7.5 We define the following languages:

- Sum of subset:

$$SOS := \{\langle a_1, a_2, \ldots, a_m, b\rangle : \exists I \subseteq \{1, 2, \ldots, m\}, \sum_{i \in I} a_i = b\}.$$

- Set partition:

$$SP := \{\langle a_1, a_2, \ldots, a_m \rangle : \exists I \subseteq \{1, 2, \ldots, m\}, \sum_{i \in I} a_i = \sum_{i \notin I} a_i\}.$$

- Bin packing: BP is the set of all strings $\langle s_1, s_2, \ldots, s_m, B \rangle$ for which
  1. $0 < s_i < 1$, for all i,
  2. $B \in \mathbb{N}$,
  3. the numbers $s_1, s_2, \ldots, s_m$ fit into B bins, where each bin has size one, i.e., there exists a partition of $\{1, 2, \ldots, m\}$ into subsets I_k, $1 \leq k \leq B$, such that $\sum_{i \in I_k} s_i \leq 1$ for all k, $1 \leq k \leq B$.

  For example, $\langle 1/6, 1/2, 1/5, 1/9, 3/5, 1/5, 1/2, 11/18, 3 \rangle \in BP$, because the eight fractions fit into three bins:

$$1/6 + 1/9 + 11/18 \leq 1, \;\; 1/2 + 1/2 = 1, \;\; \text{and } 1/5 + 3/5 + 1/5 = 1.$$

1. Prove that $SOS \leq_P SP$.
2. Prove that the language SOS is **NP**-complete. You may use the fact that the language SP is **NP**-complete.
3. Prove that the language BP is **NP**-complete. Again, you may use the fact that the language SP is **NP**-complete.

7.6 Prove that $3Color \leq_P 3SAT$.
Hint: For each vertex i, and for each of the three colors k, introduce a Boolean variable x_{ik}.

7.7 The $(0,1)$-*integer programming* language IP is defined as follows:

$$IP := \{\langle A, c \rangle : \begin{array}{l} A \text{ is an integer } m \times n \text{ matrix for some } m, n \in \mathbb{N}, \\ c \text{ is an integer vector of length } m, \text{ and} \\ \exists x \in \{0,1\}^n \text{ such that } Ax \leq c \text{ (componentwise) } \}. \end{array}$$

Prove that the language IP is **NP**-complete. You may use the fact that the language SOS is **NP**-complete.

7.8 Let φ be a Boolean formula in the variables $x_1, x_2, \ldots, x_m$.
We say that φ is in *disjunctive normal form* (DNF) if it is of the form

$$\varphi = C_1 \vee C_2 \vee \ldots \vee C_k, \tag{7.12}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \wedge \ell_2^i \wedge \ldots \wedge \ell_{k_i}^i.$$

Each ℓ_j^i is a *literal*, which is either a variable or the negation of a variable.
We say that φ is in *conjunctive normal form* (CNF) if it is of the form

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_k, \tag{7.13}$$

where each C_i, $1 \leq i \leq k$, is of the following form:

$$C_i = \ell_1^i \vee \ell_2^i \vee \ldots \vee \ell_{k_i}^i.$$

Again, each ℓ_j^i is a literal.

We define the following two languages:

$$DNFSAT := \{\langle\varphi\rangle : \varphi \text{ is in DNF-form and is satisfiable}\},$$

and

$$CNFSAT := \{\langle\varphi\rangle : \varphi \text{ is in CNF-form and is satisfiable}\}.$$

1. Prove that the language $DNFSAT$ is in $\mathbf{P}$.

2. What is wrong with the following argument: Since we can rewrite any Boolean formula in DNF-form, we have $CNFSAT \leq_P DNFSAT$. Hence, since $CNFSAT$ is $\mathbf{NP}$-complete and since $DNFSAT \in \mathbf{P}$, we have $\mathbf{P} = \mathbf{NP}$.

3. Prove directly that for every language A in $\mathbf{P}$, $A \leq_P CNFSAT$. "Directly" means that you should not use the fact that $CNFSAT$ is $\mathbf{NP}$-complete.

7.9 Prove that the polynomial upper bound on the length of the string y in the definition of $\mathbf{NP}$ is necessary, in the sense that if it is left out, then any decidable language would satisfy the condition.

More precisely, we say that the language A belongs to the class $\mathbf{D}$, if there exists a language $B \in \mathbf{P}$, such that for every string w,

$$w \in A \Longleftrightarrow \exists y : \langle w, y\rangle \in B.$$

Prove that $\mathbf{D}$ is equal to the class of all decidable languages.

Chapter 8

Problems

1. Provide a proof for the simplified Masters theorem (refer to Notes on the statement of the theorem). Present examples for each of the three cases and present an example where this theorem is not applicable.

2. 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?

3. Provide a clear, concise, and complete proof for the correctness of the BFS algorithm (refer to the notes for the algorithm).

4. 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.

5. 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.

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

7. 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 (Please refer to notes - and you can use all the lemmas which are there without reproving them).

8. 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 $1 \leq i \leq n$.

9. 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?

10. 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.

11. 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.

12. 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).

13. 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).

14. 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).

15. Construct the string matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaabaab$.

16. 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.

17. 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.

18. 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!).

19. 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.)

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

21. 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 ')'.

22. This is from [3] book and is based on Divide-and-Conquer Multiplication. (Do not use FFTs as such for this)

    (a) 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.)

    (b) Give a divide-and-conquer algorithm for multiplying two polynomials of degree n that runs in $\Theta(n^{\log 3})$. 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.

    (c) Show that two n-bit integers can be multiplied in $On^{\log 3})$ steps, where each step operates on at most a constant number of 1-bit values.

23. 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.

24. Provide a formal proof of Lemma 3.4.1.

25. 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

    (a) $w(A), w(B) \leq \frac{2}{3}$, where $w(A)$ is the sum total of weights of all the vertices in set A

    (b) $|S| \leq 4\sqrt{n}$

    (c) There is no edge in E that joins a vertex in A with a vertex in B.

    (d) Such a set S can be found in linear time.

26. Prove Lemma 6.2.2. 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.

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?

27. 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.

28. 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.

29. 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.)

30. 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.

31. 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.

32. 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.

33. Given two binary strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, 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 upto n. Present an algorithm that can sort the binary strings in $O(n)$ time. (All the strings are not of the same length!)

34. 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.

35. 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.

36. 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?

37. 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.)

38. 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.

39. 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.

40. 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!)

41. 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?

42. 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 $G = (V, E)$.)

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

44. 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).

45. 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 Dijsktra'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 Dijsktra's algorithm $O(|V|)$ times? [Hint: See Book]

46. 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.

47. 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.

48. 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!).

49. 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.

50. 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.

51. 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 versa (Use colours to illustrate the clique!)

52. 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!)

53. 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?

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

    (a) 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 $MST(G) = T$.

    (b) 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 $MST(G) = T$.

    (c) 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 $MST(G) = T$.

55. 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.

56. 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.

57. 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.

58. 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.)

59. 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 $\mathcal{P} = \mathcal{NP}$.

60. 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.

Bibliography

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley 1974.

[2] T. Chan, Backward analysis of the Karger-Klein-Tarjan Algorithm for minimum spanning tree, Information Processing Letters, 1998.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, Introduction to algorithms, McGraw Hill, 2001.

[4] D. Kozen, The design and analysis of algorithms, Springer-Verlag 1992.

[5] K.H. Rosen, Discrete Mathematics and its Applications, McGraw-Hill 1999.

[6] D.E. Knuth, The art of computer programming, Vol 1, Fundamental Algorithms, Addison Wesley, 1997.

[7] R. Graham, D.E. Knuth and O. Patashnik, Concrete Mathematics, Addison Wesley, 1998.

[8] R. Motwani and P. Raghavan, Introduction to Randomized Algorithms, Cambridge University Press, 1995.

[9] V. Strassen, Gaussian elimination is not optimal, Numerische Mathematik, 14(3):354-356, 1969.