Advanced Algorithmics

October 21, 2015

Acknowledgements This unofficial document contains notes for the lecture "Advanced Algorithmics" given by Prof. Rolf Niedermeier (TU Berlin). We assume/accept no liability for being complete, correct and/or up-to-date. This script was initiated by Meike Hatzel and Henning Seidler.

Contents

1 Introduction

Formally, an *algorithm* is a mathematical procedure that computes an *output* from some *input*. In order to measure the complexity of an algorithm, we use an abstract model for the machine it runs on. Then we count the number of basic operations relative to the input size needed to compute the output. The model we use is called *Random Access Machine* (RAM).

Definition 1.1 (Random Access Machine)

- Each basic operation $(+, -, =, \text{if}, \ldots)$ takes one time step.
- Loops and subroutines are not basic operations.
- Each memory access takes one time step (no difference between cache and disk). $\dashv$

In order to compare and categorize the running times of algorithms, we use the *big O notation*. It describes the asymptotical behavior of a function and allows us to compare their growth rates. Intuitively, we say that

- $f \in \mathcal{O}(g)$ or $g \in \Omega(f)$ if f grows at most as fast as g,
- $f \in o(g)$ or $g \in \omega(f)$ if g grows strictly faster than f,
- $f \in \Theta(g)$ if f and g have the same growth rate.

Formally, we can define the notation as follows.

Definition 1.2 (Big O Notation)

- $f(n) \in \mathcal{O}(g(n))$ if $\exists c > 0, n_0 > 0\ \forall n > n_0 : f(n) \leq c \cdot g(n)$.
- $f(n) \in o(g(n))$ if $\forall c > 0\ \exists n_0 > 0\ \forall n > n_0 : f(n) \leq c \cdot g(n)$.
- $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in \mathcal{O}(f(n))$.
- $f(n) \in \omega(g(n)) \Leftrightarrow g(n) \in o(f(n))$.
- $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \cap \Omega(g(n))$. $\dashv$

1.1 Recursion

Recursion is a method where a solution to a problem is computed from the solution of smaller instances of the same problem. In oder to terminate a recursion needs to reach a *base case*, which usually has a simple solution. We give the following examples to illustrate this method.

Example 1.3 (Towers of Hanoi)
Input: n disks, each of distinct size, 3 pegs, all disks are placed in ascending order on peg 1.
Task: Move all disks to peg 3 under the following constraints:

- Only one disk can be moved at a time.
- Only the top disk of any peg can be moved to the top of another peg.
- No disk may be placed on top of a smaller disk.

Algorithm:

1. recursively move $n - 1$ disks to peg 2,
2. move the largest disk to peg 3,
3. recursively move $n - 1$ disks from peg 2 to peg 3.

We can see that this method uses a solution of a problem instance of size $n-1$ to create a solution for an instance of size n. The number of moves needed to complete the task is

$$\begin{aligned} &T(n) = 2T(n-1)+1, \text{ with } T(0)=0 \\ \rightsquigarrow\ &T(n) = 2^n - 1. \end{aligned}$$

Example 1.4 (Euclid's Algorithm)
Input: Two integers $a, b > 0$.
Task: Find the greatest common divisor (gcd) of a and b.
Algorithm:

1. if $a = b$, return a,
2. else, return $\gcd(b, a \bmod b)$.

This algorithm makes use of the fact that $\gcd(a,b) = \gcd(b, a \bmod b)$. Note that in step 1 it checks whether a base case has been reached. Its running time is in $\mathcal{O}(n)$ if a and b have at most n digits each.

Example 1.5 (Exponentiation)
Input: Two integers a and n.
Task: Compute a^n.
Algorithm:

1. if n is even, compute $\left(a^{n/2}\right)^2$,
2. if n is odd, compute $a \cdot \left(a^{(n-1)/2}\right)^2$.

The recursive call in this example happens when computing $a^{n/2}$ and $a^{(n-1)/2}$. In each recursive step, we roughly halve the size of the instance, thus the number of multiplications we need is in $\mathcal{O}(\log n)$.

1.2 Divide and Conquer

Divide and Conquer is an algorithm design paradigm based on multi-branched recursion. A problem is recursively *divided* into several smaller sub-problems until they are simple enough to solve them directly (*conquer*). The solutions of these sub-problems are then combined to a solution of the original problem. Most of the time, the combination step is the essential part of the algorithm. Well-known examples are:

- Exponentiation.
- Merge Sort.
- Fast Fourier Transform.
- Closest Pair of Points.
- Integer Multiplication, Matrix Multiplication.
- Counting Inversions.

Another prominent example is the computation of a convex hull of a point set in the plane. This is one of the most important problems in computational geometry.

Definition 1.6 (Convex Hull)
The *convex hull* of a set S of points in the plane is the smallest convex polygon containing all points of S. (fig. 1) $\dashv$

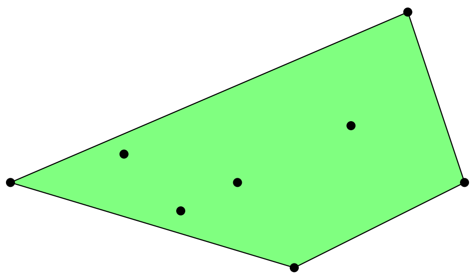


Figure 1: Convex Hull

Problem 1.7 (Convex Hull Computation in the Plane)
We assume that all points have pairwise different x-coordinates and we represent the convex hull as a double-linked list of its vertices in clockwise order. We additionally assume points of S to be sorted according to x-coordinate. Alternatively, this can be done in a preprocessing step.
Input: A set S of points in the plane.
Output: The vertices of the convex hull of S.
Algorithm:

1. If $|S| \leq 3$, return a triangle or the line.
2. Split the points into a left half S_1 and a right half S_2.
3. Compute convex hulls of S_1 and S_2 .
4. Combine the two disjoint polygons to one convex hull by adding upper and lower tangents. (fig. 2)

Note that in step 1 we check whether the problem is simple enough to solve it directly. In steps 2 and 3 we recursively divide the problem into smaller sub-problems and in step 4 we combine the solutions of the sub-problems. Let us now have a closer look at the combination step.
The upper (respectively, lower) tangent is a line connecting S_1 and S_2 such that all points of $S_1 \cup S_2$ lie below (respectively, above) it.
Computing the upper tangent: (lower tangent analogously)

1. Let p_1 be the rightmost point of S_1 and p_2 be the leftmost point of S_2.
2. Repeat as long as any of the following applies:
   - if p_1 is below line $\overline{ap_2}$, where a is the clockwise predecessor of p_1, set $p_1 \leftarrow a$.
   - if p_2 is below line $\overline{bp_1}$, where b is the clockwise successor of p_2, set $p_2 \leftarrow b$.

Note that we can evaluate the conditions in step 2 in constant time: Let $a = (a_x, a_y), b = (b_x, b_y)$ and $c = (c_x, c_y)$. A point b is below a line $\overline{ac}$ if a, b, c are oriented mathematically positively (counter-clockwise). This can be checked in $\mathcal{O}(1)$ time by testing whether the determinant is positive:

$$\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{vmatrix} > 0$$

$\rightsquigarrow$ computing the tangents can be done in $\mathcal{O}(n)$ time.
Hence we get the following overall running time for the computation of a convex hull of a point set in the plane:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n) \in \mathcal{O}(n \log n)$$

Remarks:

- The algorithm generalizes to the three dimensional case.

- It is possible to show that *Convex Hull Computation in the Plane* is in $\Theta(n \log n)$ since it is at least as hard as sorting.
- However, algorithms with a running time in $\mathcal{O}(n \log h)$ exist, where h is the number of vertices on the convex hull. $\dashv$

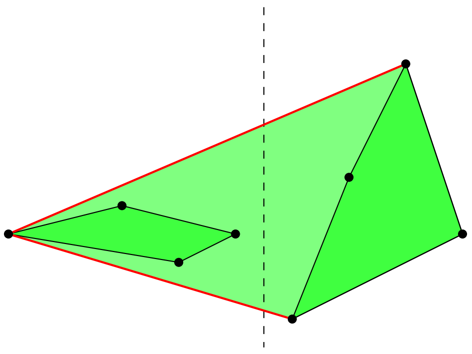


Figure 2: Convex Hull tangents construction

1.3 Dynamic Programming

Similar to Divide and Conquer, *Dynamic Programming* is a method for solving complex problems by breaking them down into collections of smaller sub-problems. In Dynamic Programming, we exploit overlaps between sub-problems or, in other words, avoid solving same sub-problems multiple times. The central idea is to work bottom-up, i.e. start with the smallest sub-problems and memorize every solution in a table. Each time we encounter a previously solved sub-problem, we look it up instead of recomputing it. There are several well-known examples:

- CYK algorithm for word problem of context-free languages.
- Levenshtein / edit distance between strings.
- Longest Common Subsequence.
- Floyd's all-pairs shortest path algorithm.
- Optimally ordering chain matrix multiplication.
- Subset Sum.

Example 1.8 (Fibonacci Numbers)
The *Fibonacci Numbers* can be defined in the following recursive way:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

It is easy to see that a naïve recursive implementation of $f(n)$ yields an algorithm with a running time in $\mathcal{O}(2^n)$. If we use a table T of size n, initiate it with $T[1] = T[2] = 1$ and iteratively compute $T[i] = T[i-1] + T[i-2]$ for $i = 3$ to n, the resulting running time is in $\mathcal{O}(n)$.

Problem 1.9 (Longest Increasing Subsequence)
Input: n integers ordered in a sequence $s_1, \ldots, s_n$.
Task: Find longest monotonically increasing subsequence.
Algorithm:
Let l_i denote the length of the longest subsequence ending with s_i.

1. Set $l_0 = 0$.
2. For $i = 1$ to n: Set $l_i = \max\limits_{j<i \text{ and } s_j \le s_i} l_j + 1$.

The length of the longest monotonically increasing subsequence is $\max_{1\le i\le n} l_i$. To compute the actual subsequence and not only its length we simply store for each s_i its predecessor p_i, i.e. the index of the element that appears immediately before s_i in a longest subsequence ending at s_i.
Time complexity: To compute each l_i we need $\mathcal{O}(n)$ comparisons, so the overall running time is in $\mathcal{O}(n^2)$.
Important note: With modified table entry definitions and binary search an improvement to a running time in $\mathcal{O}(n \log n)$ is possible. ⊣

Problem 1.10 (Knapsack)
Input:

- Items $z_1, \ldots, z_n$, where each z_i has a value v_i and a weight w_i.
- A bag that can carry a maximum weight W.

Task: Determine the maximum value that can be carried in the bag.
Algorithm:
Define a table m with entries $m[i, w]$ that store the maximum value obtainable with items from $\{z_1, \ldots, z_i\}$ and maximum weight w. Then compute the entries of m by the following rules.

$$m[0, j] = 0 \text{ for all } j \in \{0, \ldots, W\}$$
$$m[i, w] = m[i-1, w] \text{ if } w_i > w, \text{ and otherwise}$$
$$m[i, w] = \max \begin{cases} m[i-1, w] & \text{item } z_i \text{ is not chosen} \\ m[i-1, w-w_i] + v_i & \text{item } z_i \text{ is chosen} \end{cases}$$

The solution is contained in $m[n, W]$ and is computable in $\mathcal{O}(n \cdot W)$ time. This is a *pseudo-polynomial* running time. ⊣

Problem 1.11 (Travelling Salesperson (TSP)) Input: n cities and distances $d(i, j)$ between cities i and j. **Output:** A shortest-length tour visiting every city exactly once. ⊣

Example 1.12 In 2001 a shortest tour through 15,112 German cities was computed.

Algorithm 1.13 Try all permutations of n cities and, for each permutation, compute the tour length. This results in a computation time of $\mathcal{O}(n! \cdot n)$. ⊣

Algorithm 1.14 (Held/Karp,1962) Using Dynamic Programming we can achieve a time of $\mathcal{O}(2^n \cdot n^2)$. This is the fastest known exact algorithm for TSP to date (with respect to a provable upper bound). Wlog our tour starts in city 1. For a subset $S \subseteq \{2, \ldots, n\}$ and a city $c \in S$ define $T[S, c]$ to be the length of the shortest tour starting in city 1, visiting every city in S and ending in c. Then we have the recurrence

$$T[S, c] = \begin{cases} d(1, c) & : S = \{c\} \\ \min\limits_{c' \in S\setminus\{c\}} (T[S \setminus \{c\}, c'] + d(c, c')) & : \text{else} \end{cases}$$

Length of shortest tour is

$$\min_{2\le i\le n} (T[\{2, \ldots, n\}, i] + d(i, 1))$$

The table has $\mathcal{O}(2^n \cdot n)$ entries, each computable in $\mathcal{O}(n)$ time. ⊣

Figure 3: optimal tour through 15,112 German cities, University of Waterloo, Canada

1.4 Search Trees and Backtracking

Often possible solutions can be discarded because a partially constructed solution cannot be extended into a full solution.

Problem 1.15 (CNF-SAT)
Input: Boolean formula F in conjunctive normal form.
Task: Decide whether F has a satisfying truth assignment.
Useful heuristic: Expand variables from smallest clauses first. This yields a polynomial-time algorithm for the special case 2-CNF-SAT. $\dashv$

General Observation: Backtracking algorithm needs a test that quickly can declare one of three outcomes for a subproblem:

1. Failure: Subproblem has no solution.

2. Success: Solution to subproblem found.

3. Uncertainty: Continue search.

Example 1.16 (DPLL(Davis-Putman-Logemann-Loveland)) The worst case running time is $\mathcal{O}(2^n \text{poly}(n))$, but it is very successful in practice. **Algorithm:**

- Apply unit propagation as long as possible: identify a unit clause $c_i = l$, remove all clauses containing l from F and remove $\neg l$ from the remaining clauses.

- Apply pure literal elimination as long as possible: identify a literal l that occurs only in one polarity and remove all clauses containing l.

- If F is empty then return true.

- If F contains an empty clause then return false.

- Branch: select a literal l, return $\text{DPLL}(F|_{l=1}) \vee \text{DPLL}(F|_{l=0})$.

Solving k-CNF-SAT faster than $\Theta(2^n)$:

Example 1.17 (for 3-CNF-SAT) The idea is that for a clause $(l_1 \vee l_2 \vee l_3)$ of a formula F recursively evaluate the following subformulas of F:

$$F_1 \equiv F|_{l_1=1}, F_2 \equiv F|_{l_1=0,l_2=1}, F_3 \equiv F|_{l_1=0,l_2=0,l_3=1}.$$

Correctness: Obviously, if F is satisfiable, then at least one of F_1, F_2 and F_3 has to be $\rightsquigarrow$ recursive calls.
Search tree size for n variables:

$$T(n) = 1 + T(n-1) + T(n-2) + T(n-3).$$

This has the solution $T(n) \approx 1.8393^n$. The idea can be improved to $T(n) = \varphi^n$ (golden ratio).

Theorem 1.18 (ETH(exponential time hypothesis)) $2^{o(n)}$ poly-steps for CNF-SAT not possible. (not proved but believed in)

1.5 Polynomial Time versus NP-Hardness

We would like to determine the minimum effort necessary for solving problem.
→ Upper bounds via algorithms,
→ Lower bounds via "brilliant ideas".
Measures for complexity are time, space and sometimes communication. The central model for complexity theory are Turing machines. A polynomial-time RAM algorithm can be simulated by a Turing machine running in polynomial time. In complexity theory, one mainly focusses on decision (yes/no) problems (equivalently, languages), while in algorithmics one mostly deals with optimisation or search problems.
P: polynomial-time solvable decision problems.
NP: decision problems with polynomial-time verifiable solutions.

Definition 1.19 (NP) A language $L \subseteq \{0,1\}^*$ is in NP if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time algorithm A such that for all $x \in \{0,1\}*$,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} : \ A \text{ accepts input } (x,u) = 1.$$

In the case of acceptance, we call u a certificate for x (with respect to L and A). ⊣

Problem 1.20 (Independent Set)
Input: Undirected graph $G = (V,E)$, integer $k \geq 0$.
Question: Does G have an independent set of size k?
Certificate: A set with at least k vertices. ⊣

Problem 1.21 (Subset Sum)
Input: Integers $A_1, \ldots, A_n$ and T.
Question: $\exists J \subseteq \{1,\ldots,n\} : \sum_{i \in J} A_i = T$?
Certificate: J. ⊣

Problem 1.22 (Factoring)
Input: Positive integers N, L, U.
Question: Is there an integer factor M of N with $L \leq M \leq U$?
Certificate: M. ⊣

Definition 1.23 (Reduction) A language $L_1 \subseteq \{0,1\}^*$ is polynomial-time Karp-reducible (equivalently, polynomial-time many-one reducible or polynomial-time reducible) to a language $L_2 \subseteq \{0,1\}^*$, written $L_1 \leq_P L_2$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$ ⊣

Definition 1.24 (NP-hardness) A language L_2 is NP-hard if $L_1 \leq_P L_2$ for all $L_1 \in NP$. ⊣

Definition 1.25 (NP-completeness) A language L_2 is NP-complete if it is NP-hard and contained in NP. $\dashv$

- If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.
- If L is NP-hard and $L \in P$, then $P = NP$.
- If L is NP-complete, then $L \in P \Leftrightarrow P = NP$.

Proof idea: $p(n), q(n)$ polynomials $\Rightarrow p(q(n))$ is a polynomial.

Definition 1.26 (weakly NP-hard) Number problem that only becomes NP-hard if one allows "big" numbers in the input. $\dashv$

Definition 1.27 (strongly NP-hard) If a problem remains NP-hard even when every number occurring in the input has a value bounded in a polynomial in the length of the binary encoding of the total input, then the problem is strongly NP-hard. $\dashv$

Example 1.28 (3-Partition) probably best known strongly NP-hard number problem
Knapsack is weakly NP-hard $\rightsquigarrow$ pseudo-polynomial-time.

Theorem 1.29 Every Boolean function $f : \{0,1\}^l \to \{0,1\}$ can be represented by an equivalent formula in CNF with l variables and at most $l \cdot 2^l$ AND- and OR-operations.

Proof 1.30 (idea) Construct CNF formula from truth table for f. (formula may become exponentially large) $\square$

Theorem 1.31 (Cook, Levin) CNF-SAT is NP-complete.

Corollary 1.32 3-CNF-SAT is NP-complete. $\dashv$

Proof 1.33 (idea) Show CNF-SAT $\leq_P$ 3-CNF-SAT. Replace every clause with $m > 3$ literals by $m-2$ clauses with 3 literals each, introducing for each clause with m literals $m-3$ new variables $z_1, \ldots, z_m$:

$$\begin{aligned}
&(l_1 \vee l_2 \vee \ldots l_m) \rightsquigarrow \\
&\qquad (l_1 \vee l_2 \vee z_1) \wedge (\overline{z}_1 \vee l_3 \vee z_2) \wedge (\overline{z}_2 \vee l_4 \vee z_3) \wedge \ldots \\
&\qquad \wedge (\overline{z}_{m-4} \vee l_{m-2} \vee z_{m-3}) \wedge (\overline{z}_{m-3} \vee l_{m-1} \vee l_m)
\end{aligned}$$ $\square$

Theorem 1.34 If $P = NP$, then for every $L \in NP$, there is an algorithm B with polynomial running time that on input $x \in L$ produces a certificate for x.

Proof 1.35 (idea for CNF-SAT) If $P = NP$, then there is a polynomial-time algorithm A for CNF-SAT. Algorithm B works as follows:

1. Using A, check whether input formula $\phi(x_1, \ldots, x_n)$ is satisfiable.
2. If yes, recursively check whether
   a) ϕ is satisfiable for $x_1 = 0$ or
   b) ϕ is satisfiable for $x_1 = 1$.

Thus we can construct a satisfying assignment step-by-step. $\square$

In general: Most optimisation or search problems are not "significantly" harder than their "corresponding" decision problems.

Example 1.36 (Decision versus Search/Optimisation problem)

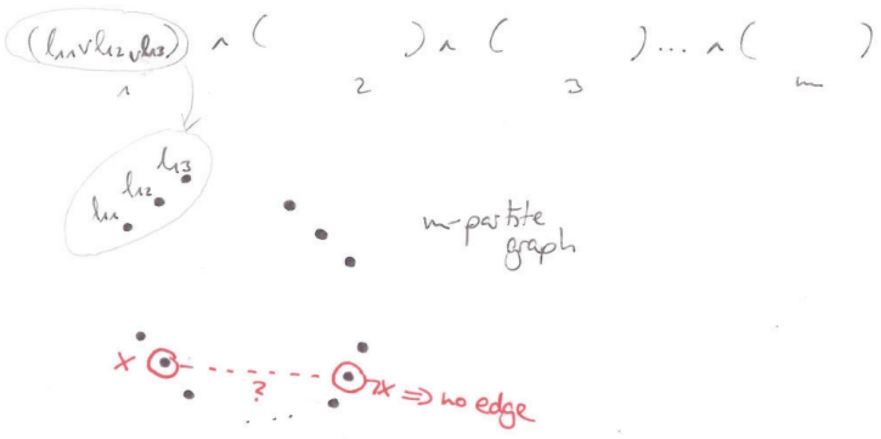
- **CNF-SAT** Decision: Is formula satisfiable? Search: Find satisfying truth assignment.

- **Knapsack** Decision: Can one carry items of value at least V in a bag of size W? Maximisation: Determine the maximum value that one can carry in a bag of size W.
- **Independent Set** Decision: Does given graph contain at least k independent vertices? Maximisation: Find maximum-cardinality independent vertex set in graph.
- **Vertex Cover** Decision: Are there at most k vertices in a given graph such that each edge has at least one of them as endpoint? Minimisation: Find minimum-cardinality vertex cover in graph.

Problem 1.37 (Clique) **Input:** Undirected graph $G = (V, E)$ and $k \geq 0$.
Question: Does G contain a complete subgraph of order k? ⊣

Theorem 1.38 Clique is NP-complete.

Proof 1.39 Obviously Clique is in NP: $\binom{n}{k} \approx n^k$.
Proof NP-hardness by reduction from 3-CNF-SAT: For a 3-CNF formula with m clauses, construct m-partite graph, where every vertex of a partition set one-to-one corresponds to a literal. There is an edge between two vertices from different partition sets iff the corresponding literals are not negations of each other. (fig. 4) Concrete example: Consider $(x_1 \vee \overline{x}_2 \vee x_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee x_3)$. (fig. 5)□



$(l_{11} \vee l_{12} \vee l_{13}) \wedge (\quad) \wedge (\quad) \ldots \wedge (\quad)$
1
2
3
m
l_{11}
l_{12}
l_{13}
m-partite graph
x
?
¬x ⇒ no edge

Figure 4: general construction

Problem 1.40 (Independent Set) **Input:** An undirected graph $G = (V, E)$ and $k \geq 0$.
Question: Does G contain at least k pairwise non-adjacent vertices? ⊣

Theorem 1.41 Independent Set is NP-complete.

Proof 1.42 Obviously Independent Set is in NP. NP-hardness follows from Clique $\leq_P$ Independent Set:

$$(G = (V, E), k) \mapsto (G' = (V, E'), k)),$$

where $E' := \{\{u, v\} | u, v \in V, \{u, v\} \notin E\}$ (complement graph). □

Problem 1.43 (Vertex Cover) **Input:** An undirected graph $G = (V, E)$ and $k \geq 0$.
Question: Can all edges of G be covered by at most k vertices, that is $\exists V' \subseteq V, |V'| \leq k : \forall \{u, v\} \in E : (u \in V') \vee (v \in V')$? ⊣

Theorem 1.44 Vertex Cover is NP-complete.

Proof 1.45 Obviously Vertex Cover is in NP. NP-hardness follows from Independent Set $\leq_P$ Vertex Cover: $G = (V, E)$ has a size-k independent set iff G has a vertex cover of size $|V| - k$. $\rightsquigarrow$ Trivial reductions ("equivalence") between both problems. □

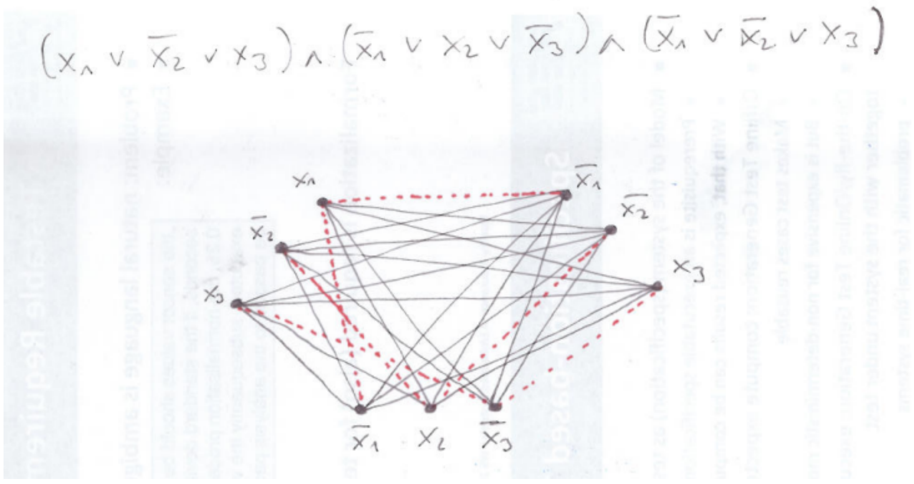

$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$
x_1
$\overline{x_1}$
$\overline{x_2}$
$\overline{x_2}$
x_3
x_3
$\overline{x_1}$
x_2
$\overline{x_3}$

Figure 5: example construction

Problem 1.46 (3-Colouring) **Input:** An undirected graph G.
Question: Can the vertices of G be coloured with three colours such that no adjacent vertices have the same colour?
The colouring number is called chromatic number.
3-CNF-SAT $\leq_P$ 3-Colouring leads to NP-hardness; containment in NP is trivial. ⊣

Problem 1.47 (Set Cover) **Input:** A family $\mathcal{F}$ of subsets of a universe U, and an integer $k \geq 0$.
Question: Are there at most k subsets from $\mathcal{F}$ such that their union yields U?
Vertex Cover $\leq_P$ Set Cover leads to NP-hardness; containment in NP is trivial. ⊣

Problem 1.48 (Data Clustering) **Task:** Partition the rows and columns of a given matrix into two non-empty subsets each such that the maximum difference between any two elements in each submatrix is minimised. ⊣

Notation 1.49 $[m] := \{1, \dots, m\}$.

Definition 1.50 ((k,l)-co-clustering) A (k,l)-co-clustering of an $m \times n$ matrix M is a pair $(\mathcal{R}, \mathcal{C})$ consisting of

- a k-partition $\mathcal{R} = \{R_1, \dots, R_k\}$ of $[m]$ (i.e. $R_s \subseteq [m], R_s \neq \emptyset$ for all $s \in [k], R_s \cap R_t = \emptyset$ for all $s \neq t \in [k]$ and $\bigcup_{s=1}^{k} R_s = [m]$) and
- an l-partition $\mathcal{C} = \{C_1, \dots, C_l\}$ of $[n]$.

For $(s,t) \in [k] \times [l]$ the set $A_{st} := \{a_{ij} \in A | (i,j) \in R_s \times C_t\}$ is called a cluster. ⊣

Problem 1.51 (Co-Clustering$_\infty$) **Input:** A matrix $M \in \mathbb{Z}^{m \times n}$, integers $k, l, c \in \mathbb{N}$.
Task: Find a (k,l)-co-clustering $(\mathcal{R}, \mathcal{C})$ of M such that

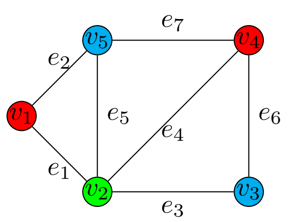
$$\max_{(s,t) \in [k] \times [l]} (\max A_{st} - \min A_{st}) \leq c.$$ ⊣

Theorem 1.52 (Bulteau/Hartung/Froese/Niedermeier, ISAAC 2014) Co-Clustering$_\infty$ is NP-hard for $k = l = 3, c = 1$, and $M \in \{0,1,2\}^{m \times n}$.

Proof 1.53 3-Colouring $\leq_P$ Co-Clustering$_\infty$. For an input graph $G = (V, E)$ of 3-Colouring, $V = v_1, \dots, v_n$ and $E = e_1, \dots, e_m$, construct a matrix $M \in \{0,1,2\}^{m \times n}$ such that M has a $(3,3)$-co-clustering of cost at most 1 iff G is 3-colourable (i.e. V can be partitioned into three independent sets V_1, V_2, V_3).
Constructing M from G:

- columns correspond to vertices V,
- rows correspond to edges E,
- for each edge $e_i = \{v_j, v_{j'}\}$ with $j < j'$, set $m_{ij} = 0$ and $m_{ij'} = 2$,
- all other entries are set to 1.

Each row corresponding to an edge $\{v_j, v_{j'}\}$ contains only 1's except for columns j and j', containing 0 and 2. Observation for correctness: Any co-clustering of M of cost at most 1 puts columns j and j' into different column blocks. (fig. 6) $\square$



$$\begin{array}{c|ccccc}
 & v_1 & v_2 & v_3 & v_4 & v_5 \\
\hline
e_1 & \mathbf{0} & \mathbf{2} & 1 & 1 & 1 \\
e_2 & \mathbf{0} & 1 & 1 & 1 & \mathbf{2} \\
e_3 & 1 & \mathbf{0} & \mathbf{2} & 1 & 1 \\
e_4 & 1 & \mathbf{0} & 1 & \mathbf{2} & 1 \\
e_5 & 1 & \mathbf{0} & 1 & 1 & \mathbf{2} \\
e_6 & 1 & 1 & \mathbf{0} & \mathbf{2} & 1 \\
e_7 & 1 & 1 & 1 & \mathbf{0} & \mathbf{2}
\end{array}$$

$V_1 = \{v_1, v_4\}$	$C_1 = \{1, 4\}$	$R_1 = \{1, 2, 7\}$
$V_2 = \{v_2\}$	$C_2 = \{2\}$	$R_2 = \{3, 4, 5\}$
$V_3 = \{v_3, v_5\}$	$C_3 = \{3, 5\}$	$R_3 = \{6\}$

Figure 6: Reduction example

Problem 1.54 (Degree Anonymity by Edge Addition) Input: An undirected graph $G = (V, E)$ and $k, s \in \mathbb{N}$.
Task: Find an edge set $E' \subseteq V^2$ with $|E'| \leq s$ such that $G' = (V, E \cup E')$ is k-anonymous, that is for every vertex $v \in V$ there are at least $k - 1$ other vertices in G' having same degree. $\dashv$

Theorem 1.55 (Hartung/Nichterlein/Niedermeier/Suchý, ICALP 2013) Degree Anonymity is NP-hard.

Proof 1.56 We reduce from the NP-hard Independent Set on three-regular graphs (all degrees equal to three). Starting from the three-regular input graph $G = (V, E)$ of Independent Set, we construct $G' = (V', E')$ as follows: First, copy G into G'. Second, add a clique of order $h + 3$.
Claim: G contains an independent set of size h iff G' can be made $(2h + 3)$-anonymous by adding $s = \binom{h}{2}$ edges.(fig. 7)
Adding all edges in an independent set of size h to make it a clique needs $\binom{h}{2}$ edges. $\square$

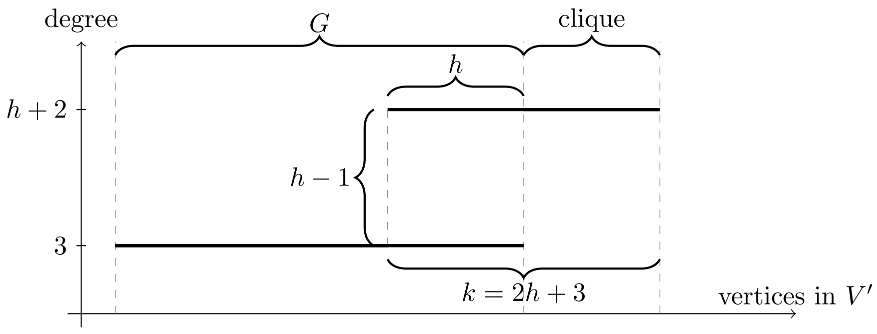


Figure 7: Construction for the Anonymisation-Construction

2 Exact and Parametrised Algorithms

2.1 Two Algorithms for the Travelling Salesperson Problem

Problem 2.1 (Travelling Salesperson (TSP)) Input: n cities and distances $d(i, j)$ between cities i and j.
Output: A shortest-length tour visiting every city exactly once.
The trivial solution would be to try all permutations of the n cities and for each compute the tour length: $\mathcal{O}(n!n)$ time.
If we use dynamic programming, using the Held-Karp algorithm from 1962, we reach a time of $\mathcal{O}(2^n \cdot n^2)$ (fastest known with provable upper bound).
For each subset $S \subseteq \{2, \dots, n\}$ of the cities and a city $c \in S$ define $T[S, c] :=$ length of the shortest tour starting in city 1, visiting every city in S and ending in c. Then

$$T[S, c] = d(1, c) \text{ if } S = \{c\} \text{ and}$$
$$T[S, c] = \min_{c' \in S \setminus \{c\}} (T[S \setminus \{c\}, c'] + d(c', c))$$

The length of the shortest tour then is $\min_{2 \leq i \leq n}(T[\{2, \dots, n\}, i] + d(1, i))$. The table has $\mathcal{O}(2^n \cdot n)$ entries, each computable in $\mathcal{O}(n)$ time.
If we have an euclidean space and thereby the triangle-inequality, we can introduce an additional parameter and solve it faster.
Parameter: Number k of cities lying inside the convex hull of all cities. ⊣

Theorem 2.2 (Deĭneko et al., Operations Research Letters, 2005) TSP with k inner points is solvable in $\mathcal{O}(2^k k^2 n)$ time.

Lemma 2.3 A shortest tour in the Euclidean plane cannot cross itself. fig. 8

Proof 2.4 Old tour T, new Tour T', tour length l:

$$\begin{aligned} l(T') = l(T) - \overline{pr} - \overline{qs} + \overline{pq} + \overline{rs} \leq l(T) \text{ since} \\ \overline{pr} = \overline{px} + \overline{rx}, \\ \overline{qs} = \overline{qx} + \overline{sx} \text{ and} \\ \overline{pq} \leq \overline{px} + \overline{qx} \text{ by triangle inequality and} \\ \overline{sr} \leq \overline{sx} + \overline{rx} \text{ by triangle inequality.} \end{aligned}$$

Conclusion: All points on convex hull visited either in clockwise or counterclockwise order. □

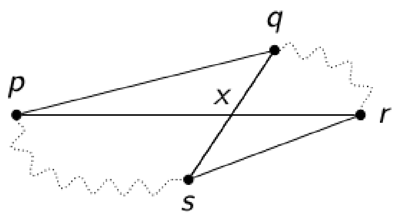


Figure 8: A shortest tour in the euclidean plane cannot cross itself

Algorithm Idea: Compute convex hull of cities. Then, compute optimal tour that visits the outer points in clockwise order. We only have to choose the right order on the "small" set I of k inner points. Use dynamic programming.

Definition 2.5 (Algorithm?) Let outer points in clockwise order be $p_1, \dots, p_{n-k}$. Let $T[i, S, r]$ for $1 \le i \le n-k, S \subseteq I$ and $r \in S \cup \{p_i\}$ be the length of the shortest path on $\{p_1, \dots, p_i\} \cup S$ such that

1. it starts at the outer point p_1,
2. it visits each point in $\{p_1, \dots, p_i\} \cup S$ exactly once,
3. it visits the outer points in clockwise order and
4. ends in r.

The length of a shortest tour is then $\min_{r \in I \cup \{p_{n-k}\}} \{T[n-k, I, r] + d(r, p_1)\}$ $\dashv$

Recurrence:

$$T[i, S, r] = \begin{cases} 0 & \text{if } r = p_1 \text{ and } S = \emptyset, \\ \infty & \text{if } r = p_1 \text{ and } S \neq \emptyset, \\ \min_{t \in S \cup \{p_{i-1}\}} T[i-1, S, t] + d(t, p_i) & \text{if } r = p_i \neq p_1, \\ \min_{t \in (S \setminus \{r\}) \cup \{p_i\}} T[i, S \setminus \{r\}, t] + d(t, r) & \text{if } r \in S. \end{cases}$$

There are $\mathcal{O}(2^k kn)$ table entries each computable in $\mathcal{O}(k)$ time. $\rightsquigarrow$ We find an optimal tour in $\mathcal{O}(2^k k^2 n)$ time which is linear time for a constant k.

2.2 Fundamentals of Parametrised Algorithms

Motivation: Solve NP-hard problems exactly and efficiently if certain input parameters are small.

Example 2.6

- Travelling Salesperson with k inner points solvable in $\mathcal{O}(2^k k^2 n)$ time for n cities.
- Job Interval Selection with j jobs is solvable in $\mathcal{O}(2^j n)$ time for n possible execution intervals.

  Problem 2.7 (Job Interval Selection) Input: A set $J \subseteq \mathbb{N}$ jobs and each job $i \in J$ having a set $S_i \subseteq \{[s_i, e_i] | s_i, e_i \in \mathbb{N}\}$ of possible execution intervals and $k \in \mathbb{N}$.
  Output: Can we select at least k pairwise non-intersecting intervals such that for each job at most one execution interval is selected? $\dashv$

  Proof 2.8 For simplicity, we will assume, that for any $s, s' \in S_i$, we never have $s \subseteq s'$, since then we could just omit s'. We initialize $T[0, C] = 0$ and $T[i, \emptyset] = 0$. Then we have the recurrence

  $$T[i+1, C] = \max \begin{cases} \max \{1 + T[s-1, C \setminus \{k\}] : k \in C, [s, i+1] \in S_k\} \\ T[i, C] \end{cases}$$

  whichever is bigger. Either we choose a new interval, ending at $i+1$, or it remains the same. In the first case job is done, so it has to be excluded from the remaining tasks. We can optimise this computation a bit, due to the fact $T[i, C] \le |C|$, so for large enough i and small C, there is no calculation to be done.

  Example 2.9 We have planes, some inspection teams and a hangar (or multiple ones). We want to inspect as many planes as possible.

  If we have many machines, just copy the intervals and append the block in a new time block.

- Knapsack with maximum weight W and n items solvable in $\mathcal{O}(Wn)$ time.

Definition 2.10 (parametrised problem) A parametrised problem is a language $L \subseteq \{0,1\}^* \times \{0,1\}^*$. The second component is called the parameter of the problem. $\dashv$

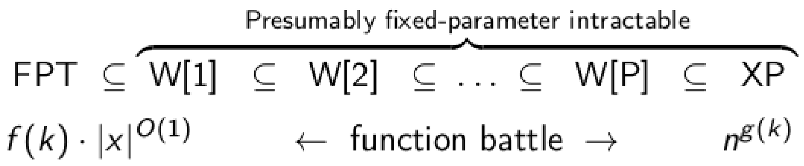

Presumably fixed-parameter intractable
FPT ⊆ W[1] ⊆ W[2] ⊆ ... ⊆ W[P] ⊆ XP
$f(k) \cdot |x|^{O(1)}$
← function battle →
$n^{g(k)}$

Figure 9: Class hierarchy

Definition 2.11 (fixed-parameter tractable(fpt)) A parametrised problem is fixed-parameter tractable if it can be determined in $f(k) \cdot |x|^{\mathcal{O}(1)}$ time whether $(x, k) \in L$, where f is a computable function only depending on k. The corresponding complexity class is called FPT.(fig. 9) $\dashv$

We now want to look at FPT-algorithms for trees and backtracking (up until now only dynamic programming).

Theorem 2.12 A vertex cover of size k in a graph with n vertices and m edges can be found in $\mathcal{O}(2^k(n + m))$ time.

Definition 2.13 (Vertex Cover (second)) A vertex subset $S \subseteq V$ of a graph $G = (V, E)$ is a vertex cover if its deletion makes the graph edge-less. $\dashv$

Proof 2.14 (Pseudocode) We define following algorithm VC(G, k, S), initially call with $S = \emptyset$:

```
if $G$ has no edges then
    output $S$
else
    if $k > 0$ then
        $\{u, w\} \leftarrow$ arbitrary edge of $G$
        VC$(G - \{u\}, k - 1, S \cup \{u\})$
        VC$(G - \{w\}, k - 1, S \cup \{w\})$
    end if
end if
```

Choice of parameter is important:

- Every decidable problem is fpt parametrised by input size n.
- Vertex Cover is fpt parametrised by size k of sought cover.
- Vertex Cover is NP-hard on graphs of maximum degree $\Delta = 3$.
  $\rightsquigarrow$ presumably not even in XP parametrised by Δ.

It is desirable to find small parameter that make a problem fpt.

- Solution size in a minimisation problem (e.g. vertex cover of size k).
- Distance from trivial instances (e.g. TSP with few inner points).
- Often more than one obvious parametrisation exists.

Problem 2.15 (DAG-partitioning) **Input:** A directed acyclic graph G, integer k.
Task: Remove at most k arcs from G so that each vertex will have a directed path to exactly one sink. Example for $k = 3$ see fig. 10. Application: Real-time tracking of trends and topics on the internet. $\dashv$

Theorem 2.16 (van Bevern et al., CIAC 2013) DAG-Partitioning is solvable in $\mathcal{O}(2^k n^2)$ time on graphs with n vertices.

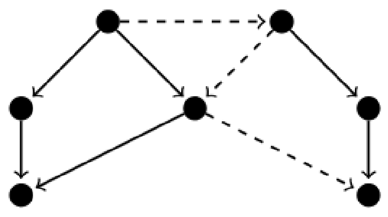


Figure 10: Example for DAG-partitioning ($k = 3$)

Proof 2.17 We use data reduction and search trees. One observation is that an optimal solution does not create new sinks. Therefore when a vertex v only has one out-neighbour u, then the arc (v, u) cannot be deleted $\rightsquigarrow$ we can contract arc (v, u) (see fig. 11).
If, after this data reduction, we find a vertex which out-going neighbours are all sinks, then

- it has arcs to $d \geq 2$ sinks,
- we want it to be connected to only one of these d sinks,
- at least $d - 1$ arcs to other sinks have to be deleted.

Such a vertex exists, because the input is a directed acyclic graph. We define the following algorithm DAGP(G, k, S), initially called with $S = \emptyset$:

```
if each vertex in $G$ is connected to exactly one sink then
    output $S$
else
    Apply data reduction rule to $G$ as long as possible.
    Choose vertex $v$, all of which out-neighbours are sinks.
    for all out-neighbours $s$ of $v$ do
        $D \leftarrow$ arcs from $v$ to vertices $\neq s$.
        if $k \geq |D|$ then
            DAGP$(G \setminus D, k - |D|, S \cup D)$.
        end if
    end for
end if
```

Running time for each call is $\mathcal{O}(n)$ time for checking whether all out-neighbours of a vertex are sinks and $\mathcal{O}(n)$ time for applying data reduction to a single vertex, which is done for at most $\mathcal{O}(n)$ vertices in the graph. Now how many recursive calls do we need ($\mathcal{O}(2^k)$ at first glance). But we show that the recursion tree has at most 2^k leaves, since each node of the recursion tree has at least two children, which brings us to the total number of $\mathcal{O}(2^k)$ nodes in the recursion tree.
Number of leaves of the recursion tree: $T(i) :=$ number of leaves when DAGP(G, k, S) is called with $k = i$. We show $T(i) \leq 2^i$.
Induction base case: $T(0) = 1$.
Induction hypothesis: $T(j) \leq 2^j$ for all $j < i$.
Induction step: Let d be number of out-neighbours of chosen vertex v.

$$T(i) \leq d \cdot T(i - (d-1)) \leq d \cdot 2^{i-(d-1)} \leq d \cdot \frac{2^i}{2^{d-1}} \leq 2^i.$$

Last inequality exploits that $d \geq 2$ and, hence $\frac{d}{2^{d-1}} \leq 1$. □

Remarks:

- Algorithm can be improved to run in $\mathcal{O}(2^k \cdot (n + m))$ time for graphs on n vertices and m arcs $\rightsquigarrow$ linear time for constant k.

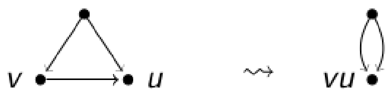


Figure 11: Contraction of only outgoing edges

- Experiments solved instances with $k \leq 190$ and $m \geq 10^7$ in five minutes (3.6 GHz Intel Xeon).

2.3 Problem Kernelisation

The idea is to solve easy parts of the problem in polynomial time and then only the hard kernel of the problem remains.

Example 2.18 (Vertex Cover) We are searching for a vertex cover of size at most k.

- All vertices with more than k neighbours have to be in the vertex cover. Delete them, decrease k for each deleted vertex.
- If the remaining graph has a vertex cover of size k, these vertices can cover at most k^2 edges.

$\rightsquigarrow$ Resulting graph has at most $k^2 + k$ vertices and k^2 edges or has no vertex cover with at most k vertices.

Definition 2.19 (Problem Kernel) Let L be a parametrised problem. A reduction to a problem kernel is a polynomial-time reduction of an instance (I, k) to an instance (I', k') (called problem kernel) such that

1. $k', |I'| \leq f(k)$ for some function f,
2. $(I, k) \in L \Leftrightarrow (I', k') \in L$.

The function f is the size of the problem kernel. (fig. 12) ⊣

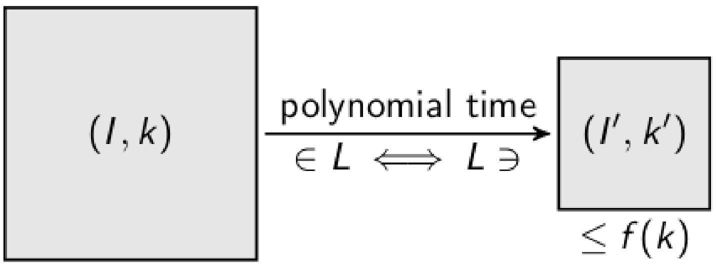


Figure 12: Problem Kernel

Problem 2.20 (Independent Rectangle Set) **Input:** Rectangles in the plane, integer k.
Task: Find set of k rectangles whose projections onto neither axis intersect.
Special case: Input is c-compact: corner points of rectangles chosen from $\{1, \ldots, c\} \times \{1, \ldots, c\}$ for some natural number c. ⊣

Theorem 2.21 (van Bevern et al., Journal of Scheduling, 2014) Independent Rectangle Set has a problem kernel comprising c^3 rectangles.

Proof 2.22 Data reduction rule: If a rectangle r is contained in a rectangle R, i.e. $r \subseteq R$, then delete R. Correctness proof: Original instance (I, k), new instance (I', k). Let (I', k) be a yes-instance. Then, there is a solution S of size k for I'. It is also a solution for I. Hence, (I, k) is a yes-instance. Let (I, k) be a yes-instance. Then, there is a solution S of size k for I. If $R \notin S$, then S is a solution of the same size for I'. If $R \in S$, then $r \notin S$. Hence, $(S \setminus \{R\}) \cup \{r\}$ is a solution of the same size for I'. In both cases, (I', k) is a yes-instance.
Proof of kernel size: We show that after applying data reduction as long as possible, at most c^3 rectangles remain. Let S_{ij} for $1 \leq i, j \leq c$ be the set of rectangles with upper-left corner point (i, j). If $|S_{ij}| > c$, then there are two rectangles r and R whose lower-right corner points have the same x-coordinates $\rightsquigarrow r \subseteq R$ or $R \subseteq r$. Hence, there are at most c^2 sets S_{ij} with $1 \leq i, j \leq c$, each $|S_{ij}| \leq c$: at most c^3 rectangles in total. □

Remark: Given an instance of Independent Rectangle Set, an equivalent c-compact representation for minimum c is computable as fast as sorting.

2.4 Parametrised Hardness

Definition 2.23 (parametrised reduction) Let $L, L' \subseteq \{0,1\}^* \times \{0,1\}^*$ be two parametrised problems. A parametrised reduction from L to L' is a function $r : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ that

1. maps (x, k) to (x', k') in $f(k) \cdot \text{poly}(|(x, k)|)$ time such that
2. $(x, k) \in L \Leftrightarrow (x', k') \in L$ and
3. $k' \leq g(k)$ for computable functions f and g. ⊣

Observation: If L' is fpt, then so is L. If L is not fpt, then L' is neither. Clique parametrised by k is $W[1]$-hard: all problems in $W[1]$ have a parametrised reduction to it. If Clique is fpt, then all problems in $W[1]$ are, but no FPT-algorithms for any $W[1]$-hard problem are known.

Example 2.24

1. A graph has a clique of size at least k iff its complement graph has an independent set of size at least $k \rightsquigarrow$ Independent Set parametrised by k is $W[1]$-hard.
2. A graph has an independent set of size at least k iff it has a vertex cover of size at most $n - k$. But this is not a parametrised reduction. Indeed, Vertex Cover is fpt parametrised by k.

Problem 2.25 (Multicoloured Clique) Input: A properly k-vertex coloured undirected graph $G = (V, E)$. **Question:** Is there a clique containing each colour exactly once? ⊣

Theorem 2.26 Multicoloured Clique parametrised by k is $W[1]$-hard.

Proof 2.27 Reduction from Clique parametrised by k. For some graph G we create k copies, one for each colour and insert all copies of edges between different colours. Then G' has a multicoloured clique of size k iff G has a clique of size k. □

Problem 2.28 (Rainbow Subgraph) Input: An undirected graph $G = (V, E)$, with p edge colours and an integer k.
Task: Find a subgraph of G that has at most k vertices and contains each edge colour exactly once. Applications in bioinformatics. ⊣

Theorem 2.29 (Hüffner, Komusiewicz, Niedermeier, Rötzschke, WG 2014) Rainbow Subgraph parametrised by the number p of colours is $W[1]$-hard.

Proof 2.30 Reduction from Multicoloured Clique: Input graph G, output G':

- Add all vertices of G to G'.

- For each edge $\{u, v\}$ of G add a path $(u, \omega_{\{u,v\}}, v)$ to G' where $\omega_{\{u,v\}}$ is a new vertex.
- For each edge $\{u, v\}$ in G with u of colour i and v of colour j colour the edge $\{u, \omega_{\{u,v\}}\}$ with colour c_{ij} and the edge $\{v, \omega_{\{u,v\}}\}$ with colour c_{ji}.

G' has at most $2\binom{k}{2}$ edge colours. Now we need to show that the graph G has a multicoloured clique of size k iff G' has a rainbow subgraph with $k + \binom{k}{2}$ vertices.

$\Rightarrow$ Let S be a clique of size k in G. The vertices in S have k pairwise different colours. Thus G' has $2\binom{k}{2}$ edges with pairwise different colours between vertices in $S' := S \cup \{\omega_{\{u,v\}} | \{u, v\} \subseteq S\}$ and $|S'| \geq k + \binom{k}{2}$.

$\Leftarrow$ Let S' be the $k + \binom{k}{2}$ vertices of a rainbow subgraph of G'. Since it covers colours c_{i1} for $1 \leq i \leq k$, S' contains $\geq k$ G-vertices. Moreover, S' has at least $\binom{k}{2}$ new vertices: needs $2\binom{k}{2}$ edges to collect all colours every edge is incident to a new vertex and each new vertex is contained in at most two edges. Hence, S' has k G-vertices and $\binom{k}{2}$ new vertices. To collect all $2\binom{k}{2}$ colours every new vertex must have degree two in rainbow subgraph. These correspond to $\binom{k}{2}$ edges between k vertices in G $\rightsquigarrow$ clique on k vertices. $\square$

3 Reduction to Solver Tools

Similar problems occur over and over with different side-constraints. So people constructed solver for whole classes of problems and we can exploit that by modelling our problems to those.

- Every solver accepts problems in a well-defined “language”.
- Describe problem as precisely as possible within the limits of the language.
- The solver decides how to find the solution.
  $\rightsquigarrow$ Declarative instead of imperative programming
- The more powerful the language, the slower the solution process.
- Some languages more suitable for given problem than others. Some combinations even impossible.

Example 3.1 (Solver examples)

- Combinatorial Optimisation/Mathematical Programming based
  - Network Flows
  - (Integer) Linear Programs
- Logic based
  - Testing satisfiability of formulas
  - Model checking for formulas

3.1 Network Flows

Let $G = (V, A)$ be a directed graph, $c : A \to \mathbb{Q}^+$ a capacity function and $s, t \in V$.

Definition 3.2 (flow) A function $f : A \to \mathbb{Q}^+$ is an $s - t$ flow in G if

- $f(u, v) \leq c(u, v)$ (capacity constraint)
- $\forall v \in V \setminus \{s, t\} : \sum_{(v,u)\in A} f(v, u) = \sum_{(u,v)\in A} f(u, v)$ (flow conversation constraint)

The value of the flow f is $\sum_{(s,v)\in A} f(s, v) - \sum_{(v,s)\in A} f(v, s)$. $\dashv$

Theorem 3.3 (Max-Flow-Min-Cut) The maximum value of an $s - t$ flow in G equals the weight of a minimum $s - t$ cut of G.

Problem 3.4 (Fixed Vertex Dissolution) **Input:** Districts D, each containing $b \in \mathbb{N}$ voters, districts $D' \subseteq D$ to be dissolved, a graph representing adjacency of all districts and an integer Δ_b.
Question: Can we dissolve D', that is move the voters of D' to adjacent districts so that each district in $D \setminus D'$ contains $b + \Delta_b$ voters? (figs. 13 and 14) $\dashv$

Theorem 3.5 (van Bevern et al., MFCS 2014) Fixed Vertex Dissolution can be solved by computing the maximum flow in a network.

Proof 3.6 (by picture) Transformation into a flow network.
Given: Instance with $b = 2$ voters per district.
Goal: Increase the number of voters per district by $\Delta_b = 3$.
Starting with the input graph. We know the districts D' to dissolve. We rearrange the districts: On the left the districts to dissolve, to the right the remaining ones. Remove edges between districts to dissolve. Replace remaining edges by arcs from left to right. Add a source s and a sink t. Add an arc from s to each dissolved district node with capacity $b = 2$ and an arc from each remaining district node to t with capacity $\Delta_b = 3$. (see fig. 15)

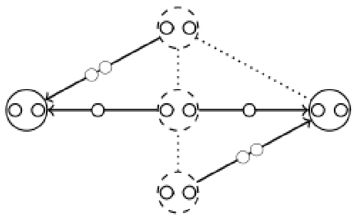


Figure 13: Districts and dissolution

Figure 14: Result

- Check that $b|D'| = \Delta_b|D \setminus D'| \rightarrow$ If not, then return no
- Build flow network as sketched previously
- There is a dissolution iff there is a flow of value $\geq b|D'|$ $\square$

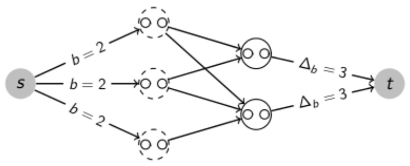


Figure 15: Resulting Flow Network

To find maximum flows there are two standard algorithms (polynomial time):

- Ford-Fulkerson/Edmonds-Karp/Dinic's
  $\rightsquigarrow$ Idea: find an $s - t$ path using only "unsaturated" edges as long as possible and send flow along this path
- Preflow-Push/Push-Relabel Algorithms (Goldberg-Tarjan's)
  $\rightsquigarrow$ Idea: Put the network on a slope: s is high, t low; then simulate water flowing simultaneously from s to t

But we want to use Linear Programming Solver.

3.2 Linear Programs

Definition 3.7 (Linear Program) A Linear Program describes a problem in the following matter. Let c, b be vectors over the reals and A be a matrix over the reals. Find real vector x maximising (or minimising) $c^T x$ subject to $Ax \leq b$.

- $c^T x$ is the objective function.
- Linear inequalities in $Ax \leq b$ are constraints.
- Vector x' is feasible if $Ax' \leq b$.
- Set of feasible vectors is the feasible region.

Often solver require standard form: $Ax = b$ and $x \geq \vec{0}$. ⊣

Example 3.8 (Network Flow as Linear Program) We recall the definition of a $s - t$ flow f in a graph $G = (V, A)$:
A function $f : A \to \mathbb{Q}^+$ is an $s - t$ flow in G if

- $f(u, v) \leq c(u, v)$,
- $\forall v \in V \setminus \{s, t\} : \sum_{(v,u)\in A} f(v, u) = \sum_{(u,v)\in A} f(u, v)$.

A Linear Program to maximise flows looks like this:
Variable $x_{u,v}$ for each $(u, v) \in A$:

$$\begin{aligned} \text{Maximise } & \sum_{(s,v)\in A} x_{s,v} - \sum_{(v,s)\in A} x_{v,s} \\ \text{Subject to } & \forall (u,v) \in A : x_{u,v} \leq c(u,v) \\ & \forall v \in V \setminus \{s,t\} : \\ & \sum_{(v,u)\in A} x_{v,u} = \sum_{(u,v)\in A} x_{u,v} \end{aligned}$$

Definition 3.9 (Simplex Algorithm) Basic idea is to traverse the edges of the polytope induced by the constraints

- Find a vertex of the polytope
- As long as possible: move along an edge of the polytope that increases the objective function

In the worst-case this takes exponential time, but in practice it is often efficient. There are polynomial-time algorithms (see Interior Points Method) ⊣

3.3 Integer Linear Programs

Many applications for Linear Programs call for boolean (0/1) or integer variables. Even deciding feasibility for integer vectors x, that is "given A and b, is there some integer vector satisfying $Ax \leq b$?", is NP-complete.

Proof 3.10 (NP-hardness) Let $F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{k_i} L_{i,j}$ be some SAT-formula. Let $x_i \in \{0, 1\}$ and regard

$$Y_{i,j} = \begin{cases} x_j & : L_{i,j} = X_j \\ 1 - x_j & : L_{i,j} = \neg X_j \end{cases}$$

$$\forall i \in \{1, \ldots, n\} : \sum_{j=1}^{k_i} Y_{i,j} \geq 1$$

This ILP has a feasible solution iff F has a satisfying assignment. Hence feasibility of ILPs is NP-hard.□

Problem 3.11 (Dissolution) Input: Districts D each containing $b \in \mathbb{N}$ voters, a graph $G = (D, E)$ representing adjacency of all districts and an integer Δ_b.
Task: Find a minimum size set of districts D' to dissolve, that is, their voters are moved to adjacent districts so that each district in $D \setminus D'$ contains at least $b + \Delta_b$ voters.
Integer Linear Program: Boolean variable $x_b \in \{0, 1\}$ for each $d \in D$: "Dissolve district d". Integer

variable $y_{u,v} \geq 0$ for each $u, v \in D$: "Move $y_{u,v}$ voters from u to v".
Optimisation goal:

$$\text{Minimise} \sum_{d \in D} x_d$$

Constraints:

$$\begin{aligned}
\forall d \in D : \sum_{v \in N(d)} y_{d,v} &= b x_d \\
\forall d \in D : \sum_{v \in N(d)} y_{v,d} &\geq \Delta_b (1 - x_d) \\
\forall d \in D : \sum_{v \in N(d)} y_{v,d} &\leq b|D|(1 - x_d)
\end{aligned}$$

$\dashv$

The solution process for integer linear programs is substantially different from Linear Programs:

- Most solvers use search tree approach
- Many rules, tweaks, heuristics to
  - Make favourable reformulations
  - Cut off parts of search tree
  - Make favourable branches

Problem 3.12 (Travelling Salesperson) Input: n cities, distances c_{ij} for each ordered pair (i, j) of them.
Task: Find a minimum-length tour through all cities.
Integer Linear Program:

- Subtour (ST) Formulation
  Boolean variable $x_{ij} \in \{0, 1\}$ for each pair i, j of cities: "arc(i, j) is part of the tour".
  Minimise $\sum_{1 \leq i,j \leq n} c_{i,j} x_{i,j}$
  Subject to

$$\begin{aligned}
\forall k = 1, \ldots, n : \sum_{1 \leq i \leq n} x_{i,k} &= 1 \\
\forall k = 1, \ldots, n : \sum_{1 \leq j \leq n} x_{k,j} &= 1 \\
&\text{Assignment constraints} \\
\forall S \subsetneq \{1, \ldots, n\}, S \neq \emptyset : \sum_{i,j \in S} x_{i,j} &\leq |S| - 1 \\
&\text{Subtour breaking constraints}
\end{aligned}$$

  $\rightsquigarrow \mathcal{O}(n^2)$ variables, $\mathcal{O}(2^n)$ constraints

- Miller-Tucker-Zemlin(MTZ) Formulation
  Boolean variable $x_{ij} \in \{0, 1\}$ for each pair i, j of cities: "arc(i, j) is part of the tour".
  Integer variable $u_i \geq 0$ for each city i: "city i comes in the u_ith place of the tour".
  Minimise $\sum_{1 \leq i,j \leq n} c_{i,j} x_{i,j}$

$$\forall k = 1, \dots, n : \sum_{1 \le i \le n} x_{i,k} = 1$$

$$\forall k = 1, \dots, n : \sum_{1 \le j \le n} x_{k,j} = 1$$

Assignment constraints

$$u_1 = 1$$

$$\forall i = 2, \dots, n : 2 \le u_i \le n$$

$$\forall i, j \in \{2, \dots, n\} : u_i - u_j + 1 \le (n-1)(1 - x_{i,j}) (x_{i,j} = 1 \Rightarrow u_i + 1 \le u_j)$$

Subtour breaking constraints

$\rightsquigarrow \mathcal{O}(n^2)$ variables, $\mathcal{O}(n^2)$ constraints ⊣

3.4 ILP Solvers and Relaxations

Inner Workings of an ILP Solver: Most Integer Linear Programming solvers are search tree algorithms: Let P be a minimisation Integer Linear Program and α be the objective value of some feasible vector x (found e.g. by heuristic or ∞ initially).

```
SolveMinILP($P, \alpha$)
if $P$ is trivial then
    return the minimum objective value of $P$.
end if
Find a lower bound $\beta$ on the minimum objective value of $P$.
if $\beta > \alpha$ then
    return $\alpha$
end if
Choose a variable $x_i$ and a value $\delta$ to branch on.
$P_1 \leftarrow P$ with constraint $x_i \le \delta$
$P_2 \leftarrow P$ with constraint $x_i > \delta$
return SolveMinILP($P_2$,SolveMinILP($P_1, \alpha$)).
```

Finding the lower bound on objective value: Relaxation of integrality constraints: Treat each integer variable x_i as a real variable: $x_i \in \mathbb{R}$ instead of $x_i \in \mathbb{Z}$.
$\rightsquigarrow$ Integer Linear Program becomes ordinary Linear Program $\rightsquigarrow$ We can solve the Linear Program efficiently...
As a consequence of this we are looking for a Integer Linear Program that gives a "tight" Linear Program as relaxation!

Definition 3.13 (valid, cutting plane)

- A constraint is valid if it does not change the feasible region of the ILP.
- Valid constraint is cutting plane if it separates the optimum value of the relaxation from the optimum of the ILP. ⊣

Now we add cutting planes on-demand:

```
SolveMinILP($P, \alpha$)
if $P$ is trivial then
    return the minimum objective value of $P$.
end if
repeat
    $\beta \leftarrow$ Minimum objective value of relaxation rel($P$) of $P$.
    $\hat{x} \leftarrow$ Corresponding vector feasible for rel($P$).
    Find valid constraints violated by $\hat{x}$ , add them to $P$.
```

```
until no new constraints are found
if $\beta > \alpha$ then
    return $\alpha$
end if
Choose a variable $x_i$ and a value $\delta$ to branch on.
$P_1 \leftarrow P$ with constraint $x_i \leq \delta$
$P_2 \leftarrow P$ with constraint $x_i > \delta$
return SolveMinILP($P_2$,SolveMinILP($P_1, \alpha$)).
```

This is called row generation.
To add subtour breaking cutting planes for TSP, we have to find strict subset S of cities so that $\sum_{i,j \in S} x_{i,j} > |S| - 1$ efficiently, which can be done using flow techniques.

Consider the complete directed graph, $|V| = n$, weight$(i,j) = x_{i,j}$. If we have a violation of subtour breaking, then we find a cut of size < 2.

$$\sum_{i \in S} \left(\sum_{j \notin S} (x_{i,j} + x_{j,i}) + \underbrace{\sum_{j \in S} (x_{i,j} + x_{j,i})}_{>2|S|-2 \text{ when summed over } i} \right) = 2|S|$$

Hence

$$\sum_{i \in S} \sum_{j \notin S} (x_{i,j} + x_{j,i}) < 2$$

but this sum is the cut between S and $V \setminus S$. The Min-Cut can be found via network flow.

Conclusion:
We now have seen three problem categories with successively more powerful solvers: Network Flows $\rightsquigarrow$ Linear Programs $\rightsquigarrow$ Integer Linear Programs and we know:

- Many different formulations for one problem may exist.
- "Best formulation" not obvious.
- For good running times it is often useful to successively improve formulation within the solution process.

3.5 Satisfiability

We have already seen that every problem in NP can be reduced to CNF-SAT. This has application in Hardware/Software verification and Planning. It is naturally suited for decision problems (in contrast to optimisation problems).

Problem 3.14 (CNF-SAT) Input: Boolean formula F in conjunctive normal form (CNF).
Task: Decide whether F has a satisfying truth assignment. $\dashv$

Problem 3.15 (Co-Clustering) Input: A matrix $M \in \mathbb{Z}^{m \times n}$, integers $k, l, c \in \mathbb{N}$.
Task: Find a (k,l)-co-clustering $(\mathcal{R}, \mathcal{C})$ of M such that $\max_{(s,t) \in [k] \times [l]} (\max M_{st} - \min M_{st}) \leq c$.
A matrix $\mathcal{U} = (u_{st}) \in \mathbb{Z}^{k \times l}$ is called a cluster boundary. A (k,l)-co-clustering satisfies a cluster boundary $\mathcal{U}$ iff for all $(s,t) \in [k] \times [l]$ and $m_{ij} \in M_{st}$ holds that $m_{ij} \in [u_{st}, u_{st} + c]$. Observation: (M, k, l, c) is a "yes"-instance iff there exists a (k,l)-co-clustering that satisfies some cluster boundary $\mathcal{U}$. (that is the case, when is satisfies the cluster boundary with the minimum value for each cluster)
Reduction to SAT-Solvers: Given a matrix M and a cluster boundary $\mathcal{U}$, define a CNF formula $\phi_{M,\mathcal{U}}$ that is satisfiable iff there exists a (k,l)-co-clustering of M satisfying $\mathcal{U}$. Variables:

- $x_{i,s}$: row i is contained in R_s (s-th row subset)

- $y_{j,t}$: column j is contained in C_t (t-th column subset)

Clauses:

- $K_i := x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,k}$: row i must be in some row subset
- $L_j ;= y_{j,1} \vee y_{j,2} \vee \cdots \vee y_{j,l}$: column j must be in some row subset
- For each $m_{ij} \notin [u_{st}, u_{st} + c]$, $U_{ijst} := \neg x_{i,s} \vee \neg y_{j,t}$: m_{ij} is not contained in cluster M_{st}

We can solve Co-Clustering by checking satisfiability of $\phi_{M,\mathcal{U}}$ for all possible cluster boundaries $\mathcal{U}$. ⊣

3.6 Model Checking

To check "Does truth assignment A satisfy boolean formula F?" is easy. A solver for this problem is not powerful. But now we increase the expressive power of the formula: Monadic Second Order Logic. We focus on graphs, so a formula may additionally contain:

- Vertex/edge, set variables (sets containing vertices or edges)
- isvertex(v), isedge(e)
- inc(v, e): "vertex v and edge e are incident"
- adj(u, v): "vertices u and v are adjacent"
- Equality for vertices/edges, set variables
- Quantifications ($\exists, \forall$) over vertices/edges, set variables

For simplicity, we allow to quantify over vertices, edges, vertex sets and edge sets. We call the vertex set V and the edge set E.

Example 3.16 (3-Colourable)

$$\begin{aligned}&\exists A_1, A_2, A_3 \subseteq V\\ &\qquad\left(\forall v \in V \left(\bigvee_{i=1}^{3} (v \in A_i) \wedge \bigwedge_{i \neq j} (\neg v \in A_i \vee \neg v \in A_j)\right) \wedge\right.\\ &\qquad\left.\forall u \in V \forall v \in V \left(\neg \mathrm{adj}(u, v) \vee \bigwedge_{i=1}^{3} (\neg u \in A_i \vee \neg v \in A_i)\right)\right)\end{aligned}$$

Problem 3.17 (Monadic Second Order Logic Model Checking for Graphs) Input: A formula F in Monadic Second Order logic and a graph G.
Task: Decide whether G is a model of F, that is, F is satisfied by G. ⊣

Problem 3.18 (Dissolution) Input: Districts D each containing $b \in \mathbb{N}$ voters, a graph $G = (D, E)$ representing adjacency of all districts and an integer Δ_b.
Task: Decide if there is a set of districts D' to dissolve, that is, their voters are moved to adjacent districts so that each district in $D \setminus D'$ contains exactly $b + \Delta_b$ voters.
First step: Multiply each edge in G b times.
Second step: Model:

$$\exists D' \subseteq D \exists M \subseteq E\big(\mathrm{movement}(D', M) \wedge \mathrm{dissolution}(D', M)\big)$$

$$\mathrm{movement}(D', M) := \forall m \in M \exists d_1 \in D \exists d_2 \in D\big(d_1 \in D' \wedge d_2 \notin D' \wedge \mathrm{inc}(d_1, m) \wedge \mathrm{inc}(d_2, m)\big)$$

$$\begin{aligned}\mathrm{dissolution}(D', M) := &\Big(\forall d \in D' \exists M' \subseteq M\big(\mathrm{card}_b(M') \wedge \forall m \in M(\mathrm{inc}(d, m) \Leftrightarrow m \in M')\big)\Big)\\ &\wedge \Big(\forall d \in D \setminus D' \exists M' \subseteq M\big(\mathrm{card}_{\Delta_b}(M') \wedge \forall m \in M(\mathrm{inc}(d, m) \Leftrightarrow m \in M')\big)\Big)\end{aligned}$$

$$\text{where } (x \Leftrightarrow y) := (\neg x \vee y) \wedge (\neg y \vee x)$$

$$\mathrm{card}_i(X) := \exists x_1 \exists x_2 \ldots \exists x_i \left(\left(\bigwedge_{j=1}^{i} x_i \in X\right) \wedge \left(\bigwedge_{j=1}^{i} \bigwedge_{k=j+1}^{i} (x_j \neq x_k)\right) \wedge \forall x \in X \left(\bigvee_{j=1}^{i} x_j = x\right)\right)$$ ⊣

A freely available solver is Sequoia.

- In general decision problem is PSPACE-complete
- Efficient on “tree-like” graphs (linear time for constant-sized formulas)
- Not touched here: Sequoia can also maximise/minimise size of an existentially quantified set variable

3.7 Conclusion and Related Topics

We did look at:

- Combinatorial Optimisation/Mathematical Programming based
  - Network Flows
  - (Integer) Linear Programs
- Logic based
  - Testing satisfiability of formulas
  - Model checking for formulas

We did not cover here:
Stochastic Programming, Constraint Programming, Quantum Computing, ...

4 Approximation Algorithms

4.1 Introduction

Remark 4.1 Most natural optimisation problems are NP-hard. Under the widely believed conjecture that $P \neq NP$ their exact solution is not computable in deterministic polynomial time. Possible ways out of that misery are

- heuristic solving methods: These do not give guarantees.
- exact and parametrised algorithms: exponential running time, But for practical applications this might suffice, because this also includes $c^k \cdot n$ for small k, or 1.001^n.
- polynomial-time approximation algorithms: Here we get non-optimal solutions, but we have a guarantee on the running time.

Example 4.2 (Minimum vertex-cover) Input: an undirected graph $G = (V, E)$
Goal: Find a minimum-cardinality vertex subset $V' \subseteq V$ such that each edge in E has at least one endpoint in V'.
A greedy heuristic is to choose a vertex of highest degree, put it into our vertex cover and then delete it from the graph. We repeat this until there is no edge left. In this example the greedy algorithm provides

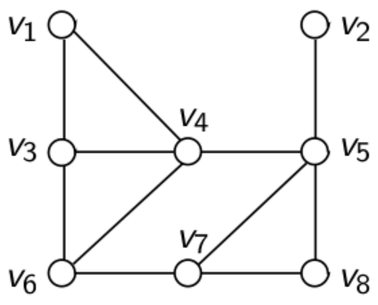


Figure 16: initial state with $C = \emptyset$

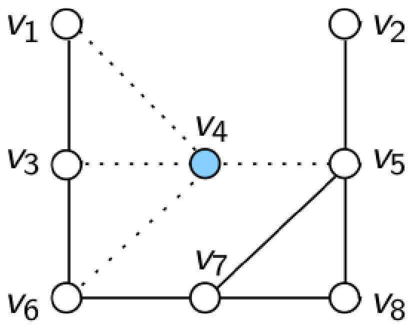


Figure 17: $C = \{v_4\}$

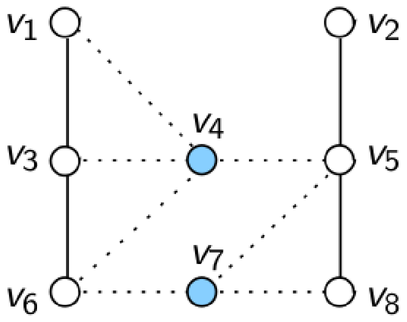


Figure 18: $C = \{v_4, v_7\}$

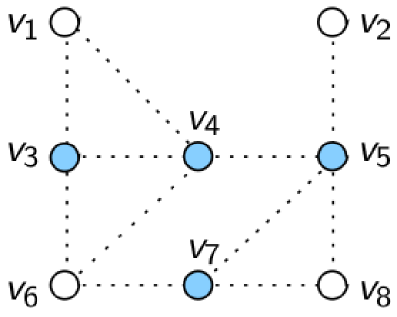


Figure 19: $C = \{v_4, v_7, v_3, v_5\}$

an optimal solution. For general input graphs, it outputs a vertex cover that can be by a factor of $\ln |V|$ greater than an optimal vertex cover.

Proof 4.3 Create a set L of n vertices. Then create sets R_i of $\left\lfloor \frac{n}{i} \right\rfloor$ vertices, each of degree i and any two nodes in R_i have disjoint neighbours in L. More formally for $R_i = \left\{u_{i,j} : j = 1, \ldots, \left\lfloor \frac{n}{i} \right\rfloor \right\}$ we have edges

$\{u_{i,1}, v_1\}, \ldots, \{u_{i,1}, v_i\}, \{u_{i,2}, v_2\}, \ldots$. Then the algorithm will delete all R_i, who have $\approx n \ln n$ nodes. The optimum would be to delete L, which has n vertices.

Therefore this is not a good approximation.

Definition 4.4 (Matching) An edge subset M of an undirected graph $G = (V, E)$ is a matching if no two edges in M share an endpoint. A matching M is maximal if there is no matching M' with $M \subsetneq M'$. A matching M is a maximum matching if there is no matching M' with $|M| < |M'|$.

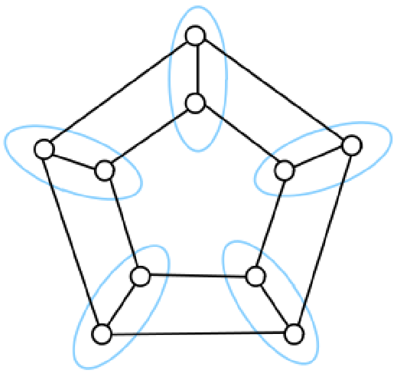


Figure 20: a maximal/maximum matching

Algorithm 4.5 (Approximation for VC via matching) Find a maximal matching for G and output the set C of matched vertices. In the graph we just regarded this algorithm might output a vertex cover

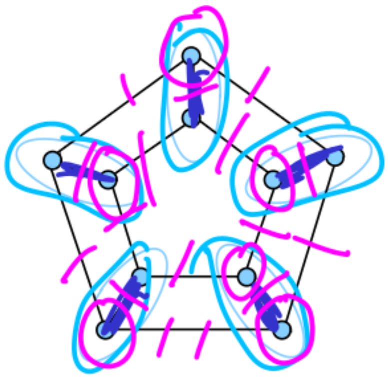


Figure 21: algorithm applied to previous example

of size 10, marked blue. But the optimal solution has size 6, marked pink. ⊣

Theorem 4.6 *Let C_{opt} be an optimal vertex cover of G and let C be the output of the described algorithm. Then, $|C| \leq 2 \cdot |C_{\text{opt}}|$.*

Proof 4.7 Let M be our maximal matching with vertex set C. Assume we have an uncovered edge $e = \{u, v\}$. Then neither u nor v lies in C. So we can extend M to get a bigger matching $M \cup \{e\}$, contradicting the maximality. Hence every edge is covered. □

Algorithm 4.8 Iteratively select both endpoints of arbitrarily chosen edges. Delete that edge from the graph and continue recursively until there is no edge left. ⊣

Theorem 4.9 *This algorithm outputs a vertex cover C with $|C| \leq 2 \cdot |C_{\text{opt}}|$ for an optimal vertex cover C_{opt}.*

Proof 4.10 For every edge $e = \{u, v\}$ we need to have $u \in C_{\text{opt}}$ or $v \in C_{\text{opt}}$. Therefore we take at most twice as many nodes as necessary. □

Remark 4.11 For the last two algorithms the ratio 2 is tight. Consider the input $K_{r,r}$. Then an optimal solution would take r nodes, whereas both algorithms choose all nodes.

Remark 4.12 For vertex cover on planar graphs there are algorithms that provide a $1 + \varepsilon$-approximation for any $\varepsilon > 0$.

Remark 4.13 It is possible to construct an optimal solution out of that greedy solution. For each edge $\{u, v\}$ we have the three possibilities

- $u \in C_{\text{opt}}$, $v \notin C_{\text{opt}}$
- $u \notin C_{\text{opt}}$, $v \in C_{\text{opt}}$
- $u \in C_{\text{opt}}$, $v \in C_{\text{opt}}$

So using a search tree we can find the optimal solution in time 3^k where k is the number of chosen edges.

Remark 4.14 It is a long-standing open problem in the field of approximation algorithms whether Minimum Vertex Cover is approximable in polynomial time within a factor $2 - \varepsilon$ for a constant $\varepsilon > 0$. There is the Unique Games Conjecture, which is used for lower bounds in approximation algorithms. If it holds, this would imply that we cannot do better than factor 2.

4.2 Basic Concepts

Definition 4.15 (NP-optimisation problem) An NP-optimisation problem Π consists of

- a set D_Π of valid instances, recognisable in polynomial time
- Each instance $I \in D_\Pi$ has a set $S_\Pi(I)$ of feasible solutions.
- Every solution $s \in S_\Pi(I)$ is of length polynomially bounded in $|I|$.
- There is a polynomial-time algorithm that, given a pair (I, s), decides $s \in S_\Pi(I)$.
- There is a polynomial-time computable objective function f_Π that assigns a non-negative rational number to each pair (I, s).
- Finally, Π is specified to be either a minimisation or a maximisation problem.

An optimal solution for an instance of a minimisation (maximisation) problem is a feasible solution that achieves the smallest (largest) objective function value. We use $\text{OPT}(I)$ to denote the objective function value of an optimal solution to instance I.
With every NP-optimisation problem, one can naturally associate a decision problem by giving a bound B on the optimal solution. Thus, the decision version of an NP-optimisation problem Π consists of pairs (I, B) with I being an instance of Π and B being a rational number. Given (I, B) we ask whether there is some $s \in S_\Pi(I)$ such that $f_\Pi(s) \geq B$ for a maximisation problem ($f_\Pi(s) \leq B$ for a minimisation problem).
Hardness for an NP-optimisation problem is established by showing that its decision version is NP-hard. In this sense, we also speak of NP-hard optimisation problems. ⊣

Remark 4.16 If the image of f_Π is known to be discrete then optimisation and decision problem are equivalent in the sense that if you can solve the decision problem in polynomial time, then you can solve the optimisation problem in polynomial time by binary search. The other direction is trivial.

Definition 4.17 (Approximation algorithm) Let Π be an optimisation problem, an algorithm $\mathcal{A}$ is called an approximation algorithm if, for any given instance I of Π, $\mathcal{A}$ returns a feasible solution. ⊣

Remark 4.18 This definition is unsatisfactory since it does not ask for guarantees. Hence, we focus on approximation algorithms which provide "near-optimal" solutions, that is, approximation algorithms with guaranteed performance.
In the remainder of this chapter, we focus on the approximation factors of the algorithms, leaving the (obvious) polynomial running times undiscussed.

Definition 4.19 (Relative Approximation) Let Π be a minimisation problem and let δ be a function, $\delta : \mathbb{N}^+ \to \mathbb{Q}^+$ with $\delta \geq 1$.
An algorithm $\mathcal{A}$ is said to be a factor-δ approximation algorithm for Π if, on each instance I , $\mathcal{A}$ produces a feasible solution $s \in S_\Pi(I)$ such that $f_\Pi(I, s) \leq \delta(|I|) \cdot OPT(I)$ and the running time of A is bounded by a fixed polynomial in $|I|$.
If Π is a maximisation problem, we require $\delta \leq 1$ and $f_\Pi(I, s) \geq \delta(|I|) \cdot OPT(I)$
The approximation factor δ is also called performance ratio or approximation guarantee.
An NP-hard optimisation problem Π is called δ-approximable if there exists a factor-δ approximation algorithm for Π. ⊣

Remark 4.20 Here our solution might differ from the Optimum by some factor. In some cases there are algorithms where we can even achieve an additive difference.

Example 4.21 (Edge Colouring) Input: A graph $G = (V, E)$
Goal: What is the minimum number k to colour the edges with numbers $\{1, \ldots, k\}$ such that no incident edges have the same colour?
An obvious lower bound is the maximum degree $\Delta(G)$. But there is also an approximation that uses at most OPT +1 colours. So here we have an additive difference.

4.3 Approximation for TSP

Problem 4.22 (Traveling Salesman Problem) Input: A set $\{1, \ldots, n\}$ of "cities" with non-negative distances $d(i, j)$ for $1 \leq i, j \leq n$, in particular a complete graph.
Goal: Find a tour of minimum length that visits each city exactly once. ⊣

Theorem 4.23 *Unless $P = NP$ there is no polynomial-time algorithm which gives a "non-trivial approximation" for TSP.*

Proof 4.24 Assume we had an approximation of factor $(1 + \varepsilon)$ for TSP. We now compute a reduction from Hamiltonian circle. Given some Graph $G = (V, E)$ we construct a complete graph G' with distances

$$d(i,j) = \begin{cases} 1 & : \{i,j\} \in E \\ n(2+\varepsilon) & : \text{else} \end{cases}$$

Now G has a Hamiltonian cycle iff G' has a TSP-tour of length n. So assuming G has a Hamiltonian cycle, our approximation algorithm will find a TSP-tour of length at most $n(1 + \varepsilon)$. But this is only possible if none of the traveled edges has weight $n(2 + \varepsilon)$. Hence the TSP tour we found was a Hamiltonian cycle in the original graph. □

4.3.1 Metric TSP

decide cost as $c : E \to \mathbb{Q}$ or $d : V \times V \to \mathbb{Q}$

Remark 4.25 We now assume that our distance function is a metric, i.e. it satisfies the triangle inequality.

$$d(u, v) \leq d(u, w) + d(w, v)$$

for all $u, v, w \in V$. Furthermore the cost of some graph G is denoted $c(G)$.

Algorithm 4.26 We will now construct a simple factor-2 algorithm.

1. Find a minimum spanning tree T of G.
2. Double every edge of T to create an Eulerian graph T'.
3. Find an Euler tour $\mathcal{T}$ through the graph T'.
4. Output the tour $\mathcal{C}$ that visits the nodes in order of their first appearance in $\mathcal{T}$. ⊣

Steps (2)-(4) can be regarded as making a depth-first search. In $\mathcal{T}$ we visit each edge of T twice. In step (4) we take shortcuts, so this is where we use the triangle inequality.

it would be nice to add an example picture

Theorem 4.27 *The above algorithm 4.26 is a factor-2 approximation for TSP.*

Proof 4.28 If we remove any edge from a TSP tour we get a spanning tree. Therefore $c(T) \leq \text{OPT}$. By doubling each edge we get $c(\mathcal{T}) = 2c(T)$ and due to the "shortcutting" we have $c(\mathcal{C}) \leq c(\mathcal{T})$, which gives us $c(\mathcal{C}) \leq 2 \cdot \text{OPT}$. □

Remark 4.29 The given bound is tight. Let $n = 2k+2$. Consider K_n with $V = \{0, \dots, 2k\} \cup \{\omega\}$. Define the cost

$$c(\{u,v\}) = \begin{cases} 1 & : u = \omega \vee u \equiv v+1 \mod (2k+1) \\ 2 & : \text{else} \end{cases}$$

So we basically have a "wheel" of cost 1 and all the other edges have cost 2. A minimum spanning tree is just a star with center ω. The order of traversal in the DFS is not deterministic, so we can end up always visiting $v+2 \mod (2k+1)$ after v. Thus all "shortcuts" have length 2. So our tour has length $2n-2$. The optimal tour goes around the circle, thus having cost n.

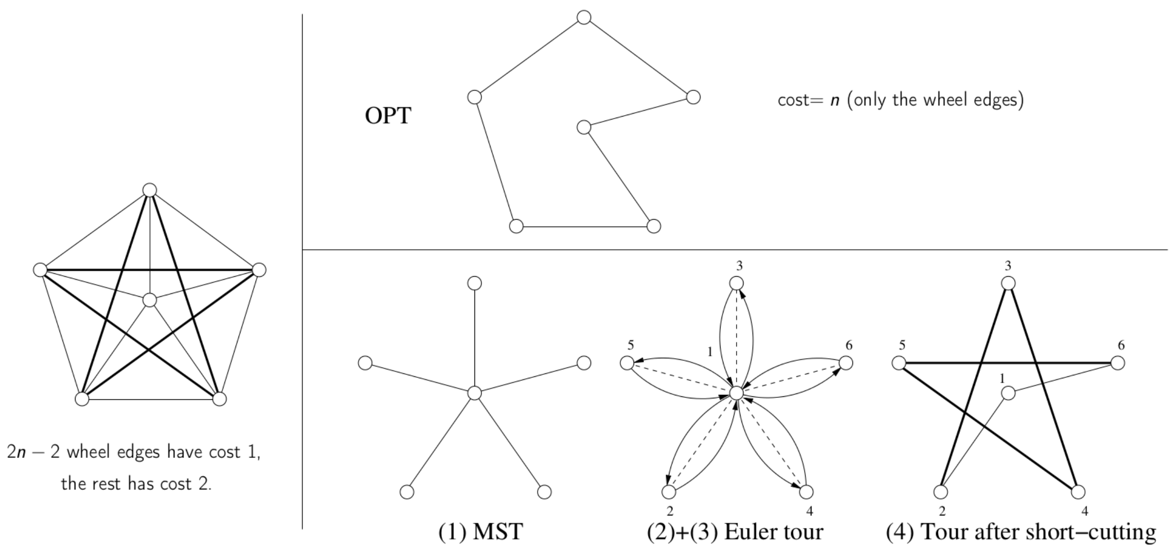

OPT
cost= n (only the wheel edges)
$2n-2$ wheel edges have cost 1,
the rest has cost 2.
(1) MST
3
5
6
1
2
4
(2)+(3) Euler tour
3
5
6
1
2
4
(4) Tour after short–cutting

Figure 22: example for $n = 6$

Algorithm 4.30 We now give a factor-$\frac{3}{2}$ approximation for TSP.
Idea: Again we start with a minimal spanning tree. But instead of doubling its edges, we only add a minimum cost perfect matching on odd-degree vertices.

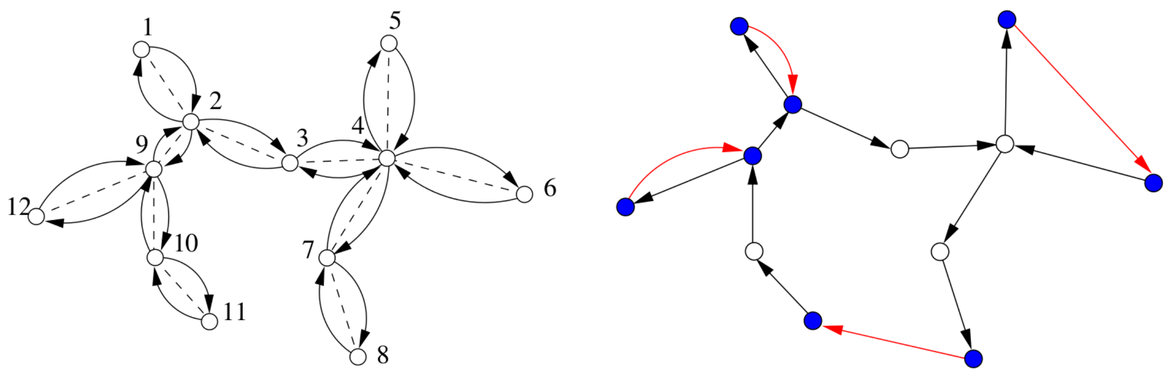


Figure 23: comparison: doubling edges vs. matching

1. Find a minimum spanning tree T of G.
2. Compute a minimum cost perfect matching M on the odd-degree vertices. Add M to T to obtain an Eulerian graph.
3. Find an Euler tour $\mathcal{T}$ in that Eulerian graph.
4. Output the tour $\mathcal{C}$ that visits the vertices of G in order of their first appearance in $\mathcal{T}$. $\dashv$

Lemma 4.31 $c(M) \leq \frac{1}{2}\,\mathrm{OPT}$

Proof 4.32 Let τ denote an optimal TSP tour in G. Let V_o be the odd degree-vertices in T. By Handshake Lemma there is an even number of them. Then we obtain another tour τ' on V_o by shortcutting τ. Since the triangle inequality holds we have $c(\tau') \leq c(\tau) \leq \mathrm{OPT}$. Now τ' is the union of two perfect matchings on V_o, each consisting of alternate edges ($|V_o|$ even). So one of these matchings must be of size $\leq \frac{1}{2}c(\tau') \leq \frac{1}{2}\,\mathrm{OPT}$, so especially M as minimum cost matching can have at most that size. $\square$

Theorem 4.33 *Algorithm 4.30 achieves an approximation guarantee of $\frac{3}{2}$ for metric TSP.*

Proof 4.34 Our tour has cost

$$c(\mathcal{T}) = c(T) + c(M) \leq \mathrm{OPT} + \frac{1}{2}\,\mathrm{OPT} = \frac{3}{2}\,\mathrm{OPT} \qquad \square$$

Remark 4.35 The given bound is tight, as fig. 24 shows. The following graph has n vertices, with n being odd. TO complete the graph add give every other edge as weight the distance of its endpoints in this graph. Thick edges represent the MST found in the first step.
Algorithm 4.30 would choose T and the long edge for a total cost $n - 1 + \lfloor \frac{n}{2} \rfloor$. The optimal solution is a circle around, having cost n.
Here our MST was a bad choice, but this choice can be made deterministic by setting the cost of the chosen edges to $1 - \varepsilon$. This does not change the arguments but robs us of our choice in the first step.

4.4 Approximating Set Cover

Problem 4.36 (Minimum Set Cover) Input: A universe U of n elements, a collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subsets of U and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$.
Goal: Find a minimum-cost subcollection of $\mathcal{S}$ that covers all elements of U. $\dashv$

Example 4.37 For simplicity take $\forall s \in \mathcal{S}. c(S) = 1$ and look at the following example for minimum set cover.

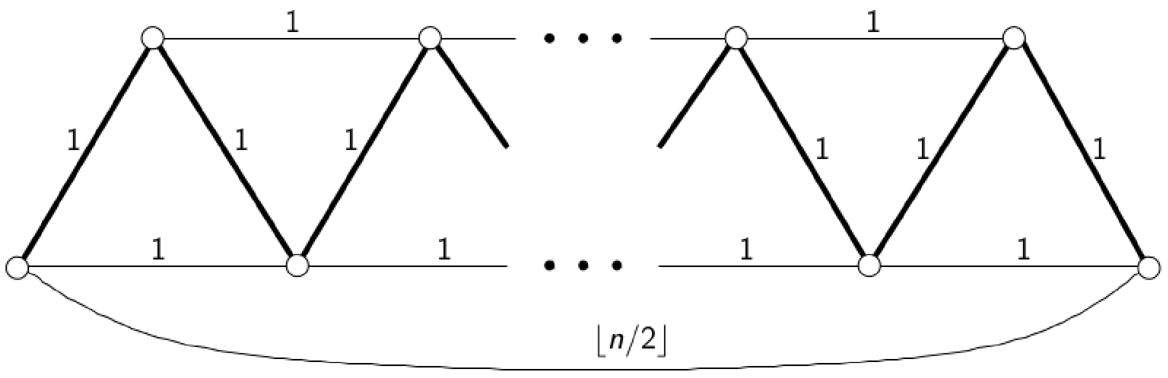
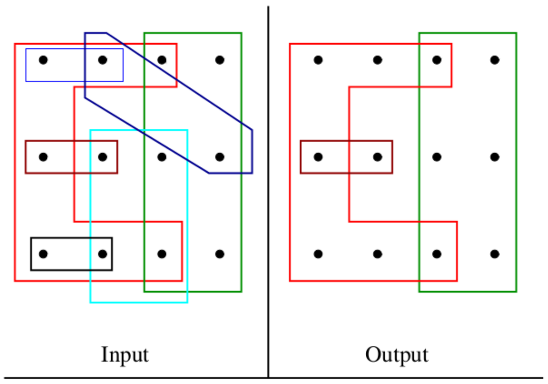


Figure 24: tight example for algorithm 4.30



Algorithm 4.38 (Greedy algorithm for minimum set cover) Let C be the set of not yet covered elements. Define cost-effectiveness of a set $S \in \mathcal{S}$ as $\frac{c(S)}{|S \cap C|}$.
Idea: Choose the set which is the most cost-effective.

```
$C \leftarrow U$
**while** $C \neq \emptyset$ **do**
    Pick the set whose cost-effectiveness is smallest, say $S$.
    $C \leftarrow C \setminus S$
**end while**
Output the picked sets
```
⊣

Example 4.39 Consider the following example. The algorithms will choose the sets in the order as they are numbered. So the algorithm outputs four weight-1 subsets, while the optimal solution has weight of $2(1+\varepsilon)$.

Definition 4.40 (Harmonic series) The harmonic series is $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$. ⊣

Theorem 4.41 *Algorithm 4.38 is an H_n-factor approximation for Minimum Cost Set Cover.*

Proof 4.42 Let $F = \{S_1, \ldots, S_r\}$ be the sets of the greedy algorithm and $\mathrm{OPT} = \{O_1, \ldots, O_l\}$. Let X_i be the elements uncovered before step i, so $X_1 = U$ and $X_{r+1} = \emptyset$. Then we have $S_i \cap X_i = X_i \setminus X_{i+1}$. By our algorithm

$$\forall j \leq l : \frac{c(S_i)}{|S_i \cap X_i|} \leq \frac{c(O_j)}{|O_j \cap X_i|} \implies \forall i \leq l : |O_j \cap X_i| \cdot \frac{c(S_i)}{|S_i \cap X_i|} \leq c(O_j)$$

$$\implies \frac{c(S_i)}{|S_i \cap X_i|} \cdot |X_i| \leq \frac{c(S_i)}{|S_i \cap X_i|} \cdot \left| \bigcup_{j=1}^{l} (O_j \cap X_i) \right| \leq \frac{c(S_i)}{|S_i \cap X_i|} \sum_{j=1}^{l} |O_j \cap X_i| \leq \sum_{j=1}^{l} c(O_j)$$

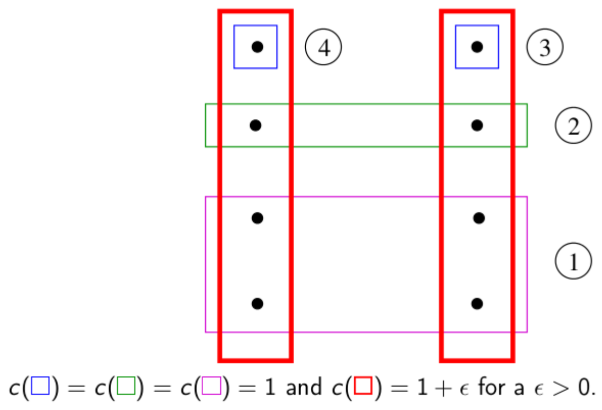

4
3
2
1
$c(\square) = c(\square) = c(\square) = 1$ and $c(\square) = 1 + \epsilon$ for a $\epsilon > 0$.

Figure 25: example for greedy algorithm for set cover

So our algorithm has cost

$$\sum_{i=1}^{r} c(S_i) \leq \text{OPT} \cdot \sum_{i=1}^{r} \frac{|S_i \cap X_i|}{|X_i|} = \text{OPT} \cdot \sum_{i=1}^{r} \frac{|X_i \setminus X_{i+1}|}{|X_i|} \overset{(*)}{\leq} \text{OPT} \cdot \sum_{k=1}^{|U|} \frac{1}{k} \approx \text{OPT} \cdot \ln |U| \qquad \square$$

where for (*) we use $|X_i| \leq n - i$ since $X_{i+1} \subset X_i$.

one could consider adding the amortised costs

Remark 4.43 This bound is tight. Consider $U = \{1, \dots, n\}$ and $\mathcal{S} = \{\{k\} : k \in U\} \cup \{U\}$ with costs $c(\{k\}) = \frac{1}{k}$ and $c(U) = 1 + \varepsilon$. Then the algorithm will choose all singleton sets, resulting in a cost of H_n. The optimal solution is U for cost $1 + \varepsilon$.

5 Online Algorithms

5.1 Introduction and Basic Concepts

Definition 5.1 (Online Algorithms) An online algorithm is an algorithm that works on data, which is given step by step. So you have to make decision without knowing the complete data. ⊣

Example 5.2 Typical applications for online algorithms are

- trading: changing prices, we do not know how the prices of shares will be in the future
- scheduling: incoming jobs, we do not know when the next job will arrive, and how large it will be
- memory management/caching: we do not know which page will be requested next
- dynamic data structures: We store our elements in a list and we may reorder the elements (e.g. put them to the back or the front). The cost for accessing an element is its index in the list. So by putting frequently accessed elements to the front we can save time. But again we do not know which element will be accessed next.
- Robot motion planning
- Searching
- energy-efficient algorithms/Green Computing: process data in a way that only few energy is consumed

Remark 5.3 When evaluating an online algorithm, we compare it with an offline algorithm and regard the ratio. Obviously the offline algorithm will be at least as good as the online algorithm. So that ratio always is at least 1.

Example 5.4 (Ski Rental) **Scenario:** We go skiing and have to decide whether to rent or to buy ski. In a few days it will thaw, but we do not know when. Still we have to decide. Renting costs 1 per day, buying has a fixed cost of b (we assume they are not resold).
Task: Decide whether and when to buy ski, without knowing the duration of the holidays.
Model: There is an "online adversary" who determines an unknown day D in the future, where our skiing holidays end.
Goal: Minimise the costs.

- Offline: D is known, so we rent the ski if $D < b$, otherwise we buy them. Hence our cost is $\min\{b, D\}$.
- Online: D is unknown, Any strategy has to following form
  1. Choose some number $t \geq 1$
  2. rent for the first $t-1$ days (or for the whole holiday, if it turns out $D < t$)
  3. buy the ski on the morning of day t

  This algorithm has cost

$$c = \begin{cases} D & : D < t \\ t-1+b & : D \geq t \end{cases}$$

  In an online algorithm, the adversary knows t. Hence we can choose the worst-case D. First we observe, that for this worst case, we still have $D \leq t$, since any $D > t$ yields the same cost as $D = t$. In the latter case we have cost $D - 1 + b$.

  For this case we compare the online cost $t-1+b$ to the offline cost $min\{b, D\}$. First observe that $t + b = \min\{b,t\} + \max\{b,t\}$. Therefore we have

$$\frac{t-1+b}{\min\{b,D\}} = \frac{t-1+b}{\min\{b,t\}} = \frac{\min\{b,t\}+\max\{b,t\}-1}{\min\{b,t\}} = 1 + \frac{\max\{b,t\}}{\min\{b,t\}} - \frac{1}{\min\{b,t\}} \tag{1}$$

Now assume the adversary chooses $D < t$. Then we get a ratio

$$\frac{D}{\min\{b, D\}} \leq \frac{t-1}{\min\{b, t-1\}} \leq \frac{t}{\min\{b, t\}} \tag{2}$$

In any case, we have (1) $\geq$ (2). Hence $D = t$ is the best choice for the adversary.

Now we still have to find the best choice for the online algorithm. To find that, we minimise (1), and get the choice $t := b$. This gives us a ratio of $2 - \frac{1}{b}$. Hence the presented algorithm has competitive ratio $2 - \frac{1}{b}$.

Remark 5.5 For $b \to \infty$, the competitive ratio goes to 2, which is the best possible for any deterministic algorithm. But we can use a randomised algorithm, where we choose a random day T at which we buy the ski. This algorithm has a competitive ratio of $\frac{e}{e-1}$, which is strictly better.
The reason for this effect is, that the adversary does not know T and therefore he cannot construct a worst case input.

Remark 5.6 The ski rental problem corresponds to power management in two-state system, where we have an active state and a sleep state, and the system experiences an idle period of unknown length. Remaining active has a continuous cost. Sleeping has negligible cost, but for waking up, we have a one-time large cost.

5.2 Competitive Analysis

Goal: Evaluate online algorithms, which must generate output without knowledge of the entire input. We already evaluated

- exact algorithms using their worst-case running times,
- polynomial-time approximation algorithms by comparing the quality of the approximate solution to that of the optimal solution.

We evaluate online algorithms using their competitive ratio: ratio between worst-case solution of the online algorithm and optimal solution of an offline algorithm on the same instance.
The computational complexity of the online algorithm is mostly neglected, since it is a secondary issue when dealing with algorithms that operate in a state of uncertainty.

Definition 5.7 (Formal model for online problems) **Input:** Online algorithm A is given a request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$. The requests must be served in the order of occurrence. When serving $\sigma(t)$, A does not know any request $\sigma(t')$ for $t' > t$.
Goal: Minimise "cost" paid for serving the entire request sequence. $\dashv$

Interpretation as request-answer game: Adversary generates requests and online algorithm has to serve them one at a time.

Definition 5.8 (Ratio for Minimisation scenarios) Let $A(\sigma)$ be the cost incurred by online algorithm A and $\mathrm{OPT}(\sigma)$ be the cost incurred by optimal offline algorithm. Then A is called c-competitive if there is a constant d such that $A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + d$ for all σ. $\dashv$

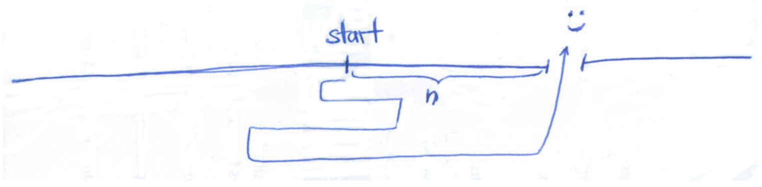
Definition 5.9 (Ratio for Maximisation scenarios) Let $A(\sigma)$ be the cost incurred by online algorithm A and $\mathrm{OPT}(\sigma)$ be the cost incurred by optimal offline algorithm. Then A is called c-competetive if there is a constant d such that $\mathrm{OPT}(\sigma) \leq c \cdot A(\sigma) + d$ for all σ. $\dashv$

Remark 5.10

1. Competitive analysis is a worst case performance measure.
2. No requirements concerning computational efficiency are made in this definition.

5.3 Search

Problem 5.11 (Linear Search/Cow Path Problem) Scenario: Find a hidden point on a discrete line. Assume that the point is n steps away from your starting point on the line.



Then there are several possibilities for the problem.

- Offline: We know the direction. So we just walk into that direction, until we find our point. Note that do not even have to know n.
- Online: We do not know the direction, but we know n. In this case we might need $3n$ steps, if we start in the wrong direction.
- Online: We know neither n nor the direction. Since this is the only non-trivial case, will will further regard this one.

Observation: Any algorithm A should explore left and right in turn. So we can split our search up into rounds, where one round ends, when we come back to our starting point (which will eventually happen, since we can only do single steps). This can be described by a function $f : \mathbb{N} \to \mathbb{N}$ where $f(i)$ is the number of steps away from the origin in the i'th round. For odd i, we move left, for even i we move to the right. For simplicity we assume that we also walk back in that round. This f is our degree of freedom. Now the performance of A entirely depends on f. Note that any reasonable f satisfies $f(i) > f(i-2)$ for all $i \geq 3$. $\dashv$

Theorem 5.12 *For the choice $f(i) = 2^i$ the online-algorithm A is 9-competitive.*

Proof 5.13 Let j be such that $2^j < n \leq 2^{j+1}$, where we assume the hidden point to be n steps away. Algorithm A makes $2f(i) = 2^{i+1}$ steps in the i'th iteration (there and back again). In the worst case we will take the wrong direction in round $j+1$. But in the last round, we will not go the full number of steps, but only until n. Thus, the number of steps of A is at most

$$\sum_{i=1}^{j+1}(2 \cdot 2^i) + n = 2\left(2^{j+2} - 1\right) + n = 2^{j+3} + n - 2 \leq 8n + n - 2 < 9n$$

compared to the offline algorithm which takes n steps. □

Remark 5.14 No deterministic online algorithm for the Cow Path problem can achieve a competetive ratio better than $9 - \Theta((\log n)^i)$ (for any i).

This ratio is negative for large n. Also it is not clear whether $\log^i(n) = \log \ldots \log n$ or $(\log n)^i$ is meant.

5.4 Trading

Financial problems are very attractive for competitive analysis: information about the future is scarce or unreliable.

Problem 5.15 (One-way trading) Scenario: Online player is a trader with initial wealth D_0 in some currency (e.g. Dollars). Every day, there is a new exchange rate for Euro per Dollar. Given the exchange rate, the trader must decide on the fraction of the remaining dollars to be exchanged for Euro ("one way").

Goal: Trade it to some other currency (e.g. Euro), maximise profit
To simplify the problem we assume that the exchange rate is drawn from some interval $[m, M]$ where $0 < m \leq M$. Thus, m is a per default guaranteed exchange rate. Let $\phi := \frac{M}{m}$ be the global fluctuation ratio.
We have two possible scenarios

- Known duration: Player knows number of trading periods.
- Unknown duration: Player does not know but is informed immediately before the last period.

Let n be the number of periods. Assume that m, M are known.
Algorithm: Accept first rate $\geq \sqrt{M \cdot m}$. If no such rate is reached, accept last day's rate. ⊣

Theorem 5.16 *Algorithm A is $\sqrt{\phi}$-competitive, which is optimal for infinite and finite time horizons and when duration is known or unknown.*

Proof 5.17 "Balancing argument": The chosen bound should be equal to the return ratio offline/online. We have to consider two cases.

- the maximal encountered rate is $\geq \sqrt{mM}$: In this case we have

$$\frac{\text{offline}}{\text{online}} \leq \frac{M}{\sqrt{mM}} = \sqrt{\frac{M}{m}} = \sqrt{\phi}$$

- the maximal encountered rate is $< \sqrt{mM}$: Then we have

$$\frac{\text{offline}}{\text{online}} \leq \frac{\sqrt{mM}}{m} = \sqrt{\phi}$$ □

What does the "Balancing Arg" say exactly? It confused us a lot since $\sqrt{\frac{M}{m}} \neq \sqrt{M \cdot m}$.

5.5 Load Balancing/Scheduling

Problem 5.18 Input: a sequence of jobs, each of which has to be assigned to some machine, Each job has a processing cost=:load. In the online algorithm each must be assigned to one machine upon arrival.
Goal: Minimise "makespan" – the completion time for all jobs. ⊣

Remark 5.19 Most load balancing and scheduling problems are NP-hard, hence one can view online algorithms for them as particular class of approximation algorithms ($\rightsquigarrow$ competitive ratio becomes approximation ratio).

Remark 5.20 Load balancing corresponds to covering problems, since all requests must be covered (contrary to packing problems, where not all requests have to be covered).

There are many possible variations in load balancing.

- identical vs nonidentical machines,
- permanent jobs (jobs can be executed at any time) vs limited duration (jobs can be executed in specific time window only),
- job reassignment allowed or not.

Definition 5.21 (Formal model for scheduling) N identical machines, sequence of job requests, job j with load value v_j, permanent jobs. ⊣

Algorithm 5.22 (Natural greedy algorithm A) Assign each incoming job to machine with currently least load. ⊣

Theorem 5.23 *Algorithm A for assigning jobs to N identical machines is 2-competitive (competitive ratio is exactly $2 - \frac{1}{N}$).*

Proof 5.24 For a lower bound choose a sequence of $N(N-1)$ load-1 jobs and a single job of load N. The offline solution has makespan N, whereas A has makespan $2N-1$. This yields our ratio $2 - \frac{1}{N}$. For the upper bound let σ be any job input sequence. W.l.o.g. the first machine shall have maximum load under A. The load situation on machine 1 is $s + w$, where w is the load of last job assigned to it (that need not be the last job of the sequence). Due to the nature of the algorithm, every other machine has load at least s. Hence the total load of all jobs is at least $N \cdot s + w$. Thus

$$\mathrm{OPT}(\sigma) \geq \frac{N \cdot s + w}{N} \qquad\qquad \mathrm{OPT}(\sigma) \geq w$$

Therefore

$$\begin{aligned}\mathrm{Makespan}(A) = w + s \leq w + \mathrm{OPT}(\sigma) - \frac{w}{N} &= \mathrm{OPT}(\sigma) + \left(1 - \frac{1}{N}\right) \cdot w \\ \leq \mathrm{OPT}(\sigma) + \left(1 - \frac{1}{N}\right) \cdot \mathrm{OPT}(\sigma) &= \left(2 - \frac{1}{N}\right) \mathrm{OPT}(\sigma)\end{aligned}$$

□

Remark 5.25 For the assignment of permanent jobs in the restricted machines model

- where every job can be run only on a specific subset of machines
- the machines in that subset are indistinguishable,

greedy algorithm A achieves a competitive ratio of $\lceil \log_2 N \rceil + 1$. The lower bound for any deterministic online algorithm is $\lceil \log_2 (N+1) \rceil$.

Remark 5.26 Now regard the case that each machine has a different speed α_i, so running job j on machine i costs $\frac{v_j}{\alpha_i}$. In this model our greedy algorithm A is $\Theta(\log N)$-competitive.

But there is a better algorithm for this case.

Algorithm 5.27 (Slowfit$_C$) Assume $\mathrm{OPT}(\sigma) \leq C$.

1. Order machines into $M_1, \ldots, M_N$ by increasing speed.
2. Let $l_j(i) :=$ load on machine i after assigning $r_1, \ldots, r_j$ by $\mathrm{Slowfit}_C$.
3. Assign job r_{j+1} to machine $i := \mathrm{argmin}_k \{l_j(k) + r_{j+1}(k) \leq 2C\}$ If no such machine i exists, then return "failure" (because $\mathrm{OPT}(\sigma) \geq C$). $\dashv$

Theorem 5.28 *If $\mathrm{OPT}(\sigma) \leq C$, then Slowfit$_C$ does not fail and* $\mathrm{Slowfit}_C(\sigma) \leq 2C$.

Proof 5.29 Assume $\mathrm{OPT} \leq C$ but slowfit fails on r_n. Let f be the fastest machine with $l_{n-1}(f) \leq \mathrm{OPT}$. It has to exists since otherwise we could not do it in OPT. Let $\mathcal{O} = \{i : i > f\}$, so all machines in $\mathcal{O}$ are overloaded.

maybe define what overloaded means

Let S_i be the set of jobs assigned to i by slowfit and S_i^* be the set assigned by OPT.

$$\begin{aligned}\sum_{\substack{i \in \mathcal{O} \\ j \in S_i}} v_j = \sum \frac{v_j}{r_j(i)} \cdot r_j(i) = \sum_{i \in \mathcal{O}} \left(\alpha_i \cdot \sum_{j \in S_i} r_j(i) \right) = \sum_{i \in \mathcal{O}} \alpha_i \cdot l_{n-1}(i) > \sum_{i \in \mathcal{O}} \alpha \cdot \mathrm{OPT} \\ \geq \sum_{i \in \mathcal{O}} \left(\alpha_i \cdot \sum_{j \in S_i^*} r_j(i) \right) = \sum_{\substack{i \in \mathcal{O} \\ j \in S_i^*}} v_j\end{aligned}$$

Hence $\emptyset \neq \mathcal{O} \neq \{1, \ldots, N\}$. There is some job r_j with $j < n$ assigned to $i \in \mathcal{O}$, but in OPT it is assigned to $i' \notin \mathcal{O}$. Therefore $i' \leq f$, so $r_j(f) \leq r_j(i') \leq \mathrm{OPT} \leq C$. Moreover $l_{j-1}(f) \leq l_{n-1}(f) \leq \mathrm{OPT} \leq C$, so $l_{j-1}(f) + r_j(f) \leq 2C$. But j is assigned to $\mathcal{O}$, although it would have fir into f. This gives our desired contradiction. □

5.6 List Accessing/List Update

Problem 5.30 Scenario: Given is an unsorted linear list of items. Each request (of a sequence of requests) specifies an item in the list. To serve the request, starting at the front, one has to linearly search through the list until the wanted item is found. Additionally, we have the possibility of reorganisation.

1. Immediately after accessed, the item can be moved with no extra cost to any position closer to the front of the list.

2. At any time, two neighboring items can be exchanged at a cost of 1.

Goal: serve the sequence of requests with minimum cost
Additional assumptions: Focus on static list accessing model, that is, no insertions or deletions in the list, which has fixed length l. ⊣

Remark 5.31

- The offline-variant is known to be NP-hard
- clearly the competitive ratio is $\leq l$

We will regard the following algorithms

- Move-To-Front (MTF): Move requested item to front of list.
- Transpose (TRANS): Exchange requested item with item before it.
- Frequency Count (FC): Sort items by number of requests.

Example 5.32 (FC) Initial list: x_1, x_2, x_3.
Request sequence: x_3, x_2, x_3, x_2.
Strategies:

- swap in the beginning $\rightsquigarrow$ 8 steps.
- "move to front" $\rightsquigarrow$ 9 steps.

Remark 5.33 (TRANS is bad) Consider the initial list $x_1, \ldots, x_l$ and the request sequence $\sigma = x_l, x_{l-1}, x_l, x_{l-1}, \ldots$ of length n. Then TRANS pays l for every request, so $\text{TRANS}(\sigma) = nl$. OPT would move both items to the front and then access each with cost 1 or 2. Hence $\text{OPT}(\sigma) = l + l + 1 + 2 + \ldots \leq 2(n + l)$. So for $n \to \infty$ we get that TRANS is $\Theta(l)$-competitive.

Remark 5.34 (FC is bad) Consider the initial list $x_1, \ldots, x_l$ and the request sequence

$$(x_1)^k, (x_2)^{k-1}, \ldots, (x_i)^{k+1-i}, \ldots$$

Then the order of the items will not be changed. Therefore x_i has cost i. So the total cost is

$$\sum_{i=1}^{l} i(k+1-i) = \frac{kl(l+1) + l(1-l^2)}{2}$$

For OPT we move each item to the front on its first request. So the first request has cost i, the following $k - i$ requests only cost 1. Thus the total cost is

$$\sum_{i=1}^{l} i + (k-i) = k \cdot l$$

For $k \to \infty$ the ratio between them becomes $\frac{l+1}{2} \in \Theta(l)$.

So now we come to the final algorithm MTF and we will show that it is significantly better.

Definition 5.35 Remember that a transposition is the switching of two consecutive items. Define for an algorithm A

$$\begin{aligned} A_P(\sigma) &:= \text{number of paid transpositions} \\ A_F(\sigma) &:= \text{number of free transpositions} \\ A_C(\sigma) &:= \text{total cost other than paid transpositions} \end{aligned}$$ ⊣

Theorem 5.36 *For a sequence σ of length n we have*

$$\mathrm{MTF}(\sigma) \leq 2 \cdot \mathrm{OPT}_C(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n$$

Before we prove this theorem, we take a closer look at a powerful tool for analysis.

5.6.1 Potential functions

Remark 5.37 **Motivation:** On the one hand it often is hard to analyse the cost of an online algorithm performance on a long sequence. On the other hand it is also hard to analyse the cost per request, since those may vary between high and low cost and the high cost would be too bad as an estimate. **Idea:** Keep track of "configurations" of OPT and A. If an action of A makes its configuration more similar to OPT, then it is allowed to cost more.

Definition 5.38 An inversion in the list of MTF with respect to the list of OPT is an ordered pair (x, y) such that

- x precedes y in MTF-list.
- y precedes x in OPT-list.

Let t_i be the cost of MTF for request i and Φ_i be the number of inversion after request i. This is our potential function. Now we define the amortized costs as $a_i := t_i + \Phi_i - \Phi_{i-1}$.
Then we have the potential function characteristics

- $\Phi_i \geq 0$ for all i
- $\Phi_0 = 0$ (lists are identical at the start)

So we have a more elegant way to compute the cost for a sequence.

$$A(\sigma) = \sum_{i=1}^{n} t_i = \sum_{i=1}^{n} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i$$ ⊣

MTF is better, but annotated slides are finished

6 Randomised Algorithms

6.1 Randomised Online One-Way Trading

Example 6.1 Recall problem 5.15 of the online one way trading. For $\phi = \frac{M}{m}$ we have constructed a $\sqrt{\phi}$-competitive algorithm.
Now we use a randomised strategy to get a competitive ratio of $\mathcal{O}(\log \phi)$.

Algorithm 6.2 For simplicity assume $\phi = 2^k$. For $0 \leq i \leq k-1$, let A_i be the deterministic online algorithm that accepts the first rate $\geq m \cdot 2^i$.
Algorithm: Randomly choose one of the algorithms A_i for $0 \leq i < k$. ⊣

Theorem 6.3 *Algorithm 6.2 is $(c(\phi) \cdot \log \phi)$-competitive on average with $c(\phi) \to 1$ as $\phi \to \infty$.*

Proof 6.4 Let $r_{\max}$ be the (posteriori) maximum rate encountered and let j be such that

$$m \cdot 2^j \leq r_{\max} < m \cdot 2^{j+1}$$

Clearly, the optimal offline algorithm chooses rate $r_{\max}$. The adversary can choose value of $r_{\max}$ and, thus, the interval $[m \cdot 2^j, m \cdot 2^{j+1})$ it is located in. Increasing $r_{\max}$ in that interval does not change the online result, but improves the offline value. Hence the worst case is $r_{\max} = m \cdot 2^{j+1} - \varepsilon$.

- for $i \leq j$, A_i chooses threshold price $m \cdot 2^i$ (or higher)
- for $i > j$, A_i obtains the final price, so at least m

So on average, the algorithms encounters rate

$$\sum_{i=0}^{j} \frac{1}{k} \cdot m \cdot 2^i + \sum_{i=j+1}^{k-1} \frac{1}{k} \cdot m = \frac{m}{k}\left(2^{j+1} + k - j - 2\right)$$

So we have a ratio

$$\frac{\text{offline}}{\text{online}} = \frac{m \cdot 2^{j+1} - \varepsilon}{\frac{m}{k}\left(2^{j+1} + k - j - 2\right)} < \underbrace{\frac{2^{j+1}}{2^{j+1} + k - j - 2}}_{=:c} \cdot k$$

(Over the real numbers the maximum is achieved for $j^* := k - 2 + \frac{1}{\ln 2}$.) Since $j < k$ we have $c \leq 1$. might be $1 + \varepsilon$
Therefore we have a competitive ratio of $k = \log \phi$. □

6.2 Fundamentals and General Observations

Remark 6.5 (Motivation) Randomisation is a natural resource to be exploited, particularly when dealing with worst-case scenarios or for coping with (malicious) adversaries.
As an essential property, the algorithmic steps depend on outcome of random experiments. Hence the behaviour of the whole process can be random to some degree. This means

- correctness/quality may depend on random processes.
- running time may depend on random processes.

Often we consider the two extreme cases:

- Las Vegas: The output is always correct, but the running time might differ greatly
  Example: Quicksort
- Monte Carlo: We have a guaranteed short running time but the result might be wrong.
  Example: Miller-Rabin primality test

By adding a timeout, each Las-Vegas-algorithm can be turned into a Monte-Carlo-algorithm. It is unknown whether the opposite is true.
For the analysis we have to consider

- Expected values of running time, error probability for output correctness, etc.
- “Guarantees” often only with high probability.

For simplicity we ignore the philosophical question whether randomness even exists, and just assume we have some black box which gives us random values.
Typical advantages of randomised algorithms are

- efficiency
- simplicity
- robustness against an adversary

6.3 Paradigms of Randomised Algorithm Design

We have a look at some concepts and methods of randomised algorithms.

- Foiling an adversary: The idea is a game theoretic view of algorithm designer versus evil opponent.
- Random sampling: We want to use random samples represent a large population. An example is randomised quicksort, where we want to find the median, or at least an element close to it, in order to partition our list.
- Abundance of witnesses: We have a large search spaces which contains many elements with desired property. Example: primality test
- Fingerprinting and hashing: We use mappings to represent large objects by a small fingerprint. With high probability different objects will have different fingerprints. Example: pattern matching, Hashmaps
- Random reordering: We use randomness to avoid pathological worst-case inputs. Example: Quicksort
- Load balancing: avoid heavy loads by random choice of resources. Example: packet routing
- Rapidly mixing Markov chains: use a random process to obtain good samples.
  Example: random walk for 2-SAT: We start with some random assignment β_0. We say that in a fulfilling assignment n variables are “correct”. Now in β_0 we have k correct variables. So $n-k$ is the Hamming distance between β_0 and a fulfilling assignment. Now we choose some unfulfilled clause, choose one variable and flip it. With probability $\geq \frac{1}{2}$ we get closer to the solution, with probability $\leq \frac{1}{2}$ we step away from it (it might be the case that we have to flip both). So this algorithm creates a random walk on $[0, n]$.
- Isolation and symmetry breaking: use randomness to avoid deadlocks in distributed computing. Example: dining philosophers
- Probabilistic methods: prove existence of objects by showing nonzero probabilities for events. Example: Lovász local lemma, Paul Erdös,

6.4 Randomised Approximation: Set Cover

Remark 6.6 We can formulate Minimum Cost Set Cover (problem 4.36) as a 0/1-ILP. We introduce variables x_S for each $S \in \mathcal{S}$ with the interpretation $x_S = 1 \Leftrightarrow S \in \mathcal{C}$.

$$\min \sum_{S \in \mathcal{S}} c(S) \cdot x_S$$
$$\forall e \in U : \sum_{S \in \mathcal{S}, e \in S} x_S \geq 1$$
$$\forall S \in \mathcal{S} : x_S \in \{0, 1\}$$

If we relax the last constraint to $x_S \geq 0$ and $x_S \leq 1$, we get a polynomial-time solvable LP.

Algorithm 6.7 (Deterministic Rounding Algorithm) Let f be the frequency of the most frequent element, that is

$$f = \max_{e \in U} |\{S \in \mathcal{S} : e \in S\}|$$

Then

1. Find an optimal solution to the LP-relaxation.
2. Pick all sets S for which $x_S \geq \frac{1}{f}$ in this solution. $\dashv$

Theorem 6.8 *The above rounding algorithm achieves an approximation factor of f for Minimum Set Cover.*

Proof 6.9 Let X be the solution of our LP and let $\mathcal{C}$ denote the collection of the sets output by the rounding algorithm.

1. $\mathcal{C}$ is a set cover: Let $e \in U$. We know

   $$\sum_{S \in \mathcal{S}, e \in S} x_S \geq 1$$

   But there are at most f such sets, so at least one of them has value $x_S \geq \frac{1}{f}$. Therefore every $e \in U$ is covered by our rounded LP-solution $\mathcal{C}$.

2. Since X is the solution for our relaxed problem, we know $c(X) \leq \text{OPT}$. From X to $\mathcal{C}$ we increase the value of a variable at most by factor f. Hence $c(\mathcal{C}) \leq f \cdot c(X) \leq f \cdot \text{OPT}$. $\square$

lower bound skipped, slide 244a/241

Definition 6.10 We define a probabilistic rounding "function" $r : [0, 1] \to \{0, 1\}$, where $r(x) = 1$ with probability x. $\dashv$

Algorithm 6.11 Take the LP from remark 6.6 and let x^* be its solution.

```
repeat
    $\mathcal{C} \leftarrow r(x^*)$
until $\mathcal{C}$ covers $U$
```
$\dashv$

Theorem 6.12 *Randomised rounding leads within expected polynomial time to a factor-$\mathcal{O}(\log n)$ randomised approximation algorithm for Minimum Cost Set Cover.*

Proof 6.13 (sketch) $n := |U|, \mathcal{C}' := \emptyset$
Algorithm:

```
repeat
    $\mathcal{C}' \leftarrow \mathcal{C}' \cup \mathcal{C}_i$ (the solution delivered by LP and random rounding)
until "$U$ is covered by $\mathcal{C}'$"
```

$\mathcal{O}(\log n)$ repetitions. $\square$

6.5 A Randomised Exact Algorithm for 3-SAT

Problem 6.14 (3-SAT) **Input:** A formula F in 3-conjunctive normal form with n variables and m clauses.
Goal: Is there a satisfying assignment for F? ⊣

Algorithm 6.15 take a random assignment

```
for $i \leftarrow 1$ to $i = m$ do
    Choose an arbitrary unsatisfied clause
    Choose one of its literals uniformly at random and flip its truth value.
end for
if satisfying assignment found then
    return true
else
    return false
end if
```

Remark 6.16 (Analysis of algorithm 6.15) Assume that given formula is satisfiable, let β be a satisfying assignment. Furthermore, let

$$A_i := \text{assignment after round } i$$
$$X_i := \text{number of variables in } A_i \text{ that have the same value as in } \beta$$

Now for $1 \le j \le n-1$ we have

$$\Pr(X_{i+1} = j-1 | x_i = j) \ge \frac{1}{3}$$
$$\Pr(X_{i+1} = j+1 | x_i = j) \le \frac{2}{3}$$

and the special case

$$\Pr(X_{i+1} = 1 | x_i = 0) = 1$$

since we cannot get more wrong. Let h_j be the expected number of steps to reach n starting from j. Then:

$$h_n = 0$$
$$h_j = \frac{2}{3}h_{j-1} + \frac{1}{3}h_{j+1} + 1$$
$$h_0 = h_1 + 1$$

for $1 \le j < n$. This has the unique solution $h_j = 2^{n+2} - 2^{j+2} - 3(n-j) \in \Theta(2^n)$. But this is trivial, since there are only 2^n possible assignments to test.
There are two key observations for improvement

1. If initial assignment is chosen uniformly at random, then number of variable values coinciding with β follows a binomial distribution with expectation $\frac{n}{2}$. Hence with some probability, the process starts with significantly more than $\frac{n}{2}$ variable values coinciding with β.

why? check

2. The longer the process runs, the more likely it has moved toward 0. So one is better off with restarting the process from time to time than letting it run for too long.

not true if multiple satisfying assignments?

if above true, then better flip all

Algorithm 6.17 **for** $i \leftarrow 1$ to M **do**
    start with random truth assignment

```
    for $j \leftarrow 1$ to $3n$ do
        Choose arbitrary unsatisfied clause,
        Choose one of its literals uniformly at random and flip its truth value.
    end for
end for
if assignment found then
    return "satisfiable"
else
    return "unsatisfiable"
end if
```

But there still remains the choice of M. It turns out that $M \in \mathcal{O}\left(\left(\frac{4}{3}\right)^n\right)$ on average yields a satisfying assignment if one exists. ⊣

Theorem 6.18 *If a 3-CNF formula F with n variables is satisfiable, then the above algorithm finds it in $\mathcal{O}\left(\left(\frac{4}{3}\right)^n \cdot n^{\frac{3}{2}}\right)$ expected time.*

Proof?

6.6 Colour Coding for Colourful Independent Set

Problem 6.19 (Job Interval Selection) **Input:** A set J of jobs, each jobs $i \in J$ having a set $S_i \subseteq \{[s_i, e_i] : s_i, e_i \in \mathbb{N}\}$ of possible execution intervals, and some number $k \in \mathbb{N}$
Goal: Can we select at least k pairwise non-intersecting intervals such that, for each job, at most one execution interval is selected? ⊣

Example 6.20 (Plane Maintenance) We have some planes that need maintenance and to do that we have different time slots. Of course we need a slot for each of the planes.

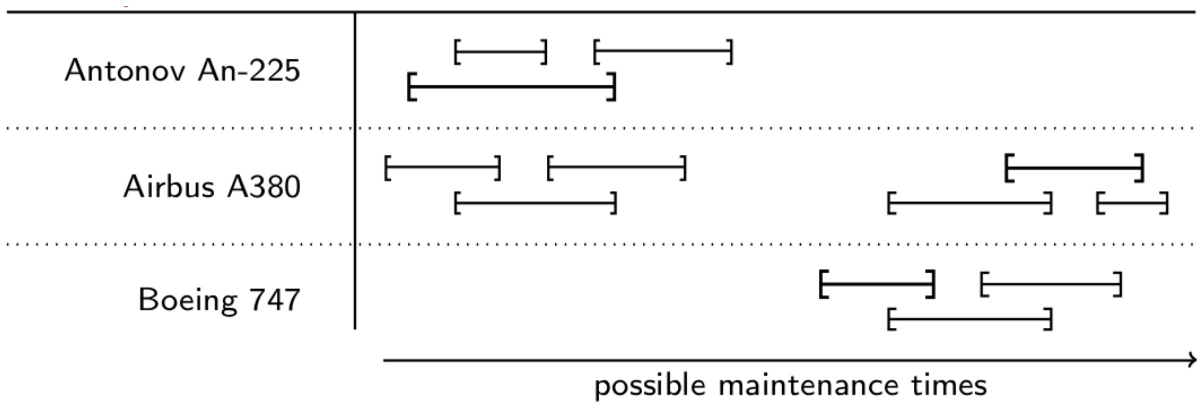


Figure 26: job interval selection for plane maintenance

Problem 6.21 (Colourful Independent Set) An equivalent formulation is:
Input: A set of intervals, each interval having a colour from some, set J and $k \in \mathbb{N}$. **Goal:** Can we select k pairwise nonintersecting intervals that have pairwise different colours? ⊣

Theorem 6.22 *Job Interval Selection is solvable in $\mathcal{O}(2^{|J|} \cdot n)$ where n is the number of input intervals.*

Proof 6.23 Exercise Sheet 1 □

Remark 6.24 Note that all yes-instances have to satisfy $k \leq |J| \rightsquigarrow$ would be nice to have fixed-parameter algorithm for smaller parameter k.

Theorem 6.25 *Colourful Independent Set is solvable with error probability ε in $\mathcal{O}((2e)^k \cdot |\ln \varepsilon| \cdot n)$.*

Proof 6.26 Replace every colour in J uniformly at random by a colour in $\{1, \ldots, k\}$. This instance with k colours is solvable in $\mathcal{O}(2^k \cdot n)$ time.
A solution for the recoloured instance is also a solution for the original instance. If the recoloured intervals have pairwise different colours, also the original intervals have.
A solution S for the original instance is a solution for the recoloured instance if the colours appearing in S are mapped to pairwise different colours, which happens with probability $p := \frac{k!}{k^k} \approx \frac{1}{e^k}$.
Now we have to repeat this $t(\varepsilon)$ times to get the error probability below ε. Using $1 + x \leq e^x$, we have

$$(1-p)^{t(\varepsilon)} \leq e^{-pt(\varepsilon)} \overset{!}{\leq} \varepsilon$$

Solving this for $t(\varepsilon)$ yields

$$t(\varepsilon) \geq \ln \varepsilon \cdot \frac{1}{-p} \approx |\ln \varepsilon| \cdot e^k$$

Therefore our total running time is $\mathcal{O}((2e)^k \cdot |\ln \varepsilon| \cdot n)$. □

7 Computational Geometry

Computational Geometry is the study of geometric problems and efficient algorithms for solving them. It has applications in Graphics, Robotics, Visualization, Geographic Information Systems (GIS), ... We exploit the geometry to arrive at efficient solutions. Examples we have seen before are Convex Hull or Closest-Pair of Points. A useful reference is the Computational Geometry Library (implementation of many CG algorithms): `https://www.cgal.org/`.

7.1 Segment Intersection

Problem 7.1 (Segment Intersection) Input: A set of n (straight) line segments in the plane.
Output: All intersection points of the n segments.
Application: geographic information system (GIS).
Algorithm FindIntersections(S):

```
for each pair of line segments $e_i, e_j \in S$ do
    if $e_i$ and $e_j$ intersect then
        report their intersection point
    end if
end for
```

There might be quadratic many intersections. We want the running time to be output-sensitive: if the output is large, it is fine to spend a lot of time, but if the output is small, we want a fast algorithm.
An idea for improving the algorithm is to not test segments that are "far apart" for intersection. Observation: Two line segments can only intersect if their y-spans have an overlap. Refined observation: Two line segments can only intersect if their y-spans have an overlap, and they are adjacent in the x-order at that y-coordinate (they are horizontal neighbours).
To keep track of segments that are close both along the y-axis and the x-axis we introduce key techniques in computational geometry: the sweep-line algorithm or plane-sweep algorithm. $\dashv$

7.2 Sweep Line

It is a type of algorithm that uses a conceptual sweep line to solve various problems in the Euclidean plane. The idea behind algorithms of this type is to imagine that a line is swept across the plane, stopping at some points, collecting information about objects that either intersect or are in the immediate vicinity of the sweep line whenever it stops. The complete solution is available once the line has passed over all objects.

Definition 7.2 (The Sweep-line Method for segment intersection) For sweep line technique imagine a horizontal line passing over the plane from top to bottom. The sweep line stops and the algorithm computes at certain positions (points), called events. The algorithm stores the relevant situation at the current position of the sweep line, we call that status. The algorithm knows everything it needs to know above the sweep line and found all intersection points there.(fig. 27)
The status of this particular plane sweep algorithm, at the current position of the sweep line, is the set of line segments intersecting the sweep line, ordered from left to right.
The events occur when the status changes and when output is generated. (event $\approx$ interesting y-coordinate)

- upper endpoint of a line segment
- lower endpoint of a line segment
- intersection point of a line segment

At each of this points the status changes; at the third also output is found.
Whenever a new event is encountered:

- Update the status of the sweep line.
- Test (changed) segments for intersection with their adjacent segments on the sweep line.

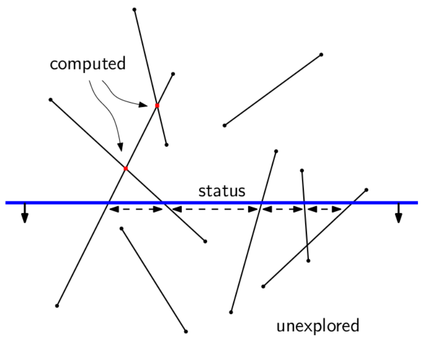


Figure 27: The sweep line algorithm knows all intersections over the sweep line.

We will exclude degenerate cases:

- No two endpoints have the same y-coordinate (in particular, no segment is horizontal).
- No more than two line segments intersect in a point.
- Two segments intersect in at most 1 point (don't overlap). ⊣

Lemma 7.3 Two line segments s_i and s_j can only intersect after (= below) they have become horizontal neighbours.

Proof 7.4 Just imagine that the sweep line is ever so slightly above the intersection point of s_i and s_j, but below any other event. □

In Computational Geometry it is common to ignore degenerate cases in the first phase of designing the algorithm. In the second phase the algorithm is adjusted to deal with them. In many cases we can integrate them into the general cases. We will see how to handle the degenerate cases for Segment Intersection later.

7.3 Algorithm

Algorithm Sketch:

- Event points: Upper endpoints, lower endpoints, intersection points
- Upper endpoint: A new segment appears on the sweep line.
  - Test it for intersection against its two neighbours on the sweep line.
  - Create a new event for each intersection point.
- Intersection point: The two segments that intersect change their order.
  - Test each segment for intersection with its new neighbour.
  - Create a new event point for each intersection point below the sweep line.
- Lower endpoint: A segment disappears and its two old neighbours become adjacent.
  - Test its old neighbours for intersection.
  - Create a new event for each intersection point below the sweep line.

Data structures:
The event list is an abstract data structure that stores all events in the order in which they occur. The status structure is an abstract data structure that maintains the current status. We use a balanced binary search tree with the line segments in the leaves as the status structure. (fig. 28)

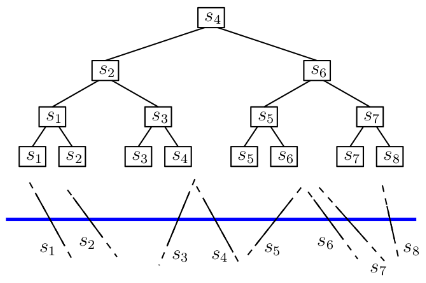


Figure 28: Data structure for the status structure

Sweep line reaches an upper endpoint of a line segment: search and insert. Sweep line reaches a lower endpoint of a line segment: delete from the status structure. Sweep line reaches intersection point: swap two leaves in the status structure (and update information). Note that each insertion, deletion, find left/right neighbour operation takes logarithmic time in the number of nodes in the tree.
The event list must be a balanced binary search tree. We know upper endpoint events and lower endpoint events beforehand; we find intersection point events when the involved line segments become horizontal neighbours.
Algorithm FindIntersections(S):

```
  Initialize an empty event queue $Q$.
  Insert the segment endpoints into $Q$ (when an upper endpoint is inserted, the corresponding segment
  should be stored with it).
  Initialize an empty status structure $T$.
  **while** $Q$ is not empty **do**
      Determine next event point $p$ in $Q$ and delete it.
      HandleEventPoint($p$)
  **end while**
```

7.3.1 Handling Upper Endpoints

If the event is an upper endpoint event and s is the line segment that starts at p:

```
  Search with $p$ in $T$ and insert $s$.
  **if** $s$ intersects its left neighbour in $T$ **then**
      Determine the intersection point and insert it in $Q$.
  **end if**
  **if** $s$ intersects its right neighbour in $T$ **then**
      Determine the intersection point and insert it in $Q$.
  **end if**
```

7.3.2 Handling Lower Endpoints

If the event is a lower endpoint event and s is the line segment that ends at p:

```
  Search with $p$ in $T$ and delete $s$.
  Let $s_l$ and $s_r$ be the left and right neighbours of $s$ in $T$ (before deletion).
  **if** they intersect below the sweep line **then**
      Insert their intersection point as an event in $Q$.
  **end if**
```

7.3.3 Handling Intersection Points

If the event is an intersection point event where s and s' intersect at p:

```
  Exchange $s$ and $s'$ in $T$.
  **if** $s'$ and its new left neighbour in $T$ intersect below the sweep line **then**
```

```
        Insert this intersection point in $Q$.
    end if
    if $s$ and its new right neighbour in $T$ intersect below the sweep line then
        Insert this intersection point in $Q$.
    end if
    Report the intersection point.
```

7.3.4 Dealing With Degenerate Cases

Two Endpoints With The Same y-Coordinate:
For two different events with the same y-coordinate, we treat them from left to right. The "upper" endpoint of a horizontal line segment is its left endpoint.
Multiply-Coinciding Event Points:
Let $U(p)$ and $L(p)$ be the line segments that have p as upper and lower endpoint and $C(p)$ the ones that contain p. When processing event p, we find $L(p) \cup U(p) \cup C(p)$. $U(p)$ is stored with p in Q and $L(p) \cup C(p)$ are adjacent in T. If $L(p) \cup U(p) \cup C(p)$ contains at least 2 segments we report p as the intersection point of all segments in $L(p) \cup U(p) \cup C(p)$. When processing event p we:

- Delete $L(p)$ from T.
- If $U(p) \cup C(p) = \emptyset$ we test the right and left neighbours of p in T for intersection and insert it as an event.
- Else reverse the order of $C(p)$ in T, insert $U(p)$ and test for intersection between the leftmost segment in $U(p) \cup C(p)$ with its left neighbour and the rightmost segment in $U(p) \cup C(p)$ with its right neighbour.

7.3.5 Running Time

Each data structure operation takes $\mathcal{O}(\log n)$ time. Initializing Q and T takes $\mathcal{O}(n \log n)$ time. For an event point p, the number of operations is $\mathcal{O}(|L(p) \cup U(p) \cup C(p)|)$, which is the number of segments containing p. It follows that the running time is: $\mathcal{O}(\log n) \times \mathcal{O}(\sum_p |L(p) \cup U(p) \cup C(p)|)$. We need to upper bound $\sum_p |L(p) \cup U(p) \cup C(p)|$. Each event p is either an endpoint of a segment or an intersection point of two or more segments. The number of endpoints is $\mathcal{O}(n)$. Let l be the number of intersection points. We construct a planar graph G whose vertices are the endpoints and the intersection points. The number of vertices in G is $\mathcal{O}(n + l)$. $|L(p) \cup U(p) \cup C(p)|$ is at most the number of edges incident to vertex p in G. Therefore,$\sum_p |L(p) \cup U(p) \cup C(p)|$ is at most the number of edges in G, which is linear in its number of vertices. Hence, $\sum_p |L(p) \cup U(p) \cup C(p)| = \mathcal{O}(n + l)$. It follows that the running time of the algorithm is $\mathcal{O}((n + l) \log n)$.

Theorem 7.5 Determining all intersections among n segments can be done in time $\mathcal{O}((n + l) \log n)$, where l is the number of intersections.

Note that if l is large (e.g. quadratic in n) then the bruteforce algorithm is more efficient. The above theorem implies that we can detect if n segments intersect in time $\mathcal{O}(n \log n)$.

7.3.6 Summary

We saw an $\mathcal{O}((n + l) \log n)$-time (output-sensitive) algorithm for the Segment Intersection problem using the sweep line technique. This algorithm is due to Bentley & Ottmann [1979] and is similar to the $\mathcal{O}(n \log n)$-time algorithm by Hoey & Shamos [1976] for detecting segment intersection. The above running time was improved to $\mathcal{O}(n \log n + l)$ by Chazelle [1992], which is optimal.

7.4 Spanners

Definition 7.6 (Euclidean graph) Let S be a set of n points in the euclidean plane. The (weighted) Euclidean graph E on S is the complete graph whose point set is S and each edge ab in E has weight equal to the Euclidean distance between its two endpoints fig. 29.

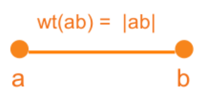


Figure 29: Edge in Euclidean graph

Definition 7.7 (geometric spanners) A spanning subgraph H of E is a geometric spanner of E if the weight of a shortest path between any pair of points a, b in H is at most a fixed constant times $|ab|$. That is, each edge in E is stretched in H by at most a fixed constant. This constant is called the stretch factor of H. ⊣

7.5 Motivation: Communication Networks

Definition 7.8 (communication network) A communication network is commonly modeled as a weighted geometric Euclidean graph in the plane:

- Devices are the points of the graph
- links are the edges of the graph
- The weight of edge ab is the cost to transmit (energy consumption) along ab ⊣

Communication in such a network along low-weight routes is desirable. Many communication protocols start by computing a sparse backbone/topology of the network that can be used for communication (e.g. spanning tree). It is desirable that this backbone approximate the shortest paths in the network, i.e., it should be a spanner with a small stretch factor.

7.6 Additional Requirements on the Spanner

- Plane: Edges do not cross
  - Planarity is useful for guaranteed-delivery routing
  - Can route along the faces of the plane spanner
- Bounded degree: Each point is incident to at most a constant number of edges
  - Bounded degree is useful for power attenuation (e.g., in wireless networks), and for
  - Minimising interference
- Lightweight: The weight of the spanner is a constant times that of the Euclidean MST
- Additional requirements on the algorithm
  - Distributed
  - Localized: The computation at a point depends only on the information in its "vicinity"

7.7 Bounded Degree Plane Spanners

Problem 7.9 Input: A set S of n points in the plane
Output: A bounded-degree plane spanner of the Euclidean graph E with point-set S ⊣

If we drop the planarity requirement, Das et al. [1993] showed how to compute a spanner of degree ≤ 3. The upper bound 3 on the degree is tight: There are Euclidean graphs with no spanner of degree ≤ 2. Most approaches for constructing bounded degree plane spanners of E can be divided into two phases:

- Phase I. Sparsification Phase: Construct a sparse, usually plane, spanner of E.
- Phase II. Bounding the Degree: In the spanner constructed in Phase I keep only a constant number of edges incident to any point (sometimes new edges need to be added).

Phase II should be implemented carefully in order not to blow up the stretch factor.

Theorem 7.10 *There are Euclidean graphs with no spanner of degree ≤ 2.*

Proof 7.11 Consider the $n \times n$-grid in Euclidean space. Then for all pairs of points we have $|ab|_1 \leq 2n$ (we use $|\cdot|_1$ for simplicity). Assume we had a spanner for these graph. A degree-2-spanner is just a Hamiltonian path or circle. Wlog we assume to have a circle, since the additional edge may only shorten distances. Still on a circle any opposite nodes have distance at least $\frac{1}{2}n^2$. Hence our stretch factor is at least $\frac{n}{4}$, which is unbounded. Hence it is not a spanner. $\lightning$ $\square$

7.7.1 Sparsification Phase

Definition 7.12 (Delaunay Triangulation) The Delaunay triangulation of S, denoted Del(S), is a triangulation whose point-set is S and such that, for each triangle in Del(S), the interior of its circumscribed circle is devoid of points of S.
Alternatively:
The Delaunay triangulation of S, denoted Del(S), is a triangulation whose point-set is S and such that, for each edge ab in Del(S), there is a circle passing through a and b whose interior is devoid of points of S.
Delaunay Triangulations are the Dual of Voronoi Diagrams.
1.998 is currently the smallest upper bound on the stretch factor of Del(S). Del(S) has the following properties:

- Connected and spanning
- Plane
- Good spanner
- Can be constructed in time $\mathcal{O}(n \log n)$
- However, it can have unbounded degree!! $\dashv$

7.7.2 Bounding the Degree

To bound the degree of a spanner of E, variants of the Yao graph have been used. The idea is that each point should keep a constant number of edges that "cover" the space around it. This way, no matter in which direction we are trying to reach, we have a "good" starting edge towards that direction. We construct a subgraph H of Del(S) satisfying that for every edge $cb \in$ Del(S): if $cb \notin H$ then $\exists ca \in H$ with $|ca| \leq |cb|$ and $\measuredangle bca$ "small".
We can then apply induction to prove the existence of a "short" path from a to b in H.

Definition 7.13 (Yao Subgraph Y_k) For every point p:

- Partition the plane around p into k disjoint cones each of size $\frac{2\pi}{k}$.
- In every cone: add to the Yao subgraph the shortest edge in Del(S) incident to p and direct it outward of p. (Directions are used only for "convenience".) $\dashv$

Yao subgraph is a good spanner of Del(S) (for large enough k) but still does not have bounded degree. It has out-degree $\leq k$ but it can have unbounded in-degree: point p selects at most k edges but an unbounded number of points may select edges having p as endpoint (head).
To bound the degree, we need to prune the edges incident to each point further so that we keep only a constant number of them. This has to be done carefully so that we don't blow up the stretch factor.
Result from Kanj & Perkovic [2010] is a $\mathcal{O}(n \log n)$ time algorithm that produces a plane graph with degree $\Delta \geq 14$ and a stretch factor $(\frac{1+2\pi}{(\Delta \cos(\pi/\Delta])}) \cdot C_{del}$. For $\Delta = 14$ that gives us a stretch factor of ≈ 3.
This result uses the Canonical Paths.

canonical path?

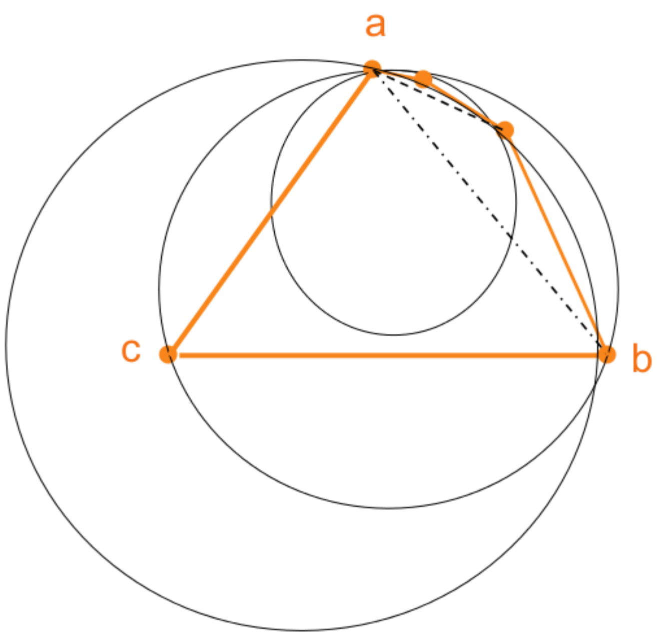


Figure 30: The outward path

Definition 7.14 (The outward path) Suppose $ca, cb \in \mathrm{Del}(S)$ such that $|ca| \leq |cb|$, $\measuredangle bca$ is small and $ab \notin \mathrm{Del}(S)$ and suppose further that Δabc is empty. Then there exists a canonical path between a and b, that we called the outward path, which can be defined as fig. 30.

Definition 7.15 (The inward path) Suppose $ca, cb \in \mathrm{Del}(S)$ such that $|ca| \leq |cb|$, $\measuredangle bca$ is small and $ab \notin \mathrm{Del}(S)$ and suppose further that Δabc is not empty. The canonical path between a and b, called in this case the inward path, is defined as fig. 31. (That uses the outward path.)

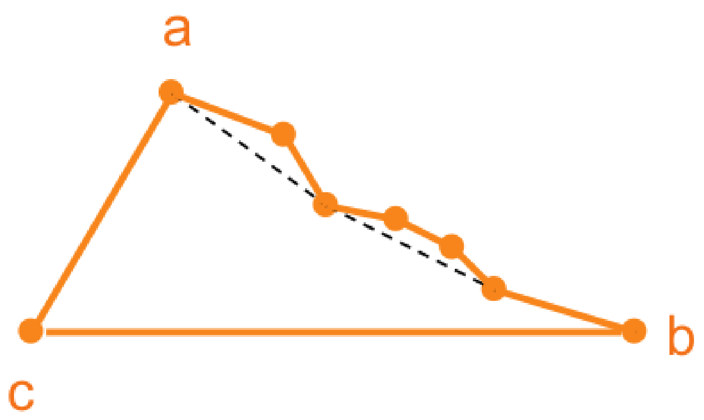


Figure 31: The inward path

The weight of the canonical path is at most $\frac{2\pi}{\Delta \cos(\pi/\Delta)}|cb|$. The interior angles on the path are large. The interior of the region determined by ca, cb and the path may contain only edges connecting c to points of the canonical path.
The idea is to make sure that each point selects its canonical paths edges; those edges make “large angles”. Every point chooses its incident edges that make large angles and performs a standard Yao step on the

remaining edges. An edge is kept iff it is chosen by both endpoints. By a careful analysis (using geometric arguments), we can prove that all canonical edges are chosen by both endpoints when the Yao sector angle is small enough.

Theorem 7.16 For every $\Delta \geq 14$ there is an \log \ time algorithm that constructs a plane spanner of E of degree bounded by Δ and stretch factor $(\frac{1+2\pi}{(\Delta\cos(\pi/\Delta])}) \cdot C_{del}$.

Bonichon et al. [2012] showed how to construct a plane spanner of E with degree 6 and stretch factor 6 In their construction they used Θ-graphs instead of Yao graphs, applied to Delaunay triangulations defined based on equilateral triangles (triangular distance) instead of circles.
Very recently [2014], Bonichon, Kanj, Perkovic', and Xia were able to obtain a plane spanner of E of degree 4 and stretch factor 157 This construction is based on the Delaunay triangulation, but under the L_∞ (or L_1) norm not the L_2 norm.

Theorem 7.17 (degree 4) There is an $\mathcal{O}(n \log n)$ time algorithm that constructs a plane spanner of E of degree at most 4 and stretch factor 157.

Proof 7.18 Start by constructing the Delaunay triangulation of S based on the L_∞ distance, denoted $\mathrm{Del}_\infty(S)$, which is a spanner of E with stretch factor $\sqrt{(}4+2\sqrt{2})$. In $\mathrm{Del}_\infty(S)$ there is an edge between a and b iff there is a rectilinear square with a and b on its boundary whose interior is empty. The advantage of using $\mathrm{Del}_\infty(S)$ over $\mathrm{Del}(S)$ is that it sets up the stage for applying the Yao construction with $k = 4$, where the space around each point is divided into 4 quadrants.
Starting with $\mathrm{Del}_\infty(S)$, each point p selects the shortest edge w.r.t. to the L_∞ distance in each of its 4 quadrants. Still, a point p selects at most 1 edge in each quadrant but it may be selected by many points. What is nice is that edges on the "Canonical Path" of the quadrant are "almost" kept (otherwise, can be handled recursively after few steps). If possible, we try to keep an "anchor" edge in each quadrant that connects p to all points on the canonical path of the quadrant. (fig. 32) The problem happens when an

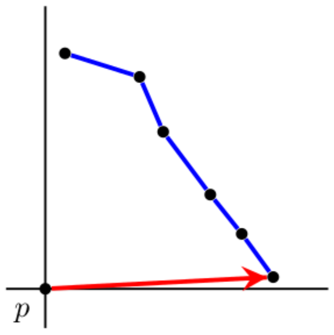


Figure 32: Anchor to canonical path

anchor is not agreed upon by the other endpoint and so on and so forth leading to propagation. We show how to limit this propagation to only few iterations so that the stretch factor does not blow up. □

As a summary of all the recent results on Bounded-Degree Plane Spanners.

Algorithm	Δ	ρ
Bose et al. [2005]	27	9
Kanj & Perkovic [2008]	14	3
Bose et al. [2009]	17	24
Bonichon et al. [2010]	6	6
Bose et al. [2012]	6	82
Bonichon et al. [2014]	4	157

7.8 Bounded Degree Plane Lightweight Spanners

The problem of constructing lightweight plane spanners of E also received considerable attention.

Definition 7.19 (Lightweight spanner) A spanner is a lightweight spanner if the weight of the spanner is a constant times that of the (Euclidean) MST of E. $\dashv$

Four results for this are:

- Levcopoulos and Lingas [1992]: For any $\lambda > 2$:
  - $\mathcal{O}(n \log n)$ time
  - Stretch factor $(\lambda - 1) \cdot C_{del}$
  - Weight at most $(\frac{1+2}{\lambda-2})\cdot$ wt(EMST)
  - Plane
  - Unbounded degree
- Althofer et al. [1993] (more general):
  - Polynomial time greedy algorithm
  - Same stretch factor and weight as Levcopoulos and Lingas
  - Plane
  - Unbounded degree
- Bose et al. [2005]:
  - $\mathcal{O}(n \log n)$ time
  - Stretch factor 9; weight = $\mathcal{O}$(wt(EMST))
  - Plane
  - Degree 27
- Kanj, Perkovic, Xia [2008]:
  - $\mathcal{O}(n \log n)$ time
  - Degree $\Delta \geq 14$
  - Weight $(\frac{1+2}{\lambda-2})\cdot$wt(EMST), where $\lambda > 2$
  - Stretch factor $(\lambda - 1) \cdot (\frac{1+2\pi}{\Delta \cos(\pi/\Delta)}) \cdot C_{del}$
  - Plane

The idea to make a spanner lightweight is to remove cycles containing a "heavy edge".

Theorem 7.20 (Althofer et al. 1993) Let G be a weighted planar graph such that for every cycle C and every edge e on C wt$(C) > \lambda\cdot$wt(e), where $\lambda > 2$, then wt$(G) \leq (\frac{1+2}{\lambda-2})\cdot$wt(MST).

Althofer et al. used the above theorem, combined with a standard greedy algorithm for constructing spanners, to obtain a polynomial-time algorithm for the problem. However, if we want an efficient algorithm, we cannot afford using Althofer et al.'s algorithm. Let G be the plane spanner of E discussed earlier of degree $\Delta \geq 14$ and stretch factor $(\frac{1+2\pi}{\Delta \cos(\pi/\Delta)}) \cdot C_{del}$. We can show that G contains the EMST of E; we will start from G.

Theorem 7.21 (A Refined Structural Theorem) Let T be a EMST of G. We call the edges in T tree edges and those not in T non-tree edges. Every non-tree edge e induces a unique fundamental cycle in $T + e$ which encloses a region in the plane called the fundamental region of e and denoted R_e. (fig. 33) Let G be a weighted plane graph, T an EMST of G and $\lambda > 2$. If for every edge e in $E(G) - T$ we have wt$(F_e) > \lambda\cdot$wt(e) then: wt$(G) \leq (\frac{1+2}{\lambda-2})\cdot$wt$(T)$.

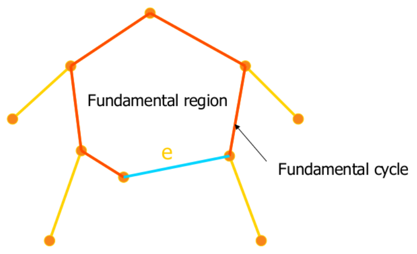


Figure 33: Fundamental Cycles

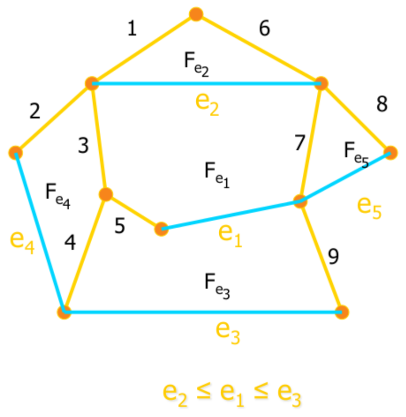


Figure 34: Ordering the Edges

According to the above theorem, we only need to remove/break the fundamental faces that contain a heavy edge. It can be shown that the above ordering is a partial ordering (fig. 34).
Let $L =< e_1, \ldots, e_s >$ be a sequence consiting of the edges in $E(G) - T$ topologically sorted.

Theorem 7.22 L can be computed in $\mathcal{O}(n \log n)$ time.

Proof 7.23 By the following algorithm:

```
Compute a EMST $T$ and $L =< e_1, \ldots, e_s >$
$G' = T$
**for** $i = 1$ to $s$ **do**
    let $F_i$ be the fundamental face of $e_i$ in $G' + e_i$
    if wt$(F_i) > \lambda\cdot$wt$(e_i)$ then $G' = G' + e_i$
**end for**
```

Putting it all together let G be the spanner of E with degree $\Delta \geq 14$ and stretch factor $\rho = (\frac{1+2\pi}{\Delta \cos(\pi/\Delta)}) \cdot C_{del}$, and let G' be the subgraph of G constructed by the previous algorithm when applied to G; let $\lambda > 2$.

Theorem 7.24 The subgraph G'

- has degree at most Δ,
- is plane,
- has weight at most $\frac{1+2}{\lambda-2}\cdot$wt(EMST),
- has stretch factor $(\lambda - 1) \cdot \rho$ and

- can be constructed in $\mathcal{O}(n \log n)$ time.

For a comparison of the results:

Algorithm	L& L	Althofer et al.	Bose et al.	KPX
Stretch factor	ρ^*	ρ^*	9	$a^*\rho^*(5.22)$
Weight factor	c^*	c^*	$\mathcal{O}(1)$	$c^*(5.22)$
Max. degree	∞	∞	27	$\Delta(14)$
Running time	$\mathcal{O}(n \log n)$	polynomial	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Problems that stay open:

- What is the tight bound on the stretch factor of Del(S)?
- Is there are plane spanner of E of degree 3?
- Design an (efficient) algorithm for constructing a plane spanner with better trade-offs between the weight, the degree and the stretch factor

8 Algorithmic Bioinformatics

9 Computational Social Choice

It lies in the intersection of computer science, economics and political science and studies computational aspects of societal decision making.

9.1 Voting

Example 9.1 Choose between the following places:

- skyline(S)
- Wetterleuchten (Main Building) (W)
- Employees Canteen (Math Building) (E)
- Marchstr (M)
- Main Canteen Hardenbergstr (H)

Four students have the following rankings:

$$\begin{gathered} S \succ E \succ W \succ M \succ H \\ H \succ E \succ W \succ M \succ S \\ W \succ E \succ S \succ H \succ M \\ S \succ M \succ H \succ W \succ E \end{gathered}$$

Which canteen should be selected depends on the voting protocol.
Voting protocols based on scoring:
Plurality: The winners are the candidates who are most often ranked first place: Skyline.
k-Approval: The winners are the candidates who are most often ranked in one of the first k positions. For $k = 2$: Employees Canteen.
Borda: The winners are the candidates who have the maximum number of total points, where for m candidates position 1 in the ranking gives $m-1$ points, position 2 gives $m-2$ points and so on. In our example with $m = 5$ Skyline wins with 10 points.
For scoring protocols, winner determination is easy.
But why should we consider so many different voting protocols? What is the best protocol? Can we mathematically prove that? No, we cannot, but we can prove the converse.

Theorem 9.2 (Arrow, J. Polit. Econ. 1950) There is no preference-based voting system that satisfies unrestricted domain, non-dictatorship, Pareto efficiency, and independence of irrelevant alternatives.

Proper subsets can be satisfied and there exist alternative formulations with different minimal sets of properties. So there is no "best voting protocol".
But are all these four properties really desirable?
Unrestricted domain: All preferences are allowed to occur.
Non-dictatorship: There is no single individual that decides on the winner.
Pareto efficiency: If there is any pair of candidates $\{a, b\}$ such that every voter prefers a over b, then the system also prefers a over b.
Independence of irrelevant alternatives: If the system prefers a over b, then, after changing the votes while keeping the relative orderings of a and b, the system still prefers a over b.

Example 9.3 (Sidney Morgenbesser) Morgenbesser, ordering dessert, is told by the waitress that he can choose between apple pie and blueberry pie. He orders the apple pie. Shortly after, the waitress comes back and says that cherry pie is also an option. Morgenbesser says "In that case I'll have the blueberry pie".

Let us concentrate on some nice property.

Definition 9.4 (Condorcet winner) A candidate who wins against all other candidates in pairwise comparisons is called Condorcet winner. ⊣

A Condorcet winner does not always exist, but is unique if it exists.

Definition 9.5 (Condorcet consistency) A voting system that always selects (uniquely) the Condorcet winner as winner (when it exists) is called Condorcet consistent. ⊣

No scoring-based voting system is Condorcet consistent.
Protocols that are Condorcet consistent:
Dodgson: Perform minimum number of swaps between neighbouring candidates to make a candidate Condorcet winner.
Young: Delete a minimum number of votes to make a candidate a Condorcet winner.
Kemeny: Determine consensus ranking that minimises the "total sum of the number of inversions" to the given rankings.

Example 9.6 (On Condorcet Winner Determination)

$$S \succ E \succ W \succ M \succ H$$
$$H \succ E \succ W \succ M \succ S$$
$$W \succ E \succ S \succ H \succ M$$
$$S \succ M \succ H \succ W \succ E$$

Pairs of candidates	# votes: $x \succ y$	# votes: $x \prec y$
$(x, y) = (S, E)$	2	2
$(x, y) = (S, W)$	2	2
$(x, y) = (S, M)$	3	1
$(x, y) = (S, H)$	3	1
$(x, y) = (E, W)$	2	2
$(x, y) = (E, M)$	3	1
$(x, y) = (E, H)$	2	2
$(x, y) = (W, M)$	3	1
$(x, y) = (W, H)$	2	2
$(x, y) = (M, H)$	2	2

⇒ No Condorcet winner.

Example 9.7 (Winner Determination in Kemeny Voting)

$$S \succ E \succ W \succ M \succ H$$
$$H \succ E \succ W \succ M \succ S$$
$$W \succ E \succ S \succ H \succ M$$
$$S \succ M \succ H \succ W \succ E$$

Determine consensus ranking that minimises the total sum of the number of inversions to the given rankings.
⇝ Two (out of 18) optimal consensus ranking with score 16:

$$S \succ E \succ W \succ H \succ M$$
$$E \succ W \succ S \succ M \succ H$$

Definition 9.8 (KT-Distance) The Kendall Tau distance between two votes v and w:

$$\mathrm{dist}(v, w) = \sum_{\{c,d\} \subseteq C} d_{v,w}(c, d)$$

$$\text{where } d_{v,w}(c, d) = \begin{cases} 0 & \text{if } v \text{ and } w \text{ rank } c \text{ and } d \text{ in the same order,} \\ 1 & \text{otherwise.} \end{cases}$$ ⊣

Example 9.9

$$v : a \succ b \succ c$$
$$w : b \succ c \succ a$$

$$\begin{aligned}\operatorname{dist}(v,w) &= d_{v,w}(a,b) + d_{v,w}(a,c) + d_{v,w}(b,c)\\ &= 1+1+0\\ &= 2\end{aligned}$$

Problem 9.10 (Kemeny Score) **Input:** An election and a positive integer k.
Question: Is there a ranking r with Kemeny score at most k, that is, the sum of KT-distances of r to all input rankings is at most k? ⊣

This has applications in ranking of web sites (meta search machine), in sport competitions, databases or bioinformatics.
There are several algorithms:

- factor 8/5-approximation, randomised: factor 11/7
- PTAS
- exact algorithms, heuristics, branch and bound and experiments.

Theorem 9.11 Kemeny score is NP-complete even for four votes.

Proof 9.12 We proof is by reduction from Feedback Arc Set.

Problem 9.13 (Feedback Arc Set) **Input:** An directed graph $G = (U, A)$ ad a positive integer h.
Question: Are there at most h arcs whose deletion makes G acyclic? ⊣

We make use of the observation that acyclic graphs have a topological ordering. Create one candidate for each vertex and each arc.

$$\begin{aligned}
v_1 &: u_1 \succ \overrightarrow{\mathrm{out}(u_1)} \succ u_2 \succ \overrightarrow{\mathrm{out}(u_2)} \succ \cdots \succ u_n \succ \overrightarrow{\mathrm{out}(u_n)}\\
v_2 &: u_n \succ \overleftarrow{\mathrm{out}(u_n)} \succ \cdots \succ u_2 \succ \overleftarrow{\mathrm{out}(u_2)} \succ u_1 \succ \overleftarrow{\mathrm{out}(u_1)}\\
v_3 &: \overrightarrow{\mathrm{in}(u_1)} \succ u_1 \succ \overrightarrow{\mathrm{in}(u_2)} \succ u_2 \succ \cdots \succ \overrightarrow{\mathrm{in}(u_n)} \succ u_n\\
v_4 &: \overleftarrow{\mathrm{in}(u_n)} \succ u_n \succ \cdots \succ \overleftarrow{\mathrm{in}(u_2)} \succ u_2 \succ \overleftarrow{\mathrm{in}(u_1)} \succ u_1
\end{aligned}$$

Where $\overrightarrow{\mathrm{out}\,/\,\mathrm{in}(u_i)}$ denotes some fixed ordering of the candidates representing out/ingoing arc of vertex u_i and $\overleftarrow{\mathrm{out}\,/\,\mathrm{in}(u_i)}$ denotes it reverse ordering.
Claim: There is a Feedback Arc Set of size at most $h \Leftrightarrow$ there is a ranking with Kemeny Score $k' + 2h$ ($k' := 2\binom{n}{2} + 2\binom{m}{2} + 2m \cdot (n-1)$).
A pair of two vertex candidates contributes two $\Rightarrow 2\binom{n}{2}$.
A pair of two arc candidates contributes two $\Rightarrow 2\binom{m}{2}$.
Consider pair (e, w) when $e = (u, v), w \neq u$ it contributes one from $\{v_1, v_2\}$,
when $e = (u, v), w \neq v$ it contributes one from $\{v_3, v_4\}$, $\Rightarrow 2m(n-1)$.
Take a topological ordering from $G - F$ where F denotes a Feedback Arc Set.
Observe that "non-removed" arcs have no additional contribution. "Removed" arcs have extra contribution of two. □

Now we have a look at the parametrised complexity of Kemeny Score.

parameter	complexity	comment
the number of votes n	NP-c	for $n = 4$
number of candidates m	FPT	$\mathcal{O}^*(2^m)$
Kemeny Score k	FPT	$\mathcal{O}^*(2^{\mathcal{O}(\sqrt{k})})$
max. range of cand. pos. r_m	FPT	$\mathcal{O}^*(32^{r_m})$
avg. range of cand. pos. r_a	NP-c	for $r_a \geq 2$
avg. KT-distance d_a	FPT	$\mathcal{O}^*(5.823^{d_a})$, $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{d_a})})$
partial kernel:		$\frac{16}{3} d_a$ candidates
max. KT-distance d_m	FPT	$\mathcal{O}(4.829^{d_m})$, $\mathcal{O}^*(2^{\mathcal{O}(\sqrt{d_m})})$

For the partial kernelisation we view Kemeny Score as a two-dimensional problem with dimensions "number n of voters" and "number m of candidates". The basic idea is to shrink a instance by polynomial-time executable data reduction rules to an equivalent smaller instance such that the size of one dimension only depends on the parameter. We will bound the number of candidates m. Recall at this point that Kemeny Score is NP-hard for $n = 4$ and fixed-parameter tractable with respect to m ($\mathcal{O}^*(2^m)$ dynamic programming algorithm).

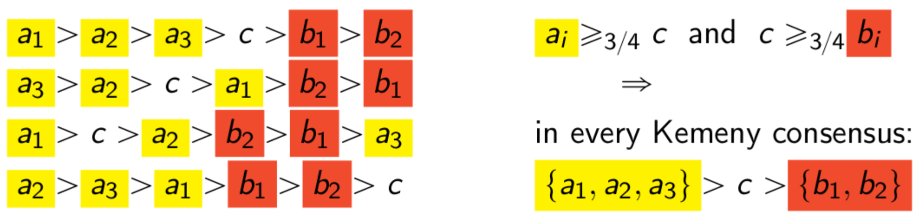
how does the alg work?

The idea is based on majority relations. We find the candidate pairs that are in the same relative order in at least 3/4 of the votes. Their relative order in every Kemeny consensus is then fixed analogously.

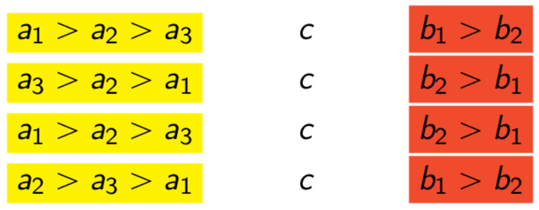
Definition 9.14 (dirty candidates) A candidate c is non-dirty if for every other candidate c' either $c' \geq_{\frac{3}{4}} c$ or $c \geq_{\frac{3}{4}} c'$. Otherwise c is dirty. ⊣

Lemma 9.15 For a non-dirty candidate c and candidate $c' \in C \setminus \{c\}$: If $c \geq_{\frac{3}{4}} c'$, then $c > c'$ in every Kemeny consensus. If $c \geq_{\frac{3}{4}} c'$, then $c' > c$ in every Kemeny consensus.

Data reduction rule: If there is a non-dirty candidate c, then delete c and partition the instance into two subinstances accordingly.



$a_1 > a_2 > a_3 > c > b_1 > b_2$
$a_3 > a_2 > c > a_1 > b_2 > b_1$
$a_1 > c > a_2 > b_2 > b_1 > a_3$
$a_2 > a_3 > a_1 > b_1 > b_2 > c$
$a_i \geqslant_{3/4} c$ and $c \geqslant_{3/4} b_i$
$\Rightarrow$
in every Kemeny consensus:
$\{a_1, a_2, a_3\} > c > \{b_1, b_2\}$



$a_1 > a_2 > a_3$ c $b_1 > b_2$
$a_3 > a_2 > a_1$ c $b_2 > b_1$
$a_1 > a_2 > a_3$ c $b_2 > b_1$
$a_2 > a_3 > a_1$ c $b_1 > b_2$

We can also use the average KT-distance between the input votes as parameter for Kemeny Score.

$$d_a := \frac{2}{n(n-1)} \cdot \sum_{\{u,v\} \subseteq V} \operatorname{dist}(u, v).$$

Theorem 9.16 A Kemeny Score instance with average KT-distance d_a can be reduced in polynomial time to an equivalent instance with less than $16/3 \cdot d_a$ candidates.
In parametrised terms: Kemeny Score yields a "partial problem kernel" with $16/3 \cdot d_a$ candidates.

We have a look at experimental results on meta search engines. We considered four votes: Google, Lycos, MSN Live Search and Yahoo. We always took the top 1000 hits each of candidates that appear in all four rankings.

search term	cand.	sec.	red. inst. solved/unsol.		
aff. action	127	0.41	[27]	> **41** >	[59]
alcoholism	115	0.21	[115]		
architecture	122	0.47	[36]	> **12** > [30] > **17** >	[27]
blues	112	0.16	[74]	> **9** >	[29]
cheese	142	0.39	[94]	> **6** >	[42]
class. guitar	115	1.12	[6]	> **7** > [50] > **35** >	[17]
Death Valley	110	0.25	[15]	> **7** > [30] > **8** >	[50]
field hockey	102	0.21	[37]	> **26** > [20] > **4** >	[15]
gardening	106	0.19	[54]	> **20** > [2] > **9** > [8] > **4** >	[9]
HIV	115	0.26	[62]	> **5** > [7] > **20** >	[21]
lyme disease	153	2.61	[25]	> **97** >	[31]
mutual funds	128	3.33	[9]	> **45** > [9] > **5** > [1] > **49** >	[10]
rock climbing	102	0.12	[102]		
Shakespeare	163	0.68	[100]	> **10** > [25] > **6** >	[22]
telecomm.	131	2.28	[9]	> **109** >	[13]

But winner determination is not the only problem we face. There are voting problems beyond that.
For example we could face problems caused by uncertainties in the voters' preferences (the votes only provide partial instead of linear orders):
Possible Winner: Can some distinguished candidate win if one extends the partial orders to total linear orders?
Necessary Winner: Does some distinguished candidate win for every possibility of extending the partial orders?
Or there could be problems caused through attacks on the outcome of the voting system:
Manipulation: Can voters get better off by voting insincere?
Bribery: "Pay" voters to change their votes in order to achieve a desired outcome.
Control: Make a distinguished candidate win by changing the structure of the election (e.g. by deleting some candidates or votes.)

Definition 9.17 A voting system is called immune to some attack if the attack can not change the outcome. ⊣

Voting systems should be immune to attacks like manipulation, but that turns out to be much more difficult then it sounds.

Theorem 9.18 (Gibbard-Satterthwaite) One of the following must hold for every voting system:

- The voting system is dictatorial or
- there is some candidate who can never win in the system or
- the voting system is not immune to manipulation.

So what to do? The way out of this is designing voting protocols for which it is difficult to find a successful manipulation.

Definition 9.19 (resistant) A voting system is resistant to some attack if it is computationally hard (NP-hard) to recognize opportunities for a successful attack; otherwise it is vulnerable. ⊣

Let us look at an example. Recall the Borda voting protocol we introduced earlier.
For each vote the Borda rule assigns

0	points to lowest-ranking candidate,
...	...
$m-1$	points to highest-ranking candidate.

In the end the winner(s) is/are the candidate(s) with highest total score.
This system was independently discovered by Ramon Llull (1232–1315) and Nicholas of Cusa (1401–1464).

It is intuitive and yields a good representation of the voters preferences. But the winner is not always a Condorcet winner and it is "easy" to manipulate in practice.
Still it is used in political elections, e.g. in Slovenia, Micronesia, in the peace process in Northern Ireland or educational institutions or the Eurovision Song Contest (in a modified version).
A possible manipulation is to add a few votes to make a distinguished candidate win.

Example 9.20

$$\begin{aligned}
&5 \text{ votes with } A \succ B \succ C \succ D\\
&5 \text{ votes with } C \succ D \succ A \succ B\\
&3 \text{ votes with } B \succ D \succ C \succ A\\
&3 \text{ votes with } D \succ B \succ C \succ A
\end{aligned}$$

Can we make D win by adding one vote?
YES: Add the vote $D \succ A \succ B \succ C$.

A recent result is that the minimisation problem "Borda Manipulation" is NP-hard.

Problem 9.21 (Borda Manipulation) Input: An election (C, V) and a distinguished candidate $c^* \in C$.
Task: Find a minimum-size "coalition" W of votes over C such that c^* wins the election $(C, V \cup W)$ according to the Borda rule. ⊣

Tractability results for Borda Manipulation:

- Polynomial-time solvable for one manipulated vote.
- Polynomial-time greedy algorithm to find $x + 1$ manipulated votes for an instance where x manipulated votes suffice: "Additive +1 approximation".
- Typically efficiently solvable on average.
- Fixed-parameter tractable wrt. the parameter "# candidates".

Hardness results for Borda Manipulation:

- NP-hard for weighted votes.
- NP-hard for three input votes and two manipulated votes. Plus a few specific (fixed-parameter) tractability results.
- NP-hard for two manipulated votes and many input votes.(simpler proof and new approximation methods)

A common feature of the NP-hardness proof is the reduction from the NP-hard scheduling related problem 2-Numerical Matching with Target Sums (2NMTS).
The discussion of course is whether this can be seen as protection against manipulation in practice.
We want to concentrate on a different aspect of voting systems. We recall the following theorem.

Theorem 9.22 (Arrow's Impossibility Theorem) There is no preference-based voting system that satisfies unrestricted domain, non-dictatorship, Pareto efficiency, and independence of irrelevant alternatives.

Unrestricted domain means that all preferences may occur at the same time. But for some elections this might be simply unnecessary.

Example 9.23 (Elections with restricted domain)

- Voting on the seminar room temperature.
- Voting on the tuition fee.

Example 9.24 (Restricted domains) "Single-peaked" and "single-crossing" preferences.

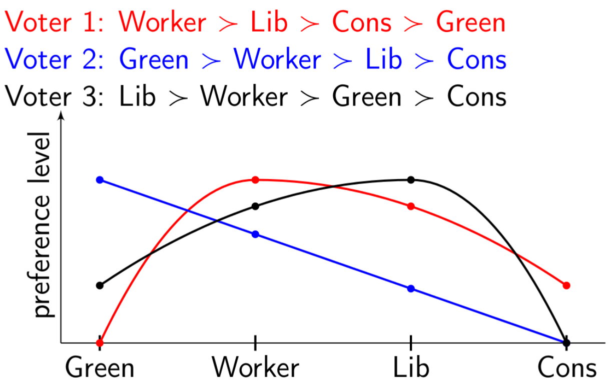


Figure 35: Election for political parties.

9.2 Single-peaked Profiles

Duncan Black discovered and analysed these types of preferences in political elections in 1948. It received a lot of attention from political science and economics.

Definition 9.25 (Single-peaked profiles) There is an arrangement of the candidates on an axis such that each voter i has an ideal axis point p_i and for every two candidates that are both to the right or both to the left of p_i, voter i prefers the one which is closer to p_i. ⊣

We can interpret that by looking at the axis as representation of the decision criterion of the voters.

Example 9.26 An illustrating example is to see the candidates as political parties arranged from left to right wing.(fig. 35)

In single-peaked preferences the arrangement of the candidates' axis does not need to be unique, but some bounds on the number of compatible axes are known. Single-peaked preferences with m candidates have at most 2^{m-1} distinct preference lists. The axis is computable in linear time.
The consequences in algorithmic issues are that several NP-hard voting problems become polynomial-time solvable in case of single-peaked preferences. Some remain NP-hard.
Already 1929, Harold Hotelling notes in passing that political candidates' platforms seem to converge during majoritarian elections. Duncan Black provided a formal analysis of majority voting and proved Hotellings assumptions.

Theorem 9.27 (Median Voter Theorem) If the voters' preferences are single-peaked, then the candidate which is most preferred by the "median voter" is preferred to each other candidate by at least half of the voters.

Corollary 9.28 Restricted to single-peaked preferences, all weak Condorcet-consistent voting systems coincide and are polynomial-time solvable. ⊣

Definition 9.29 (weak Condorcet winner) A weak Condorcet winner is preferred to each other candidate by at least half of the voters and is not necessarily unique. For an odd number of voters every weak Condorcet winner is a Condorcet winner. ⊣

Now we want to characterise single-peakedness by some forbidden substructures.

Definition 9.30 (Worst-diverse configuration) A set of three candidates a, b, c and three votes v_1, v_2, v_3 with

$$\begin{aligned}\{b,c\} &\succ_{v_1} \{a\},\\ \{a,c\} &\succ_{v_2} \{b\} \text{ and}\\ \{a,b\} &\succ_{v_3} \{c\}\end{aligned}$$

⊣

Example 9.31 Consider the following preference lists:

$$a > b > c$$
$$b > c > a$$
$$a > c > b$$

Which candidate to put in the middle?

Definition 9.32 (α-configuration) A set of four candidates a, b, c, d and two votes v_1, v_2 with

$$a \succ_{v_1} b \succ_{v_1} c \text{ and } d \succ_{v_1} b \text{ and}$$
$$c \succ_{v_2} b \succ_{v_2} a \text{ and } d \succ_{v_2} b. \qquad \dashv$$

Example 9.33 Consider the following preference lists:

$$a > b > d > c$$
$$c > b > d > a$$

Theorem 9.34 An election is single-peaked if and only if it contains no worst-diverse configuration and no α-configuration.

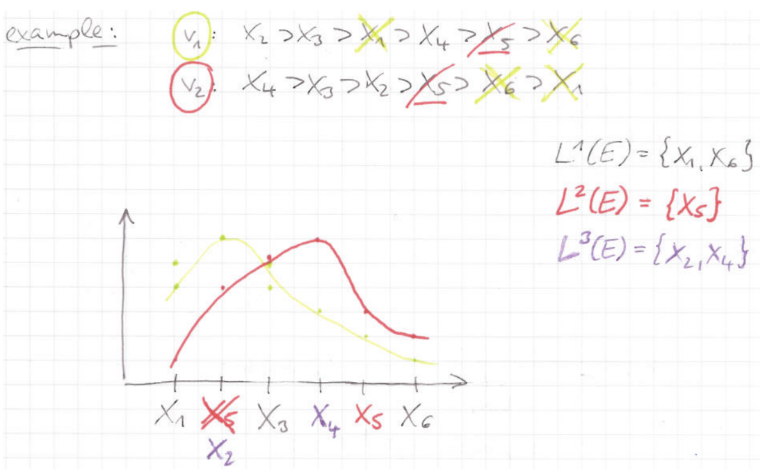
So now how do we detect whether a profile is single-peaked or not?

Theorem 9.35 There is a linear-time algorithm that detects whether a given profile is single-peaked and, if so, outputs a compatible axis.

Proof 9.36 With theorem 9.34 an algorithm of $\mathcal{O}(n^4)$ is trivial.
An better algorithm is to iteratively build up the axis from the borders by considering the candidates being "worst positioned" by some voter.
One worst-positioned candidate can be placed immediately right to the leftmost candidate on the axis or immediately left to the rightmost candidate on the axis; or one encounters a contradiction.
Two worst-positioned candidates can be placed such that one is immediately right to the leftmost candidate on the axis and the other immediately left to the rightmost candidate on the axis; or one encounters a contradiction.
If there are more than two worst-positioned candidates we have a worst-diverse configuration.
Correctness: It is easy to see that a found axis is single-peaked. It is a bit more technical to show, that if there is an axis that it is fond this way.
For illustration see example 9.37.
Single-peaked Detection Algorithm:
Input: An election $E = (C, V)$
Output: A compatible axis or "no".
$L(E)$ returns the set of candidates being in the last position in some vote.

```
repeat
    if $L(E) = \{x\}$ then
        if all voters are compatible then
            place $x$ left
        else if all voters are compatible then
            place $x$ right
        else
            return "no"
        end if
        remove $L(E)$ from the election $E$
    else if $L(E) = \{x, y\}$ then
        if all voters are compatible then
            place $x$ left and $y$ right
        else if all voters are compatible then
            place $x$ right and $y$ left
        else
            return "no"
        end if
        remove $L(E)$ from the election $E$
    else($|L(E)| \geq 3$)
        return "no"
    end if
until all candidates are placed on the axis
return the axis
```
□

Example 9.37



example:
v_1: $x_2 > x_3 > \cancel{x_1} > x_4 > \cancel{x_5} > \cancel{x_6}$
v_2: $x_4 > x_3 > x_2 > \cancel{x_5} > \cancel{x_6} > \cancel{x_1}$
$L^1(E) = \{x_1, x_6\}$
$L^2(E) = \{x_5\}$
$L^3(E) = \{x_2, x_4\}$
x_1 $\cancel{x_6}$ x_3 x_4 x_5 x_6
x_2

9.3 Single-Crossingness

Kevin Roberts discovered and analysed these types of preferences appearing in voting over income tax schedules in 1977. It received a lot of attention from political science and economics.

Definition 9.38 (Single-cossing ordering) Ordering $\langle v_1, \ldots, v_n \rangle$ is single-crossing if for each pair $\{a, b\}$ of candidates there is a voter v_i such that

v_1	$\cdots$	v_{i-1}	v_i	v_{i+1}	$\cdots$	v_n
a	$\cdots$	a	b	b	$\cdots$	b
b	$\cdots$	b	a	a	$\cdots$	a

Attention: Columns represent voters and preferences are top down! ⊣

Example 9.39 Three voters and three proposed tax rates

Student	Banker	Professor
12	5	5
10	10	12
5	12	10

For the order $\langle \text{Banker}, \text{Professor}, \text{Student} \rangle$ it is single-crossing.

Again we want to characterise Single-crossingness by forbidden substructures.

Definition 9.40 (γ-configuration) Three voters v_1, v_2, v_3 and three pairs $\{a, b\}, \{c, d\}, \{e, f\}$ of candidates with:

v_1	v_2	v_3
a	a	b
b	b	a
c	d	c
d	c	d
e	f	f
f	c	e

$\dashv$

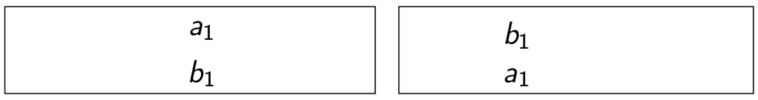
Definition 9.41 (δ-configuration) Four voters v_1, v_2, v_3, v_4 and two pairs $\{a, b\}, \{c, d\}$ of candidates with:

v_1	v_2	v_3	v_4
a	a	b	b
b	b	a	a
c	d	c	d
d	c	d	c

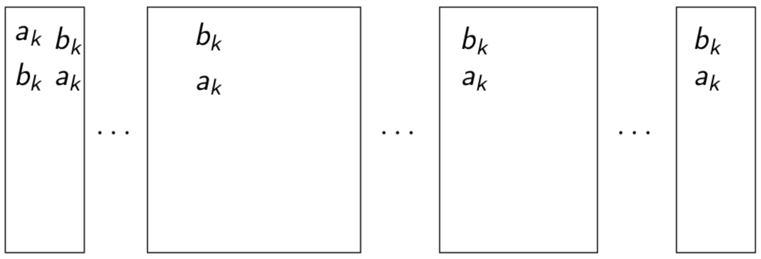
$\dashv$

Theorem 9.42 An election is single-crossing if and only if it contains no γ-configuration and no δ-configuration.

Proof 9.43 (Sketch) "$\Rightarrow$" holds since ($\exists\gamma$-configuration$\vee\exists\delta$-configuration) $\Rightarrow \neg$ single-crossing.
"$\Leftarrow$" Use a simple algorithm to prove this: Construct a single-crossing ordering of the voters step by step. If the algorithm gets stuck, then it identifies a γ-configuration or a δ-configuration. Otherwise, it outputs a single-crossing ordering of the voters.
Idea for the construction:
Enumerate all pairs of candidates $\{a_k, b_k\}, 1 \leq k \leq \binom{m}{2}$.
Initial($\{a_1, b_1\}$): partition the voters into two blocks:

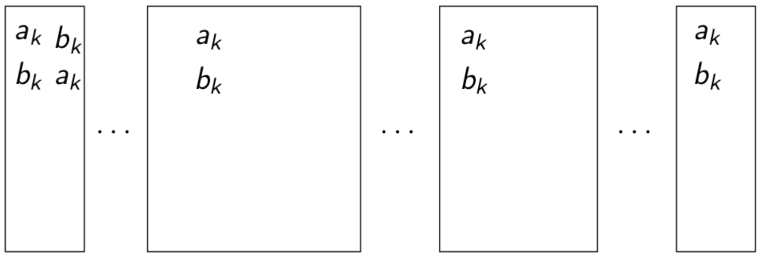


Loop($\{a_k, b_k\}$): Find the first mixed block, reorder and partition:
Case 1.1:



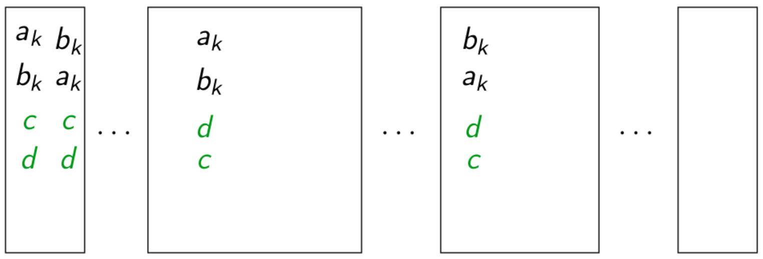
Case 1.1

Case 1.2:



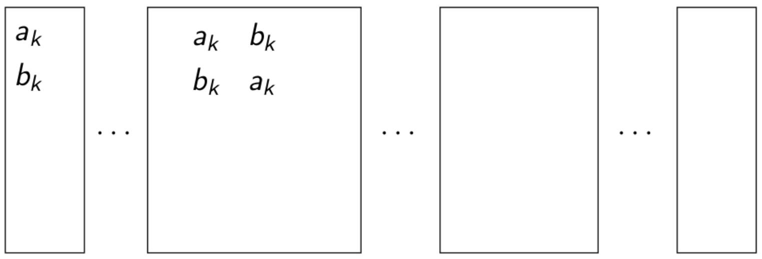
Case 1.2

Case 1.3: $\Rightarrow \delta$-configuration



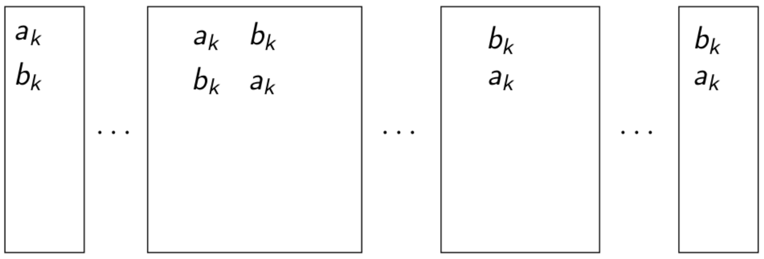
Case 1.3

Case 2:



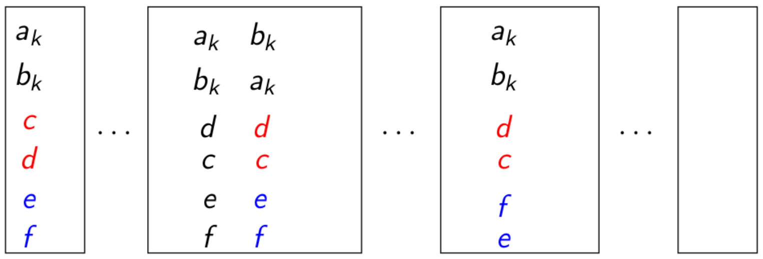
Case 2

Case 2.1:



Case 2.1

Case 2.2: $\Rightarrow$ γ-configuration



Case 2.2

Let $X := \langle X, \dots, X_p \rangle$ be the computed blocks, then

- $\forall v, w \in X_i : v = w$ (and the ordering inside X_i is arbitrary),
- the reverse of X yields also a single-crossing ordering,
- $|X| \leq \frac{1}{2}m(m-1) + 1$ and
- the running time is $\mathcal{O}(m^2 \cdot n)$. □

Example 9.44 Single-crossing election with a max. number of distinct voters with four candidates $1, 2, 3$ and 4.

V_1	V_2	V_3	V_4	V_5	V_6	V_7
1	2	2	2	3	3	4
2	1	3	3	2	4	3
3	3	1	4	4	2	2
4	4	4	1	1	1	1

But of course restricted domains have their drawbacks. Many real-world instances are neither single-peaked nor single-crossing. One single maverick voter which behaves strange can destroy the nice property. One single outlier candidate which cannot be arranged on the axis can destroy the nice property.
So maybe can we relax the properties? Is the election almost single-peaked/single-crossing? How to define "closeness to single-peakedness/single-crossingness"? We can measure distance by the number of maverick voter/outlier candidates.
A hot topic in research is to determine whether the nice properties still hold then.
So how do we detect outliers? Let Π be one of the two properties single-peaked and single crossing.

Problem 9.45 (Π Maverick Detection) Input: A profile with n voters and an integer $k \leq n$.
Question: Can we delete at most k voters such that the resulting profile has the Π-property? $\dashv$

Problem 9.46 (Π Candidate Deletion) Input: A profile with m candidates and an integer $k \leq m$.
Question: Can we delete at most k candidates such that the resulting profile has the Π-property? $\dashv$

Alternatively we can define the distance by #swaps of candidates, #axis, etc.
The idea for Single-crossing Maverick Detection is to identify maverick voters by extending a single-crossing detection algorithm.
We already know that given an ordering $v_1 > v_2 > \cdots > v_n$ of voters, we can build s directed graph with vertices representing the voters and an arc (v, w) if any pair $a \succ_v b$ and $b \succ_w a$ implies $a \succ_{v_1} b$.
This graph is acyclic, that is, it is a DAG.
Basically we are ruling out γ-configurations. If two voters disagree, then the first one must agree with v_1.
We contract equals voters. If our graph has a Hamiltonian path, then this path also is a single.crossing ordering.
For each voter v we choose them as v_1, construct the graph and search for the longest path. This can be done by giving each edge weight -1 and then searching for the shortest path to any other node. To compute this we use Floyd-Warshall. All this can be done in polynomial time.
Then a single-crossing ordering of maximum number of voters must correspond to a DAG with maximum number of vertices.

10 Algorithmic Gametheory

10.1 Motivation

Algorithmic Game Theory (AGT) is a close relative of Computational Social Choice. Both concerned with processes of decision making. AGT possibly closer to economics, where Computational Social Choice is closer to politics.
Tools helping to understand phenomena when agents interact. Mathematical models of conflict and cooperation between intelligent agents. An aspect is multi-agent decision making. It is the interface between theoretical computer science and game theory.
Central topics are:

- Games, strategies and equilibria
- Cooperative vs. non-cooperative games
- Optimality concepts such as Pareto optimality and social optimality
- Mechanism design and auctions
- Price of anarchy, selfish agents
- Fair allocation.

We compare with other lecture topics:

- Lack of unbounded computational resources $\rightsquigarrow$ complexity analysis, approximation algorithms, parametrised algorithmics.
- Lack of (future) information $\rightsquigarrow$ online algorithms.
- Lack of a dictator $\rightsquigarrow$ computational social choice.
- Lack of coordination $\rightsquigarrow$ algorithmic game theory; price of anarchy.

10.2 Games

Definition 10.1 (Game) A formal representation of a situation in which ≥ 2 agents strategically interact. It comprises

- the players (who is involved?);
- the rules (who moves when, what do agents know, what can they do?);
- the outcome for each set of players' actions;
- the payoffs (what are the players' preferences over the possible outcomes?). $\dashv$

Example 10.2 (Preparation) A student has an exam and a presentation(together with a partner) next day, but time only suffices to prepare exactly one of them.
Assumption: The expected outcome is easy to predict once it is decided what to prepare:

- Exam:
  - Without preparation: 80 points.
  - With preparation: 92 points.
- Presentation: To be done together with the partner who is in the same situation.
  - Both prepare: 100 points.
  - Only one prepares: 92 points.

– Neither prepares: 84 points.

Problem: The students cannot contact each other so they have to decide independently. Naturally the goal for each student is to maximise his points.
Possible Cases:

- Both prepare for the presentation: 100 points for presentation, 80 points for exam, $\rightsquigarrow$ 90 points on average for each student.
- Both prepare for the exam: 84 points for presentation, 92 points for exam, $\rightsquigarrow$ 88 points on average for each student.
- Exactly one prepares for the exam and one for the presentation:
  92 and 92 $\rightsquigarrow$ average 92 points for Student 1.
  92 and 80 $\rightsquigarrow$ average 86 points for Student 2.

Payoff matrix:

Stud.1 ↓ 2 →	Presentation	Exam
Presentation	$90, 90$	$86, 92$
Exam	$92, 86$	$88, 88$

Definition 10.3 (Dominant Strategie) A (strictly) dominant strategy is a strategy that is (strictly) better than all other options, regardless of what other players do. $\dashv$

In the previous example, "studying for the exam" is a strictly dominant strategy for both players.

Example 10.4 (Prisoner's Dilemma) Two suspects interrogated by police in separate rooms, accused for robbery, done jointly. Options offered by police:

1. If you confess and your partner does not, then you are released and the other goes to jail for 10 years.
2. If you both confess, then both of you go to jail for four years.
3. If you both don't confess, then both go to jail for one year (for resisting arrest).

Payoff matrix:

	not confess	confess
not confess	$-1, -1$	$-10, 0$
confess	$0, -10$	$-4, -4$

Confessing is strictly dominant strategy. $\rightsquigarrow$ Expected payoff -4 under rational play.

Example 10.5 (Product Marketing) There are two types of consumers in the market: Those who are interested in low-price product and those who prefer the upscale version of a product. That leads to two possible production strategies.

Payoff matrix:

Firm 1 ↓ 2 →	Low-Priced	Upscale
Low-Priced	$0.48, 0.12$	$0.60, 0.40$
Upscale	$0.40, 0.60$	$0.32, 0.08$

60% of the people prefer the low-priced version. Firm 1 is more popular than Firm 2, that is, Firm 1 gets 80% of the sales and Firm 2 gets 20% of the sales of one product version.
A strictly dominant strategy for Firm 1 is to go Low-Priced. But Firm 2 has no dominant strategy: "Low-Priced" would be the best response if Firm 1 plays "Upscale" and "Upscale" would be the best response if Firm 1 plays "Low-Prized". However, if Firm 2 can predict that Firm 1 will play "Low-Prized", then it plays "Upscale".

Example 10.6 (Product Marketing (again and different)) Now, two firms are hoping to do business with one of three large clients A, B and C.

Payoff matrix:

Firm 1 ↓ 2 →	A	B	C
A	$4, 4$	$0, 2$	$0, 2$
B	$0, 0$	$1, 1$	$0, 2$
C	$0, 0$	$0, 2$	$1, 1$

Neither firm has a dominant strategy. So how to reason about the outcome of this play?

10.3 Equilibria

Due to John Nash [1950], 1994 Nobel Prize in Economics:

Definition 10.7 (Nash Equilibrium) Assume that Player 1 chooses strategy S and Player 2 chooses strategy T. Then (S,T) is a Nash Equilibrium if S is a best response to T and T is a best response to S.⊣

In the above example, (A,A) is the only Nash equilibrium.
Possibilities to compute Nash equilibria:

- Check for all pairs of strategies whether the strategies are best responses to each other.
- Compute each player's best response(s) to each strategy of the other player and then find strategies that are mutually best responses.

Example 10.8 (Coordination Game) Talk to be prepared by two students, one cannot reach the other, but needs to start. Make the slides in LaTeX or PowerPoint? $\rightsquigarrow$ Coordination game since two players share the goal of coordinating on the same strategy.

Payoff matrix:

Student 1 ↓ 2 →	LaTeX	PowerPoint
LaTeX	$1,1$	$0,0$
PowerPoint	$0,0$	$1,1$

Two Nash equilibria: (LaTeX, LaTeX), (PowerPoint, PowerPoint).

Example 10.9 (Multiple Nash Equilibria) There are different variants:

- Two Nash Equilibria: $\begin{array}{c|c} 2,2 & 0,0 \\ 0,0 & 1,1 \end{array}$
- Two Nash Equilibria "Battle of Sexes": $\begin{array}{c|c} 1,2 & 0,0 \\ 0,0 & 2,1 \end{array}$
- Two Nash Equilibria "Stag Hunting":

Hunter 1 ↓ 2 →	Stag	Hare
Hunt Stag	$4,4$	$0,3$
Hunt Hare	$3,0$	$3,3$

  If the hunters miscoordinate, then the one trying for the stag hunting gets penalised more than the other.

There are also games without a Nash equilibrium.

Example 10.10 (Matching Pennies) Payoff matrix:

Player 1 ↓ 2 →	Heads	Tails
Heads	$-1,+1$	$+1,-1$
Tails	$+1,-1$	$-1,+1$

This kind of game is called Zero-sum game since the payoffs of the two players sum to zero in every outcome.

We observe that in the above example there is no pair of strategies that are best responses to each other, So there is no Nash equilibrium if the only two strategies are simply Heads or Tails.
With randomisation we can define socalled Mixed strategies. Each player chooses a probability p with which to play Heads. This leads to expected payoff values.
Pure strategies are a special case where either $p = 1$ or $p = 0$.
Observe that in the example above in any Nash equilibrium both players may not use a pure strategy.
We recall the payoff matrix from our Exam-or-Presentation example game before.

Stud.1 ↓ 2 →	Presentation	Exam
Presentation	$90,90$	$86,92$
Exam	$92,86$	$88,88$

We now want to reason whether an outcome is "good for society" instead of "individually".

Definition 10.11 (Pareto optimality) A choice of strategies (one by each player) is Pareto optimal if there is no other choice of strategies in which all players receive payoffs at least as high and at least one player receives a higher payoff. ⊣

In above example (Exam, Exam) is not Pareto optimal, all others are. Here, the only outcome not Pareto optimal is the unique Nash equilibrium.

Definition 10.12 (Social optimality) A choice of strategies (one by each player) is a social welfare optimiser or socially optimal if it maximises the sum of the players' payoffs. ⊣

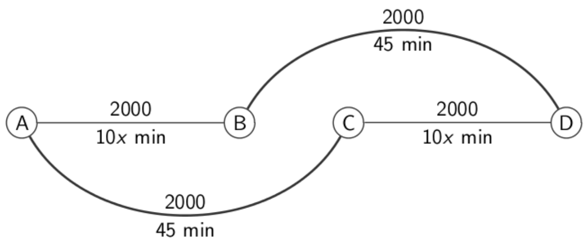
In above example, assuming that we simply sum up the individual payoffs, (Pres, Pres) is the socially optimal outcome with payoff $90 + 90 = 180$.
It holds that socially optimal outcomes are pareto optimal.

Theorem 10.13 Under "suitable complexity assumptions" (somewhat weaker than $P \neq NP$): There is no polynomial-time algorithm for computing a Nash equilibrium in general (non-zero) games.

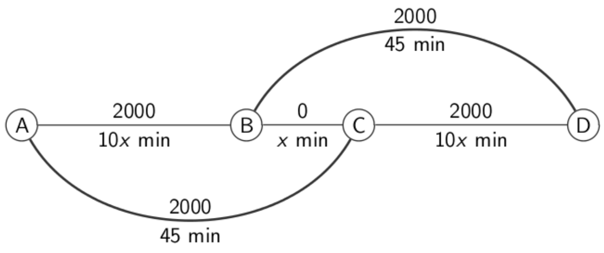
10.4 Price of Anarchy

Some games occur in everyday life, like on road networks or on the internet. A central question is, when selfish behaviour is harmless and when it leads to worse outcome for everyone.

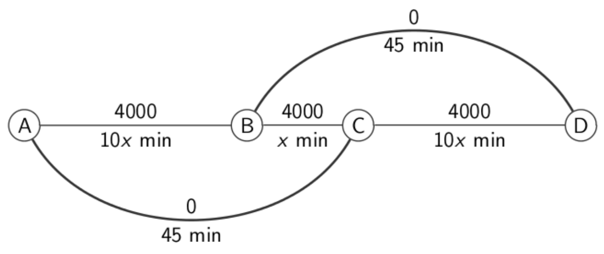
Example 10.14 (Braess' paradox) We have a road network with 400 drivers who want to travel from A to D. Travel time depends on the number x of drivers in thousands.



The travel time is 65 minutes for each driver and nobody can improve $\rightsquigarrow$ Nash equilibrium. Now a bridge from B to C is built with travel time x minutes.



Now the fastest connection from A to C is via B and nobody uses the direct connection from A to C, analogously for B to D.



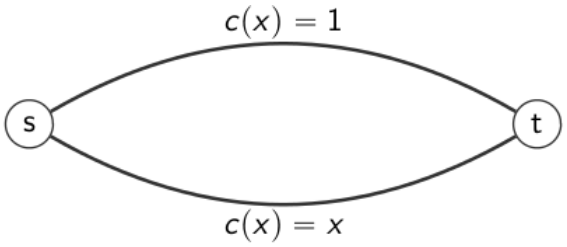
Each driver now needs 84 instead of 65 minutes, that means, introducing the additional connection B–C slowed everyone down.

Braess' paradox shows that selfish routing does not minimise the commute time of the drivers.
An altruistic dictator however could dictate routes and thus improve the travel time for each driver.

Definition 10.15 (Price of Anarchy(POA)) The price of anarchy (POA) is the ratio between the system performance with strategic (selfish) players and the best possible system performance. ⊣

In our example (Braess) the price of anarchy is at least $\frac{84}{65}$.
For designing systems and mechanisms a POA close to 1 is desirable.

Example 10.16 (Pigou's example) Travelling along a link costs $c(x)$, where x is the fraction of traffic using the link.



Every driver's dominant strategy is to use the lower link. Even when congested with all of the traffic it takes travel time 1. Any other solution would be socially better. An altruistic dictator would split traffic equally between both links. ⇒ average travel time = $\frac{3}{4}$ ⇒ POA = $\frac{4}{3}$.

10.5 Mechanism Design and Auctions

Example 10.17 (Women's Badminton, Olympics, London 2012) Failed tournament design, similar to European Championship in soccer: four groups A, B, C and D of four teams each. Two phases: 1. Round-robin phase within each group, each team playing against the three others, 2. Elimination phase: knockout tournament, four quarter finals, two semifinals, final. Quarter finals: Winner-A against Second-C, Winner-C against Second-A and the same for B/D. Problem: Group D was surprisingly won by the Danish team. The Chinese team only made 2nd place, but everyone thought that the overall strongest team would be the Chinese one. So nobody wanted to become winner of group B. Two teams tried to lose against each other.

Mechanism design applications include

- internet search auctions,
- wireless spectrum auctions,
- matching residents to hospitals,
- matching children to schools,
- kidney exchange markets.

Definition 10.18 (Single-Item Auctions) Situation: 1 item to sell, n bidders (e.g. eBay auctions). **Assumptions:** Each bidder i has a valuation v_i for the item. Valuation is private, unknown to seller and other bidders. We have a Quasilinear utility model: If the bidder looses the auction, the utility is 0. If the bidder wins at price p, the utility is $v_i - p$. ⊣

Definition 10.19 (Sealed-Bid Auctions) Sealed-bid auctions consist of three steps:

1. Each bidder i privately communicates a bid b_i.
2. Auctioneer decides who gets the item.
3. Auctioneer decides on selling price.

The most natural solution for 2. is to give the item to the highest bidder. But what is the most natural solution for 3.? There are several possibilities which influence the bidders. ⊣

Definition 10.20 (First-Price Auctions) One possibility for step 3 is to let the winning bidder pay his bid. First-price auctions are hard to reason about. For the participants it is hard to find out how to bis and for the seller it is hard to predict the outcome.
Example experiment: Valuation for each of two bidders: number of birth month + day of birth, that is, possible range is $[2, 43]$.
How to bid to maximise quasilinear utility? Would the answer change if there were three instead of two bidders? $\dashv$

Much easier to reason about are second-price auctions.
Like eBay auctions. EBay uses a proxy bidder, that knows your highest bid and bids on your behalf. It bids until your maximum value is reached or you are the highest bidder, so in the end you pay the bid of the second highest bidder plus some small value.
So if you win the auction, the price equals almost the second highest bid.

Definition 10.21 (Second-Price Auctions) A second-price or Vickrey auction is a sealed auction where the sale price equals the second-highest bid. $\dashv$

Theorem 10.22 In a second-price auction every bidder has a dominant strategy: set the bid b_i to the true valuation v_i.

Proof 10.23 We show that bidder i's utility is maximised. Let $B :=$ max bid of the other bidders.
If $v_i < B$, then maximum possible utility of bidder i is $\max\{0, v_i - B\} = 0$, which is achieved by bidding truthfully ($b_i := v_i$) and loosing.
If $v_i \geq B$, then the maximum possible utility of bidder i is $\max\{0, v_i - B\} = v_i - B$. which is achieved by bidding truthfully and winning. $\square$

Compare this to first-price auctions. There you should never bid your valuation ($\rightsquigarrow$ zero utility) and the ideal amount to underbid depends on the other players.
Second-price auctions are easy to participate in, because you can always bid truthfully.

Theorem 10.24 In a second-price auction, every truth-telling bidder is guaranteed non-negative utility.

Proof 10.25 Losers all get zero utility.
Winners get utility $v_i - p$, where p is the second highest bid. Since i wins and bids its true valuation, we have $p \leq v_i$, thus $v_i - p \geq 0$.
Finally note that second-price auctions can be performed in linear time. $\square$

There are also more complicated auctions:

- more complex allocations (sponsored search, Google auctions)
- maximising "social surplus" $\sum_{i=1}^{n} v_i x_i$, where $x_i = 1$ if i wins and $x_i = 0$ otherwise.

11 Massive Data

We work every day with increasing data sizes and computation speed. The balance between computation time and data access time has changed. The data exchange has itself become either an expensive resource to be optimised (external memory and cache-oblivious models of computation) or is extremely constrained (e.g. streaming model: only single pass over data is allowed and only a tiny fraction of it can be stored). We focus here on a single processor model; things would become even more challenging when dealing with multiple processors on vast amounts of distributed data. Then, communication instead of access would have to be modeled.

11.1 External Memory Model

The most significant difference between the RAM model of computation and real-world computer is the memory structure. The RAM model works with uniform memory access times, while real computer have a complex memory hierarchy. In particular they have slow external memory.
In the External memory model we have:

- fast internal memory with M words and
- unlimited-size external memory, where I/O operations can transport blocks of B subsequent words between internal and external memory.

Typical values would be $M = 2\text{GB}$, $B = 2\text{MB}$ and one I/O operation may take 10ms, that is $2 \cdot 10^7$ clock cycles of a 2GHz processor. With other values for M and B one can model different access times between hardware cache and main memory (the difference here is smaller).

Definition 11.1 (Principles of External Memory Algorithm Design)

- **Internal efficiency:** internal work done should be comparable to the best internal memory algorithms.
- **Spacial locality:** when a block is accessed, it should contain as much useful data as possible.
- **Temporal locality:** Once data is in internal memory, as much useful work as possible should be done before it is written back to external memory.

Which of these criteria is most important depends a lot on the application and the hardware used. $\dashv$

Scanning and sorting are applied in virtually all I/O-efficient algorithms. These two techniques will be studied now. Without loss of generality, we assume that $\frac{N}{B}$ is integer. Where N is the overall data size.

11.2 Scanning

Scanning is the simplest of all paradigms in I/O-efficient algorithms.
Consider N data items: Accessing in sequential order takes $\mathcal{O}(\frac{N}{B})$ I/Os. Accessing in random order costs $\Omega(N)$ I/Os (worst case).

Example 11.2 (Prefix Sums) Input: Array (or stream) A with numbers.
Output: Array (or stream) A' with $A'[i] = \sum_{j=1}^{i} A[j]$.
Internal memory: Trivial in linear time. **External memory:** $\frac{N}{B}$ I/Os after simple modification:

- Read $A[1], \ldots, A[B]$ into a buffer for A.
- Read elements of A always from this buffer.
- If buffer is empty, then read next B elements of A into buffer.
- Analogous strategy for output A'.

11.3 Sorting: MergeSort

The previous technique generalizes to more than two streams, as long as the internal memory is large enough to hold the buffers.
The case of many input streams and one output stream occurs in an I/O-efficient variant of MergeSort, to be looked at next.
Sorting is often used for I/O efficient algorithms as a subprocedure in order to eliminate random disk accesses.
We consider two categories of algorithms for external memory sorting: Algorithms based on the merging paradigm (MergeSort) and algorithms based on the distribution paradigm (QuickSort).

Problem 11.3 (MergeSort for External Memory)
Input: Array with N elements in external memory.
Output: Sorted array in external memory.
Assume that $\frac{N}{M}$ is integer.
Basic approach:

- Phase 1:
  - Move a subarray of size M into internal memory.
  - Sort it with standard fast internal memory algorithm.
  - Move sorted subarray, called run, back to external memory.
- Phase 2:
  - Repeatedly merge pairs of shorter runs into larger runs.

We have $\lceil \log(\frac{N}{M}) \rceil$ repetitions of Phase 2.
Using buffers as seen for Prefix Sums, each merge phase requires $\frac{N}{B}$ read and $\frac{N}{B}$ write operations to external memory.
In total, we need $(2\frac{N}{B})(1 + \lceil \log \frac{N}{M} \rceil)$ I/Os, provided that $M \geq 3B$. ⊣

Now the idea is to exploit the fact that typically $M > 3B$ and so more blocks fit into internal memory. We want to make better use of this in the merging phase.
We introduce the k-Way Merge. We merge k input runs into one output run. This works easily if $k + 1$ buffers and some additional space are available in internal memory.
To find the smallest of k values each time, use a priority queue, for which insert/delete of one element takes $\mathcal{O}(\log k)$ time.
A straightforward implementation using buffers as for Prefix Sum is possible.
But how do we choose k? The internal memory space requirement is $k+1$ buffers plus space for k-element priority queue. Hence $(k + 1)B + \mathcal{O}(k) \leq M$, that is, $k \in \mathcal{O}(\frac{M}{B})$. We only have $\lceil \log_k \frac{N}{M} \rceil$ merging subphases, yielding in total $2\frac{N}{B} \cdot (1 + \lceil \log_{M/B} \frac{N}{M} \rceil) = \mathcal{O}((\frac{N}{B}) \cdot \log_{M/B} \frac{N}{B})$ I/Os.

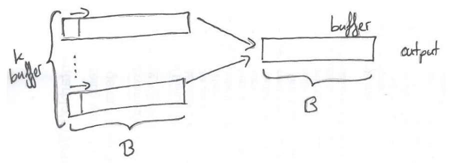


Figure 36: Multi-way Merge Sort

Theorem 11.4 Sorting can be done with $\mathcal{O}((\frac{N}{B}) \cdot \log_{M/B} \frac{N}{B})$ I/Os and needs $\Omega(\frac{N}{B} \cdot \log_{M/B} \frac{N}{B})$ I/Os.

This statement looks weird. Why not use Θ? Why are there brackets in only one case?

Proof 11.5 (idea) Upper bound: see above.
Lower bound: show a lower bound for the number of I/Os needed to generate a permutation of the input; this is

$$\geq 2 \cdot \frac{N}{B} \cdot \frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B} \text{ I/Os.} \qquad \square$$

11.4 Sorting: SampleSort

Samplesort is quicksort for external memory. It is a randomised method with the same expected asymptotic running time as the previous method.
The advantage is, that it is easier to adapt to multiple hard disks and parallel processors.
The idea is instead of using one pivot as in QuickSort, we now use $k-1$ splitters $s_1, \dots, s_{k-1}$ to partition input into k buckets. The bucket i holds the elements e with $s_{i-1} \leq e < s_i$ ($s_0 := -\infty, s_k := \infty$). We sort the buckets recursively.
The goal is that all buckets size roughly $\frac{N}{k}$, so we need good splitters.

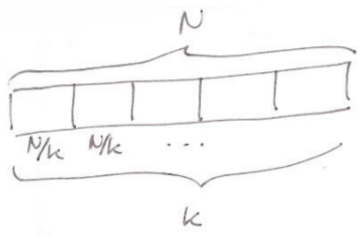


Figure 37: Sample Sort partition into k buckets, preferably equal size

A randomised strategy for finding the buckets is to randomly choose a sample of $(a+1)k-1$ elements from the input for some integer $a \geq 0$. Then internally sort this sample increasingly into an array $S[1 \dots (a+1)k-1]$ and choose $S[(a+1)i]$ as splitters for $1 \leq i \leq k-1$.
So our next question is how to choose the sample size. If we use $a = 0$, we have a small sample, but non-uniform bucket sizes. If we use $a = N$ we have a perfect distribution, but the sample is too big. $a \in \mathcal{O}(\log k)$ is a good choice.
Most expensive is the distribution of all elements to the k buckets.

- Use one buffer block for input and k further buffer blocks for buckets, similarly as for k-way Merge Sort.
- Keep k splitters in sorted array such that each element can be assigned to its block in $\mathcal{O}(\log k)$ time using binary search.

Theorem 11.6 SampleSort performs an expected number of $\mathcal{O}((\frac{N}{B}) \cdot \lceil \log_{M/B} \frac{N}{M} \rceil)$ I/Os and $\mathcal{O}(N \log N)$ internal operations.

Proof 11.7 (Sketch) There are $k = \Theta(\min\{\frac{N}{M}, \frac{M}{B}\})$ buckets and a size-$\mathcal{O}(k \log k)$ sample.
Then only with small probability we obtain a bucket of size far beyond average. More precisely one can show the central claim: Let $k \geq 2$ and $a + 1 = 12 \ln k$. Then with probability $\geq \frac{1}{2}$ no sample contains more than $4\frac{N}{k}$ elements. (Proof of this Claim uses Chernoff bounds.)
Using the claim the theorem "easily" follows. $\square$

11.5 Cache-Oblivious Model

Up until now we looked at a cache-aware model.
Cache-oblivious algorithms perform well on any memory hierarchy without knowing its parameters. Data structures are particularly important in this context. The principle is designing external memory algorithms without knowing B and M. In this model our memory consists of cache("internal") and disk("external"). The consequences are:

- If a cache-oblivious algorithm performs well in the two-level model, then it performs well between any two adjacent levels of a memory hierarchy with more levels.
- Self-tuning: other than for standard I/O algorithms, one does not need to know hardware-specific parameters as B and M for different hierarchy levels.
- Loss of freedom to explicitly manage internal memory/caches.

We make a few assumptions in this model:

- Optimal page replacement: The page replacement strategy knows the future and always evicts the page accessed farthest in the future.
- Full associativity: Any block can be stored anywhere in the cache.
- Tall cache: $\frac{M}{B} > B$, that is, $M = \Omega(B^2)$.

To justify this model we can use some "reductions" that modify ideal-cache algorithms to operate on more realistic models. In most cases, this only costs constant factors. One can show that LRU and FIFO page replacement strategies do just as well as optimal replacement up to a constant factor of memory transfers and up to a constant factor of cache wastage. For some constant $\alpha > 0$, a size-αM LRU cache with block size B can be simulated in M space such that an access to a block takes $\mathcal{O}(1)$ time.

11.5.1 Scanning

The idea is quite simple: we lay out all N elements in a contiguous segment of the memory.

Theorem 11.8 Scanning N elements costs at most $(\frac{N}{B} + 1)$ I/Os.

Proof 11.9 Main issue is where the block boundaries are relative to the beginning and the end of the contigous segment of memory. In the worst case the first and the last block have only one element. That means we habe $\frac{N}{B} - 1$ full blocks. (fig. 38) $\frac{N}{B} - 1$ full blocks and 2 non-full blocks lead us to $\frac{N}{B} + 1$ I/Os.$\square$

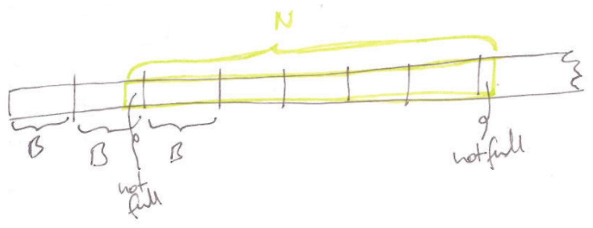


Figure 38: Scanning in the cache-oblivious model

Note that the algorithm does not use B and M, but the analysis does!

11.5.2 Divide and Conquer

The basic idea of divide and conquer is the repeated refinement of the problem size until the problem fits into our cache (size $\leq M$) and later fits into a single block (size $\leq B$).

Example 11.10 (Binary Search) The standard recurrence for the running time is: $T(N) = T\left(\frac{N}{2} + \mathcal{O}(1)\right)$, which has the solution $T(N) \in \Theta(\log N)$. The first idea would be to exploit the "stronger base case" $T(\mathcal{O}(B)) = \mathcal{O}(1)$. Unfortunately this does not help much because this just reduces the number of levels in the recursion tree by $\Theta(\log B)$, that is

$$T(N) = \Theta(\log N) - \Theta(\log B).$$

in contrast to $\mathcal{O}(\log_B N)$ ext. memory using B-trees.

Due to the information-theoretic lower bound, at least $\log_B N + \mathcal{O}(1)$ I/Os and $\log N + \mathcal{O}(1)$ comparisons are required in the average case to search for a desired element.

Theorem 11.11 Search in a complete binary tree with the van Emde Boas layout incurs $\leq 2 + 4\log_B N$ I/Os and $\log N$ comparisons.

Proof 11.12 The idea is to construct a complete binary tree storing N elements in "search tree order". We store the tree sequentially in memory using a recursive layout called "van Emde Boas layout": the tree is split at its middle depth level resulting in one top recursive tree and $\approx \sqrt{N}$ bottom recursive trees.(fig. 39)

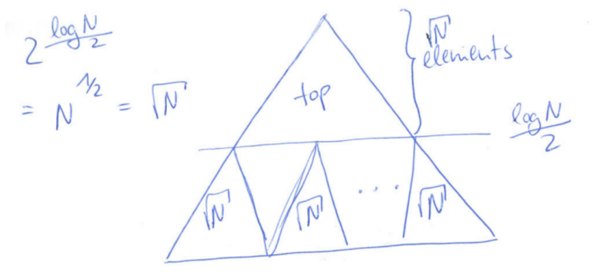


Figure 39: The van Emde Boas layout

Then, recursively layout top subtree followed by recursive layouts of bottom subtrees. Important is that each recursive subtree is laid out in a single memory segment and segments are stored together without gaps.
Then we use the common binary search algorithm. Analysis:
We stop splitting once the subtrees have size $\leq B$.
$\Rightarrow$ each subtree occupies ≤ 2 blocks.
Each recursive subtree (except top one) has same height: $\geq \frac{(\log B)}{2}$ (for subtree size $= \mathcal{O}(\sqrt{B})$).
Now the algorithm searches along root-to-leaf path, which has length $\leq \log N$. One the way it visits a sequence of subtrees, all of heihgt at least $\frac{\log B}{2}$.
$\Rightarrow \leq 1 + 2 \cdot \frac{(\log N)}{(\log B)} = 1 + 2\log_B N$ subtrees that are visited.
Since each subtree incurs 2 I/Os it follows the wanted result. □

11.5.3 Sorting

funnel sort-skipped in lecture .

413/414

11.6 Streaming Model

We now consider an extremely constrained data access. Only a single pass over the data is permitted and only a tiny fraction of what has been read can be stored.
A natural technique is sampling "on the fly". As each piece of data is seen, randomness is used to decide whether to add it to the sample or not. Typically, the probability of adding the piece of data to the sample depends on its value. The idea or motivation here is to only sketch the data.

Example 11.13 (Simple Streaming) **Task:** For a stream of n positive reals $a_1, \ldots, a_n$, draw a sample element a_i so that the probability of picking a_i is proportional to a_i's value.
Sampling method: Upon seeing $a_1, \ldots, a_i$ keep track of $a := \sum_{j=1}^{i} a_j$ and a sample a_k, $k \leq i$, drawn with probability proportional to its value.
On seeing a_{i+1} replace a_k by a_{i+1} with probability $\frac{a_{i+1}}{a+a_{i+1}}$ and update a.

Case 1 a_k is replaced by a_{i+1}. Then clearly a_{i+1} is selected with probability $a_{i+1}/\sum_{j=1}^{i+1} a_j$. $\rightsquigarrow$ OK!

Case 2 a_k is not replaced by a_{i+1}. Then probability of a_k being selected is $\boxed{\frac{a_k}{\sum_{j=1}^{i} a_j}} \cdot \left(1 - \frac{a_{i+1}}{\sum_{j=1}^{i+1} a_j}\right) =$

$\frac{a_k}{\sum_{j=1}^{i+1} a_j}$(blue by induction). $\rightsquigarrow$ OK!

Problem 11.14 (Streaming for Number of Distinct Elements) Stream S**:** integers $a_1, \dots, a_n$, where $1 \le a_i \le m$; n and m very large.
Goal: Determine the number of distinct a_i.
Any deterministic algorithm must use $\ge m$ bits of memory. (there are 2^m possibilities and since it has to be right, it has to distinguish them $\Rightarrow \ge \log(2^m)$ bits)
The randomised approach is to estimate the number of distinct elements in S using only $\mathcal{O}(\log m)$ space.
Assume that the elements are selected uniformly at random from $\{1, \dots, m\}$.
Algorithm: Let min := min$\{S\}$. Elements of S partition $\{1, \dots, m\}$ into $|S| + 1$ subsets, each of size $\approx \frac{m}{|S|+1}$. That implies that min $\approx \frac{m}{|S|+1}$ and thereby that $\frac{m}{\min} - 1$ estimates $|S|$.
If our assumption from before does not hold, this algorithm would fails for some cases, e.g. if S is obtained by selecting the $|S|$ smallest elements of $\{1, \dots, m\}$.
An idea is to use hash functions $h : \{1, \dots, m\} \to \{0, \dots, M - 1\}$ and count $\{h(a_1), h(a_2), \dots\}$ using previous algorithm because this now "behaves like a random subset".
But now M has to be selected: larger M, less collisions, etc. $\dashv$

Theorem 11.15 *Let d be the number of distinct elements in S. If $M > 100d$, then with probability $\frac{2}{3}$ we have $\frac{d}{6} \le \frac{M}{\min} \le 6d$, that is, $\frac{M}{\min}$ is a "good" estimate for d.*

12 Distributed Algorithms

12.1 Distributed Computing

Remark 12.1 In this chapter we regard networks with multiple agents/nodes. These have to communicate with each other to achieve a common goal. In some scenarios this goal must be reached even if some of the agents fail or become malicious. In other cases the agents compete with each other.

Example 12.2 As “old school” examples we have

- parallel computers
- internet

More recently there are

- peer-to-peer systems
- sensor networks
- multi-core architectures

Remark 12.3 There are several possible degrees of freedom for nodes.

- own hardware
- own code
- own independent task/goal
- May share common resources and information

Also we commonly distinguish between the following

Synchronous	vs.	asynchronous
Homogeneous	vs.heterogeneous systems	
Message passing	vs.	shared memory

The fundamental issues are

- Communication,
- Coordination,
- Fault-tolerance,
- Locality,
- Parallelism,
- Symmetry breaking,
- Synchronization,
- Uncertainty.

12.2 Vertex Colouring in Graphs

Problem 12.4 Input: undirected graph $G = (V, E)$
Task: Colour the vertices so that adjacent vertices get different colours, using a minimal number of colours. ⊣

Remark 12.5 Colouring may help in breaking symmetries. Vertex Colouring is NP-hard, so the worst-case is intractable even in the nondistributed setting. Hence distributed algorithms are often happy with suboptimal solutions.
Assumption: Nodes have unique identifiers $1, \ldots, n$ encoded with $\log n$ bits each and each vertex 1:1 corresponds to a distributed computation node.
Observation: A graph with maximum degree Δ can always be coloured using $\Delta + 1$ different colours, using a greedy algorithm.

Definition 12.6 (synchronous distributed algorithm) In a synchronous distributed algorithm, nodes operate in synchronous rounds, where in each round each processor executes the following steps:

1. Do some local computation.
2. Send messages to neighbours.
3. Receive messages from neighbours. ⊣

Algorithm 12.7 Assuming that nodes have pairwise different IDs, there is a simple synchronous algorithm colouring a graph with $\Delta + 1$ colours using n rounds.
Each node starts with colour 1. Each round each node sends its colour to all its neighbours. If a node receives his own colour from a node with higher ID he chooses the next colour he has not received so far. ⊣

Remark 12.8 On trees we can do faster. Since trees are bipartite, they can always be coloured with two colours. Starting with tree root, such a colouring can be found in tree height many rounds. This algorithm does not need to be synchronous.

Definition 12.9 In an asynchronous distributed algorithm, nodes cannot access a global clock. The algorithms are event-driven ("upon receiving message ..., do ...") and messages will arrive in finite but unbounded time. ⊣

Remark 12.10 Synchronous and asynchronous models are the two cornerstones in distributed computing. Every other model is inbetween these extremes.

Definition 12.11 (Message Complexity) The message complexity of a distributed algorithm is the number of messages exchanged between nodes. ⊣

12.3 A "log-star" Algorithm for Tree Colouring

Definition 12.12 ($\log^*$) We define the iterated logarithm as

$$\forall x \leq 2 : \log^*(x) = 1$$
$$\forall x > 2 : \log^*(x) = 1 + \log^*(\log x))$$

It is an extremely slowly growing function. E.g. $\log^*(10^{80}) = 5$. It basically counts how many times you have to apply the logarithm base 2, to get a value at most 2. ⊣

Algorithm 12.13 Idea: Start with colour labels having $\log n$ bits. In each synchronous round compute a new label with exponentially smaller size, still guaranteeing vertex colouring Initialise nodes with n different colours, each represented with $\log n$ bits.

```
root is coloured 0: $v$ has colour $c_v$
for all $v \in V \setminus \{\text{root}\}$ do
    synchronously send $c_v$ to all children
    repeat
        receive $c_p$ from parent
        Look at bitstrings of $c_v$ and $c_p$, both in format $c(k), \dots, c(1), c(0)$ and let $i$ be the smallest index where bitstrings of $c_v$ and $c_p$ differ.
        The new label of $c_v$ is the bitstring of $i$ followed by the bit $c_v(i)$.
        Send $c_v$ to all children
    until $\forall w \in V : c_w \in \{0, \dots, 5\}$
end for
```

The relabeling is where the "logarithmic shrink" happens. ⊣

Example 12.14

$$\begin{aligned} c_p &: 1010010000 \\ c_v &: 0110010000 \end{aligned}$$

$i = 8 \rightsquigarrow$ new $c_v : \underbrace{1000}_{8}1.$

Lemma 12.15 Algorithm 12.13 terminates in $\mathcal{O}(\log^*(n))$ rounds and colours the tree with at most 6 colours.

Proof 12.16 The number of colours is obvious since otherwise the algorithm does not stop. The number of rounds follows from the fact that in each round the label is replaced by one logarithmically in the former size. □

Theorem 12.17 Three-colouring a sixcoloured tree works in $\mathcal{O}(\log^* n)$.

Proof 12.18

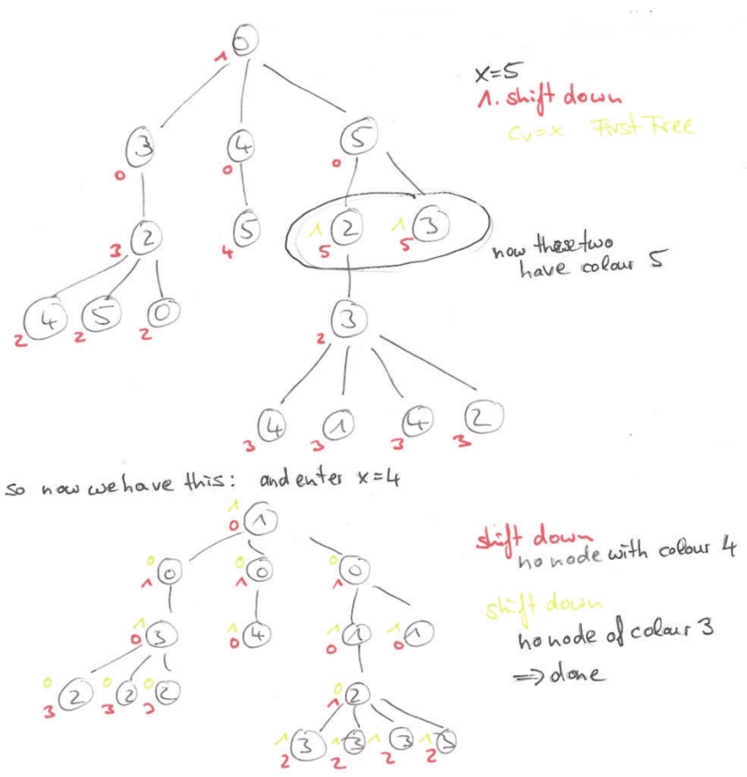
```
compute 6-colouring
for x=5,4,3 do
  ShiftDown()
  if c(v)=x: c(v):=FirstFree()

ShiftDown:
  c(v) := colour of parent
  c(root) = FirstFree

FirstFree:
  c(v) = smallest colour not on parent or child
```

After using ShiftDown, all siblings have the same colour. Hence FirstFree will always give a result from $\{0, 1, 2\}$. So this algorithm eliminates all occurrences of 3,4,5, so in the end we just have 3 colours. For illustration see example 12.19. □

Example 12.19



Remark 12.20

- Algorithm 12.13 generalizes to ring topology.
- Idea for degree-bounded graph colouring with $\Delta + 1$ colours in $\mathcal{O}(\log^* n)$ rounds: In each step, node compares its label to each of its neighbors,
- again constructing a "logarithmic difference tag"
  - New node label := concatenation of all these tags.
  - Yields 3Δ-colouring in $\mathcal{O}(\log^* n)$ rounds.
  - Can be further reduced to $\Delta + 1$ colours in further $2^{3\Delta}$ rounds.
- General graphs, other approach $\rightsquigarrow$ $\Delta + 1$ colours in $\mathcal{O}(\log n)$ rounds.
- $\log^* n$ running time is asymptotically optimal.

12.4 Leader Election

Problem 12.21 (Leader Election) Determine exactly one leader such that every node knows whether it is the leader or not. ⊣

Remark 12.22

- Computing a leader among nodes is a very simple form of symmetry breaking.
- Typical drawback of "leader-based" algorithms: low degree of parallelism, thus slow.
- Many interesting challenges in distributed algorithmics already occur for the ring topology. We concentrate on rings in the following.

Definition 12.23

- A system is anonymous if nodes do not have unique identifiers.
- An algorithm is uniform if the number n of nodes is unknown to it. Otherwise it is called non-uniform. ⊣

Definition 12.24 (symmetric networks) A graph in which the extended neighbourhood of each node has the same structure is symmetric.

extended neighbourhood definition?

Remark 12.25 We distinguish between symmetric networks (rings, complete graphs, complete bipartite graphs, etc.) and asymmetric networks (stars, single nodes with highest degree, etc.).
In a synchronous ring (symmetric network) *anonymous* leader election is impossible, even in the non-uniform case, since symmetry can always be maintained.
The situation may change once allowing randomisation (but not for uniform leader election because such an algorithm cannot distinguish between three or six nodes).

Remark 12.26 In the non-anonymous case by forwarding and comparing IDs, leader election on an asynchronous ring can be solved in $\mathcal{O}(n)$ phases with message complexity $\mathcal{O}(n^2)$.
But we can improve the message complexity.

Algorithm 12.27 Call a node active when it still may become a leader. All nodes are initially active. For our messages we introduce a TTL (time to live). Now let all nodes execute the following code.

1. If v sees a message from node w with $w > v$, then v becomes passive.
2. If active, send "probe messages" into both directions: (ID of sender, leader bit, TTL), starting with TTL $= 1$.
3. For received probe message, decrement TTL and forward modified message if TTL > 0: set leader bit to 0 if own ID is higher than message ID. If TTL $= 0$ then send "reply message" to sender.
4. For received reply message: if no higher ID seen, then double TTL compared to previous message and send new probe message. Otherwise v becomes passive.
5. If v receives its own message (not reply), then v decides to be the leader. ⊣

Theorem 12.28 *Assuming that nodes are non-anonymous, Leader Election on an asynchronous ring can be solved in $\mathcal{O}(n)$ phases with message complexity $\mathcal{O}(n \log n)$.*

Proof 12.29 We use algorithm 12.27. As number of rounds we regard the longest causal chain of messages. As a phase, we regard a collection of rounds using the same TTL, so phase r consists of $2 \cdot 2^r$ rounds. Our phases consist of TTL $1, 2, 4, \dots, 2^k$ with $\log n \leq k < \log(n+1)$. So we have $\mathcal{O}(n)$ rounds. Next we regard the number of messages. As shown before, the number of phases is in $\mathcal{O}(\log n)$. Note that if v survives phase r, then no other vertex with distance $\leq 2^r$ to v can have survived. So there are at most $\frac{n}{2^r}$ active nodes in phase r. Hence the total number of messages in phase r is at most

$$2 \cdot 2^r \cdot \frac{n}{2^r} = 2n$$

So we have $\mathcal{O}(\log n)$ phases with $\mathcal{O}(n)$ messages each, giving a total complexity of $\mathcal{O}n \log n)$. □

Definition 12.30 An execution of a distributed algorithm is a list of events sorted by time. An event is a tuple (time, node, type, message), where type $\in \{\text{send}, \text{receive}\}$. ⊣

Remark 12.31 We assume that no two events happen exactly at the same time. To achieve this, ties can be broken arbitrarily.
To get a lower bound for the algorithm, we make the following assumptions about our model.

- We have an Asynchronous ring. Nodes wake up arbitrarily but at the latest when receiving first message.

- We have uniform algorithms where node with maximum ID can become leader. Every node different from leader must finally know leader.
- Respect FIFO conditions for links.

Theorem 12.32 *Any uniform Leader Election algorithm for asynchronous rings has $\Omega(n \log n)$ message complexity.*

12.5 A Glimpse on Distributed Sorting

Definition 12.33 For distributed sorting, in a graph with n nodes $v_1, \ldots, v_n$, each initially storing a value, the goal is that for all k, node v_k finally stores the k^{th} smallest value. ⊣

Lemma 12.34 If an oblivious comparison-exchange algorithm (that is, the exchange of two elements must exclusively depend on the relative order of their values) sorts all inputs of 0's and 1's, then it sorts arbitrary inputs.

Proof 12.35 (idea) Assume that such an algorithm does sort not an arbitrary input. Then construct corresponding 0-1 input it does not sort as well. □

Algorithm 12.36 Assume our nodes lie on a path.

```
repeat
    Compare and exchange values at nodes $i$ and $i+1$ if $i$ is odd
    Compare and exchange values at nodes $i$ and $i+1$ if $i$ is even
until done
```
⊣

Theorem 12.37 *Odd/Even Sort sorts correctly in n steps.*

Proof 12.38 (idea) Use lemma 12.34 and induction □

13 Communication Complexity

13.1 Motivation and Basic Definitions

In distributed computing, multiple network agents jointly solve certain tasks. In order to achieve their goal, they have to communicate (expensive, time-consuming).
The communication complexity specifies how many communication bits need to be exchanged by agents in a distributed system in order to perform a given task (e.g. compute the value of a function)?
It provides mostly "impossibility" results, that is, lower bounds on what can be achieved by algorithms and simple models to prove strong lower bounds.
Communication models are:

- deterministic (two-party, multi-party),
- probabilistic,
- non-deterministic and
- quantum.

Applications are:

- networks, distributed and parallel computing,
- streaming algorithms (space lower bounds),
- circuit complexity (depth lower bounds),
- VLSI(Very-large-scale integration) design (area-time trade-offs) and
- data structures (space-time trade-offs).

Definition 13.1 (The Two-Party Model) A simple (deterministic) model comprising two communicating parties (Alice and Bob) computing a two-argument Boolean function (Yao 1979).
Input: A Function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, Alice's input $x = x_1 \dots x_n \in \{0,1\}^n$ and Bob's input $y = y_1 \dots y_n \in \{0,1\}^n$.
Task: Compute $f(x, y)$. $\dashv$

Example 13.2 (Example functions)

- Parity: $\mathrm{PAR}(x, y) := (\#_1(x) + \#_1(y)) \bmod 2$
- Equality: $\mathrm{EQ}(x, y) := 1$ if $x = y$ (0 otherwise)
- Disjointness: $\mathrm{DISJ}(x, y) := 0$ if $x_i = y_i = 1$ for some $1 \leq i \leq n$ (1 otherwise)

Definition 13.3 (Communication Protocol) A communication protocol specifies which messages (strings of bits) Alice and Bob alternatingly send to each other.(fig. 40) Protocol $\mathcal{P}$ *computes* function f

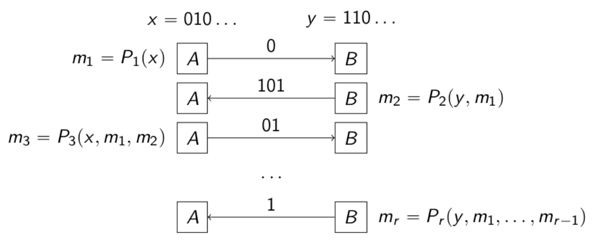


Figure 40: Communication Protocol

in r rounds if $m_r = f(x, y)$ holds for every pair (x, y). $\dashv$

Definition 13.4 (Transcript) A transcript $s_{\mathcal{P}}(x,y) := (m_1,\dots,m_r)$ is the sequence of exchanged messages during execution of protocol $\mathcal{P}$ on input (x,y). By $|s_{\mathcal{P}}(x,y)| := \sum_{i=1}^{r} |m_i|$ we denote the overall number of bits exchanged. ⊣

Definition 13.5 (Communication Complexity)
The communication complexity of a protocol $\mathcal{P}$ is

$$C(\mathcal{P}) := \max_{x,y\in\{0,1\}^n} |s_{\mathcal{P}}(x,y)|.$$

The communication complexity of a function f is

$$C(f) := \min_{\mathcal{P} \text{ computing } f} C(\mathcal{P}).$$ ⊣

Note that we do not care about computational resources for computing the messages m_i.
We have a trivial upper bound for the communication complexity: $C(f) \le n+1$ for all $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$. Because we can formulate the trivial protocol: Alice send her input x to Bob (n bits) and Bob (holding input y) determines $f(x,y)$ and sends the answer back to Alice.
So the central question is, whether we can do better.

13.2 Introductory Examples

Problem 13.6 (Parity) $\mathrm{PAR}(x,y) := (\#_1(x) + \#_1(y)) \bmod 2$ ⊣

We observe that $C(\mathrm{PAR}) \le \log n + 1$ using the protocol: Alice sends $\#_1(x)(\log n$ bits $)$ and Bob computes $(\#_1(x) + \#_1(y)) \bmod 2$.

Theorem 13.7 $C(\mathrm{PAR}) = 2$.

Proof 13.8 With the protocol:

1. Alice sends $p_x := \#_1(x) \bmod 2$.
2. Bob computes $p_y := \#_1(y) \bmod 2$ and sends $(p_x + p_y) \bmod 2$.

Clearly $C(f) \ge 2$ holds for all functions depending non-trivially on both inputs. □

Problem 13.9 (Halting Problem)

$$H(x,y) := \begin{cases} 1 & x = 1^n, y = \mathrm{encode}(M),\ \text{TM } M \text{ holds on } x \\ 0 & \text{else.} \end{cases}$$ ⊣

Theorem 13.10 $C(H) = 2$.

Proof 13.11 With the protocol:

1. Alice sends 1 if $x = 1^n$ and 0 otherwise.
2. Bob determines whether M halts on 1^n (if Alice sent 1) and sends the answer. □

Note that we assume that the two parties have unbounded computational power.

Problem 13.12 (Arithmetic Mean) $x,y \subseteq [n] = \{1,\dots,n\}, x \cap y = \emptyset$
$\mathrm{MEAN} : 2^{[n]} \times 2^{[n]} \to \mathbb{Q}$
$\mathrm{MEAN}(x,y) :=$ arithmetic mean of $x \cup y (= \frac{1}{|x\cup y|} \sum_{i\in x\cup y} i)$. ⊣

The trivial protocol send $\mathcal{O}(n \log n)$ bits.

I thought the trivial protocol need $n+1$ and that is always our upper bound? The problem is the representation. Presenting x,y as $\{0,1\}^n$ we have complexity $\le n+1$.

Can we achieve $\mathcal{O}(\log n)$ bits?

Theorem 13.13 $C(\text{MEAN}) \in \mathcal{O}(\log n)$.

Proof 13.14 With the protocol:

1. Alice sends $\overline{x} := \frac{1}{|x|}\sum_{i\in x} i$ ($\log n$ bits) and $|x|$ ($\log n$ bits).
2. Bob sends $\frac{1}{|x|+|y|}(|x| \cdot \overline{x} + |y| \cdot \overline{y})$. □

Problem 13.15 (Median) $x, y \subseteq [n] = \{1, \dots, n\}, x \cap y = \emptyset$
$\text{MED} : 2^{[n]} \times 2^{[n]} \to [n]$
$\text{MED}(x, y) :=$ median of $x \cup y$. ⊣

Theorem 13.16 $C(\text{MED}) \in \mathcal{O}((\log n)^2)$.

Proof 13.17 With the protocol:

1. Alice sends $|x|$ ($\log n$ bits).
2. Alice sends $|\{i \in x | i \geq \frac{n}{2}\}|$($\log n$ bits).
3. Bob determines whether $\text{MED}(x, y) \geq \frac{n}{2}$ or $< \frac{n}{2}$ and notifies Alice (1 bit).
4. Repeat 2. and 3. (binary search).

Step 2. and 3. are repeated at most $\mathcal{O}(\log n)$ times and in each round Alice and Bob send $\mathcal{O}(\log n)$ bits.□

13.3 Lower Bounds

Problem 13.18 (Equality)

$$\text{EQ}(x, y) := \begin{cases} 1 & , x = y \\ 0 & , \text{else.} \end{cases}$$

It is a simple and "natural" function of great importance e.g. in distributed computing (consistency checks of files/strings). ⊣

We recall that $C(\text{EQ}) \leq n + 1$ (trivial). Now we want to know: Is that optimal? Intuitively it is. With fewer bits Alice and Bob cannot correctly determine the equality of each single bit.
We now want to introduce a formal framework for proving communication complexity lower bounds such as

Theorem 13.19 $C(\text{EQ}) \geq n$.

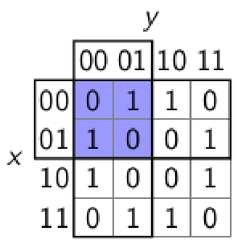
13.3.1 The Matrix View

Consider a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ as a $2^n \times 2^n$ binary matrix M_f over the input space $\{0,1\}^n \times \{0,1\}^n$.
The idea is to analyse the combinatorial structure of M_f to prove communication complexity lower bounds.

Definition 13.20 (Combinatorial and monochromatic rectangle) A combinatorial rectangle is a subset $R = A \times B \subseteq \{0,1\}^n \times \{0,1\}^n$, where $A, B \subseteq \{0,1\}^n$. A rectangle R is f-monochromatic if for every $x \in A$ and $y \in B$ the value $f(x, y)$ is the same. ⊣

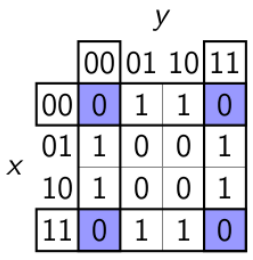
Example 13.21 (M_{PAR})

$R = \{00, 01\} \times \{00, 01\}$



$x \backslash y$	00	01	10	11
00	0	1	1	0
01	1	0	0	1
10	1	0	0	1
11	0	1	1	0

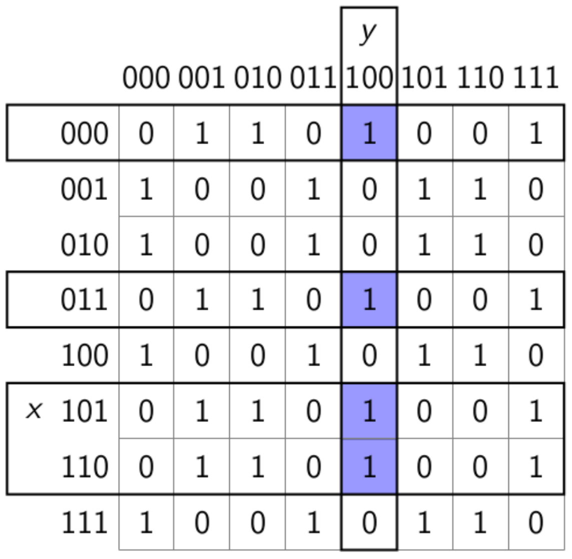
$R = \{00, 11\} \times \{00, 11\}$



$x \backslash y$	00	01	10	11
00	0	1	1	0
01	1	0	0	1
10	1	0	0	1
11	0	1	1	0

A protocol for f defines f-monochromatic rectangles.

Example 13.22 (Parity)



$x \backslash y$	000	001	010	011	100	101	110	111
000	0	1	1	0	1	0	0	1
001	1	0	0	1	0	1	1	0
010	1	0	0	1	0	1	1	0
011	0	1	1	0	1	0	0	1
100	1	0	0	1	0	1	1	0
101	0	1	1	0	1	0	0	1
110	0	1	1	0	1	0	0	1
111	1	0	0	1	0	1	1	0

Alice sends $\#_1(x) \bmod 2 = 0$. Bob sends $0 + \#_1(y) \bmod 2 = 1$.

Lemma 13.23 Let $\mathcal{P}$ be a protocol computing function f and $(m_1, \ldots, m_r)$ be a sequence of messages. The set $\{(x, y) | s_{\mathcal{P}}(x, y) = (m_1, \ldots, m_r)\}$ of inputs producing transcript $(m_1, \ldots, m_r)$ is an f-monochromatic rectangle.

Proof 13.24 We show: For every $i = 1, \ldots, r$ the set $R_i := \{(x, y) | s_{\mathcal{P}}(x, y) = (m_1, \ldots, m_r)\}$ is a rectangle.
For $i = 1$: $R_1 := A \times \{0, 1\}^n$, $A \subseteq \{0, 1\}^n$ (Alice' first msg restricts the rectangle).
Let R_i be the set of inputs in which $\mathcal{P}$ starts with $(m_1, \ldots, m_i)$ $R_i = A \times B, A, B \subseteq \{0, 1\}^n$.
Assume w.l.o.g. that Alice sends m_{i+1}.
Let $A' \subseteq A, A' := \{x \in A | \mathcal{P}_{i+1}(x, m_1, \ldots, m_i) = m_{i+1}\}$. Then clearly $R_{i+1} = A' \times B \Rightarrow R_r$ is a rectangle.
Since $\mathcal{P}$ computes f (correctly), it follows $\forall (x, y) \in R_r \quad f(x, y) = m$. □

Hence, a protocol for f partitions the input space into f-monochromatic rectangles.

Definition 13.25 For a function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ we define $\chi(f)$ to be the minimum number of f-monochromatic rectangles that partition the input space $\{0,1\}^n \times \{0,1\}^n$. $\dashv$

Lemma 13.26 For every function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ it holds $C(f) \geq \log \chi(f)$.

Proof 13.27 The idea is that n communication bits define at most 2^n rectangles.(proof 13.27)
We know from lemma 13.23 that every protocol $\mathcal{P}$ computing f partitions $\{0,1\}^n \times \{0,1\}^n$ into monochromatic rectangles. (Every possible transcript determines a rectangle.) The number of different transcripts is $2^{C(\mathcal{P})}$.

$$\begin{aligned}
&\Rightarrow \chi(f) \leq 2^{C(\mathcal{P})} \text{ for every protocol } \mathcal{P} \text{ computing } f \\
&\overset{C(f)=\min_{\mathcal{P}} C(\mathcal{P})}{\Rightarrow} \chi(f) \leq 2^{C(f)} \\
&\Rightarrow C(f) \geq \log \chi(f).
\end{aligned}$$
$\square$

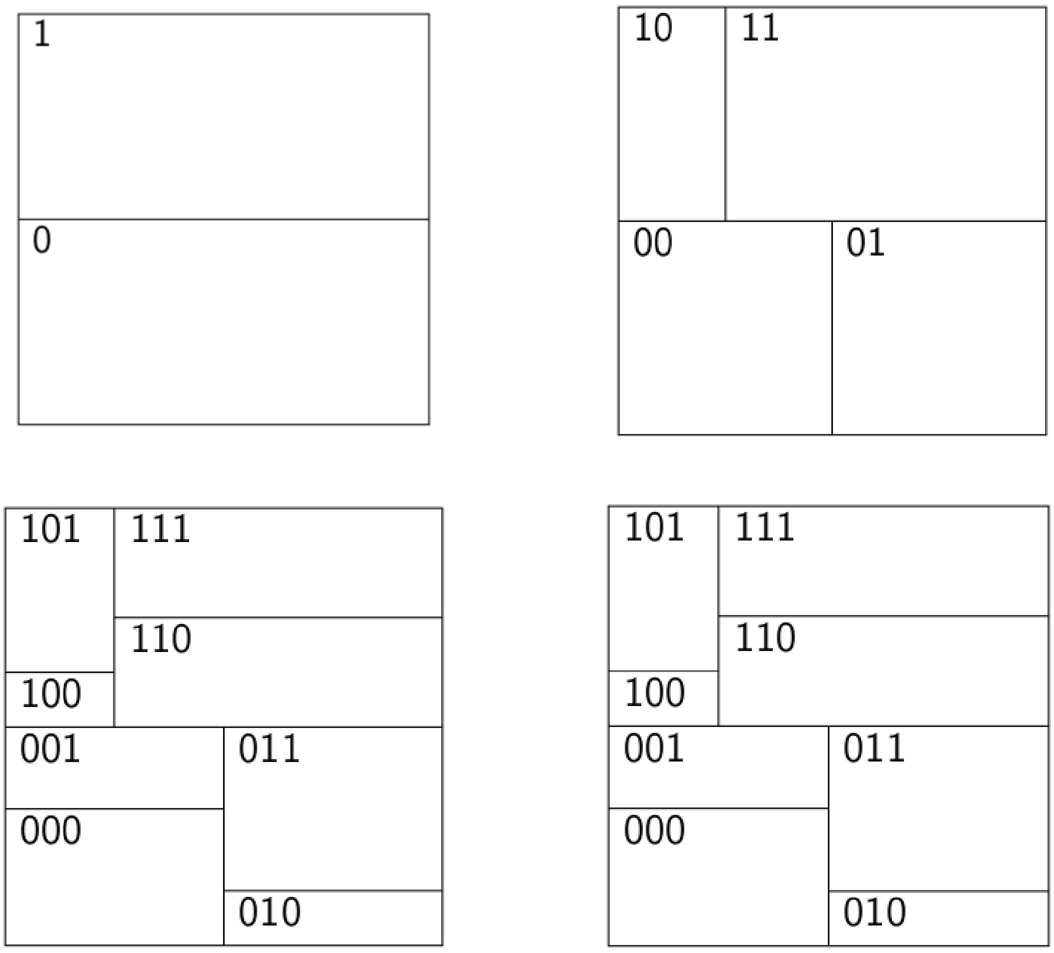


Figure 41: n communication bits define at most 2^n rectangles

Note that computing $\chi(f)$ is NP-hard (biclique partition problem).

Corollary 13.28 Lower bounds on $\chi(f)$ imply lower bounds on $C(f)$. $\dashv$

That is to prove lower bounds on the communication complexity $C(f)$, we can prove lower bounds on the minimum number $\chi(f)$ of f-monochromatic rectangles required to partition the space of inputs.
In the following, we show two methods to prove lower bounds:

1. fooling set method (Yao 1979, Lipton & Sedgewick 1981),
2. rank method (Mehlhorn & Schmidt 1982).

13.3.2 The Fooling Set Method

The idea is to find a large set of inputs such that no two of them are contained in the same f-monochromatic rectangle.

Definition 13.29 (Fooling Set) A set of input pairs $\{(x_1, y_1), \dots, (x_l, y_l)\}$ is called a fooling set (of size l) with respect to f if there exists a $b \in \{0,1\}$ such that

- for all $i \in \{1, \dots, l\}$, $f(x_i, y_i) = b$ and
- for all $i, j \in \{1, \dots, l\}$ with $i \neq j$, $f(x_i, y_j) \neq b$ or $f(x_j, y_i) \neq b$. (fig. 42) $\dashv$

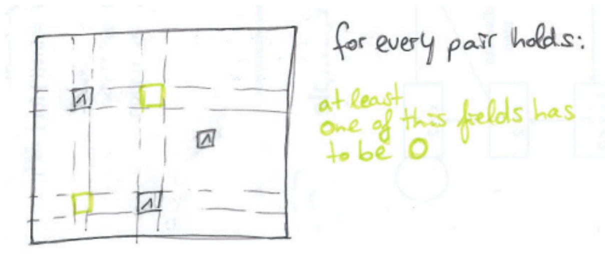


Figure 42: Fooling Set

Lemma 13.30 If there exists a fooling set of size l with respect to f, then $C(f) \geq \log l$.

Proof 13.31 We show that $\chi(f) \geq l$ using the definition of a fooling set.
Let S be a fooling set of size l. Show that $\chi(f) \geq l$.
Assume $\chi(f) < l \Rightarrow \exists (x_i, y_i), (x_j, y_j) \in S$ such that $(x_i, y_i) \in R, (x_j, y_j) \in R$ for a f-monochromatic rectangle $R = A \times B$.
$\overset{R \text{ rectangle}}{\Rightarrow} (x_i, y_j), (x_j, y_i) \in R$
$\overset{\text{Def. fooling set}}{\Rightarrow} f(x_i, y_i) = (x_j, y_j) = b \in \{0,1\}$ but $F(x_i, y_j) \neq b$ or $f(x_j, y_i) \neq b$ which is a contradiction to R being f-monochromatic. $\square$

We can now give a (simple) proof for the equality lower bound using fooling sets.

Theorem 13.32 $C(\text{EQ}) \geq n$.

Proof 13.33 The set $\{(\alpha, \alpha) | \alpha \in \{0,1\}^n\}$ is a fooling set of size 2^n, since for all α $\text{EQ}(\alpha, \alpha) = 1$, while for all $\beta \neq \alpha$ $\text{EQ}(\alpha, \beta) = \text{EQ}(\beta, \alpha) = 0$. Hence $C(\text{EQ}) \geq \log(2^n) = n$. $\square$

Theorem 13.34 $C(\text{DISJ}) \geq n$.

Proof 13.35 We arrange the columns and rows for the matrix in ascending order. Put $N = 2^n - 1$. Then k and $N - k$ are disjoint, since they form complements. Hence on the second diagonal we have all ones. Furthermore going down any column will increase the number. This means we have an additional element, compared to k. But then the corresponding set is not disjoint to $N - k$. Hence the lower right triangle has all zeros. Therefore our matrix has full rank 2^n, which gives complexity n.
Alternatively we have the fooling set $\{(A, \overline{A}) : A \subseteq \mathbb{N}\}$. Then $f(A, \overline{A}) = 1$ for all A. Assume $A \neq B$ with $f(A, \overline{B}) = f(\overline{A}, B) = 1$. This implies $A \cap \overline{B} = \emptyset \Rightarrow A \subseteq B$ and analogously $B \subseteq A$ ↯. $\square$

13.3.3 The Rank Method

The idea is to bound the number $\chi(f)$ of monochromatic rectangles partitioning M_f from below by the rank of M_f. This allows us to apply known results from linear algebra.
Recall that the rank of a matrix M is the maximum number $\operatorname{rank}(M)$ of linearly independent columns/rows in M.

Lemma 13.36 Let $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ then $C(f) \geq \log(\,M_f)$.

For the rank of a matrix we have the property of subadditivity: $\operatorname{rank}(A+B) \leq \operatorname{rank}(A) + \operatorname{rank}(B)$.

Proof 13.37 We show that $\chi(f) \geq \operatorname{rank}(M_f)$ by writing M_f as $M_f = \sum_{i=0}^{t} M_i$ where M_i is a rank-1-matrix corresponding to a 1-monochromatic rectangle R_i (that is a f-monochromatic rectangle that contains 1's).
Let $R_1, \ldots, R_t$ be the 1-monochromatic rectangles of an optimal partition.
It is sufficient to show $t \geq \operatorname{rank}(M_f)$.
For $R_i (i = 1, \ldots, t)$, define a $2^n \times 2^n$ matrix M_i with $M_i(x,y) = 1$ if $(x,y) \in R_i$ and 0 otherwise.
$\Rightarrow M_f = \sum_{i=1}^{t} M_i \Rightarrow$ Since R_i is 1-monochromatic it follows that $\operatorname{rank}(M_f) = 1$
$\overset{\text{subadditivity}}{\Rightarrow} \operatorname{rank}(M_f) \leq t$. □

An improvement can be made considering $M_{\overline{f}}$ (all entries flipped): $C(f) \geq \log(\operatorname{rank}(M_f) + \operatorname{rank}(M_{\overline{f}}))$.
Using the rank lower bound, we obtain another easy proof of the lower bound for the equality function EQ.

Theorem 13.38 $C(\mathrm{EQ}) \geq n$.

Proof 13.39 The matrix M_{EQ} clearly equals the $2^n \times 2^n$ identity matrix. Therefore $\operatorname{rank}(M_{\mathrm{EQ}}) = 2^n$ and thus $C(\mathrm{EQ}) \geq \log(2^n) = n$. □

The rank of a matrix can be computed efficiently (in time polynomial in the size of the matrix).

Problem 13.40 (Inner Product) $\mathrm{IP} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$
$\mathrm{IP}(x,y) := (\sum_{i=0}^{n} x_i \cdot y_i) \bmod 2$

	000	001	010	011	100	101	110	111
000	0	0	0	0	0	0	0	0
001	0	1	0	1	0	1	0	1
010	0	0	1	1	0	0	1	1
011	0	1	1	0	0	1	1	0
100	0	0	0	0	1	1	1	1
101	0	1	0	1	1	0	1	0
110	0	0	1	1	1	1	0	0
111	0	1	1	0	1	0	0	1

M_{IP} is a so-called Hadamard matrix with rank $2^n - 1$. Thus $C(\mathrm{IP}) \geq n$. ⊣

One of the greatest open problems in communication complexity is to upper bound the communication complexity polylogarithmically in the rank.

Conjecture 13.41 (Log Rank Conjecture (Lovasz & Saks 1988)) There is a constant $c > 1$ such that for every function $f : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$:

$$C(f) \leq (\log \operatorname{rank}(M_f))^c + 2.$$

(Nisan & Wigderson 1995) proved lower bound $c \geq 1.63$.

13.4 Applications

Problem 13.42 (Streaming Space Lower Bound) Goal: Derive space lower bounds on streaming algorithms.
Approach: Show that small-space streaming algorithms imply protocols of low communication complexity (which cannot exist). ⊣

We use the following simple streaming model:
Input: Stream $x_1, x_2, \dots, x_m \in U$ of elements from universe $U := \{1, \dots, n\}$ arriving one-by-one.
Task: Compute in one pass a function f of $x_1, \dots, x_m$.
For example frequency moments.

Definition 13.43 For $j \in U$ let $f_j \in \{0, \dots, m\}$ be the number of times that j occurs in the stream and let $F_k := \sum_{j \in U} f_j^k$.
Then F_0 is the number of distinct elements and $F_\infty := \max_{j \in U} f_j$ is the maximum frequency of an element in the stream. ⊣

We saw in one of the previous lectures that F_0 requires $\Omega(n)$ space (deterministic) and $\mathcal{O}(\log n)$ space (randomised).

Theorem 13.44 Every (deterministic) streaming algorithm computing F_∞ uses space $\Omega(n)$.

Recall: $\text{DISJ}(x, y) := 0$ if $x_i = y_i = 1$ for some $1 \leq i \leq n$ (1 otherwise).

Proof 13.45 Let S be a space-s streaming algorithm for F_∞. Consider the following protocol $\mathcal{P}$ for DISJ:
Given $(x, y) \in \{0,1\}^n \times \{0,1\}^n$,

1. Alice feeds $\{i | x_i = 1\}$ into S and sends the current memory state σ to Bob (s bits).
2. Bob resumes S with the memory state σ and feeds $\{i | y_i = 1\}$ into S.
3. Bob sends a 1 if and only if S's answer is at most 1 (0 otherwise).

Note that for every $i \in \{1, \dots, n\}$ the frequency f_i in the data stream induced by (x, y) is either

- 0 (if $x_i = y_i = 0$),
- 1 (if $x_i + y_i = 1$) or
- 2 (if $x_i = y_i = 1$).

Thus F_∞ if this data stream is 2 if and only of $\text{DISJ}(x, y) = 0$. Hence protocol $\mathcal{P}$ computes DISJ using $C(\mathcal{P}) = s + 1$ bits. Since $C(\text{DISJ}) \geq n$ (theorem 13.34), it follows that $s \in \Omega(n)$. □

The above result can even be strengthened further.

Theorem 13.46 Every randomised streaming algorithm that, for every data stream, computes a factor 1.2 approximation of F_∞ with probability at least 2/3 uses $\Omega(n)$ space.

13.5 Randomised Communication Complexity

Now Alice and Bob are allowed to "toss coins", that is, the message sent in each round is a probabilistic function of the input and the communication so far.
A randomised protocol $\mathcal{P}$ computes a function f if for every input (x, y) the probability that the output of $\mathcal{P}$ equals $f(x, y)$ is at least 2/3.
The randomised communication complexity $R(\mathcal{P})$ is the worst-case number of bits exchanged (over all inputs (x, y) and over all random choices).
Some functions have a significantly smaller randomised communication complexity than the deterministic lower bounds proved (e.g. $R(\text{EQ}) \in \mathcal{O}(\log n)$).
For many other functions, the same lower bounds can be proven even for the randomised case (e.g. $R(\text{DISJ}) \in \Omega(n), R(\text{IP}) \in \Omega(n)$).
An example application is to find lower bounds on threshold circuits (discrete neural computation).

Theorem 13.47 Any threshold circuits that computes the inner-product function IP has $\Omega(n/(\log n)^2)$ gates.

13.6 Variable Partition Model

Consider a single-argument function $f : \{0,1\}^m \to \{0,1\}$ where $m = 2n$.
In the variable partition model Alice and Bob are allowed to choose the partition of the m input bits among them (n for each). Note that the partition is based on f and not on a specific input.
Define $C^{\text{best}}(f)$ as the (deterministic) communication complexity of the best protocol for f with respect to the best partition of the m inputs into two sets (clearly, $C^{\text{best}}(f) \leq C(f)$, e.g. $C^{\text{best}}(\text{EQ}) = 2$).
An example application are area-time trade-offs for VLSI chips.

Theorem 13.48 For every VLSI chip with area A and time T computing a function $f : \{0,1\}^m \to \{0,1\}(m = 2n)$ it holds $AT^2 \geq (D^{\text{best}}(f))^2)$.

13.7 Outlook

There are other models:

- Randomised protocols (public vs. private coins)
- Non-deterministic protocols (overlapping rectangles, monochromatic coverings, relation to deterministic communication complexity $C(f) \in \mathcal{O}(N(f)^2)$)
- Multi-party protocols ("number on the forehead" model)

The is a so-called "Discrepancy method" to find deterministic lower bounds.

14 Resilient Algorithms and Data Structures

14.1 What Are We Dealing With?

The world is not (computationally) perfect. So we have to deal with errors. We consider here memory errors, that is, a bit is read incorrectly. Why does that happen? It might be due to interference (electrical, magnetical, ...) or due to hardware problems (memory) or due to cable problems (communication). Memory errors occur quite often. An experimental estimation was shown by [Tezzaron Semiconductor, White Paper 2004], a few thousands memory errors per billion hours per megabit is fairly typical. Implying [Christiano et al., WADS 2011] roughly one error every five hours on a modern PC with 24 gigabytes of memory. Big companies (Google, Apple, etc. ...) have huge data center with millions of cheap PCs ....

14.2 Error Correcting Code

The general idea is repetition to achieve redundancy. Full redundancy would be the naive approach, just store or send the data several times. This is very simple, but also very expensive in space, time and communication.
A better approach is partial redundancy. We store a function of the handled data, the parity bit. Specifically, for each 7 bits store another bit (0 stands for even, 1 for odd). This approach is much cheaper. But, it does not detect $\mathbb{N}_{\text{even}}$ number of faults and what happens when the parity bit itself is faulty?
We can use Hamming codes. Here the parity bit is smartly generalised. Several parity bits are stored at specific places each storing the parity bit of a specific subset of the data bits. This is already much better than the normal parity bit, but still only detects low error-rates.
A different approach is Reed-Solomon. Here we think about the data as a polynomial. Remember that any k distinct points uniquely determine a polynomial of degree at most $k-1$. So here writing is done by over-sampling the polynomial and reading is done by polynomial interpolation (approximation). This method can deal nicely with lots of errors, but it is also more complex due to non-trivial computation.
So to sum up this approach is very effective, can also be implemented in hardware and works best for communication (transmission corruption). But on the other hand it usually does not deal with corruptions during computation (unaware of exact computation) and it only gives probabilistic guarantees.

14.3 Other Models

Model 14.1 (Pointer Model) **Scenario:** Pointers can get corrupted during the computation.
Goal: Minimise the amount of lost data.
Idea: Add pointers at/to specific position.
Example: Linked-list with one pointer corruption.

Model 14.2 (Parallel and Distributed Computation) **Scenario:** Faulty processors or faulty communication links.
Goal: Decide on something together or compute something together.
Example: Consensus (agree on a value).

14.4 Faulty RAM Model (FRAM)

The goal is to deal with errors during computation, to give a worst-case guarantee and to be cheap. The FRAM builds upon the RAM model (each memory word is $\log(n)$ bits).

Model 14.3 (FRAM)

1. An upper bound δ on the number of faults that can happen during computation is given to the algorithm.
2. Corrupted and uncorrupted cells are indistinguishable.
3. Corruptions are adversarial (can happen at any time and place).

4. (Relaxation) Assume there are $\mathcal{O}(1)$ reliable memory cells (needed, for example, for storing the code itself).

Correctness definition: "Be as correct as possible", meaning correct on the set of uncorrupted values and else give the α-closest value (α is the actual number of faults).

The following algorithms are known in the literature:

- Resilient searching (uncorrupted elements guaranteed to be found)
- Resilient dictionaries (dynamic search)
- Resilient sorting (uncorrupted subset guaranteed to be sorted)
- Resilient priority queues (deletemin returns the minimum or a corrupted element)
- Resilient counters (Supports inc and get operations and return an α-additive approximation)
- Resilient selection (return the $k \pm \alpha$ order-statistics)

14.5 Resilient Search

Problem 14.4 Given: A sorted array A of size n and a number x.
Task: Return true iff $x \in A$. ⊣

Correctness in normal environments:

- Is 7 in the array $[2, 3, 5, 7, 8]$? YES
- Is 6 in the array $[2, 3, 5, 7, 8]$? NO

Algorithms for normal environments:

- Naive: $\mathcal{O}(n)$
- Binary-Search: $\mathcal{O}(\log n)$

Correctness in corrupted environments (by example):

- At the beginning, the input is: $[2, 3, 5, 7, 8]$
- After some time: $[2, 3, 1, 7, 8]$ (first corruption)
- After some time: $[2, 3, 1, 6, 8]$ (second corruption)
- We should be able to find $2, 3, 8$ (they were always present)
- Maybe we will (falsely) say NO for $5, 7$ (they were not always present)
- Maybe we will (falsely) say YES for $1, 6$ (they were not in the input, but they were present for some time)

If we start with the naive resilient algorithm (run through the array from left to right), what can go wrong? Firstly the executing code can get corrupted, secondly we need a counter and that could get corrupted. This can be solved by putting both in the $\mathcal{O}(1)$ safe memory. Now it is correct (on the non-faulty subset), but expensive ($\mathcal{O}(n)$).
So we try with binary search. Now further things can go wrong, because through corruptions the sorting might get lost (remember the input $[2, 3, 5, 7, 8]$ that got corrupted to $[2, 3, 1, 7, 8]$ and try to find 3). The solution is to store each input number $2\delta + 1$ times. Reading is then done by reading all $2\delta + 1$ copies and taking the majority. This also is correct now, but still expensive ($\mathcal{O}(\delta \log(n))$, multiplicative factor both in time and space).
Lets do some sidetracking for a moment and have a look at MAJORITY.

Problem 14.5 (MAJORITY) **Given:** An array of n elements.
Find: The majority element (if exists) (where an element x is the majority element of an array A of size n if it occurs at least $\frac{n}{2} + 1$ times in A). ⊣

Example 14.6 3 is the majority element of $[2, 6, 3, 3, 5, 3, 7, 3, 3, 3]$. There is no majority element for $[2, 6, 4, 7, 7]$.

Now the questions are "How can you solve majority in normal environments?" and "How can you solve majority in corrupted environments?".
We now look at (almost) optimal resilient searching. The main idea is to keep the input as it is and perform mostly fast steps which can be incorrect and once in a while perform a slow, but correct, step. When such a slow step reveals errors made in the past, we repeat from the last "checkpoint" (last slow step made).(fig. 43)

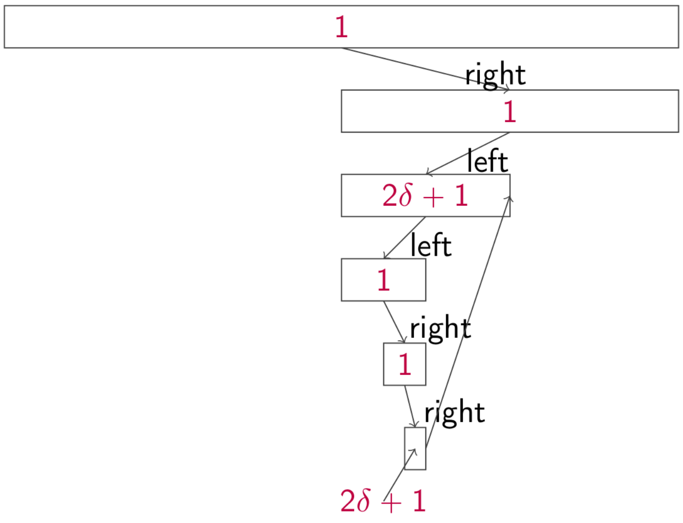


Figure 43: (almost) optimal resilient search visualisation

First we have a look at the correctness of this algorithm. When the algorithm does a slow step it has a closer look at the current part of the array. It looks at the $2\delta + 1$ most left elements and the $2\delta + 1$ most right elements and checks whether the searched value is still in between.(fig. 44) Since there can be at most δ corruption this leads to the correctness of our algorithm.

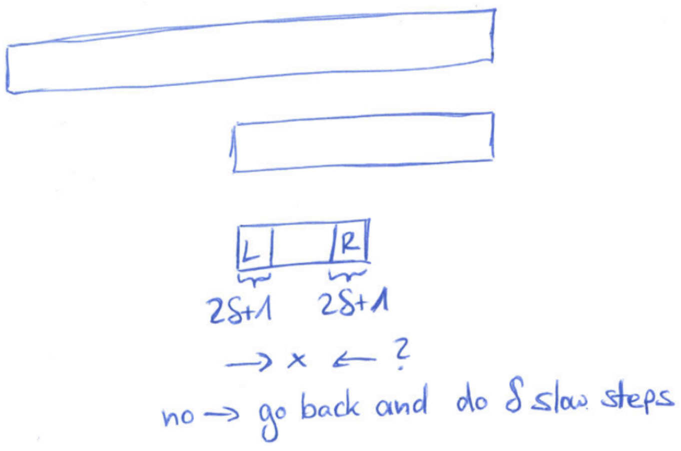

L
R
$2\delta+1$
$2\delta+1$
$\rightarrow x \leftarrow ?$
no $\rightarrow$ go back and do δ slow steps

Figure 44: (almost) optimal resilient search correctness

Now we have a look at the running time. If there are no repetitions, we have one slow step once in δ fast steps, so “for free” (δ constant). We can blame each repetition on a different fault (at most δ slow steps, therefore δ^2 extra work per 1 fault). So we have running time $\mathcal{O}(\log n + \alpha\delta^2)$. With further tricks one can reach the natural lower bound of $\mathcal{O}(\log n + \alpha)$.

14.6 Optimal Resilient Selection

Problem 14.7 Find the k^{th}-smallest element in an unsorted array of n elements. ⊣

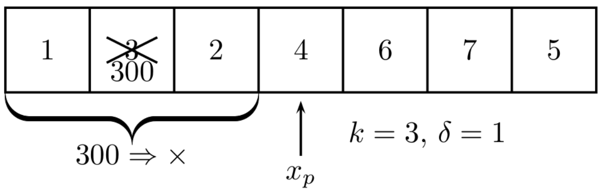
Correctness in “normal” environments:

- 3 is the 2-order statistics in the array $[7, 4, 2, 8, 3, 6]$
- 5 is the $\frac{n}{2}$-order statistics in the array $[4, 7, 2, 6, 1, 3, 5, 9, 8]$ (generalising the median)

Non-resilient algorithms:

- Sorting and picking: $\mathcal{O}(n \log n)$
- Randomised - quickselect: $\mathcal{O}(n)$
- Deterministic - median of medians: $\mathcal{O}(n)$

Using Quickselect as idea works as follows. A random pivot x_p is picked and the input array is partitioned by x_p. Then we continue in one of the parts or halt if the part is empty. This works correctly and has time complexity $\mathbb{E}[T(n)] = \mathcal{O}(n)$ (good chance of cutting roughly in the middle). But what can go wrong in the presence of faults?



We can overcome this by introducing two variables lb and ub, a lower and an upper bound on the “values that make sense” in the safe memory. Initially their are $-\infty$ and ∞. Then we pick the random pivot x_p and partition the input array by x_p. We continue in one part or halt in case it is empty. If we continue in the left part we set $ub \leftarrow x_p$, otherwise $lb \leftarrow x_p$. Reading x_p is done by $\min(\max(\text{lb}, x_p), \text{ub})$. Now this is correct (when we remember to use the $\mathcal{O}(1)$ safe memory). For the time complexity we distinguish between corruptions to the pivot and corruptions to other elements. If other elements are corrupted we do not have any time cost. If the pivot element is corrupted the time cost is n' (current array size), but this happens with probability $\frac{1}{n'}$, so the time cost is roughly 1. In total we have $\mathcal{O}(n + \alpha)$, which is optimal. Now for an approach that works deterministically.
We use median of medians , firstly in non-corrupted environments. We compute the median of each group of 5 consecutive elements and then recursively compute the median of medians x_p. We partition the input array by x_p and continue in one part (or halt in case its empty). This is correct and has time complexity $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + \mathcal{O}(n) = \mathcal{O}(n)$ (also works for other magic numbers other than 5). First we have the same problem like before in the randomised case, which we can solve the same way, by introducing lb and ub. The second problem is, that using recursion we have $\omega(1)$ frame pointers, which can get corrupted and thereby destroy our result. The main idea to solve this is verification like in resilient searching. It is achievable by carefully implementing the recursion. To have more structure in the stacks we want to always divide the current array by a fixed ratio. Specifically we always cut at $\approx \frac{7n}{10}$. For the remainder we use dummies. We gain structure in the recursion implementation. Due to local computation we have an easily invertible size function.(fig. 45)
This way we have a reliable and a faulty stack, reliable global variables and the functions push and pop. The information that is fully reliable is thereby: $\&X[0]$, n, the return value and the program counter. lb, ub and k are n'-reliable. This results in a $\mathcal{O}(n')$ time overhead.

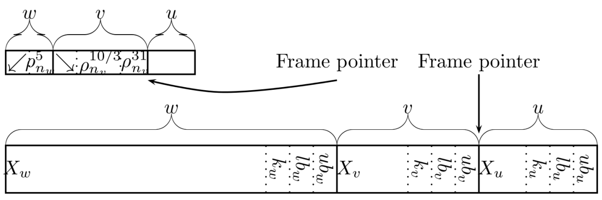


Figure 45: pointer and direction are stored in safe memory

```
Algorithm 2: Deterministic-Select($X$, $n$, $k$, $lb$, $ub$)
1  # The algorithm uses the recursion implementation from Lemma 3
2  repeat
3      # Let $f$ denote $\lfloor \frac{3n}{10} \rfloor - \lfloor \frac{n}{11} \rfloor - 6$
4      begin First Phase
5          repeat
6              $X_m \leftarrow []$
7              for $i \in [1..\lceil n/5 \rceil]$ do
8                  $X_m[i] \leftarrow$ median of $X[5i, 5i + min(4, n - 5i)]$
9              $x_p \leftarrow$ Deterministic-Select($X_m$, $\lceil |X_m|/2 \rceil$, $lb$, $ub$)
10             partition $X$ around $x_p$ # using the algorithm from Lemma 2
11             # Let $p$ denote $rank^{c}_{X}(x_p)$
12         until $p \in [f, n - f]$
13     begin Second Phase
14         if $p = k$ then
15             return $e = min(max(x_p, lb), ub)$
16         else if $p > k$ then
17             $e \leftarrow$ Deterministic-Select($X[1, n - f]$, $k$, $lb$, $x_p$)
18         else if $p < k$ then
19             $e \leftarrow$ Deterministic-Select($X[f, n]$, $k - f$, $x_p$, $ub$)
20 until $rank^{c}_{X}(e) \in [k \pm n_v]$ # $v$ is a second type child of the node
21 return $e = min(max(e, lb), ub)$
```

Figure 46: Pseudocode for median of medians

We repeat until the pivot is "good" ($\in [\approx \frac{3n}{10}, \approx \frac{7n}{10}]$) and the element is "good" ($\in [k_v \pm n_u]$).
We have the three variables k_w, lb_w and ub_w (the current k, lb and ub). These we call auxiliary variables. n' is the size of the subarray considered by the child.
We distinguish two cases: First case is when our auxiliary variables are non-faulty (majority gives correct value). There is a "1 to 1" correspondence between any corrupted cell (in the child's array), to a "shift" in the element returned from it. So each deviation of "one" from the correct answer is caused by "one" corruption.
The second case is where some of the auxiliary things are corrupted (majority gives the wrong value). Therefore, we must have at least n' corruptions that corrupted at least n' of these copies in order to get a faulty auxiliary variable. So $\alpha' \geq n'$, where α is the number of corruptions during the computation of this one child.
Finally, this means that, after we have passed the verification step, then we had at least as many corruptions as we are slightly "off" the correct value, and therefore correctness follows.
For the complexity we get, when there are no faults there are no repetitions, then we have $\mathcal{O}(n)$. Pivot repetitions cost us $\mathcal{O}(n')$ and $\Omega(\frac{n'}{33}) = \Omega(n')$ faults, so in the end $\mathcal{O}^*(1)$. Element repetitions cost us $\mathcal{O}(n')$ extra work and $\Omega(\frac{7n'}{10}) = \Omega(n')$ faults, so in the end $\mathcal{O}^*(1)$. The faults are disjoint so this results in $T(n) = \mathcal{O}(n + \alpha)$.
An improvement idea would be to maintain a counter which is a lower bound on the number of faults encountered and then halt when the counter is too big. The counter is initialised to 0 and incremented

by one on each repetition. We halt when the counter is $> n$ and return 42. We then get $T(n) = \mathcal{O}(n)$.

14.7 Resilient k-dimensional Trees

We look here at a data structure that partitions n points in a k-dimensional space. It supports orthogonal range queries in $\mathcal{O}(n^{1-\frac{1}{k}} + t)$ time. It is used in practice to compute resilient k-means in space. k for order statistics is not equal to k for dimensions.
We start with a full binary tree which root holds all n points. Using round robin we split by the (depth $\bmod k$)-median. Using the median of medians the construction time is $\mathcal{O}(n \log n)$. Resiliently we store $2\delta + 1$ copies for the inner nodes. The leaves hold $\mathcal{O}(\delta)$ points in an array. We store the nodes in order of a BFS in the memory, so we have no pointers. The deterministic construction time using resilient sorting is $\mathcal{O}(n(\log n)^2 + \alpha\delta)$. Using resilient selection we get $\mathcal{O}(n \log n)$.

14.8 Resilient Randomised In-place Sorting

We recall quicksort. We pick a random pivot x_p and partition X by x_p and recursively sort both parts. Then we merge both parts. This is correct and has complexity ($\mathbb{E}[T(n)] = \mathcal{O}(n \log n)$). The problem is again, that we have pointer to $\omega(1)$ partition locations that can get corrupted. We solve that by splitting at the exact median instead of at $\frac{n}{2} \pm \alpha$ and go iteratively from top to bottom and from left to right.
So we have a look a resilient splitting algorithms.

Definition 14.8 (Splitter) Partitions the array such that the uncorrupted elements in $[1 \dots k]$ are smaller then the uncorrupted elements in $[k \dots n]$. $\dashv$

Theorem 14.9 (Sandboxed Splitting Algorithms) There exists a [deterministic/randomised and in-place] resilient splitting algorithm with [worst-case/expected] time complexity $\mathcal{O}(\alpha n)$.

Proof 14.10 We take a non-destructive, non-resilient algorithm as input. Then we start it in a memory sandbox (boundaries) and a time sandbox (counter). We use a resilient verification procedure. We need at most $\alpha + 1$ iterations. General sandboxing $\rightarrow$ atomic swaps (or input duplication). $\square$

Theorem 14.11 (Efficient Splitting Algorithms) There exists a [deterministic/randomised and in-place] resilient splitting algorithm with [worst-case/expected] time complexity $\mathcal{O}(n + \alpha\delta)$.

Proof 14.12 We resilient select the $(k - \delta)$ order statistic and partition by it. Then we resilient select the $(k + \delta)$ order statistic and partition by it. Then we sandbox-split the remaining $\mathcal{O}(\delta)$ elements and take the lower bound. $\square$

Theorem 14.13 (Resilient Quicksort Algorithms) There exists a [deterministic/randomised and in-place] resilient splitting algorithm with [worst-case/expected] time complexity $\mathcal{O}(n + \alpha\delta)$.

Proof 14.14 Correctness (uncorrupted subset is sorted): For every a, b with $a < b$ there is a pivot element such that a is in the left and b in the right partition.
The complexity is like that because of disjoint faults. $\square$

14.9 Summary

The world is not perfect but we still want to compute correctly. We like deterministic algorithms with worst-case guarantees against any adversary. One nice model for this is the FRAM model. You can search, select, and sort with no real (asymptotic) time and space cost.
In the future we want to achieve δ-obliviousness and resilient graph algorithms.

14.10 Exercises

- Majority
  1. How to compute majority (non-resiliently)?
  2. What can go wrong in FRAM?
  3. (BONUS) How to compute majority (resiliently)? (hint: run over it and store only the most frequent element)
- Quicksort
  1. Assume you have resilient splitter, which, given an array and an element, split the array into two parts, one part with elements smaller than the given element and the other part with elements larger than the given element.
     What can go wrong in FRAM for quicksort?
  2. How to implement quicksort (resiliently)? (hint: implement recursion with iterations)