ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Computational Thinking

Roger Wattenhofer

wattenhofer@ethz.ch

Fall 2022

Contents

Introduction

Computation is everywhere, but what is computation really? In this lecture we will discuss the power and limitations of computation.

Understanding computation lies at the heart of many exciting scientific and social developments. Computational thinking is more than programming a computer – rather it is thinking in abstractions. Consequently, computational thinking has become a fundamental skill for everyone, not just computer scientists. For example, designing an electronic circuit relates directly to computation. Mathematical functions which can easily be computed but not inverted are at the heart of understanding data security and privacy. Machine learning on the other hand has given us fascinating new tools to teach machines how to learn function parameters. Thanks to clever heuristics, machines now appear to be capable of solving complex cognitive tasks. In this class, we study all these and more problems together with the fundamental theory of computation.

While computation is predominantly an engineering discipline, some of our insights are going to be philosophical. One may even claim that this class is going to discuss all major connections between computation and philosophy.

The weekly lectures will be based on blackboard discussions and programming demos, supported by a script and coding examples. The course uses Python as a programming language. Python is popular and intuitive, a programming language that looks and feels like human instructions. The lecture will feature weekly exercises, paper and Python.

→ notebook

Have fun!

Chapter 1

Algorithms

The term "computer" used to be a job description for a person doing the same tedious computations over and over, hopefully without error. When electrical computers became available, these human computers often transitioned to become computer programmers. Instead of doing the computations themselves, they told the computer what to do.

Definition 1.1 (Algorithm)**.** *An **algorithm** is a sequence of computational instructions that solves a class of problems. Often the algorithm computes an output for a given input, i.e., a mathematical function.*

Remarks:

- While the number of algorithms is theoretically unlimited, surprisingly many problems can be solved with just a few algorithmic paradigms that we will review in this chapter. A simple yet powerful algorithmic concept is recursion. Let us start with an example.

1.1 Recursion

You have won an *all-you-can-carry* run through an electronics store. The rules are simple: Whatever you manage to carry, you can have for free. Being well-prepared you bring a high-capacity backpack to the event. Which items should you put into your backpack such that you can carry the maximum possible value out of the store?

Problem 1.2 (Knapsack)**.** *An item is an object that has a* `name`*, a* `weight` *and a* `value`*. Given a list of* `items` *and a knapsack with a weight* `capacity`*, what is the maximal value that can be packed into the knapsack?* → notebook

Remarks:

- An algorithm solving Knapsack computes a function; the inputs of this function are the set of possible `items` and the `capacity` limit of the knapsack, the output is the maximal possible `value`.

- A simple way to solve Knapsack is to check for every `item` whether it should be packed into the knapsack or not, expressed as the following recursion:

→ notebook

```python
1 def knapsack(items, capacity):
2   if len(items) == 0:
3     return 0
4   first, *rest = items
5   take = 0
6   if first.weight <= capacity:
7     take = knapsack(rest, capacity-first.weight) + first.value
8   skip = knapsack(rest, capacity)
9   return max(take, skip)
```

Algorithm 1.3: A recursive solution to Knapsack.

Remarks:

- Algorithm 1.3 may look like pseudo-code, but really is correct Python.
- In Lines 7 and 8, the algorithm calls itself. This is called a recursion.

Definition 1.4 (Recursion)**.** *An algorithm that splits up a problem into sub-problems and invokes itself on the sub-problem is called a* ***recursive algorithm****. A* ***recursion*** *ends when reaching a simple base case that can be solved directly. Also, see Definition 1.4.*

Remarks:

- In mathematics, we find a similar structure in some prominent inductive functions such as the Fibonacci function.
- Recursive algorithms are often easy to comprehend, but not necessarily fast.
- How can we measure "fast"?

Definition 1.5 (Time Complexity)**.** *The* ***time complexity*** *of an algorithm is the number of basic arithmetic operations ($+$, $-$, $\times$, $\div$, etc.) performed by the algorithm with respect to the size n of the given input.*

Remarks:

- Each variable assignment, `if` statement, iteration of a `for` loop, comparison (`==`, `<`, `>`, etc.) or `return` statement also counts as one basic arithmetic operation, and so do function calls (`len()`, `max()`, `knapsack()`).
- Unfortunately, there is no agreement on how the size of the input should be measured. Often the input size n is the number of input items. If input items get large themselves (e.g., the input may be a single but huge number), n refers to the number of bits needed to represent the input.

- We are usually satisfied if we know an approximate and asymptotic time complexity. The time complexity should be a simple function of n, just expressing the biggest term as n goes to infinity, ignoring constant factors. Such an asymptotic time complexity can be expressed by the "big O" notation.

Definition 1.6 ($\mathcal{O}$-notation)**.** *The $\mathcal{O}$-**notation** is used to denote a set of functions with similar asymptotic growth. More precisely,*

$$\mathcal{O}(f(n)) = \left\{ g(n) \,\middle|\, \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty \right\}.$$

Remarks:

- In other words, $\mathcal{O}(f(n))$ is the set of functions $g(n)$ that asymptotically do not grow much faster than $f(n)$.

- For example, $\mathcal{O}(1)$ includes all constants and $\mathcal{O}(n)$ means "linear in the input size n".

- In other words, the $\mathcal{O}$-notation is quite crude, but nevertheless useful, both in theory and practice.

- Other useful asymptotic notations are $\Omega()$ for lower bounds, but also $o()$, $\omega()$, $\Theta()$, etc.

Lemma 1.7. *The time complexity of Algorithm 1.3 is $\mathcal{O}(2^n)$.*

Proof. Each call of the `knapsack()`-procedure performs constantly many basic arithmetic operations itself and makes (at most) two additional calls to the `knapsack()`-procedure. Hence, it suffices to count the total number of `knapsack()`-invocations. We get 1 invocation on the first item, at most 2 on the second, 4 on the third, ..., and 2^{n-1} on the last. Hence, there are less than 2^n invocations of the `knapsack()`-function. □

Remarks:

- The time complexity of Algorithm 1.3 is exponential in the number of items. Even if there were only $n = 100$ items to be evaluated, the currently fastest supercomputer in the world would take 2^{100} ops$/(148 \cdot 10^{15}$ ops/s$) \approx 271\,000$ years to compute our `knapsack` function. So for many realistic inputs, Algorithm 1.3 is not usable. We need a better approach!

1.2 Greedy

What about sorting all the items by their `value`-to-`weight` ratio, and then simply greedily packing them!?

→ notebook

```
1 def knapsack(items, capacity):
2   items.sort(key=lambda item : -item.value/item.weight)
3   value = 0
4   for item in items:
5     if item.weight <= capacity:
6       capacity -= item.weight
7       value += item.value
8   return value
```

Algorithm 1.8: A naive greedy algorithm for Knapsack.

Remarks:

- Algorithm 1.8 is fast, with a time complexity of $\mathcal{O}(n \log n)$, just for calling the sorting function in Line 2. So a large input is no problem.
- Also, the output of Algorithm 1.8 often seems reasonable. However, Algorithm 1.8 does not solve Knapsack optimally. For example, assume a capacity 6 knapsack, two items each with value 3 and weight 3, and one higher-ratio item with value 5 and weight 4.
- Can we gain a speed-up from first sorting the elements?

1.3 Backtracking

Definition 1.9 (Backtracking)**.** *A **backtracking** algorithm solves a computational problem by constructing a candidate solution incrementally, until either a solution or a contradiction is reached. In case of a contradiction, the algorithm "backtracks" (i.e. reverts) its last steps to a state where another solution is still viable. Efficient backtracking algorithms have two main ingredients:*

- ***Look-ahead:*** *We order the search space such that the most relevant solutions come up first.*
- ***Pruning:*** *We identify sub-optimal paths early, allowing to discard parts of the search space without explicitly checking.*

Remarks:

- Algorithm 1.3 was an inefficient backtracking algorithm.
- Our look-ahead idea is to sort the items by `value-to-weight` ratio as in Algorithm 1.8.
- The algorithm prunes the solution space if it cannot possibly achieve the best solution so far.

→ notebook

```
def knapsack(items, capacity):
  items.sort(key=lambda item: -item.value/item.weight)
  return bt(items, capacity, 0)

def bt(items, capacity, missing):
  if len(items) == 0:
    return 0
  first, *rest = items
  if first.value / first.weight * capacity < missing:
    return 0  # branch is worse than the best previous solution
  take = 0
  if first.weight <= capacity:
    take = bt(rest, capacity-first.weight, missing-first.value)
    take += first.value
  skip = bt(rest, capacity, max(take, missing))
  return max(take, skip)
```

Algorithm 1.10: An efficient backtracking solution to Knapsack.

Remarks:

- The `missing` parameter is the additional value that is required to surpass the previously best solution.
- The time complexity of Algorithm 1.10 is still $\mathcal{O}(2^n)$ in the worst case. Can we do better?

1.4 Dynamic Programming

Definition 1.11 (Dynamic Programming)**.** ***Dynamic programming (DP)*** *is a technique to reduce the time complexity of an algorithm by utilizing extra memory. To that end, a problem is divided into sub-problems that can be optimized independently. Intermediate results are stored to avoid duplicate computations.*

Remarks:

- Knapsack can be solved with dynamic programming. To that end, we store a value matrix `V` where `V[i][c]` is the maximum value that can be achieved with capacity `c` using only the first `i` items.

→ notebook

```
def knapsack(items, capacity):
  n = len(items)
  V = zero matrix of size (n+1)×(capacity+1)
  for item i in items:
```

```
5     for c in range(capacity+1):
6       V[i+1][c] = max(V[i][c-item.weight] + item.value, V[i][c])
7   return V[n][capacity]
```

Algorithm 1.12: A dynamic programming solution to Knapsack.

Remarks:

- Note that Algorithm 1.12 is not correct Python. Line 3 is just pseudocode, far from actual Python notation. Line 4 could be Python, but unfortunately needs an extra `enumerate()` function.
- Line 6 is incorrect: If `item.weight > c`, `c-item.weight` becomes negative. The programmer of Algorithm 1.12 assumed that accessing a negative index of an array returns 0; however, most programming languages return an error. We can fix Line 6 by adding the conditional expression `if c >= item.weight else 0` to the first term of the `max()` function.
- The time complexity of Algorithm 1.12 is $\mathcal{O}(n \cdot \texttt{capacity})$. In Definition 1.5 we postulated that the time complexity should be a function of n. So the DP approach only makes sense when `capacity` is a natural number with $\texttt{capacity} < 2^n/n$. .

Definition 1.13 (Space Complexity)**.** *The **space complexity** of an algorithm is the amount of memory required by the algorithm, with respect to the size n of the given input.*

Remarks:

- As for Definition 1.5, we are usually satisfied if we know the approximate (asymptotic) space complexity.
- Also, the amount of memory can be measured in bits or memory cells.
- The space complexity of Algorithm 1.12 is $\mathcal{O}(n \cdot \texttt{capacity})$.
- For reasonably small `capacity`, Algorithm 1.12 is faster than Algorithms 1.3–1.10, but is it correct?

Lemma 1.14. *Assuming that all* `items` *have integer* `weights`*, Algorithm 1.12 solves Knapsack correctly.*

Proof. We show the correctness of each entry in the matrix `V` by induction. As a base case, we have `V[0] = [0, ..., 0]` since without item, no value larger than 0 can be achieved. For the induction step, assume that `V[i]` correctly contains the maximum values that can be achieved using only the first `i` items. When we set a value `V[i+1][c]`, we can either include the item `i+1` or select the optimal solution for Knapsack with capacity using only the first `i` items. Algorithm 1.12 stores the `max()` of these two values in `V[i+1][c]` (for all $\texttt{c} \in \{0, \ldots, \texttt{capacity}\}$), which is optimal.

Hence, the value `V[n][capacity]` contains the maximum value that can be achieved with the weight `capacity`, using any combination of the n items. □

Remarks:

- Line 6 of Algorithm 1.12 is typical for dynamic programming algorithms: either the previous best solution can be improved, or it remains unchanged. This is called Bellman's principle of optimality.

- The computation order of Algorithm 1.12 is important. For example, we can only compute the entry `V[i+1][c]` once we have computed both `V[i][c-item.weight]` and `V[i][c]`.

- The sub-problem dependencies can be visualized as a dependency graph. In order to apply dynamic programming, this graph must be a directed acyclic graph (DAG).

- Algorithm 1.12 is a so-called *bottom-up* dynamic programming algorithm as it begins computing the entries of matrix `V` starting with the simple cases.

- But do we really need to compute the entire matrix `V`?

Definition 1.15 (Memoization)**.** ***Memoization*** *generally refers to a technique that avoids duplicate computations by storing intermediate results.*

→ notebook

```
 1  def knapsack(items, capacity, memo={}):
 2    index = (len(items), capacity)
 3    if index in memo:
 4      return memo[index]
 5    if len(items) == 0:
 6      return 0
 7    first, *rest = items
 8    take = 0
 9    if first.weight <= capacity:
10      take = knapsack(rest, capacity-first.weight, memo)
11      take += first.value
12    skip = knapsack(rest, capacity, memo)
13    memo[index] = max(take, skip)
14    return memo[index]
```

Algorithm 1.16: A top-down DP solution to Knapsack.

Remarks:

- Memoization can be used to implement *top-down* DP algorithms.

- This is not so different from our initial Algorithm 1.3!

- We only changed Line 1 and added Line 2 to set up memoization, which is then used in Lines 3–4 and 13–14.

- Top-down DP is inheriting the best of recursion and bottom-up DP. Consequentially, the time complexity of Algorithm 1.16 is

$$\mathcal{O}(\min(2^n, n \cdot \texttt{capacity}))\,.$$

- So far we have learned a family of related algorithmic techniques: recursion, backtracking, dynamic programming, and memoization. Together, this family can help solving many demanding algorithmic problems.

- However, there are powerful algorithmic paradigms beyond this family of techniques, for instance linear programming.

1.5 Linear Programming

So far, we were only considering unsplittable items. However, for liquid goods, Knapsack can be solved quickly using a greedy method (Algorithm 1.8). What if we had more than one constraint?

Problem 1.17 (Liquid Knapsack)**.** *A* `beverage` *has a* `name`*, a* `value` *per liter and a preparation* `time` *per liter. Given t hours to prepare for a party and a fridge with a storage* `capacity`*, what is the maximal value that can be prepared and stored in the fridge?* → notebook

Remarks:

- With more than one constraint, the greedy method does not work.

- However, this problem has a nice property: the objective and the constraints are linear functions of the quantity of each prepared beverage. We call such problems *linear programs.*

Definition 1.18 (Linear Program or LP)**.** *A* ***linear program (LP)*** *is an optimization problem with n variables and m linear inequalities* → notebook

$$\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & \leq & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & \leq & b_2 \\
\vdots & & \vdots & & & & \vdots & & \vdots \\
a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & \leq & b_m
\end{array}$$

We are interested in finding a point $\boldsymbol{x} = (x_1, \dots, x_n)^T$ with $x_i \geq 0$, respecting all these constraints, and maximizing a linear function

$$f(\boldsymbol{x}) = c_1x_1 + c_2x_2 + \cdots + c_nx_n$$

where a_{ij}, b_i and c_i are given real-valued parameters. We call the point $\boldsymbol{x}$ an ***optimum*** *of the LP.*

Remarks:

- There is also a short hand notation using linear algebra

  $$\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \mathbf{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0\},$$

  where $\mathbf{A}$ is the matrix with entries a_{ij} and $\boldsymbol{b}$ and $\boldsymbol{c}$ the vectors given by the b_i and c_i, respectively.

- In general, if you have the problem of maximizing or minimizing a linear function under constraints that are linear (in)equalities, there is a way to formulate it in above canonical form. For instance, a constraint $\boldsymbol{a}^T\boldsymbol{x} = b$ can be rewritten as a combination of $\boldsymbol{a}^T\boldsymbol{x} \leq b$ and $\boldsymbol{a}^T\boldsymbol{x} \geq b$ which itself can be rewritten as $-\boldsymbol{a}^T\boldsymbol{x} \leq -b$. Also, minimizing a linear function with coefficients $c_1, \dots, c_n$ is the same as maximizing a linear function with coefficients $-c_1, \dots, -c_n$.

- It is possible to model some functions which do not look linear at first sight. For example, minimizing an objective function $f(x) = |x|$ can be expressed as $\min\{t | x \leq t, -x \leq t\}$.

Definition 1.19 (Feasible Point)**.** *Given an LP, a point is* ***feasible*** *if it is a solution of the set of constraints.*

Remarks:

- Geometrically, the set of feasible points of an LP corresponds to an n-dimensional convex polytope. The hyperplanes bounding the polytope are given by the restricting inequalities.

- Polytopes are a generalization of 2D polygons to an arbitrary number of dimensions. Convexity, however, deserves a more formal definition.

Definition 1.20 (Convex Set)**.** *A set of points in $\mathbb{R}^n$ is* ***convex*** *if for any two points of the set, the line segment joining them is also entirely included in the set.*

Lemma 1.21. *The set of feasible points of an LP is convex.*

Proof. Given two feasible points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, any point in the line segment joining them can be written as $\boldsymbol{x}_1+\lambda(\boldsymbol{x}_2-\boldsymbol{x}_1)$ for $\lambda \in [0,1]$. For any constraint $\boldsymbol{a}^T\boldsymbol{x} \leq b$, we compute

$$\boldsymbol{a}^T[\boldsymbol{x}_1 + \lambda(\boldsymbol{x}_2 - \boldsymbol{x}_1)] = (1-\lambda)\boldsymbol{a}^T\boldsymbol{x}_1 + \lambda\boldsymbol{a}^T\boldsymbol{x}_{\boldsymbol{2}} \leq (1-\lambda)b + \lambda b = b.$$

□

Definition 1.22. *Given an LP, we call* ***polytope*** *the set of feasible points. A constraint $\boldsymbol{a}^T\boldsymbol{x} \leq b$ is* ***tight*** *at $\boldsymbol{x}$ if $\boldsymbol{a}^T\boldsymbol{x} = b$. For an LP with n variables, feasible points activating n (resp. $n-1$) linearly independent constraints are called the* ***nodes*** *(resp.* ***edges****) of the polytope. Each edge links two nodes $\boldsymbol{x}_{\boldsymbol{1}}, \boldsymbol{x}_{\boldsymbol{2}}$ with $n-1$ common activating constraints; we say that the two nodes $\boldsymbol{x}_{\boldsymbol{1}}, \boldsymbol{x}_{\boldsymbol{2}}$ are* ***neighbors****.* → notebook

Remarks:

- A polytope can be *unbounded*, i.e. infinitely large. If the convex polytope is unbounded, it is often rather called a convex polyhedron. In some cases, it is even possible to have an infinitely large solution, e.g., $\max\{x|x \geq 0\}$. Following our definition, the LP does not admit an optimum in this case.

- In order to solve an LP, one has to find a point in the polytope that maximizes our objective function $f(\boldsymbol{x})$.

Theorem 1.23. *If the polytope of an LP is bounded, then at least one node of the polytope is an optimum of the LP.*

→ notebook

Proof. For any value y that the objective function can take, the set of points reaching this value is given by the hyperplane $\boldsymbol{c}^T\boldsymbol{x} = y$. We can find an optimum of the LP by sliding this hyperplane until the boundary of the polytope is reached, which happens at some node of the polytope. □

Remarks:

- One popular method exploiting Theorem 1.23 for solving LPs is the simplex algorithm. The idea is simple: starting from a node of the LP polytope, greedily jump to a neighboring node having a better objective until you cannot improve the solution anymore.

```python
def simplex(polytope, f, x):
  for y in neighbors(x, polytope):
    if f(y) > f(x):
      return simplex(polytope, f, y)
  return x
```

→ notebook

Algorithm 1.24: Simplex Algorithm.

Remarks:

- While the simplex algorithm performs well in practice, there are instances where its time complexity is exponential in the size of the input. Other LP algorithms known as interior point methods are provably fast.

- In practice, we do not build and store the whole polytope of the LP, as the polytope could have an exponential number of nodes! Instead, we represent a node as a set of tight constraints. To find its neighbors, we remove a constraint of the set, add another constraint and check if the point is feasible.

- The node returned by the simplex algorithm is better than any neighboring node by construction, but how can we convince ourselves that no other point anywhere in the feasible polytope is better?

Definition 1.25 (Local Optimum)**.** *A feasible node $\boldsymbol{x}$ is a **local optimum** if $f(\boldsymbol{x}) \geq f(\boldsymbol{y})$ for any neighboring node $\boldsymbol{y}$.*

Remarks:

- In contrast to a local optimum, an optimum from Definition 1.18 is called *global optimum.*
- While it is easy to find a local optimum, finding a global optimum is often difficult. However, it turns out that every local optimum of an LP is also a global optimum!

Theorem 1.26. *The node $\boldsymbol{x}^*$ returned by the simplex algorithm is an optimum.*

Proof. Let us consider the hyperplane $\boldsymbol{c}^T\boldsymbol{x} = f^*$, where $f^* = \boldsymbol{c}^T\boldsymbol{x}^*$. We know that all the neighbors of node $\boldsymbol{x}^*$ are on the side $\boldsymbol{c}^T\boldsymbol{x} \leq f^*$. Since the polytope is convex, we know that the whole polytope must be on this side of the hyperplane. Hence no node x' in the polytope can be on the side $\boldsymbol{c}^T\boldsymbol{x} > f^*$, and hence the node $\boldsymbol{x}^*$ is a global optimum. □

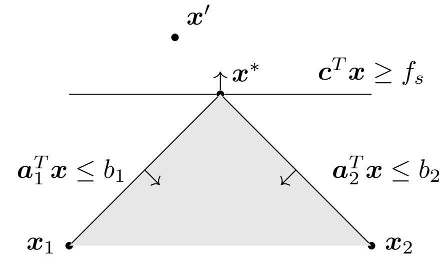

$\boldsymbol{x}'$
$\boldsymbol{x}^*$
$\boldsymbol{c}^T\boldsymbol{x} \geq f_s$
$\boldsymbol{a}_1^T\boldsymbol{x} \leq b_1$
$\boldsymbol{a}_2^T\boldsymbol{x} \leq b_2$
$\boldsymbol{x}_1$
$\boldsymbol{x}_2$

Figure 1.27: Illustration of Theorem 1.26. The neighbors of $\boldsymbol{x}^*$ are x_1 and x_2.

Remarks:

- So we have seen that every local optimum of an LP is also a global optimum. This important property in optimization is true for convex functions in general, and as such LPs are only a special case of convex optimization.
- We call Algorithm 1.24 with $\boldsymbol{x}$ being any node of the polytope. But wait, how do we find such a start node?! It turns out that we can construct an auxiliary LP:

Definition 1.28 (Phase 1 LP)**.** *Given an LP*

$$\max\{\boldsymbol{c}^T\boldsymbol{x} \mid \mathbf{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0\},$$

*we build the so-called **phase 1 LP** by replacing every constraint $\boldsymbol{a}_i^T\boldsymbol{x} \leq b_i$ with $\boldsymbol{a}_i^T\boldsymbol{x} - y_i \leq b_i$, introducing a new artificial variable y_i. If we minimize all artificial variables y_i, we get:*

$$\max\{-\mathbf{1}^T\boldsymbol{y} \mid \mathbf{A}\boldsymbol{x} - \mathbf{I}\boldsymbol{y} \leq \boldsymbol{b}, \boldsymbol{x} \geq 0, \boldsymbol{y} \geq 0\}.$$

Lemma 1.29. *Setting each $x_i = 0$ and each $y_i = \max(0, -b_i)$ yields a feasible node of the phase 1 LP.*

Proof. With each original variable $x_i = 0$, each constraint is reduced to $-y_i \leq b_i$, which is satisfied when $y_i = \max(0, -b_i)$.

Also, this point is a node of the polytope: Algebraically, a point is a node if at least n linearly independent constraints are tight at this point. The constraint $x_i \geq 0$ is tight for each original variable x_i and either $\boldsymbol{a}_i\boldsymbol{x} - y_i \leq b_i$ or $y_i \geq 0$ is tight for each artificial variable y_i, depending on the sign of b_i. Thus, the number of tight constraints is at least equal to the number of variables, and this point is a node of the polytope. □

Lemma 1.30. *If the original LP is feasible, then the phase 1 LP will find a feasible node.*

Proof. If the original LP is feasible, then its polytope is not empty, i.e., there exists a feasible node $\boldsymbol{x}$ in the original LP. Together with $\boldsymbol{y} = 0$, node $\boldsymbol{x}$ is also feasible in the phase 1 LP. Since $\max\{-\mathbf{1}^T\boldsymbol{y}\} = \min\{\texttt{sum}(\boldsymbol{y})\}$ is optimal for $\boldsymbol{y} = 0$, node $\boldsymbol{x}$ is optimal in the phase 1 LP. With Theorem 1.26, we know that the phase 1 LP will find such a node $\boldsymbol{x}$. □

Remarks:

- Algorithm 1.31 is the complete procedure to solve an LP. This process is often called the *two-phase simplex algorithm.*
- In Python, one can solve an LP using the function `linprog` from the module `scipy.optimize`. → notebook

→ notebook

```
def solveLP(A, b, c):
  x,y = simplex(polytope([A -I], b), −1, (0, max(0,−b)))
  if sum(y) == 0:
    return simplex(polytope(A, b), c, x)
  else:
    return 'no solution'
```

Algorithm 1.31: Two-phase simplex algorithm to solve LPs.

1.6 Linear Relaxation

Linear programming is covering a broad class of problems, but we are often confronted with *discrete* tasks, for which we need an integer solution.

Definition 1.32 (Integer Linear Programming or ILP)**.** *An* ***integer linear program (ILP)*** *is an LP in which all variables are restricted to integers.*

Remarks:

- In a lot of combinatorial problems, variables are restricted to just two values $\{0, 1\}$. Such variables are called *indicator* ("to be or not to be") variables. We call such programs *binary ILPs.*
- Apart from LP and ILP, there exist many other optimization techniques: Mixed Integer Linear Programming (MILP) with both integer and continuous variables, Quadratic Programming (QP), Semidefinite Programming (SDP), ...

Problem 1.33 (ILP Knapsack)**.** *We can model Knapsack (Problem 1.2) with capacity c and n items of value v_i and weight w_i as a binary ILP, using indicator variables x_i:*

$$\begin{aligned} \textit{maximize} \quad & \sum v_i \; x_i \\ \textit{subject to:} \quad & \sum w_i \; x_i \le c \\ & x_i \in \{0,1\}. \end{aligned}$$

Remarks:

- Unlike LPs, no efficient algorithm solving ILPs is known.
- It is tempting to relax the constraints $x_i \in \{0, 1\}$ to $0 \le x_i \le 1$, apply the simplex algorithm, and round the possible solution to the nearest feasible point.

Definition 1.34 (Linear Relaxation)**.** *Given a binary ILP, we construct the* ***linear relaxation*** *of the LP by replacing the constraint $\boldsymbol{x} \in \{0, 1\}^n$ with the constraint $0 \le x_i \le 1$.*

Remarks:

- However, in general, there is no guarantee that a linear relaxation finds the optimum.
- In the case of Knapsack, the solution of the linear relaxation is similar to Algorithm 1.8. All items i with a high value-to-weight ratio will get an indicator variable $x_i = 1$, all items with a low value-to-weight ratio will get an indicator variable $x_i = 0$. The critical item(s) in the middle will get a non-integer indicator variable which we must round down to 0 to get a valid solution. This solution can be arbitrarily bad, as the best (highest value-to-weight ratio) item might already be too heavy; we might end up without any object in the knapsack.
- However, a linear relaxation sometimes has the same optimum as its ILP. In particular, this is true for some classes of constraint matrices, e.g., totally unimodular matrices.
- A matrix is totally unimodular if every square submatrix has determinant -1, 0 or $+1$. This is a non trivial property to check. For a certain class of problems we know that the constraint matrices are always totally unimodular.

Problem 1.35 (Assignment Problem). *Given a list of customers and a list of cabs, how to match customers to cabs in order to minimize the total waiting time?* → notebook

Algorithm 1.36. *This problem can be modeled as an ILP. We denote the waiting time of customer i for cab j by $w_{i,j}$. Also, we introduce a set of indicator variables $x_{i,j}$ describing the assignment: $x_{i,j} = 1$ if and only if customer i is assigned to cab j. We get:* → notebook

$$\begin{array}{lllll} \textit{minimize} & \sum_{i,j} & x_{i,j}w_{i,j} & & \\ \textit{subject to:} & \sum_{j} & x_{i,j} & = 1 & \textit{for each customer } i \\ & \sum_{i} & x_{i,j} & \leq 1 & \textit{for each cab } j \\ & & x_{i,j} & \in \{0,1\} & \end{array}$$

This ILP can be solved optimally with linear relaxation: the constraint matrix is totally unimodular.

1.7 Flows

Graphs and flows are useful algorithmic concepts, related to LPs and linear relaxations.

Definition 1.37 (Graph). *A graph G is a pair (V, E), where V is a set of* ***nodes*** *and $E \subseteq V \times V$ is a set of* ***edges*** *between the nodes. The number of nodes is denoted by n and the number of edges by m.*

Remarks:

- A *directed* graph $G = (V, E)$ is a graph, where each edge has a direction, i.e., we distinguish between edges (u, v) and (v, u). If all edges of a graph are undirected, then the graph is called *undirected.*
- In a directed graph, we note $\texttt{in}(u)$ (resp. $\texttt{out}(u)$) the set of edges entering (resp. leaving) node u.
- A *weighted* graph $G = (V, E, \omega)$ is a graph, where $\omega : E \to \mathbb{R}$ assigns a weight $\omega(e)$ for each edge $e \in E$.
- Weights can for instance be used for delay $d(e)$ or capacity $c(e)$ of an edge.
- In the rest of this chapter, we consider capacitated directed graphs.
- Consider a company that wants to optimize the flow of goods in a transportation network from their factory to a customer.

Definition 1.38 (Flow). *Formally, an* ***s-t-flow*** *from a source node s to a target node t is given as a function $f : E \to \mathbb{R}_{\geq 0}$ such that*

$$f(u,v) \leq c(u,v) \qquad \textit{for all } (u,v) \in E \qquad \textit{(capacity constraints)}$$

$$\sum_{e \in \textbf{\textit{in}}(u)} f(e) = \sum_{e \in \textbf{\textit{out}}(u)} f(e) \;\textit{for all } u \in V \setminus \{s,t\} \;\textit{(flow conservation)}$$

We call the total flow reaching t the ***value*** *of f, i.e. $|f| = \sum_{(u,t)\in E} f(u,t)$.*

Problem 1.39 (Max-Flow)**.** *What is the maximum flow that can be established between a* `source` *and a* `target` *node in a network?*

Remarks:

- Max-Flow can be written as an LP maximizing the value of the flow.
- Flows are also useful to model discrete (integral) data. Imagine traffic flow for example: every road as some capacity of cars and at each intersection, and every whole car getting in is expected to eventually get out!
- Fortunately, we can use the linear relaxation of the ILP and be guaranteed to have the optimal solution!

Theorem 1.40 (Integral Flow Theorem)**.** *If the capacity of each edge is an integer, then there exists a maximum flow such that every edge has an integral flow.*

Proof. Assume you have an optimal but non-integral flow. If there is a path from s to t with every edge being non-integral, we can increase the flow on that path, so our original flow was not optimal. Hence, there cannot be a non-integral path from s to t.

Let u be a node adjacent to an edge e with non-integral flow. Then u needs at least another edge e' with non-integral flow because of flow conservation at node u. We can follow these non-integral edges. Since they cannot include both s and t, we must find a cycle C of non-integral edges. All edges in C can both change their flow by $\pm\varepsilon$, without changing the flow from s to t. We change the flow of all edges in C until a first edge in C has integral flow. Now we have one edge less with non-integral flow. If there is still an edge with non-integral flow, we repeat this procedure, until all edges have integral flow.

□

Remarks:

- Thanks to Theorem 1.40, we can solve a discrete maximum flow problem with the linear relaxation of the ILP formulation and the simplex algorithm!
- There are also more efficient algorithms, known as augmenting paths algorithms.

Lemma 1.41. *The following LP can be used to solve the flow problem:*

$$\begin{array}{lrl}
\textit{maximize} & \sum_{(u,v)} x_{(u,v)} & \textit{for each edge } (u,v) \textit{ in } \texttt{out}(s) \\
\textit{subject to:} & x_{(u,v)} > 0 & \textit{for each edge } (u,v) \in E \\
 & x_{(u,v)} \leq c(u,v) & \textit{for each edge } (u,v) \in E \\
 & \sum_{e \in \boldsymbol{in}(u)} x_e - \sum_{e \in \boldsymbol{out}(u)} x_e = 0 & \textit{for all } u \in V \setminus \{s,t\}
\end{array}$$

Proof. The flow f is represented by one variable $x_{(u,v)}$ for every directed edge $(u, v) \in E$ that indicates the value on that edge, i.e. $f(u, v) = x_{(u,v)}$. We maximize the total flow value by looking at the flow that leaves s. The first constraint ensures that the flow is non-negative, while the second enforces the capacity constraint and the third one flow conservation. □

Definition 1.42 (Augmenting Path)**.** *We define an **augmenting path** as a path from s to t such that the flow of each edge does not reach its capacity or flow can be pushed back. This is the case if the residual capacity on every edge of the path is greater than 0, where the residual capacity r of an edge is defined as:*

$$\mathit{residual}(u, v) = c(u, v) - f(u, v) + f(v, u)$$

Remarks:

- We can find an augmenting path in linear time, using a recursive algorithm!

- Instead of using the residual capacity defined above, we can also add all missing directed edges to the graph and give them capacity 0. Then, when we add flow to an edge (u, v) we decrease the flow on the reverse edge $f(v, u)$ by the same amount. In this case $c(u, v) - f(u, v) > 0$ if and only if $c(u, v) - f(u, v) + f(v, u)$ and we can use the former check for finding edges with non-zero residual capacity.

→ notebook

```
def find_augmenting_path(u, t, G, flow, visited):
  visited.insert(u)
  for v in G.neighbors(u):
    if v is not in visited and residual[u, v] > 0:
      path = find_augmenting_path(v, t, G, flow, visited)
      if len(path) > 0 or v == t:
        path.append((u, v))
        return path
  return []
```

Algorithm 1.43: Find augmenting path

Remarks:

- If the network has an augmenting path, then none of the edges of this path is at full capacity and we can add some flow on this path. This gives us a greedy algorithm: Find an augmenting path, push as much flow as possible on this path, then try again. This is known as the *Ford-Fulkerson* algorithm.

→ notebook

```
def max_flow(s, t, G):
  while there is an augmenting path:
    visited = set()
    path = find_augmenting_path(s, t, G, visited):
    flow = update(G, flow, path)
  return flow # no augmenting path anymore
```

Algorithm 1.44: Ford-Fulkerson algorithm

Remarks:

- The maximum flow is closely related to the minimum cut.

Definition 1.45 (Cut)**.** *An s-t cut is a partition of the vertices V into two sets S and $T = V\ S$, such that $s \in S$ and $t \in T$. Each valid cut has a set of edges C which point from nodes in S to nodes in T. The minimum s-t cut is cut where the edges in set C have minimum total capacity.*

Theorem 1.46 (Max-Flow Min-Cut)**.** *The maximum s-t flow (Definition 1.38) is equal to the minimum s-t cut (Definition 1.45).*

Chapter Notes

The word algorithm is derived from the name of Muhammad ibn Musa al-Khwarizmi, a Persian mathematician who lived around AD 780–850. Some algorithms are as old as civilizations. A division algorithm was already used by the Babylonians around 2500 BCE [2]. Analyzing the time efficiency of recursive algorithms can be a difficult task. An easy but powerful approach is given by the master theorem [1]. Linear programming is an old concept whose origins lie in solving logistic problems during World War 2. Back in the days, the term *programming* meant optimization, and not *coding*. Maximum flow has been studied since the 1950s, when it was formulated to study the Soviet railway system. The classic algorithm is by Ford and Fulkerson [4]. However just recently there has been progress, and Chen et al. [3] managed to solve maximum flow in pretty much linear time. This chapter was written in collaboration with Henri Devillez and Roland Schmid.

Bibliography

[1] Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, September 1980.

[2] Jean-Luc Chabert, editor. *A History of Algorithms*. Springer Berlin Heidelberg, 1999.

[3] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time, 2022.

[4] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. 8:399–404, 1956.

Chapter 2

Complexity

In Chapter 1 we learned some nifty algorithmic techniques. However, sometimes we are dealing with a problem where none of these techniques seem to work! Should we give up? And what do we do after giving up?

2.1 P and NP

Definition 2.1 (Computational Problem)**.** *A **computational problem** is defined as a (possibly infinite) set of inputs X and a computational goal. An algorithm solving the problem takes as input $x \in X$ and produces an output satisfying the goal.*

Definition 2.2 (Decision Problem)**.** *A **decision problem** is a computational problem specified by a partition of $X = X_{yes} \cup X_{no}$ into "yes" instances X_{yes} and "no" instances X_{no}. An algorithm solves the decision problem if, when run on $x \in X_{yes}$ outputs "yes" and when run on $x \in X_{no}$ outputs "no".*

Definition 2.3 (Function Problem)**.** *A **function problem** is a computational problem specified by a function Y, mapping each input $x \in X$ to a (possibly empty) set of feasible outputs Y_x for that input. An algorithm solves the function problem if, when run on $x \in X$, produces an output $y \in Y_x$, or outputs "no" in case $Y_x = \varnothing$.*

Remarks:

- For any function problem, asking the question "is Y_x nonempty" is a decision problem, which is no harder than the function problem: if we can find $y \in Y_x$ or tell that $Y_x = \varnothing$, then we can also state whether $Y_x \neq \varnothing$.

Definition 2.4 (Optimization Problem)**.** *An **optimization problem** is a function problem enhanced with a quality measure function $q(x, y)$. An algorithm solves the optimization problem if, when run on $x \in X$, produces an output $y \in Y_x$ which maximizes/minimizes $q(x, y)$, or outputs "no" in case $Y_x = \varnothing$.*

Remarks:

- Solving the optimization problem is no easier than solving the function problem: if we can find the best solution, we can also find a solution.
- We have seen several optimization problems in the previous chapter: What is the maximum value that can be packed into a knapsack? What is the optimal value of flow in a given network?
- Any linear program (Definition 1.18) is an optimization problem: an input to the problem is of the form $x = (A, b, c) \in X$, the set of admissible outputs for x is $Y_x = \{y \mid Ay \leq b\}$. The function to be maximized is $q(x, y) = c^T y$.
- The distinction between maximization and minimization problems is insignificant, since maximizing $q(x, y)$ is the same as minimizing $-q(x, y)$.
- In general, one can reformulate any optimization problem as a decision problem. The corresponding decision problem asks a "yes"/"no" question about the quality of a solution. For example: Can we pack items of total value at least 100 into a knapsack? Is there a flow of value at least 5 in a given network?

Definition 2.5. *Let (X, Y, q) be a maximization optimization problem, then the corresponding decision problem has as inputs pairs of the form (x, k) with $x \in X$, and (x, k) is a yes instance iff there exists $y \in Y_x$ such that $q(x, y) \geq k$.*

Lemma 2.6. *By solving the optimization problem, we can find a solution for the corresponding decision problem, or report none exist.*

Proof. We show this for the maximization variant, minimization being similar. Upon solving the optimization problem, one of the following two will hold:

- Either the optimization problem returns that $Y_x = \varnothing$, in which case we know that the answer is "no" for the decision problem;
- Or the optimization problem returns $y \in Y_x$ such that $q(x, y)$ is maximized. If any solution $y' \in Y_x$ satisfies $q(x, y') \geq k$, then the returned y will satisfy this as well, since $q(x, y) \geq q(x, y')$. Therefore, in this case we can just check whether $q(x, y) \geq k$ and return "yes" if so, and "no" otherwise. □

Remarks:

- In this chapter, we assume that $q(x, y)$ is efficient to evaluate, so the step where we check whether $q(x, y) \geq k$ in the proof of Lemma 2.6 does not add too much overhead. But there are optimization problems where finding the optimal $y \in Y_x$ is feasible within reasonable time, but computing the quality $q(x, y)$ is actually difficult.
- The opposite direction of Lemma 2.6 is usually also possible: if we have an algorithm for a decision problem, we can run it for many different decision values k in order to determine the optimal value. If we can then also find a feasible solution with this optimal value of k, then we are done. This way, we can solve the corresponding optimization problem, but it is not clear how efficiently.

- In the first part of this chapter, we will focus on decision problems.
- In the previous chapter, we have discussed a measure for discussing the running time of algorithms — the time complexity (Definition 1.5). Algorithms for Knapsack seemed to have exponential time complexity, while network flow and matching were polynomial.
- When studying time complexity, we are first and foremost interested in whether a problem can be solved in polynomial time. All problems that can be solved in polynomial time are considered "easy", and grouped together in a class.

Definition 2.7 (Complexity Class P)**.** *The **complexity class** P contains all decision problems that can be solved in deterministic polynomial time, i.e. where there exists an algorithm whose running time is in $\mathcal{O}(\mathsf{poly}(n))$, where n is the size of the input.*

Remarks:

- There are many problems that belong to class P: checking if an array is sorted, checking whether a graph is bipartite, checking whether an integer exists in an array, etc.
- A common misconception is that all problems that can be solved in polynomial time belong to class P; e.g. sorting an array, computing the product of two matrices, finding the maximum in an array. The reason this is *not* true is that they are *not* decision problems, but function problems.
- In order to show that a problem is in P, we usually give an explicit algorithm and analyze the running time directly. There is also an indirect way of showing that a problem is in P: if we can show that a problem known to be in P can be used to solve our problem. Then, our problem is "at most as difficult" as a problem in P, and is therefore also in P.

Definition 2.8 (Polynomial Reduction)**.** *Let A and B be computational problems (decision, function or optimization). A polynomial-time reduction from problem A to problem B is an algorithm that solves problem A using a polynomial number of calls to a subroutine for problem B, and polynomial additional time. If such an algorithm exists, then we write $A \leq B$.*

Remarks:

- Note that the reduction is not allowed to look inside the subroutine solving B, it should work no matter how the subroutine is implemented.

Lemma 2.9. *"Sorting an Array" $\leq$ "Maximum of an Array." Moreover, "Sorting an Array" can be solved in polynomial time.*

Proof. Assume we have a subroutine implementing "Maximum of an Array." Then, we can compute the maximum value iteratively, n times, at each step removing the maximum from the array. Putting together these maxima, we get the sorted array. We can find a maximum in linear time, and we call the maximum n times, which completes the proof. □

Remarks:

- We can use polynomial reductions between problems in order to determine classes of problems that have roughly the same difficulty.

Problem 2.10 (Clique)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***clique (decision) problem*** *we want to know whether there exists a subset of k nodes in G such that there is an edge between any two nodes.*

Problem 2.11 (Independent Set)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***independent set problem*** *we want to know whether there exists a subset of k nodes in G such that there is* ***no*** *edge between any two nodes in the subset.*

Theorem 2.12. *Clique $\leq$ Independent Set and Independent Set $\leq$ Clique.*

Proof. Let $K = \{\{u, v\} \in V^2 \mid u \neq v\}$ denote all possible edges in a graph G. Given a graph $G = (V, E)$, remove all edges E of this graph and add all edges in $K \setminus E$. Then, all cliques in G will become independent sets in the new graph, and vice-versa. This transformation can be computed in polynomial time. As a result, if we have an algorithm solving Independent Set, we can solve Clique by calling it on the new graph, and vice-versa. □

Problem 2.13 (Vertex Cover)**.** *The input is a graph $G = (V, E)$ with n nodes, and an integer $k < n$. In the* ***vertex cover problem*** *we want to know whether there exists a subset S of k nodes in G such that every edge in E has at least one of its two endpoints in S.*

Theorem 2.14. *Vertex Cover $\leq$ Independent Set and Independent Set $\leq$ Vertex Cover.*

Proof. For any independent set S, all other nodes $V \setminus S$ will be a vertex cover. This is because the nodes in S are not connected to each other, by definition, so the nodes $V \setminus S$ must cover all edges. Likewise, if the nodes $V \setminus S$ cover all edges, then the nodes S cannot have any edges between them, so S is an independent set. If independent set S has size ℓ, then vertex cover $V \setminus S$ has size $n - \ell$, so the maximum size S corresponds to the minimum size $V \setminus S$. It is now easy to see that if we have a subroutine that solves one of the two problems in polynomial time, then we can also solve the other in polynomial time. □

Lemma 2.15. *The reduction relation $\leq$ is transitive, i.e., if $A \leq B$ and $B \leq C$, then $A \leq C$.*

Proof. If there is an algorithm that can solve A using polynomially many calls to a procedure solving B and polynomial time outside those calls and an algorithm that can solve B using polynomially many calls to a procedure solving C and polynomial time outside those calls, then, by inlining the algorithm for B inside the one for A, we get that there exists an algorithm that can solve A using polynomially many calls to a procedure solving C and polynomial time outside those calls. □

Remarks:

- Therefore, there are polynomial reductions between Clique and Vertex Cover as well. If you know how to solve one of these problems in polynomial time, you can solve the other two in polynomial time as well! Unfortunately, nobody knows whether these problems (and many others!) are in P.

- Using polynomial reductions, we can define a hierarchy of different problems.

- Recall the decision variant of Knapsack: Can we pack items of total value at least k into the knapsack? Even though we could not find an efficient (polynomial) algorithm for Knapsack, we can try to see whether a simpler decision problem can be solved efficiently: Assume a friend has spent a considerable amount of time trying to pack items of total value at least k into the knapsack, and they claim that the answer is "yes". Can they prove this to you without redoing all the work they went through? Of course! They just need to show you which items to pack, and you can check for yourself that this is indeed a valid solution.

Definition 2.16 (NP). *The class of non-deterministic polynomial problems* (NP) *contains decision problems $X = X_{yes} \cup X_{no}$ such that there is a polynomial algorithm A, called a polynomial verifier, which takes as input two arguments (x, y) with $x \in X$, and has the following properties:*

1. *For all $x \in X_{yes}$, there is a solution y of length polynomial in the length of x such that A outputs "yes" on (x, y).*

2. *For all $x \in X_{no}$, A outputs "no", irrespective of y.*

Remarks:

- The solution y in Definition 2.16 is also called a certificate or witness.

- Note how decision problems do *not* necessarily have a "native" notion of "solutions", in contrast to function/optimization problems; e.g. think what would be a feasible solution to "is an array sorted?"

- However, decision problems coming from function/optimization problems will oftentimes hint at an implicit notion of solutions; e.g. for Knapsack solutions are sets of items to pack that do not exceed the weight constraint.

- The definition above essentially states that "a decision problem is in NP if there is a way to define solutions for it, whose validity can be checked in polynomial time".

- The term NP ("non-deterministic polynomial") comes from an equivalent definition of NP, which is outside our scope.

- The class NP contains a large set of computational problems. But, we should also remember that there are also problems that are not in NP. For example, given any chess configuration, we not know how to determine in polynomial time whether there exists a chess move that guarantees that we will win the game (assuming both players are playing optimally).

Lemma 2.17. *Knapsack is in* NP*.*

Proof. For Knapsack, a solution is a set of items. In order for Knapsack to be in NP, we would be given a set of items and a total value, for example 100. Verifying whether a given set S of items has a total value of $\geq$ 100 is just a sum. We also need to make sure that S is admissible; i.e. it does not exceed the maximum knapsack weight constraint. This can be also done in linear time. Therefore, Knapsack is in NP. □

Lemma 2.18. *Clique is in* NP*.*

Proof. Solutions are subsets of nodes which form a clique. Given a graph and a proposed set of k nodes for a clique, we need to check whether the k proposed nodes are connected. This can be done by going through the list of all edges and counting the edges between the k nodes. If we counted $k(k-1)/2$ edges, the proposed node set forms a clique. □

Lemma 2.19. $\mathsf{P} \subseteq \mathsf{NP}$.

Proof. For any problem in P, we actually do not need the solutions to tell yes instances apart from no instances: we can just use $y = 0$ value (or any other value) as an artificial universal witness, and then checking whether $x \in X$ is a yes instance is done solely by solving the instance x in polynomial time and outputting accordingly (without relying on the witness y). □

Remarks:

- The definitions of P and NP only concern deterministic algorithms. A problem for which there exists a randomized algorithm running in polynomial time is not necessarily in P.

- Observe that all three problems (Clique, Independent Set, Vertex Cover) are in the class NP, and they seem to be more difficult than the problems in P. Are there even more difficult problems? Is Knapsack more difficult? What is the most difficult problem in NP?!

2.2 NP-hard

Definition 2.20 (NP-hard)**.** *A decision problem H is called* NP*-hard if there exists a polynomial reduction from every problem in* NP *to H.*

Remarks:

- In other words, an NP-hard problem is at least as difficult as *all* the problems in NP.
- NP-hard problems do *not* necessarily have to be in NP.
- Are there also NP-hard problems in NP?

Lemma 2.21. *Let B be an* NP*-hard problem and C be a problem for which there exists a polynomial reduction $B \leq C$. Then, C is* NP*-hard.*

Proof. Since B is NP-hard, there exists a polynomial reduction from every problem in NP to B. By using Lemma 2.15, we get that there also exists a polynomial reduction from every problem in NP to C. Therefore, C is NP-hard. □

Definition 2.22 (NP-complete)**.** *A decision problem is called* NP*-complete if it is* NP*-hard and it is contained in* NP*.*

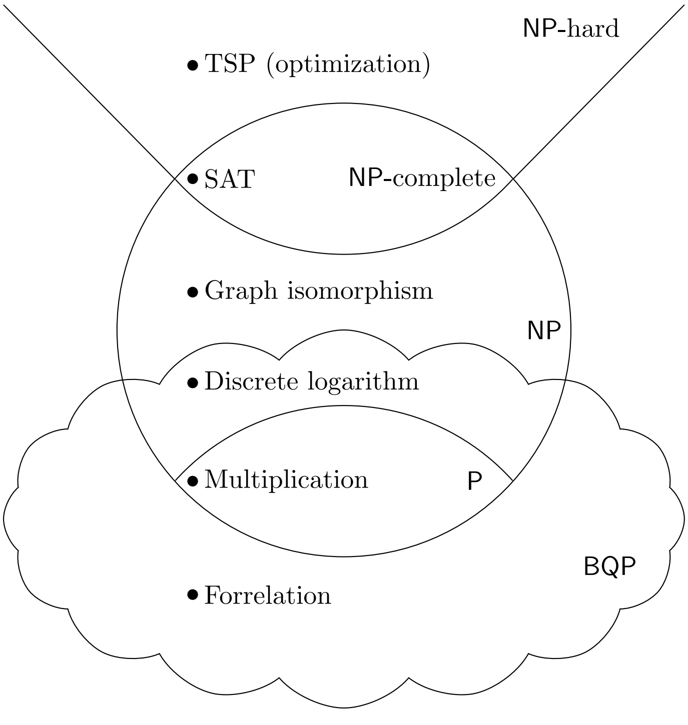


Figure 2.23: A complexity "pet" zoo: Diagram with different complexity classes and sample problems inside each class. The boolean satisfiability problem (SAT) will be introduced in the next section; the optimization version of the traveling salesperson problem (TSP) will be introduced in Section 2.5; the discrete logarithm and the graph isomorphism problems will be discussed in Chapter 3.

Remarks:

- Can we identify an NP-complete problem? If yes, we could use polynomial reductions to show that there are many more NP-complete problems. We will postpone this question to Section 2.3.
- Figure 2.23 visualizes some of the complexity classes and how they are related to each other. The figure just represents our current knowledge of how these classes relate to each other. There are still many open questions in complexity theory that could change the landscape once an answer is found. Also, there are many more classes not discussed.
- It is generally believed that there are some problems which are in NP but not in P. Or simply:

Conjecture 2.24. $\mathsf{P} \neq \mathsf{NP}$.

Remarks:

- The P versus NP question is one of the most important open scientific problems. There were many proof attempts, however, so far, without success.
- Another open question is whether quantum computers will allow to solve all difficult problems efficiently.

Definition 2.25 (BQP)**.** *A decision problem is in the class of* ***bounded-error quantum polynomial time*** (BQP) *problems, if it can be solved on a quantum computer in polynomial time and if for any input the probability to compute the wrong answer is at most* $1/3$.

Remarks:

- It is known that some problems from NP are also in BQP.
- The relation between the class of NP-complete problems and problems from BQP is unknown.
- It has been shown that there is a problem in BQP that is not in NP — the forrelation problem. In this problem, the question is whether a boolean function correlates to a Fourier transform of another boolean function. This result implies that only a quantum computer could solve (or even just verify) Forrelation efficiently.

2.3 Boolean Formulas

Definition 2.26 (Boolean Formula)**.** *Let* $\boldsymbol{x} = (x_1, \ldots, x_n)$ *be a vector of boolean variables; i.e. each variable can take one of the values True/False, or, equivalently, 1/0. A* ***boolean formula*** f *is an expression consisting of boolean variables and the logical connectives/operations AND, OR and NOT, denoted by* $\wedge$, $\vee$ *and* $\neg$*, respectively. Given an assignment of True/False values to each of the variables in* $\boldsymbol{x}$*, the formula* f *can be evaluated to yield a truth value True/False (or, equivalently,* $1/0$*). We say that a boolean formula is* ***satisfiable*** *if there* *exists an assignment of variables $\boldsymbol{x}$, such that $f(\boldsymbol{x}) = 1$. We will call x_i and its negation, $\neg x_i$, **literals**. Literals that are all connected by a logical OR (AND) operation we will call an **OR-clause** (AND-clause).*

Example 2.27. *Consider variables x_1, x_2, x_3, then an example of a boolean formula is $f = x_1 \wedge (x_2 \vee \neg x_3)$. Formula f evaluates to True if and only if x_1 is True and either x_2 is True or x_3 is False. In particular, this means that $\boldsymbol{x} = (1,1,1)$ is a satisfying assignment, while $\boldsymbol{x} = (0,1,0)$ and $\boldsymbol{x} = (1,0,1)$ are not. The subformula $(x_2 \vee \neg x_3)$ is an OR-clause.*

Definition 2.28 (Conjunctive Normal Form (CNF))**.** *A boolean formula f is in **conjunctive normal form (CNF)**, if it consists of OR-clauses that are connected by AND operations. A CNF formula is satisfiable if there is an assignment of variables such that all OR-clauses of the formula are satisfied.*

Remarks:

- Clause usually means OR-clause, unless stated otherwise.
- Any boolean formula can be rewritten as a CNF.

Problem 2.29 (SAT)**.** *In the boolean satisfiability problem (SAT), the task is to determine whether a given CNF formula is satisfiable.*

Problem 2.30 (3-SAT)**.** *3-SAT is a special case of SAT, where the given formula is a CNF that has exactly three literals in each OR-clause.*

Theorem 2.31. *SAT is in* NP.

Proof. We can use assignment vectors as solutions. Assume we are given an arbitrary boolean formula and an assignment of its variables that acts as a solution, we then need to verify in polynomial time whether the given assignment satisfies the formula. This is straight-forward to do in polynomial time. □

Remarks:

- Intuitively, SAT seems difficult to solve. With n variables, there are 2^n possible assignments to check. If only one assignment satisfies the boolean formula, we are trying to find a needle in a haystack.
- The satisfiability problem with DNF (disjunctive normal form) formulas, on the other hand, is in P. In a DNF formula, the operations AND and OR are swapped. Therefore, only one AND-clause has to be satisfied in order to satisfy the whole formula. To do so, it is sufficient to verify that there is an AND-clause that contains no variable in its negated and non-negated form simultaneously.
- Note that CNF $\leq$ DNF is not true. While every CNF can be expressed as DNF, the DNF may be exponentially bigger than the CNF, so a reduction like in Definition 2.8 would not be polynomial.

Theorem 2.32 (The Cook-Levin Theorem)**.** *SAT is* NP*-complete.*

Proof. The proof of this theorem is beyond the scope of this script and hence omitted here. The theorem is usually proven by using the alternative definition of the class NP via non-deterministic Turing machines. Using this definition one can show that it is possible to encode a Turing Machine as a SAT formula in polynomial time. □

Lemma 2.33. *3-SAT is NP-complete.*

Proof. Since 3-SAT is a special case of SAT, Theorem 2.31 also shows that 3-SAT is in NP. To show that 3-SAT is NP-hard, we reduce from SAT (i.e. Lemma 2.21). Assume we have a SAT formula where some clauses do not have 3 literals. We want to construct a 3-SAT formula that is satisfiable if and only if the SAT formula was satisfiable. Afterwards, we can use a subroutine solving 3-SAT to complete the proof. OR-clauses that contain more than three literals need to be split into smaller clauses, while clauses with less literals need to be filled up. Here, we will only consider some small examples. A clause with two literals $(x_1 \vee x_2)$ can simply be replaced by $(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$, where y is a new variable. A clause of four literals $(x_1 \vee x_2 \vee x_3 \vee x_4)$ we can rewrite as $(x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \neg z)$, where z is a new variable. Successively applying such transformations achieves our goal. □

Remarks:

- Not all instances of SAT are hard. In 2-SAT, every clause contains exactly two literals. In contrast to 3-SAT, instances of 2-SAT can be solved in polynomial time.

- In order to show that some of the previously introduced problems (e.g. Independent Set, Clique or Vertex Cover) are also NP-complete, we will search for polynomial-time reductions from SAT (or 3-SAT) to these problems.

Theorem 2.34. *Independent Set is NP-hard.*

Proof. To show hardness, we show the reduction 3-SAT $\leq$ Independent Set. Therefore, we will model boolean formulas as graphs. Given a 3-CNF formula f, we construct a graph G as follows:

1. For each clause consisting of 3 literals in f (e.g. $\neg x_1 \vee x_2 \vee x_3$) we add a cluster of 3 nodes to the graph. Each literal from the clause corresponds to one node in the cluster. We label the three nodes with their literals (e.g. $\neg x_1$, x_2 and x_3). Note that, after processing multiple clauses, it might be that multiple nodes get the same label: such nodes are considered distinct.

2. For the edges, we connect the following:

   (a) All nodes inside the same cluster.

   (b) All pairs of nodes that represent the same variable in negated and non-negated form (e.g. x and $\neg x$).

Note that G can be constructed in polynomial time.

Next, we show that f is satisfiable if and only if the corresponding graph G has an independent set of size m, where m is the number of clauses in f.

"$\implies$" Assume that f is satisfiable; i.e. there is an assignment $\boldsymbol{x}$ such that in each clause there is a literal that evaluates to True. Select one such literal from each of the m clauses. Note that, in graph G, no two of these literals will be connected. This is because the corresponding nodes are all in different clusters and because nodes with labels x and $\neg x$ cannot both be True in the boolean assignment. Therefore, these m nodes form an independent set in G.

"$\impliedby$" Assume there exists an independent set of size m in G. This independent set cannot contain nodes from the same cluster, as such nodes would be connected. By definition of G, the independent set will also not contain nodes with labels x and $\neg x$. Therefore, the independent set of size m will have exactly one node from each cluster, and no variable will be present in both its negated and non-negated form. By setting the values of the node labels in the independent set to True, we get an assignment of variables that satisfies the boolean formula. Note that this might not set a value to all variables, but in that case those variables can just be set arbitrarily. □

Remarks:

- Note that the opposite direction, Independent Set $\leq$ 3-SAT, does not directly follow from the above proof. Not all graphs G can be reached with the construction. In order to show Independent Set $\leq$ SAT, one would have to show that every graph G can be transformed into a boolean formula in polynomial time.
- In Lemma 2.18, we showed that Independent Set is in NP. Therefore, Independent Set is also NP-complete.
- Clique and Vertex Cover are also NP-complete, since there are reductions in both directions between Independent Set and these problems, which we have shown in Theorem 2.12 and Theorem 2.14.
- In other words, all these problems are the most difficult problems in NP. If you could solve one of them in polynomial time, you could solve all problems in NP in polynomial time!
- What about Knapsack? We need yet another problem.

Problem 2.35 (Subset Sum)**.** *Given a number $s \in \mathbb{N}$ and a set of n natural numbers $x_1, \ldots, x_n$, does there exist a subset of these numbers that has a sum of s?*

Theorem 2.36. *Subset Sum is NP-complete.*

Proof. It is easy to verify whether a given subset of elements has the desired sum s. Therefore, Subset Sum is in NP. In order to show that Subset Sum is NP-hard, a reduction 3-SAT $\leq$ Subset Sum is shown. This reduction is non-trivial and will be omitted in this script. □

Theorem 2.37. *The Knapsack (decision) problem is NP-complete.*

Proof. Subset Sum is a special case of Knapsack: let the natural numbers be items with values $x_1, \ldots, x_n$ and weights $x_1, \ldots, x_n$, and let the knapsack have capacity s. If items of value s can be packed into the knapsack, then the corresponding items form a subset of sum s. Therefore, Subset Sum $\leq$ Knapsack. Since Subset Sum is NP-complete, Knapsack has to be NP-hard. In Lemma 2.17, we already showed that Knapsack is in NP. □

Remarks:

- In fact, thousands of interesting problems in many different areas are NP-complete: Achromatic Number, Battleship, Cut, Dominating Set, Equivalence Deletion, …, Super Mario Bros, Tetris, …, Zoe.
- Figure 2.38 below shows the reductions of NP-complete problems that we discuss in this chapter.

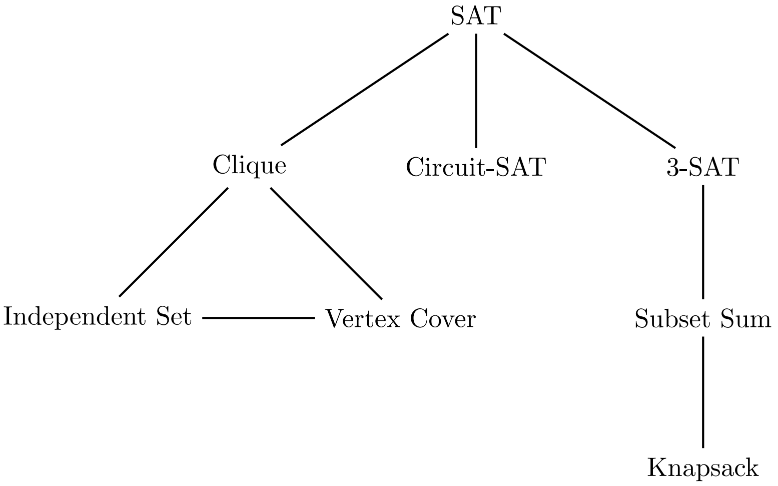


Figure 2.38: Reductions between NP-complete problems.

2.4 Boolean Circuits

Definition 2.39 (Boolean Circuit)**.** *Let $n, m \in \mathbb{N}$. A* ***boolean circuit*** *is a directed acyclic graph (DAG) with n boolean* ***input nodes*** *and m* ***output nodes****. The input nodes have no incoming edges, while the output nodes have no outgoing edges. The nodes of the graph that are not input nodes represent logical operations (AND, OR, NOT) and are called* ***gate nodes****. We will label the corresponding nodes $\wedge, \vee$ and $\neg$, respectively. Each NOT gate has in-degree 1. The in- and out- degrees of the other gates do not have to be bounded. We further define the* ***size*** *of the circuit to be its total number of gates, and the* ***depth*** *of the circuit to be the length of the longest (directed) path from an input to an output node.*

A boolean circuit C represents a mapping $C : \{0,1\}^n \to \{0,1\}^m$. The value of an input node is the value of the corresponding input bit of C. The value of *a gate node is determined recursively by applying the logic operation that this gate represents to the values received through incoming edges. The output of the boolean circuit are the values of the output nodes.*

Remarks:

- Boolean circuits provide a nice tool for understanding the complexity of computation. They are limited in computation power, yet may help to answer the $\mathsf{P} \neq \mathsf{NP}$ conjecture.

- In the literature, it is often assumed that AND and OR gates have a bounded number of input values, or that the circuit has a bounded depth.

- In the following, we will consider boolean circuits that have only one output value (True or False).

Problem 2.40 (Circuit-SAT)**.** *Let $C : \{0,1\}^n \to \{0,1\}$ be a boolean circuit with only one output value. In* ***Circuit-SAT****, the task is to determine whether for a given circuit C there exists an input vector $\boldsymbol{z} \in \{0,1\}^n$, such that $C(\boldsymbol{z}) = 1$.*

Theorem 2.41. *Circuit-SAT is in NP.*

Proof. We can use the vector $\boldsymbol{z}$ as a certificate. To check the it, we can just evaluate $C(\boldsymbol{z})$ in polynomial time. □

Theorem 2.42. *Circuit-SAT $\leq$ SAT.*

Proof. We need to construct a boolean formula that represents the gates of the given circuit. First, we introduce a new variable g_i representing gate i for each gate of the circuit. Next, for each gate i, we differentiate between the three possible gate types:

- If i is a NOT gate of gate j, then we add the clauses $(g_i \vee g_j) \wedge (\neg\, g_i \wedge \neg g_j)$ to the formula.

- If i is an OR gate with inputs from two gates, j and k, we add the following clauses to the formula: $(\neg g_i \vee g_k) \wedge (\neg g_j \vee g_k) \wedge (g_i \vee g_j \vee \neg g_k)$.

- If i is an AND gate with inputs from two gates, j and k, we add the following clauses to the formula: $(\neg g_k \vee g_i) \wedge (\neg g_k \vee g_j) \wedge (\neg g_i \vee \neg g_j \vee g_k)$.

Observe that an AND or an OR gate with k inputs can be replaced by $k-1$ gates with two inputs by increasing the depth of the circuit, so our construction also applies to circuits where some gates have more than 2 inputs.

The resulting boolean formula is satisfiable if and only if the circuit evaluates to 1 for some vector $\boldsymbol{z}$. The construction was polynomial in the input size, and the resulting boolean formula can be transformed into a CNF with minimal overhead. □

Theorem 2.43. *Circuit-SAT is* NP*-complete.*

Proof. Previously, we have shown that Circuit-SAT is in NP. In order to show that Circuit-SAT is NP-hard, we will reduce SAT to Circuit-SAT; i.e. SAT $\leq$ Circuit-SAT. Let each variable of the boolean formula be an input to the circuit. For each negated variable, we can add a NOT gate. Then, for each clause in the formula, we add an OR gate that has all variables from the clause as input to the gate. And, finally, we add one AND gate that has values from all OR gates as its input. This gives a polynomial construction of a boolean formula as a circuit, thus showing that Circuit-SAT is NP-complete. □

Problem 2.44 (Minimum Circuit Size Problem (MCSP))**.** *Given a boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$, represented as a truth table, the task of the **Minimum Circuit Size Problem (MCSP)** is to determine whether f can be represented by a boolean circuit of size at most s; i.e. a circuit with at most s gates.*

Remarks:

- There are several approaches of how boolean formulas can be simplified. Often the laws from boolean algebra are applied (Distributive law, Idempotent law, Identity law, Complement law, DeMorgan's law, Karnaugh maps). These minimization rules however do not guarantee that the resulting boolean formula uses the minimum number of gates.

- How difficult is MCSP?

Lemma 2.45. *MCSP is in* NP*.*

Proof. In essence, we can use as witness the circuit of size at most s that implements the function f. □

Remarks:

- Is MCSP also NP-hard? As of now, there is no answer to this question. In fact, there is evidence that this result might be difficult to establish. However, we know that the problem is NP-hard if the circuit is allowed to have multiple outputs.

- Some complexity classes are defined with respect to boolean circuits.

Definition 2.46 (Class AC^0)**.** *The class AC^0 contains all decision problems that can be decided by a family of circuits $\{C_n\}$, where C_n has a constant depth and the size of the circuit is polynomial in n.*

Remarks:

- Note that we need to consider a circuit family, because the input size of each circuit is fixed, while the input size to a problem can vary.

- Note that we do not allow the boolean circuits to save output bits or use recursive operations.

- There exist many other classes based on the boolean circuits: The class NC^0 is defined in the same way as AC^0, but the number of incoming edges to the AND and the OR gate is restricted to only two edges. Note that "two" can really be replaced by any constant, since, e.g., a three-input AND can be simulated with two binary ANDs. Both classes can be extended to versions that allow a non-constant depth of the circuit: AC^i and NC^i have depth $\mathcal{O}\big(\log^i n\big)$.

- Since the NAND gate (negated AND) can simulate all three of NOT, OR and AND, we could have defined AC and NC equivalently using only NAND gates. Note that this would fail to be true for the XOR gate (exclusive OR), since XOR can simulate neither AND nor OR.

2.5 Solving Hard Problems

We have seen that many important computational problems are known to be NP-hard. So what do we do if we encounter such a hard problem?

Remarks:

- We will now consider optimization problems instead of decision problems. This often makes sense in practice, where you usually want to find the best solution to some problem, instead of just deciding if the problem has a solution with a specific cost.

- So how can we approach a hard problem in practice? So far, we have mostly considered exact algorithms.

Definition 2.47 (Exact Algorithm)**.** *An **exact algorithm** always returns the optimal solution to the problem.*

Remarks:

- We have seen that for NP-complete problems we do not know if there is an exact algorithm with polynomial running time.

- Even when the worst-case running time of an algorithm is exponential, one can often use tricks to considerably reduce this running time in practice, e.g. by using look-ahead or pruning techniques as in Definition 1.9.

- What if we also accept weaker-than-optimal solutions while requiring that our algorithm always has a reasonable running time?

Definition 2.48 (Heuristic)**.** *A **heuristic** is an algorithm that is guaranteed to have a polynomial running time, at the cost of returning a suboptimal solution.*

Remarks:

- One example for a heuristic is our greedy algorithm for Knapsack, Algorithm 1.8.
- Good heuristics are usually based on insights into the structure or the behavior of the problem. Some heuristics are known to usually provide good solutions in practice.
- However, while heuristics are often good on some inputs, they may return weak solutions on other inputs.
- In general, it is better to have algorithms which provide guarantees that they always return a solution which is reasonably close to the optimum. We want to understand algorithms with such guarantees.

2.6 Vertex Cover Approximation

Let us first revisit the optimization version of Vertex Cover: we want to find a vertex cover $S \subseteq V$ in an input graph $G = (V, E)$, with $|S|$ as small as possible. Can we come up with an algorithm where we can prove that the size of the returned vertex cover is always "reasonably close" to the optimum?

Remarks:

- A natural approach is the greedy method of Algorithm 2.49.

```
def VertexCover_Greedy_Naive(G):
    $S = \emptyset$
    while $E \neq \emptyset$:
        select an arbitrary edge $(u, v) \in E$
        $S = S \cup \{u\}$
        remove all edges from $E$ that are adjacent to $u$
    return $S$
```

Algorithm 2.49: Naive greedy algorithm for Vertex Cover.

Lemma 2.50. *Algorithm 2.49 returns a vertex cover.*

Proof. An edge from E is only removed when one of its incident nodes is included in the set S. □

Remarks:

- However, the size of this vertex cover can be far from the optimal size.

Theorem 2.51. *The size $|S|$ of the vertex cover returned by Algorithm 2.49 can be an $n - 1$ factor larger than the optimum.*

Proof. Consider a "star" graph with n nodes and $n-1$ edges, where $n-1$ distinct leaf nodes u_1, u_2, …, u_{n-1} are connected by an edge to the same center node v. Whenever Algorithm 2.49 selects an edge (u_i, v) in this graph, it can happen that it always chooses the leaf node u_i. In this case, the algorithm only removes a single edge (u_i, v) in each step, and the algorithm lasts for $n-1$ iterations of the loop. In the end, the algorithm returns the vertex cover $S = \{u_1, \ldots, u_{n-1}\}$, which consists of $n-1$ nodes.

On the other hand, the set $\{v\}$ is an optimal vertex cover of cost $c^* = 1$ in this graph, so we have $\frac{|S|}{c^*} = n - 1$. □

Remarks:

- It is a natural idea to try to improve Algorithm 2.49 with a clever tie-breaking rule: for example, to always select the higher-degree adjacent node of the chosen edge, or the highest-degree node altogether among the available nodes. This solves the star. However, there are more complicated counterexamples which show that even in this case, the solution we obtain can be a $\log n$ factor worse than the optimum.

- However, there is a slightly different greedy approach that provides much better guarantees.

```
def VertexCover_Greedy(G):
    $S = \emptyset$
    while $E \neq \emptyset$:
        select an arbitrary edge $(u, v) \in E$
        $S = S \cup \{u, v\}$
        remove all edges from $E$ that are adjacent to $u$ or $v$
    return $S$
```

Algorithm 2.52: Greedy algorithm for Vertex Cover.

Theorem 2.53. *Algorithm 2.52 always returns a vertex cover with* $|S| \leq 2 \cdot c^*$.

Proof. Algorithm 2.52 returns a correct vertex cover: an edge is only removed from E if at least one of its incident nodes is inserted into S.

Assume that the algorithm runs for c iterations of the main loop, i.e. it selects c different edges (u_i, v_i), and inserts both endpoints u_i and v_i of these edges into S for $i \in \{1, \ldots, c\}$. Note that none of these edges (u_i, v_i) share a node, because all edges adjacent to either u_i or v_i are removed when (u_i, v_i) is selected (in graph theory, such a set of edges is called a *matching*). This means that in the optimal vertex cover S^*, each vertex can only be incident to at most one of the edges (u_i, v_i); thus, in order to cover all the edges (u_i, v_i), we already need at least c nodes. As a result, we get that $c^* = |S^*| \geq c$.

On the other hand, our algorithm returns a vertex cover of size $|S| = 2 \cdot c$, so we have $|S| = 2 \cdot c \leq 2 \cdot c^*$. □

Remarks:

- In a graph of $\frac{n}{2}$ independent edges, the solution returned by Algorithm 2.52 is indeed 2 times worse than the optimum, so our analysis is tight.
- This algorithm was somewhat counter-intuitive; one might expect Algorithm 2.49 to be more efficient. However, as we have seen, Algorithm 2.52 is always at most a factor 2 away from the optimum c^*, while Algorithm 2.49 can be an $(n-1)$-factor away.
- Algorithms that are proven to always be within an α factor from the optimum are called *approximation algorithms.* Theorem 2.53 shows that Algorithm 2.52 is a 2-approximation algorithm for Vertex Cover.
- For our formal definition of this notion, we assume a minimization problem.

Definition 2.54 (Approximation Algorithm)**.** *We say that an algorithm $\mathcal{A}$ is an α-**approximation algorithm** if for every possible input of the problem, it returns a solution with a cost of at most $\alpha \cdot c^*$, where c^* is the cost of the optimal solution (the one with minimal cost).*

Remarks:

- The definition is similar for maximization problems. In this case, we denote the value of the optimal solution (the one with highest value) by v^*. For an approximation algorithm, we require that the returned solution always has a value of at least v^*/α.
- This section only considers approximations algorithms that run in polynomial time. That is exactly the point of these algorithms: to find a reasonably good solution without having an unreasonably high running time.

Definition 2.55. *We say that a problem is α-approximable if there exists a polynomial-time algorithm $\mathcal{A}$ that is an α-approximation algorithm for the problem.*

Remarks:

- In general, α can be any constant value with $\alpha > 1$, or it can also be a function of n (the size of the input), e.g. $\alpha = \log n$ or $\alpha = n^{3/4}$.
- In many cases, we can already get some approximability results from the most trivial algorithms. Independent Set naturally has $v^* \leq n$. Also, we can find a solution of value 1 by simply outputting an arbitrary single node. This shows that Independent Set is (at least) n-approximable.
- This allows us to classify hard problems, based on how well their optimum solution can be approximated. E.g. the complexity class of problems that are α-approximable for some constant α is called APX. Since Theorem 2.53 has $\alpha = 2$, Vertex Cover is in APX.

- For Vertex Cover, there is currently no $2 - \varepsilon$ approximation algorithm known for any $\varepsilon > 0$, so the algorithm above is indeed the best we have. Also, it is proven that no approximation better than 1.3606 is possible at all unless $\mathsf{P} = \mathsf{NP}$. This means that unless $\mathsf{P} = \mathsf{NP}$, we are unable to approximate the best solution of this problem arbitrarily well (in polynomial time).

2.7 Bin Packing Approximation

Let us now look at some other problems that are α-approximable to some constant α. We continue with the bin packing problem, which is similar to Knapsack.

Problem 2.56 (Bin Packing)**.** *We have a set of n items of sizes $x_1, \ldots, x_n$, and an unlimited number of available bins, each having a capacity of B. A set of items fits into a bin if their total size is at most B. Our goal is to put all of the items into bins, using the smallest possible number of bins.*

→ notebook

Remarks:

- You can easily imagine immediate applications of this, e.g. packing files on disks.
- For this problem, c^* denotes the number of bins that are used in the optimal solution.
- In Bin Packing, a simple greedy heuristic already allows us to obtain a 2-approximation.

```
def FirstFit(items, B):
    for each item in items:
      place item in the first bin where it still fits
      if item does not fit into any bin:
        open a new bin, and insert item into the new bin
```

Algorithm 2.57: First Fit algorithm for Bin Packing.

Lemma 2.58. *In Algorithm 2.57, at most 1 bin is filled at most half.*

Proof. Assume that there are at least 2 bins b_1 and b_2 that are filled at most half. This means that b_1 still has free space of at least $\frac{B}{2}$, and b_2 only contains items of size $x_i \leq \frac{B}{2}$. However, in this case, the items sorted into b_2 would also fit into b_1. This contradicts the First Fit algorithm, which only opens a new bin when the next item does not fit into any of the previous bins. □

Theorem 2.59. *Algorithm 2.57 is a 2-approximation.*

Proof. Assume that the First Fit algorithm uses m bins. Due to Lemma 2.58, at least $m-1$ of these bins are filled to a capacity of more than $\frac{B}{2}$. This implies

$$\sum_{i=1}^{n} x_i > (m-1) \cdot \frac{B}{2}.$$

On the other hand, we know that

$$c^* \geq \frac{\sum_{i=1}^{n} x_i}{B},$$

since all the items have to be sorted into a bin, and every bin can only take at most B. The two inequalities imply

$$B \cdot c^* > (m-1) \cdot \frac{B}{2},$$

and thus

$$2 \cdot c^* > m-1.$$

Since c^* and m are integers, this implies $2 \cdot c^* \geq m$, proving our claim. □

Remarks:

- One can even improve on this algorithm by first sorting the elements in decreasing order, and then applying the same First Fit rule. With only a slightly more detailed analysis, one can show that this improved algorithm achieves an approximation ratio of 1.5.
- Can we get a better approximation ratio than 1.5? Unfortunately not, unless we have $\mathsf{P} = \mathsf{NP}$. One can show that getting a better than 1.5 approximation is already an NP-hard problem.
- To prove this, we present a reduction to Partition.

Problem 2.60 (Partition)**.** *In the Partition Problem, given a set of n items of positive size $x_1, \ldots, x_n$, we want to decide if we can partition them into two groups such that the total sum is equal in the two groups.*

Remarks:

- Partition is a special case of Subset Sum where $s = \frac{1}{2} \cdot \sum_{i=1}^{n} x_i$.
- It is known that this special case is still NP-complete.

Theorem 2.61. *Subset Sum $\leq$ Partition.*

Proof. The reduction is not too complex, but a bit boring, so we do not discuss it here. □

Theorem 2.62. *It is NP-hard to solve Bin Packing with an α-approximation ratio for any constant $\alpha < \frac{3}{2}$.*

Proof. Given an input of Partition with integers $x_1, \ldots, x_n$, we convert it into a Bin Packing problem: we consider items of size $x_1, \ldots, x_n$, and we define $B = \frac{1}{2} \cdot \sum_{i=1}^{n} x_i$.

If the original Partition problem is solvable, then this Bin Packing problem can also be solved with 2 bins. In this case, an α-approximation algorithm with $\alpha < \frac{3}{2}$ must always return a solution with $m \leq c^* \cdot \alpha < 2 \cdot \frac{3}{2} = 3$ bins; since m is an integer, this means a solution with $m = 2$ bins.

On the other hand, if the items cannot be partitioned into two sets of size B, then the algorithm can only return a solution with $m \geq 3$ bins.

Hence an α-approximation algorithm for Bin Packing would also solve Partition in polynomial time: we can just solve the converted Bin Packing problem, and if $m = 2$, then output 'Yes', whereas if $m \geq 3$, then output 'No'. □

Remarks:

- Thus, Bin Packing is a problem that can be approximated to some constant $\alpha = 1.5$, but not arbitrarily well, i.e. not for any $\alpha > 1$.
- Let us consider a classical graph question.

2.8 Metric TSP Approximation

Problem 2.63 (Traveling Salesperson or TSP)**.** *The input is a clique graph $G = (V, E)$ (there is an edge between any two nodes of G), with positive edge weights, i.e. a function $d : E \to \mathbb{R}^+$. The goal of* ***TSP*** *is to find a Hamiltonian cycle in G where the total weight of the edges contained in the cycle is minimal.* → notebook

Remarks:

- In general, solving the traveling salesperson problem is NP-hard.
- Even approximating TSP to a constant factor is already NP-hard. We will prove this in the next section.
- On the other hand, there is a special case of the problem that does allow a constant-factor approximation.

Problem 2.64 (Metric TSP)**.** *In Metric TSP, the distances between any three nodes v_1, v_2, v_3 must satisfy the triangle-inequality: $d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)$.*

Remarks:

- Note that the triangle inequality is a realistic assumption in real-world applications.
- This is another possible approach to solving hard problems in practice: maybe we can show that our actual problems are restricted to only a special case of the original problem, and that this special case is computationally more tractable.
- A Metric TSP already has a 2-approximation algorithm.

```
def Tree_Based_TSP(G)
    find a Minimum Spanning Tree $T$ in $G$
    form a sequence of nodes $P_0$ by traversing all nodes of $T$
    simplify $P_0$ to $P$ by only keeping the first occurrence of each node
    return $P$
```

Algorithm 2.65: Tree-based approximation for TSP.

Remarks:

- The Minimum Spanning Tree (MST) is, intuitively speaking, the subset of edges with smallest total weight which already connects the whole graph, i.e. there is a path between any two nodes through these edges. An MST in a graph can easily be found in polynomial time with some simple algorithms.

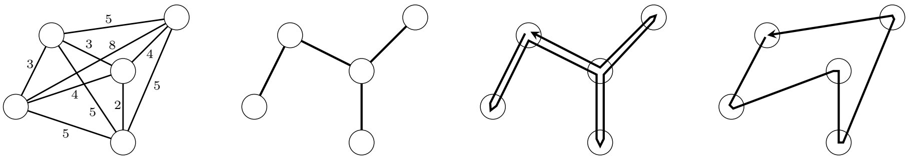
Theorem 2.66. *Algorithm 2.65 is a 2-approximation.*

Proof. See also Figure 2.67. Let t^* denote the total cost of the MST. Note that if we delete a single edge from any Hamiltonian cycle, we get a spanning tree, so t^* must be smaller than the cost of any Hamiltonian cycle. This implies $t^* < c^*$.

Now consider the cost of the Hamiltonian cycle returned by Algorithm 2.65. Since the traversal visits each edge of the spanning tree exactly twice, the total cost of P_0 is $2 \cdot t^*$. By only keeping the first occurrence of each node in P_0, we create "shortcuts": whenever we delete a node from P_0, we only shorten our tour due to the triangle inequality. As such, for the total costs we have

$$cost(P) \leq cost(P_0) = 2 \cdot t^* < 2 \cdot c^*.$$

□



(a) (b) (c) (d)

Figure 2.67: A weighted complete graph as an input to TSP (a), an MST in this graph (b), the tour P_0 obtained from this MST, starting from an arbitrary node (c), and the simplified tour P after removing repeated occurrences (d).

Remarks:

- Being more clever even allows a 1.5-approximation.
- For TSP, it is often reasonable to assume the even more special case of Euclidean TSP, where the nodes of the graph corresponds to specific points in a plane, for example. That is, each node v is associated with two coordinates v_x and v_y, and the weight of the edge between u and v is the Euclidean distance of u and v, i.e. $\sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$.

2.9 Non-Approximability: TSP

We now return to the general version of TSP. By a reduction to the following problem, we prove that approximating TSP up to a constant factor is already NP-hard.

Problem 2.68 (Hamiltonian Cycle)**.** *The input is a graph $G = (V, E)$. In the* ***Hamiltonian cycle problem*** *we want to know whether there exists a cycle in G which contains each node of the graph exactly once.*

Remarks:

- Hamiltonian cycle was among the first problems that were shown to be NP-complete.

Theorem 2.69. *In general graphs, TSP is NP-hard to approximate to any constant factor α.*

Proof. We provide a reduction to Hamiltonian Cycle, which is NP-complete.

Assume that we have an α-approximation algorithm for TSP. Given an input graph $G = (V, E)$ for Hamiltonian Cycle, we turn this into an instance of TSP on V. For each pair of nodes u and v, we define the weight of edge (u, v) in the TSP in the following way:

- if $(u, v) \in E$, then we assign $d(u, v) = 1$ in the TSP,
- if $(u, v) \notin E$, then we assign $d(u, v) = \alpha \cdot n + 1$ in the TSP.

If the initial graph had a Hamiltonian cycle, then the resulting TSP has a cycle of total weight n as the optimal solution. In this case, our approximation algorithm is guaranteed to return a solution of cost at most $\alpha \cdot n$. On the other hand, if there was no Hamiltonian cycle in the original graph, then the optimal TSP cycle must contain at least one edge of weight $\alpha \cdot n + 1$, and thus $c^* \geq \alpha \cdot n + 1$.

Hence an α-approximation returns a solution of cost at most $\alpha{\cdot}n$ if and only if the original graph had a Hamiltonian cycle; thus running such an approximation algorithm allows us to solve the Hamiltonian cycle problem. This shows that finding a polynomial-time α-approximation of TSP for *any* constant α is NP-hard. □

Remarks:

- The most difficult problems are those that cannot be approximated to any constant α, but only to a factor $\alpha = f(n)$ depending on n, e.g. $\alpha = \log n$ or $\alpha = \sqrt{n}$. This means that as the size of the problem becomes larger, the difference between the optimum and our solution also grows larger.
- For TSP, not even this is possible in polynomial time. The proof above works for any function $f(n)$ that can be computed in polynomial time.
- Remember the Independent Set problem? It turns out that this is also one of the most difficult problems from this perspective: approximating Independent Set to any factor that is slightly smaller than n would already imply that $\mathsf{P} = \mathsf{NP}$. (The proof of this fact is more involved than our previous claims, so we do not discuss it here.)

- This means that for any approximation algorithm $\mathcal{A}$, there are some graphs where $\mathcal{A}$ returns weak solutions.

- This $n^{1-\delta}$ factor is indeed huge. For example, it might be that the largest independent set contains $n^{0.99}$ nodes, but $\mathcal{A}$ returns an independent set of only 1 or 2 nodes. As such, an approximation algorithm like this is not useful in practice.

- The same result also holds for the Clique problem.

- Recall from Theorem 2.14 that as a decision problem, Independent Set $\leq$ Vertex Cover; however, we have seen that Vertex Cover is 2-approximable. This shows that even if the decision version of two problems are equivalent, the approximability of the optimization version may differ significantly.

- So while all NP-complete decision problems have the same difficulty in theory, their optimization variants are substantially different in practice.

2.10 FPTAS: Knapsack

We have now seen many problems that are α-approximable for a specific constant α. Sometimes we can even get arbitrarily close to the optimum in polynomial time: there exists a $(1+\varepsilon)$-approximation for any $\varepsilon > 0$.

Definition 2.70 (FPTAS). *We say an algorithm $\mathcal{A}$ is a fully polynomial-time approximation scheme (FPTAS) if for any $\varepsilon > 0$*

- *$\mathcal{A}$ is a $(1+\varepsilon)$-approximation algorithm, and*
- *the running time of $\mathcal{A}$ is polynomial in both n and $\frac{1}{\varepsilon}$.*

Remarks:

- While an FPTAS is not as good as a polynomial-time exact algorithm, it is almost as good: for any desired error ε, we can efficiently find a solution that is only ε away from the optimum.

- There is also a slightly weaker notion of PTAS (polynomial-time approximation scheme), where we only require the running time to be polynomial in n, but not in $\frac{1}{\varepsilon}$. That is, the running time is still polynomial in n for any fixed $\varepsilon > 0$, but it might increase quickly in $\frac{1}{\varepsilon}$; for example, it includes a factor of $n^{2^{1/\varepsilon}}$. These algorithms are not as useful in practice: if we want to get really close to the optimum by selecting a small ε, the running time still becomes unreasonably large.

- As an example for a nicely approximable problem, we revisit the Knapsack problem, and present an FPTAS for it.

- Since we now study the problem from more of a mathematical than a programming-based perspective, we introduce a shorter notation for the inputs of the problem.

Definition 2.71 (Knapsack notation)**.** *We will use C to denote the capacity of our Knapsack, and we denote the value and weight of the i^{th} item as v_i and w_i, respectively. We denote the number of items by n as before. Finally, let us use $v_{\max}$ to denote the value of the highest value item, i.e. $v_{\max} := \max v_i$.*

Remarks:

- Let us use v^* to denote the maximal value we can fit into the knapsack, i.e. the optimal solution.
- Note that if any item has $w_i > C$, then it can never fit into the knapsack, so we might as well remove such items. Hence, we will assume that even the largest items still fits into the knapsack, i.e. $v_{\max} \leq C$.
- Recall that we have discussed a dynamic programming solution for Knapsack, where each cell $V[i][c]$ of our DP table stored the maximum value that can be achieved with capacity c using only the first i items. The starting point of our FPTAS algorithm will be a slightly different variant of this DP method: in our new table, $W[i][v]$ will store the weight of the lowest-weight subset of items $1, \dots, i$ that has a total value of v.
- One can show that this table W can also be computed with a similar dynamic programming method to Algorithm 1.12. As before, the table has n rows. The number of columns is now at most $n \cdot v_{\max}$, which is an upper bound on the value of any subset; we have n items, and each of them has value at most $v_{\max}$. From this alternative DP table, we can find the optimum v^* by taking the maximal v value in the table where $W[n][v] \leq C$. → notebook
- The main difference from the original DP algorithm is that now each column expresses the *value* of a specific subset of items, instead of the *weight* of a specific subset of items as before.
- We can now use this alternative DP algorithm to develop an FPTAS for Knapsack. The main idea is to slightly round up the values of the items, which results in some inaccuracy for the algorithm, but it also reduces the number of columns in our DP table.

```
def Knapsack_FPTAS(items, C):
    $k = \varepsilon \cdot v_{\max} \,/\, n$
    consider the modified Knapsack problem where item $i$ has weight $w_i$ and value $\hat{v}_i = \lceil \frac{v_i}{k} \rceil \cdot k$
    run the alternative DP algorithm on this problem, with columns only corresponding to multiples of $k$
```

Algorithm 2.72: FPTAS algorithm for Knapsack.

Lemma 2.73. *Algorithm 2.72 has a running time of $\mathcal{O}\big(n^3 \cdot \frac{1}{\varepsilon}\big)$.*

Proof. Given the parameter ε, the algorithm introduces a scaling parameter $k = \varepsilon \cdot v_{\max} / n$, and defines the *rounded-up value* of item i as $\hat{v}_i = \lceil \frac{v_i}{k} \rceil \cdot k$. This means that whichever subset of nodes we select, the total value of the subset is a multiple of k.

On the other hand, any subset of the items has a total value of $n \cdot v_{\max}$ at most. If each column represents a multiple of k, then the table has at most $n \cdot v_{\max} / k = n^2 \cdot \frac{1}{\varepsilon}$ columns. The table still has n rows, so the total running time of the DP algorithm with these rounded values is $O(n^3 \cdot \frac{1}{\varepsilon})$. □

Theorem 2.74. *Algorithm 2.72 is an FPTAS.*

Proof. The polynomial running time has already been established in Lemma 2.73. Let us use S to denote the set of items chosen by Algorithm 2.72. Since we round the values to multiples of k, the value of each item has been overestimated by k at most, so $\hat{v}_i - k \leq v_i$. Hence for the total value of items in S we have

$$\sum_{i \in S} v_i \geq \sum_{i \in S} (\hat{v}_i - k) = \sum_{i \in S} \hat{v}_i - |S| \cdot k = \sum_{i \in S} \hat{v}_i - |S| \cdot \frac{\varepsilon \cdot v_{\max}}{n} \geq \sum_{i \in S} \hat{v}_i - \varepsilon \cdot v_{\max}.$$

To simplify this last expression even more, we make two more observations. First, note that $\sum_{i \in S} \hat{v}_i$ is the optimal solution of the rounded problem found by our algorithm. Since we only rounded values upwards, this is larger or equal to the solution of the original problem v^*. Second, choosing the item with maximal value $v_{\max}$ is always a valid solution, so we also have $v_{\max} \leq v^*$. Hence for the total value in S we have

$$\sum_{i \in S} \hat{v}_i - \varepsilon \cdot v_{\max} \geq v^* - \varepsilon \cdot v^* = (1 - \varepsilon) \cdot v^*.$$

Note that this shows $v^* / v(S) \leq \frac{1}{1-\varepsilon}$, while our original FPTAS definition requires a slightly different relation: that $v^* / v(S) \leq (1 + \varepsilon)$. However, since we now have an $\alpha = \frac{1}{1-\hat{\varepsilon}}$-approximation algorithm for any $\hat{\varepsilon} > 0$, we can obtain $\alpha = (1 + \varepsilon)$ by choosing $\hat{\varepsilon} = \frac{\varepsilon}{1+\varepsilon}$. □

Chapter Notes

The first formal definition of computation was provided by Alan Turing in 1936 [20]. In this paper, Turing presents an automatic machine (now known as Turing machine) that can compute certain classes of numbers. We will discuss the Turing machine in more detail in the last chapter. The birth of computational complexity as a field is attributed to Hartmanis and Stearns for their paper "On the computational complexity of algorithms" [14]. They proposed to measure time with respect to the input size and showed that some problems can only be solved if more time is given.

The first problem that was shown to be in P was the maximum matching problem, see Edmonds [9]. In another paper, Edmonds introduced a notion that is equivalent to the class NP [10]. In 1971, Cook [6] showed that SAT was NP-complete. Two years later, independently of Cook's result, Levin [17] showed that six problems were NP-complete, the so-called universal search problems. Both authors have formulated the famous P versus NP problem in computer science. The name NP was however given to the class a year later by Karp [16], who proved that 21 (combinatorial) problems were NP-complete. He thereby first used the technique of polynomial reductions in these proofs. Since then, thousands of problems in different areas of science have been shown to be NP-hard. Also many new complexity classes have been proposed in the literature. Different "zookeepers" around the world keep track of the newly introduced complexity classes in their complexity zoos [1]. The question $\mathsf{P} \neq \mathsf{NP}$ still remains open. In 2000, the P versus NP problem was announced as one of the seven Millennium Prize Problems by the Clay Mathematics Institute [4]. Solving P versus NP (or BQP versus NP) will be a major milestone in science, in case of equality ($\mathsf{BQP} = \mathsf{NP}$) there will be amazing practical consequences.

Circuit theory is almost 200 years old, being studied by electrical engineers and physicists. In the 1940s, the invention of semiconductor devices and later the transistor further boosted the quest for understanding circuits. The computational model of boolean circuits was introduced by Claude Shannon [19] in 1949, who showed that most boolean functions require circuits of exponential size. It was a natural step to restrict the circuits in size and depth and consider problems that are still computable on restricted circuits. Furst, Saxe and Sipser [11] for example showed that Parity is not in AC^0, but in NC^1. The first connection between circuits and Turing machines has been made by Savage [18]. It is generally believed that proving bounds using boolean circuits is easier than by using Turing machines.

While all known exact algorithms for NP-hard problems have a superpolynomial runtime in the worst case, there are numerous possible tricks and optimizations that can make these algorithms viable on special cases of the problem, even for large inputs. For example, there are yearly SAT-competitions where SAT-solver algorithms try to decide the satisfiability of SAT formulas from real-life applications, with many of these formulas containing millions of variables and tens of millions of clauses [2].

Heuristic solutions to hard problems have also been extensively studied. In particular, there is a wide range of so-called metaheuristics, which are general approaches and techniques for developing a heuristic solution to a problem. This includes some simpler approaches like local search or gradient descent, and also some more sophisticated ones like simulated annealing, tabu search or genetic algorithms [13]. Note that many techniques in machine learning are also developed for this purpose: to provide heuristic solutions to hard problems.

The 2-approximation algorithm for Vertex Cover has long been known, discovered by both Gavril and Yannakakis independently [12]. The FPTAS algorithm for Knapsack has also been around for a long time, and it is one of the most popular FPTAS examples [21].

Bin Packing has also been studied for multiple decades [12]. A recent analysis of the First Fit heuristic is available by Dósa and Sgall, proving the even stronger result that First Fit is a 1.7-approximation [8]. The variant which first sorts the items in decreasing order is known to even provide a $\frac{11}{9}$-approximation up to an additive constant, and this bound is known to be tight [7].

For Traveling Salesperson, a slightly improved version of our approximation algorithm is due to Christofides, which improves the approximation ratio to only 1.5 [5]. In the special case of an Euclidean TSP, there even exists a (rather complicated) PTAS algorithm, which received a Gödel prize [3].

The inapproximability result for Independent Set was the final result of a line of lower bounds in the early 2000s; John Hastad also received a Gödel prize for this result [15, 22].

This chapter was written in collaboration with Darya Melnyk, Pál András Papp, and Andrei Constantinescu.

Bibliography

[1] Complexity zoo. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

[2] The international SAT competition. `http://www.satcompetition.org`.

[3] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, September 1998.

[4] James A Carlson, Arthur Jaffe, and Andrew Wiles. *The millennium prize problems*. American Mathematical Society Providence, RI, 2006.

[5] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[6] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[7] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd(i) ¡ 11/9 opt(i)+6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11, Berlin, Heidelberg, 2007.

[8] György Dósa and Jiri Sgall. First Fit bin packing: A tight analysis. In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *LIPIcs*, pages 538–549, Dagstuhl, Germany, 2013.

[9] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[10] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[11] Merrick Furst, James B Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical systems theory*, 17(1):13–27, 1984.

[12] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[13] Fred W Glover and Gary A Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

[14] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[15] Johan Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 627–636. IEEE, 1996.

[16] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[17] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[18] John E Savage. Computational work and time on finite machines. *Journal of the ACM (JACM)*, 19(4):660–674, 1972.

[19] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[20] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[21] Vijay V Vazirani. *Approximation algorithms.* Springer Science & Business Media, 2013.

[22] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, page 681–690, New York, NY, USA, 2006.

Chapter 3

Cryptography

In Chapter 2 we learned that some functions are really hard to compute. This might seem like terrible news, but enables modern cryptography!

3.1 Perfect Encryption

We start with the oldest problem in cryptography: How can we send a secret message?

Definition 3.1 (Perfect Security)**.** *An encryption algorithm has perfect security, if the encrypted message reveals no information about the plaintext message to an attacker, except for the possible maximum length of the message.*

Remarks:

- If an encryption algorithm offers perfect security, any plaintext message of the same length could have generated the given ciphertext.
- Sometimes perfect security is also called information-theoretic security.
- Is there an algorithm that offers perfect security?

```
# $m$ = plaintext message Alice wants to send to Bob
# $k$ = random key known by Alice and Bob, with len($k$) = len($m$)
# $c$ = ciphertext, the encrypted message $m$

def encrypt_otp_Alice($m, k$)
    Alice sends $c = m \oplus k$ to Bob   # $\oplus$ = XOR

def decrypt_otp_Bob($c, k$)
    Bob computes $m' = c \oplus k$
```

Algorithm 3.2: One Time Pad

Remarks:

- In cryptography, it's always Alice and Bob, with a possible attacker Eve.

Theorem 3.3. *Algorithm 3.2 is correct.*

Proof. $m' = c \oplus k = (m \oplus k) \oplus k = m$. □

Theorem 3.4. *Algorithm 3.2 has perfect security.*

Proof. Given a ciphertext c, for every plaintext message m there exists a unique key k that decrypts c to m, that is $m = c \oplus k$. Therefore, if k is uniformly random, every plaintext is equally likely and thus, ciphertext c reveals no information about plaintext m. □

Remarks:

- Algorithm 3.2 only works if the message m has the same length as the key k. How can we encrypt a message of arbitrary length with a key of fixed length?
- Block ciphers process messages of arbitrary length by breaking them into fixed-size blocks and operating on each block.

```
# $m, k, c$ as defined earlier, now with len($k$) << len($m$)

def encrypt_ECB($m, k$)
    Split $m$ into $r$ len($k$)-sized blocks $m_1, m_2, \dots, m_r$
    for $i = 1, 2, 3, \dots, r$:
        $c_i = m_i \oplus k$
    $c = c_1; c_2; \dots; c_r$   # ; stands for concatenation
    return $c$
```

Algorithm 3.5: Electronic Code Book

Remarks:

- In Algorithm 3.5, blocks of the same plaintext result in the same ciphertext, because the same key k is reused to encrypt every block. Furthermore, reusing the same key reveals information about m_1 and m_2: Suppose you have two messages m_1, m_2 encrypted with the same key k, resulting in c_1, c_2. We now have $c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2$. So, reusing the same key k in Algorithm 3.2 is insecure. → notebook
- But there are better block based encryptions. CTR-AES (Advanced Encryption Standard with Counter Mode of Operation) is the current state of the art.
- For encryption, Alice and Bob need to agree on a key k first! While this may be feasible for, e.g., secret agents, it is quite impractical for everyday usage.

3.2 Key Exchange

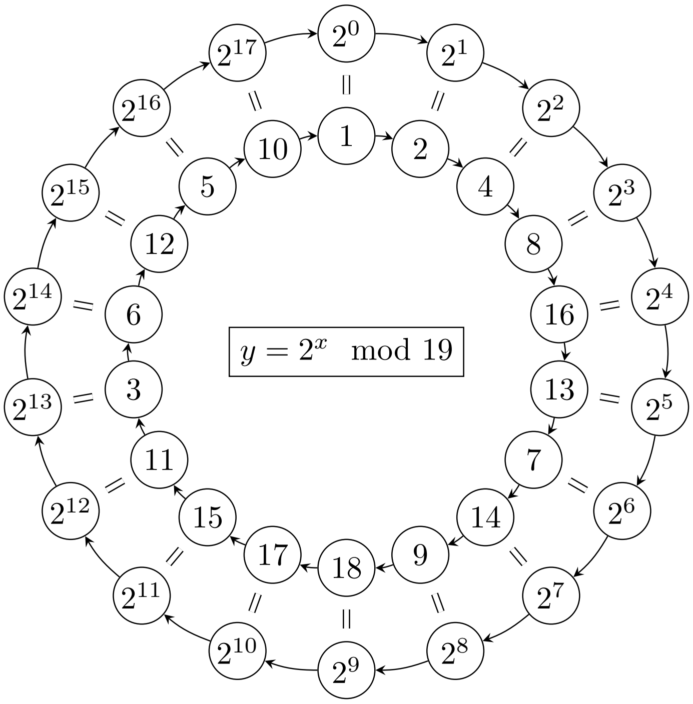
How to agree on a common secret key in public, if you never met before?

Definition 3.6 (Primitive Root). *Let $p \in \mathbb{N}$ be a prime. Then $g \in \mathbb{N}$ is a primitive root of p if the following holds: For every $y \in \mathbb{N}$, with $1 \leq y < p$, there is an $x \in \mathbb{N}$ such that $g^x = y \mod p$.*

→ notebook

Remarks:

- An example for $p = 19$ with $g = 2$:



→ notebook

```
# p = publicly known large prime number
# g = publicly known primitive root of p
def Diffie_Hellman_Alice():
    Pick a random secret key $a \in \{1, 2, \dots, p-1\}$
    Send $k_a = g^a \mod p$ to Bob
    Receive $k_b$ from Bob
    Calculate $k = (k_b)^a \mod p$

def Diffie_Hellman_Bob():
    # same as Alice, swapping all $a, b$.
```

Algorithm 3.7: Diffie-Hellman Key Exchange

Theorem 3.8. *In Algorithm 3.7, Alice and Bob agree on the same key k.*

Proof. Everything mod p, we have

$$k = (k_b)^a = \left(g^b\right)^a = g^{b\cdot a} = g^{a\cdot b} = (g^a)^b = (k_a)^b = k.$$

Actually, Alice receives $\left(g^b \bmod p\right)$ and computes $\left(g^b \bmod p\right)^a \bmod p$, however under modulo operations $\left(g^b \bmod p\right)^a = \left(g^b\right)^a$. In fact, all the following operations are well-defined in the modulo operation:

$$\begin{aligned}(a+b) \mod p &= ((a \bmod p) + (b \bmod p)) \mod p\\ (a-b) \mod p &= ((a \bmod p) - (b \bmod p)) \mod p\\ (a\cdot b) \mod p &= ((a \bmod p) \cdot (b \bmod p)) \mod p\\ (g^a \bmod p)^b \bmod p &= (g^a)^b \bmod p\end{aligned}$$

□

Remarks:

- Algorithm 3.7 does not have perfect security, but instead only computational security.

Definition 3.9 (Computational Security)**.** *An algorithm has computational security, if it is secure against any adversary with polynomial computational resources.*

Remarks:

- The definition of security differs from one cryptographic primitive to another (e.g., encryption, signatures, etc.), but they are typically reduced to the difficulty of a computational problem.
- The computational security of Algorithm 3.7 is based on the difficulty of the discrete logarithm.

Problem 3.10 (Discrete Logarithm or DL)**.** *Given a prime* $p \in \mathbb{N}$*, a primitive root* g *of* p*, and* $y \in \mathbb{N}$ *with* $1 \le y < p$*, find an* $x \in \mathbb{N}$ *such that* $g^x = y \mod p$*.*

Remarks:

- In Algorithm 3.7, an adversary overhears g^a and g^b and wants to learn the secret key g^{ab}. But it is unknown whether the adversary actually needs to know how to compute the discrete logarithm to extract the key. Maybe there is another way. Therefore, the following (stronger) assumption captures the security of the protocol better.

Problem 3.11 (Computational Diffie Hellman or CDH)**.** *Given a prime* $p \in \mathbb{N}$*, a primitive root* g *of* p*, and* $g^a, g^b \in \mathbb{N}$ *with* $1 \le g^a, g^b < p$*, compute* $g^{ab} \mod p$*.*

Lemma 3.12. *CDH* $\le$ *DL.*

Proof. We just compute the discrete logarithms of g^a, g^b and then compute g^{ab}. □

Remarks:

- The discrete logarithm (resp. computational Diffie-Hellman) assumption states that it is infeasible to solve DL (CDH) with computationally bounded resources.
- We have no proof that CDH (or DL) is hard, but there is also no known efficient algorithm.
- It is not known whether the opposite direction (DL $\leq$ CDH) holds, though in certain special cases it does.
- Conversely, modular exponentiation can be done in polynomial time using repeated squaring. → notebook
- So far, we have assumed the adversary only listens on the communication channel. This is known as passive security.
- What about stronger adversaries?

Definition 3.13 (Man in the Middle Attack)**.** *A man in the middle attack is defined as an adversary Eve deciphering or changing the messages between Alice and Bob, while Alice and Bob believe they are communicating directly with each other.*

Theorem 3.14. *The Diffie-Hellman Key Exchange from Algorithm 3.7 is vulnerable to a man in the middle attack.*

Proof. Assume that Eve can intercept and relay all messages between Alice and Bob. That alone does not make it a man in the middle attack, Eve needs to be able to decipher or change messages without Alice or Bob noticing. Indeed, Eve can emulate Alice's and Bob's behavior to each other, by picking her own a', b', and then agreeing on common keys $g^{a \cdot b'}$, $g^{b \cdot a'}$ with Alice and Bob, respectively. Thus, Eve can relay all messages between Alice and Bob while deciphering and (possibly) changing them, while Alice and Bob believe they are securely communicating with each other. □

Remarks:

- It is a bit like concurrently playing chess with two grandmasters: If you play white and black respectively, you can essentially let them play against each other by relaying their moves.
- How do we fix this? One idea is to personally meet in private first, exchange a common secret key, and then use this key for secure communication. However, having a key already completely defeats the purpose of a key exchange algorithm.
- Can we do better? Yes, with public key cryptography.

3.3 Public Key Cryptography

Definition 3.15 (Public Key Cryptography)**.** *A public key cryptography system uses two keys per participant: A public key k_p, to be disseminated to everyone, and a secret (private) key k_s, only known to the owner. A message encrypted with the public key of the intended receiver can be decrypted only with the corresponding secret key. Also, messages can be digitally signed; a message verifiable with a public key must have been signed with the corresponding secret key.*

Remarks:

- Popular public key cryptosystems include RSA, Elliptic Curve Cryptography, etc.
- We study a public key cryptosystem based on the DL problem.
- In Diffie-Hellman Key Exchange algorithm (Algorithm 3.7), Alice picked a secret number a and computed a public number $k_a = g^a \mod p$ which Alice sent to Bob. We use the exact same idea in Algorithm 3.16 to generate a pair of public and secret keys (k_p, k_s).

```
# $p, g$ as defined earlier

def generate_key():
    Pick a random secret key $k_s \in \{1, 2, \ldots, p-1\}$
    $k_p = g^{k_s} \mod p$
    return $k_p, k_s$
```

Algorithm 3.16: Key Generation

3.4 Public Key Encryption

Public key or asymmetric encryption schemes allow users to send encrypted messages directly.

Definition 3.17. *A public key encryption scheme is a triple of algorithms:*

- *A key generation algorithm that outputs a public/secret key pair k_p, k_s.*
- *An encryption algorithm that outputs the encryption c of a message m using the receiver's public key k_p.*
- *A decryption algorithm that outputs the message m using the secret key k_s.*

```
# $p, g, m, k_p, k_s$ as defined earlier

```

```
def encrypt($m, k_p$):
    Pick a random nonce $x \in \{1, 2, \ldots, p-1\}$
    $c_1 = g^x \mod p$
    $c_2 = m \cdot k_p^x \mod p$
    return $c_1, c_2$   # encryption $c = (c_1, c_2)$

def decrypt($c_1, c_2, k_s$):
    $m' = c_2 \cdot c_1^{k_s \cdot (p-2)} \mod p$
    return $m'$
```

Algorithm 3.18: ElGamal Encryption Algorithm

Theorem 3.19. *ElGamal encryption scheme (Algorithms 3.16, 3.18) is correct.*

Proof. Alice can recover the message: $m' = c_2 \cdot c_1^{k_s \cdot (p-2)} = \left(m \cdot k_p^x\right) \cdot g^{x \cdot k_s \cdot (p-2)} = m \cdot \left(k_p^x\right)^{p-1} = m$. The last step uses the following theorem. □

Theorem 3.20 (Fermat's Little Theorem)**.** *Let p be a prime number. Then, for any $x \in \mathbb{N}$: $x^p = x \mod p$. If x is not divisible by p, then $x^{p-1} = 1 \mod p$.*

Remarks:

- What about the security of ElGamal encryption? In the context of public encryption schemes, we want that an adversary who listens in the communication channel to not be able to extract the message m.

Problem 3.21 (Breaking-ElGamal-Encryption)**.** *Given $(c_1, c_2) = (g^x, m \cdot g^{k_s \cdot x})$ and the public key $k_p = g^{k_s}$, compute the message m.*

Remarks:

- The computational security of Algorithm 3.18 is based on the difficulty of CDH.

Theorem 3.22. *CDH $\leq$ Breaking-ElGamal-Encryption*

Proof. Given (g^a, g^b), we create a problem instance for Breaking-ElGamal-Encryption by setting $c_1 = g^a$, c_2 a random value and $k_p = g^b$. From the definition of the problem we can infer that $c_1 = g^x = g^a$, $k_p = g^{k_s} = g^b$ and thus $c_2 = m \cdot g^{k_s \cdot x} = m \cdot g^{a \cdot b}$. We assume that we can break ElGamal, so we know m. Now we simply compute $m^{p-2} \cdot c_2$ and we get

$$m^{p-2} \cdot c_2 = m^{p-2} \cdot m \cdot g^{a \cdot b} = m^{p-1} \cdot g^{a \cdot b} = g^{a \cdot b}.$$

□

Remarks:

- In other words, we have shown that CDH is “easier” than Breaking-ElGamal-Encryption. As long as the CDH is hard, so is Breaking-ElGamal-Encryption.
- The other direction (Breaking-ElGamal-Encryption $\leq$ CDH) holds as well.
- However, there is a problem with reductions: An encryption scheme that reveals 90% of the plaintext is still considered secure with a reduction approach, as long as the remaining 10% is hard to find. This is clearly unacceptable. We want extracting even a single bit of information to be difficult.
- Both CDH and DL assumptions are not enough to show this. The decisional Diffie Hellman assumption is an even stronger assumption that we rely on.

Problem 3.23 (Decisional Diffie-Hellman or DDH)**.** *Given a prime* $p \in \mathbb{N}$*, a primitive root* g *of* p*, and* $g^a, g^b, g^c \in \mathbb{N}$ *with* $1 \leq g^a, g^b, g^c < p$*, decide if* $c = a \cdot b$*.*

Remarks:

- Note that DDH $\leq$ CDH.
- How can we prove security of ElGamal based on the DDH assumption?
- The idea is to compare ElGamal to a perfectly secure scheme. We have seen an example for perfect security (OTP) which uses XOR. To make the protocols comparable, we now present another version of OTP that uses modulo computation.

```
# $p, g, m, k_p, k_s$ as defined earlier
# $k$ = random key known by Alice and Bob (shared secret)

def encrypt_modulo_otp_Alice($m, k$)
    Pick a random nonce $x \in \{1, 2, \ldots, p-1\}$
    $c_1 = g^x \mod p$
    $c_2 = m \cdot k \mod p$
    return $c_1, c_2$   # encryption $c = (c_1, c_2)$

def decrypt_modulo_otp_Bob($c_1, c_2, k$)
    Bob computes $m' = c_2 \cdot k^{p-2} \mod p$
```

Algorithm 3.24: Modulo-OTP

Theorem 3.25. *Algorithm 3.24 is correct.*

Proof. $m' = c_2 \cdot k^{p-2} = (m \cdot k) \cdot k^{p-2} = m \cdot k^{p-1} = m.$ □

Theorem 3.26. *Algorithm 3.24 has perfect security.*

Proof. Given a ciphertext c, for every plaintext message m there exists a unique key k that decrypts c_2 to m, that is $m = c_2 \cdot k^{p-2}$. Therefore, if k is uniformly random, every plaintext is equally likely and thus, ciphertext c_2 reveals no information about plaintext m. □

Remarks:

- Note that the key c_1 is not used at all for encryption and thus is independent of the message. The sole reason why we have that is to make a protocol very similar to ElGamal which we will use in the following proof.
- The idea is to show that in presence of polynomially bounded adversaries ElGamal is just as hard as Modulo-OTP. Since Modulo-OTP is impossible to crack, so is ElGamal (under DDH assumption).

Problem 3.27 (Distinguish(Modulo-OTP, ElGamal))**.** *Given a protocol where Alice sends an encrypted message to Bob, decide whether Alice and Bob are using Modulo-OTP (Algorithm 3.24) or ElGamal (Algorithm 3.18).*

Theorem 3.28. *DDH $\leq$ Distinguish(Modulo-OTP, ElGamal)*

Proof. Given g^a, g^b, g^c, we create an instance for Distinguish(Modulo-OTP, ElGamal), where we set $g^a = g^{k_s}$ and $g^b = g^r$ (both publicly known) and we encrypt any message m with g^c, i.e. $c_2 = g^c \cdot m$. If $g^c = g^{ab}$, then we have the exact situation of ElGamal encryption, since there we encrypt by using $c_2 = g^{k_s \cdot r} \cdot m$. If $g^{ab} \neq g^c$, then we have the exaction situation of Modulo-OTP encryption, where we can set $k = g^c$ since g^c has no relation to public information like g^a and g^b. □

Remarks:

- ElGamal-Encryption has some other features, e.g., it is homomorphic.

Definition 3.29 (Homomorphic Encryption Schemes)**.** *An encryption scheme is said to be homomorphic under an operation $*$ if $E(m_1 * m_2) = E(m_1) * E(m_2)$.*

Remarks:

- In other words, we can directly compute with encrypted data!
- $m_1 * m_2$ and $E(m_1) * E(m_2)$ indicates that ciphertexts and messages can both be operated upon using the same operation. This depends on the representation of ciphertexts, and is not always precisely defined. In the case of ElGamal encryption's homomorphism, we use pairwise vector multiplication to multiply ciphertexts: $E(m_1) \cdot E(m_2) = (c_{11}, c_{12}) \cdot (c_{21}, c_{22})^T$.

Lemma 3.30. *The ElGamal encryption scheme (Algorithms 3.16, 3.18) is homomorphic under modular multiplication.*

Proof. We refer the encryption of message m with public key k_p, large prime p, generator g, and a random nonce x as $E(m) = (c_1, c_2) = (g^x, m \cdot k_p^x)$

$$\begin{aligned} E(m_1) \cdot E(m_2) &= (g^{x_1}, m_1 \cdot k_p^{x_1}) \cdot (g^{x_2}, m_2 \cdot k_p^{x_2})^T \\ &= (g^{x_1+x_2}, (m_1 \cdot m_2) k_p^{x_1+x_2}) = E(m_1 \cdot m_2) \end{aligned}$$

□

Remarks:

- Not every public encryption scheme is homomorphic under all operations. If an encryption scheme is homomorphic only under some operations, it's called a *partial homomorphic encryption scheme.* For example, we have:
  - Modular multiplication: ElGamal cryptosystem, RSA cryptosystem.
  - Modular addition: Benaloh cryptosystem, Pallier cryptosystem.
  - XOR operations: Goldwasser–Micali cryptosystem.
- There are fully homomorphic encryption schemes that support all possible functions, like Craig Gentry's lattice-based cryptosystem.
- Homomorphic encryption is used in electronic voting schemes to sum up encrypted votes.

3.5 Digital Signatures

Definition 3.31 (Digital Signature Scheme)**.** *A digital signature scheme is a triple of algorithms:*

- *A key generation algorithm that outputs a public/secret key pair k_p, k_s.*
- *A signing algorithm that outputs a digital signature σ on message m using a secret key k_s.*
- *A verification algorithm that outputs* `True` *if the signature σ on the message m is valid using the public key k_p of the signer, and* `False` *otherwise.*

Definition 3.32 (Correctness)**.** *A signature scheme is correct if the verification algorithm on input σ, m, k_p returns* `True` *only if σ is the output of the signing algorithm on input m, k_s.*

Remarks:

- All algorithms (key generation, signing, and verification) should be efficient, i.e., computable in polynomial time.
- Digital signatures offer *authentication* (the receiver can verify the origin of the message), *integrity* (the receiver can verify the message has not been modified since it was signed), and *non-repudiation* (the sender cannot falsely claim that they have not signed the message).

- Widely known signature schemes are ElGamal, Schnorr, and RSA.

```
1  # $p, g, m$ as defined earlier
2  # $h$ = cryptographic hash function like SHA256
3  # $k_p, k_s$ = Alice's public/secret key pair
4  # $s, r$ = the signature sent by Alice
5
6  def sign_Alice($m, k_s$):
7      Pick a random $x \in \{1, 2, \ldots, p-2\}$
8      $r = g^x \mod p$
9      $s = x \cdot h(m) - k_s \cdot r \mod p-1$
10     return $s, r$   # signature $\sigma = (s, r)$
11
12 def verify_Bob($m, s, r, k_p$):
13     return $r^{h(m)} == k_p^r \cdot g^s \mod p$
```

Algorithm 3.33: ElGamal Digital Signatures

Remarks:

- The key generation algorithm for ElGamal signatures ElGamal encryption scheme (Algorithm 3.16).

Theorem 3.34. *The ElGamal digital signature scheme (Algorithms 3.16, 3.33) is correct.*

Proof. The algorithm is correct, meaning that a signature generated by (an honest) Alice will always be accepted by Bob. That is because,

$$k_p^r \cdot g^s = g^{k_s \cdot r} \cdot g^{x \cdot h(m) - k_s \cdot r} = g^{k_s \cdot r + x \cdot h(m) - k_s \cdot r} = g^{x \cdot h(m)} = r^{h(m)} \mod p.$$

□

Remarks:

- The random variable x in Line 7 is often called a *nonce* – a number only used once.

- Writing `mod` $p-1$ in Line 9 is not a typo. In the exponent, we always compute modulo $p-1$, since that will make sure that values larger than $p-1$ will be truncated (Theorem 3.20).

- The function $h()$ in Line 9 is a so-called cryptographic hash function. If we did not use $h()$, we had a problem:

Theorem 3.35. *ElGamal signatures without cryptographic hash functions are vulnerable to existential forgery.*

Proof. Let s, r be a valid signature on message m. Then, $(s', r') = (sr, r^2)$ is a valid signature on message $m' = rm/2$ (as long as either m or r is even), because → notebook

$$k_p^{r'} \cdot g^{s'} = \left(g^{k_s}\right)^{r^2} \cdot g^{s\cdot r} = g^{r^2\cdot k_s} \cdot g^{r\cdot(x\cdot m-r\cdot k_s)} = g^{r^2\cdot k_s+r\cdot x\cdot m-r^2\cdot k_s} =$$
$$g^{r\cdot x\cdot m} = (g^x)^{r\cdot m} = r^{2r\cdot m/2} = \left(r^2\right)^{r\cdot m/2} = (r')^{m'} \mod p.$$

□

Remarks:

- Existential forgery is the creation of at least one message-signature pair (m, s), when m was never signed by Alice. Hashing the message in the computation makes the inversion difficult.
- Craig Wright used Satoshi Nakamoto's key in Bitcoin and signed a random message attempting to impersonate the famous creator of Bitcoin. However, when Wright was asked to sign "I am Satoshi" he could not deliver!
- So what is a cryptographic hash function?

3.6 Cryptographic Hashing

Definition 3.36 (Cryptographic Hash Function)**.** *A cryptographic hash function is a function that maps data of arbitrary size to a bit array of a fixed size (the **hash value** or **hash**). A cryptographic hash function is called one-way if it is easy to compute but hard to invert and a cryptographic hash function is called collision-resistant if it is difficult to find two different values that are mapped to the same hash.*

Remarks:

- A hash function is deterministic: the same message always results in the same hash value.
- SHA2, SHA3 (Secure Hash Algorithm 2/3), RIPMED, and BLAKE are some example families of cryptographic hash functions. SHA256 is a specific implementation of the SHA2 construction which outputs a 256 bit output for arbitrary sized inputs. Earlier constructions like MD5 or SHA1 are considered broken/weak now.
- One-way and collision-resistance properties can be phrased as computational problems.

Problem 3.37 (Collision)**.** *Find two different values x and x' such that $h(x) = h(x')$.*

Problem 3.38 (Inversion)**.** *Given a value y, find a value x such that $h(x) = y$.*

Lemma 3.39. *Collision $\leq$ Inversion.*

Proof. Let us choose some random value x and compute $h(x)$. If we invert $h(x)$, it is highly likely we get a value x' such that $x' \neq x$ (collision), since hash functions maps data of arbitrary size to fixed-size values, and hence there many potential inputs. □

Lemma 3.40. *Existence of one-way functions* $\Rightarrow \mathsf{P} \neq \mathsf{NP}$.

Proof. Suppose there exists a function h that is one-way. We define a decision problem (Definition 2.2) as follows: Given an input $(\bar{x}, y)$, decide whether there is a input x such that $h(x) = y$ with $\bar{x}$ being a prefix of x. This decision problem is in NP: Given $(\bar{x}, y)$ as input and x as possible solution, one can check in polynomial time that $h(x) = y$.

If the decision problem was in P, we could invert h one bit at a time. We start by checking whether $(0, y)$ or $(1, y)$ is true. Whichever is true determines the initial prefix $\bar{x}$. We continue to adding a bit to $\bar{x}$ such that the decision with that added bit continues to be true. This way we construct all bits of x. But since we assumed that h was one-way, the decision problem is not in P. We have now a decision problem that is in NP but not in P and thus $\mathsf{P} \neq \mathsf{NP}$. □

Remarks:

- Since we do not know $\mathsf{P} \neq \mathsf{NP}$, we even less know whether one-way functions exist. But that does not keep us from using them.
- What about the security of digital signatures? In the context of digital signatures, only the owner of the secret key should be able to produce valid signatures.

Problem 3.41 (Forging-ElGamal-Signatures). *For a given digital signature scheme, produce a valid message-signature pair without having access to the secret key.*

Remarks:

- The computational security of Algorithm 3.33 is based on the difficulty of inverting one-way functions and computing the discrete logarithm. Intuitively, to forge a signature, a malicious Bob can either find a collision in the hash function, $h(m) = h(m') \mod p-1$, or extract Alice's secret key k_s. Therefore, Bob must either solve Collision or DL. Both problems are assumed to be hard. However, there is no proof that these are the only ways of forging ElGamal digital signatures.

Conjecture 3.42. *(Collision or DL)* $\leq$ *Forging-ElGamal-Signatures*

Remarks:

- Note that Alice must choose a different x for *each* signature, keeping x secret. Otherwise, security can be compromised. In particular, if Alice uses the same nonce x and secret key k_s to sign two different messages, Bob can compute k_s.
- Similarly to digital signatures, message authentication codes are used to ensure a message received by Bob is indeed sent by Alice.

Definition 3.43 (Message Authentication Code or MAC)**.** *A message authentication code is a bitstring that accompanies a message. It can be used to verify the authenticity of the ciphertext in combination with a secret authentication key k_a (different from k) shared by the two parties.*

Remarks:

- Eve should not be able to change the encrypted message and/or the MAC, and get Bob to believe that Alice sent the encrypted message.
- MACs are symmetric, i.e., they are generated and verified using the same secret key.
- Algorithm 3.44 shows a hash based MAC construction.

```
# $m, k_p, c$ as defined earlier
# $k_a$ = key to authenticate $c$

def encrypt_then_MAC($m, k_p, k_a$):
    $c$ = encrypt($m, k_p$)
    $a = h(k_a; c)$
    return $c, a$
```

Algorithm 3.44: Hash Based Message Authentication Code

Remarks:

- Bob accepts a message c only if he calculates $h(k_a; c) = a$.
- With some hash functions (e.g., SHA2), it is easy to append data to the message and obtain another valid MAC without knowing the key. To avoid these attacks, in practice we use $h(k_a; h(k_a; c))$.
- Now Alice and Bob can securely communicate over the insecure communication channels of the internet, due to the known public keys.
- But how does Bob know that Alice's public key really belongs to Alice? What if it is really Eve's key? Quoting Peter Steiner: "*On the Internet, nobody knows you're a dog.*"

3.7 Public Key Infrastructure

"*Love all, trust a few.*" – William Shakespeare

What can we do, unless we personally meet with everyone to exchange our public keys? The answer is trusting a few, in order to trust many.

Remarks:

- Let's say that you don't know Alice, but both Alice and you know Doris. If you trust Doris, then Doris can verify Alice's public key for you. In the future, you can ask Alice to vouch for her friends as well, etc.
- Trust is not limited to real persons though, especially since Alice and Doris are represented by their keys. How do you know that you give your credit card information to a shopping website, and not an attacker? You probably don't know the owner of the shopping website personally.

Definition 3.45 (Public Key Infrastructure or PKI)**.** *Public Key Infrastructure (PKI) binds public keys with respective identities of entities, like people and organizations. People and companies can register themselves with a certificate authority.*

Definition 3.46 (Certificate Authority or CA)**.** *A certificate authority is an entity whose public key is stored in your hardware device, operating system, or browser by the respective vendor like Apple, Google, Microsoft, Mozilla, Ubuntu, etc.*

Remarks:

- A certificate is an assertion that a known real world person, with a physical postal address, a URL, etc. is represented by a given public key, and has access to the corresponding secret key.
- You can accept a public key if a certificate to that effect is signed by a CA whose public key is stored in your device.
- CA's whose public keys are stored in your device are also called root CA's. Sometimes, there are intermediate CA's whose certificates are signed by root CA's, and who can sign many other end-user certificates. This enables scaling, but also introduces vulnerabilities.
- If a CA's secret key is compromised by a malicious actor, they can sign themselves a certificate saying that they are someone else (say, Google), and then impersonate Google to innocent browsers which trust this CA. A CA's key can be revoked if this happens, or CA's keys can have shorter expiry times.
- Another problem is that your own set of root certificates might be compromised, e.g., if malicious software replaces your browser's root certificates with fakes.

3.8 Transport Layer Security

To communicate securely over the internet, we simply combine the cryptographic primitives we learned so far!

Remarks:

- Alice and Bob don't want Eve to be able to read their messages. Therefore, they encrypt their messages using block based encryption (Section 3.1).
- For the encryption algorithm, they need to agree on a secret key using a key exchange protocol (Section 3.2).
- When Alice receives a message, how can she be sure that the message hasn't been modified on the way from Bob to her? Alice and Bob use message authentication (Section 3.5) to ensure integrity of the communication.
- Let's assume that Alice hasn't met Bob in person before. How can she be sure that she is really communicating with Bob and not with Eve? She would ask Bob to authenticate himself (Sections 3.5, 3.7).

Protocol 3.47 (Transport Layer Security, TLS)**.** *TLS is a network protocol in which a client and a server exchange information in order to communicate in a secure way. Common features include a bulk encryption algorithm, a key exchange protocol, a message authentication algorithm, and lastly, the authentication of the server to the client.*

Remarks:

- TLS is the successor of Secure Sockets Layer (SSL).
- HTTPS (Hypertext Transfer Protocol Secure) is not a protocol on its own, but rather denotes the usage of HTTP via TLS or SSL.
- What other problems can we solve using crypto? The answer is surprisingly many! In the next sections we will discuss some of the most exciting cryptographic primitives beyond TLS.

3.9 Zero-Knowledge Proofs

Problem 3.48 (Waldo)**.** *Peggy and Vic play Where's Waldo. Can Peggy prove she found Waldo without revealing Waldo's location to Vic?*

Remarks:

- In the physical world, Peggy can cover the picture with a large piece of cardboard that has a small, Waldo-shaped hole in its center. She can then place the cardboard such that only Waldo is visible through the hole and therefore prove to Vic she has found Waldo without revealing any information regarding Waldo's location.
- In Zero-Knowledge Proofs (ZKP), the *prover*, Peggy, wants to convince the *verifier*, Vic, of the knowledge of a secret without revealing any information about the secret to Vic.

Definition 3.49 (Zero-Knowledge Proof)**.** *A pair of probabilistic polynomial time interactive programs P, V is a zero-knowledge proof if the the following properties are satisfied:*

- ***Completeness:*** *If the statement is true, then an honest verifier V will be convinced by an honest prover P.*
- ***Soundness:*** *If the statement is false, a cheating prover P cannot convince the honest verifier V that it is true, except with negligible probability.*
- ***Zero-knowledge:*** *If the statement is true, a verifier V learns nothing beyond the statement being true.*

Remarks:

- Soundness concerns the security of the verifier, and zero-knowledge the security of the prover.

Problem 3.50 (Color-Blind)**.** *Vic has two spheres: one red and one blue. Vic is color-blind and thus he cannot differentiate between these two spheres; they look exactly the same to him. Peggy, on the other hand, is not color-blind and wants to prove to Vic that she can differentiate between these two spheres. How can she do this?*

Remarks:

- Peggy wants to convince Vic in zero-knowledge, meaning that she wants to give Vic absolutely no additional information except the fact that she can differentiate these two spheres. For example, she does not want Vic to know which sphere is red and which one is blue.

```
# n = security parameter
def ColorBlind(n):
    repeat n times:
        Vic takes the spheres behind his back
        Vic either switches the spheres or not
        Vic shows the spheres to Peggy
        Peggy answers whether Vic switched the spheres or not
```

Algorithm 3.51: Color-blind

Theorem 3.52. *Algorithm 3.51 is complete.*

Proof. If Peggy knows how to differentiate the spheres (i.e. if she is not color-blind) and Vic behaves according to the protocol, Peggy can always tell whether Vic switched the spheres or not. □

Theorem 3.53. *Algorithm 3.51 is sound.*

Proof. If Peggy cannot differentiate the colors, then she has only 50% of guessing correctly whether Vic switched the spheres or not. In such a case, Peggy is caught with a probability 1/2, and the probability that she survives n rounds of the protocol undetected is negligible (2^{-n}).

□

Remarks:

- The main intuition is that Peggy's answers do not reveal anything about the spheres. In each round, Peggy simply answers whether the spheres are switched or not, and none of these answers tell which one is red and which one is blue. It seems that this proves that our protocol is zero knowledge, but...

Theorem 3.54. *Algorithm 3.51 is not zero-knowledge.*

Proof. Consider a malicious Vic who has not 2 but 4 spheres: One red and one blue (as before), plus a second set of spheres: Red and Blue. These upper-case spheres look exactly the same as the lower-case spheres, but Vic knows their color, because somebody told him. In the first round Vic shows the red and blue spheres, just as in the normal protocol. In the second round however, Vic shows the Red and Blue spheres. Now Peggy's answer (switched or not) will directly reveal the color of the original (lower-case) spheres to Vic. So the protocol was not zero-knowledge. □

Remarks:

- Peggy is interested to show that the protocol is not zero-knowledge. So she comes up with a "movie script" to prove that Vic can learn something. She gives the script to Vic. Vic is the director, the only actor (Vic also plays Peggy), and the cutter. If Vic can produce any movie Peggy wants to see, the protocol is zero-knowledge.

- The color-blind protocol is not zero-knowledge because Vic could not produce the movie with the 4 spheres.

Definition 3.55. *A protocol consisting of a pair of probabilistic polynomial time interactive programs P, V is zero-knowledge, if every script requested by P can be generated by V.*

Remarks:

- An example of a ZKP is Hamiltonian Cycle (HC), see Problem 2.68. HC is particularly interesting because HC is NP-complete. This means that a ZKP for HC can thus be used as a ZKP for every problem in NP.

```
# n = security parameter
# G = large graph
def ZKP_HamiltonianCycle(G):
    repeat $n$ times:
        Peggy creates graph $H$ = permutation of $G$
        Peggy hides each entry of $H$
        Vic tosses a coin $c$ = [perm, cyc]
        if $c$ == perm:
            Peggy opens $H$ and gives the permutation
            Vic verifies that it is the original graph $G$
        elif $c$ == cyc:
            Peggy opens only the entries of the cycle
            Vic verifies it is a cycle
```

Algorithm 3.56: Hamiltonian Cycle ZKP

Remarks:

- A permutation H of G is the same graph, but we only permute the names of the nodes of G.

- If $c =$ `cyc`, the prover should be able to open only the cycle. This can be done, for example, by using a matrix representation of graphs as illustrated in Figure 3.59.

Theorem 3.57. *Algorithm 3.56 is complete.*

Proof. If Peggy knows the cycle in G, she can satisfy Vic's demand in both cases. In case $c =$ `perm`, Peggy opens the whole matrix and also returns the renaming of G's nodes in H. In case $c =$ `cyc`, Peggy can easily construct and return a cycle in H by applying the permutation in the original cycle in G. □

Theorem 3.58. *Algorithm 3.56 is sound.*

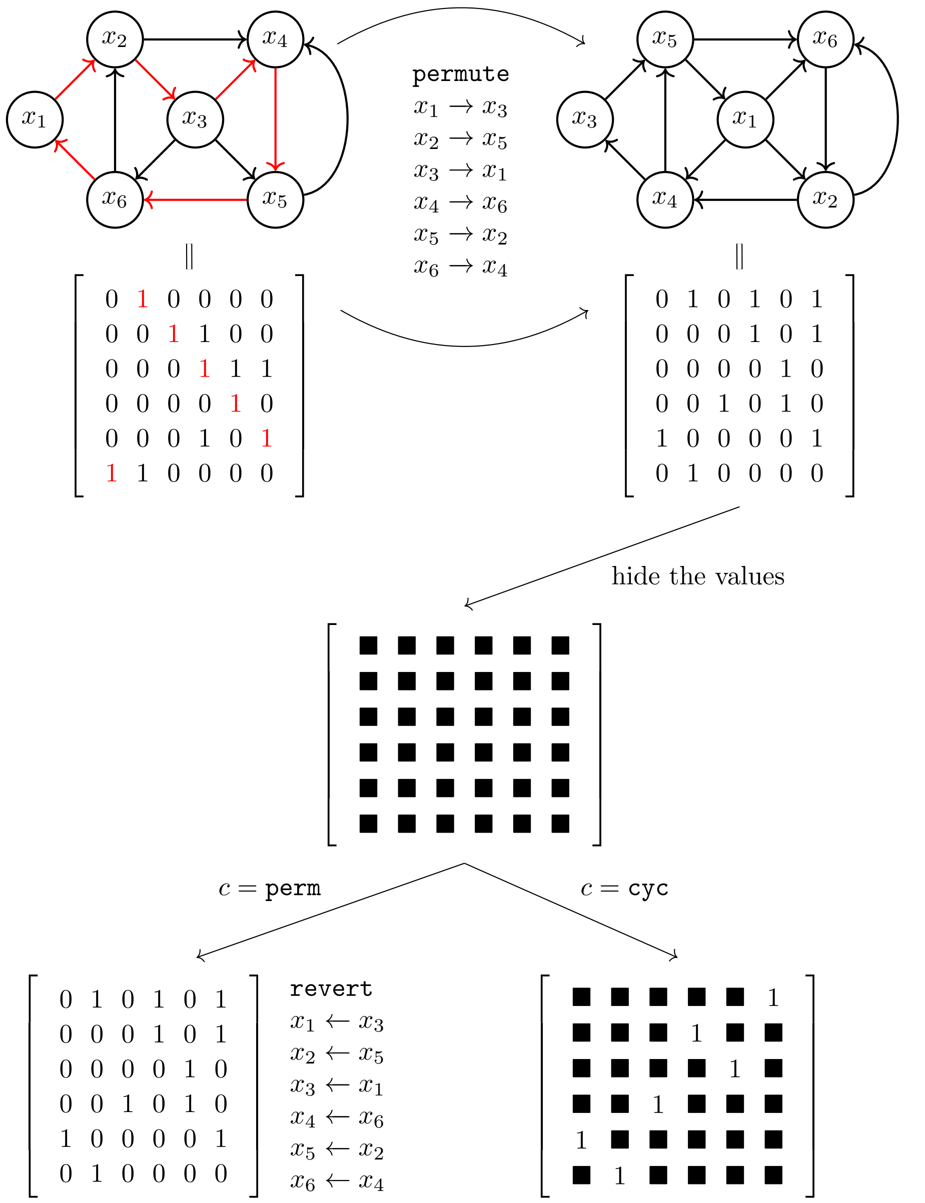
Proof. If someone knew how to answer both questions, then they can construct the cycle: Take the cycle in H and do the reverse permutation to get the cycle in G.

If Peggy does not know the cycle, the previous argument implies that she can only answer one of these questions. In such a case, Peggy is caught with probability 1/2, and the probability that she survives n rounds of the protocol undetected is negligible (2^{-n}).

□

Remarks:

- What about the zero-knowledge property? The main intuition is that Peggy's answers do not reveal the original cycle in G. In each round, if the challenge is $c =$ `perm` Vic only sees a permutation of G. On the other hand, if the challenge is $c =$ `cyc`, Vic sees that there is a cycle in the hidden graph. However, for all we know, the hidden matrix could be filled all with ones. So in both cases, zero information about the cycle is revealed. As explained in Theorem 3.54, such an argument is not enough. In the following we prove that HC is zero-knowledge based on our Definition 3.55.



$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare & 1 \\ \blacksquare & \blacksquare & \blacksquare & 1 & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare & 1 & \blacksquare \\ \blacksquare & \blacksquare & 1 & \blacksquare & \blacksquare & \blacksquare \\ 1 & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & 1 & \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{bmatrix}$$

Figure 3.59: An illustration of the ZKP for Hamiltonian Cycle

Theorem 3.60. *Algorithm 3.56 is zero-knowledge.*

Proof. We only sketch this proof – formally proving zero-knowledge is usually difficult and beyond the scope of this lecture. We need to show that Vic can produce any script Peggy throws at him. Vic uses his cutter abilities. He can produce two scenes, one hiding a random permutation of G, the other hiding a matrix of 1's only. If the script demands a permutation, Vic shows the permutation scene. If the script demands a cycle, Vic shows the scene revealing a cycle in the 1-only matrix. If Vic showed the wrong scene, he will cut it and do the other one instead. □

Remarks:

- But how can we hide the entries of a matrix? We can use commitment schemes.

3.10 Commitment Schemes

Commitment schemes are the digital analogue of a safe.

Definition 3.61 (Commitment Scheme)**.** *A commitment scheme is a two-phase interactive protocol between Alice, the sender, and Bob, the receiver.*

- ***Commit phase:*** *Alice commits to a message m by producing a public commitment c and a secret decommitment d. Alice sends c to Bob.*
- ***Reveal phase:*** *Alice sends m and d to Bob. Bob verifies that the message m corresponds to the commitment c.*

A commitment scheme must be correct, binding and hiding.

- ***Correctness:*** *If both Alice and Bob follow the protocol, then Bob always returns* `True` *in the reveal phase.*
- ***Hiding:*** *A commitment scheme is hiding if Bob cannot extract any information about the committed message before the reveal phase.*
- ***Binding:*** *A commitment scheme is binding if Alice cannot change her commitment after the commit phase.*

Remarks:

- Is there a simple way to create a commitment scheme? How about using hash functions?

```
# $m, h$ as defined earlier
# $n$ = security parameter

def commit_Alice($m$):
    Pick a random $n$-bit string $r$
    Send $c = h(r;m)$ to Bob

def reveal_Bob($c, m, r$): # Bob receives $m, r$ from Alice
    return $c == h(r;m)$
```

Algorithm 3.62: A Simple Commitment

Theorem 3.63. *Any cryptographic hash function can produce a (computationally binding and computationally hiding) commitment scheme.*

Proof. Assume Alice committed to a value m by sending $h(r, m)$.

- Computationally binding: To change her commitment, Alice needs to find a collision (i.e. an r', m') such that $h(r'; m') = h(r; m)$. This is hard for a computationally bounded Alice.

- Computationally hiding: In order for Bob to find what is the committed value, he needs to invert the given hash function. This is hard for a computationally bounded Bob.

□

Remarks:

- One might think that this scheme is perfectly hiding, since there are infinitely many values that could be committed. We then deduce that even if Bob has unlimited computational power he cannot find the committed value. This is not the case, because we have not specified how the hash function looks like. For example, it might happen that for a particular value there is no collision, and thus Bob with unlimited computational power would be able to find the committed value.

- What if Alice or/and Bob are computationally unbounded?

```
# $p, g, m, n$ as defined earlier
# $y$ = a random value in $\{1, 2, \ldots, p-1\}$
# $x$ with $y = g^x \mod p$ is unknown

def commit_Alice($m$):
    Pick a random $r \in \{1, 2, \ldots, p-1\}$
    $c = g^m \cdot y^r \mod p$
    Send $c$ to Bob

def reveal_Bob($m, c, r$): # Bob receives $m, c, r$ from Alice
    return $c == g^m \cdot y^r \mod p$
```

Algorithm 3.64: Pedersen Commitment

Theorem 3.65. *Pedersen commitments are correct.*

Proof. Given m, c, r, y, Bob can verify $c = g^m \cdot y^r \mod p$. Thus, the Pedersen commitment scheme is correct. □

Theorem 3.66. *Pedersen commitments are perfectly hiding.*

Proof. Given a commitment c, every message m is equally likely to be the committed message to c. That is because given m, r and any m', there exists (a unique) r' such that $g^m \cdot y^r = g^{m'} \cdot y^{r'} \mod p$. With $y = g^x \mod p$ (such a value x always exists since g is a primitive root), we just have to solve the linear equation $m' + x \cdot r' = m + x \cdot r \mod p - 1$. □

Theorem 3.67. *Pedersen commitments are computationally binding.*

Proof. We show that DL $\leq$ ChangingCommitment, where ChangingCommitment is defined as the problem where Alice needs to find different values m', r' that have the same commitment $c = g^m \cdot y^r$ (i.e. $g^m \cdot y^r = g^{m'} \cdot y^{r'}$).
Given $g, y = g^x$ our goal is to find x. We create an instance for ChangingCommitment by first choosing any m, r and commit to $c = g^m \cdot y^r$. We assume we can solve ChangingCommitment and thus Alice can find m', r' such that $g^m \cdot y^r = g^{m'} \cdot y^{r'}$. The previous equation can also be written as $g^{m-m'} = y^{r'-r}$.
We can now compute $x = (m - m') \cdot (r' - r)^{p-2} \mod p - 1$, because

$$g^x = g^{(m-m')\cdot(r'-r)^{p-2}} = \left(g^{(m-m')}\right)^{(r'-r)^{p-2}} = y^{(r'-r)\cdot(r'-r)^{p-2}} = y^{(r'-r)^{p-1}} = y.$$

The last step assumes that $p - 1$ is prime and thus we can apply Fermat's Theorem. In practice, the group used always has a size that is a prime.

□

Remarks:

- If Alice sends both c, r as a commitment to Bob, Pedersen commitments can be perfectly binding and computationally hiding.
- But why compromise at all? Ideally, we want both: perfectly hiding and perfectly binding.

Theorem 3.68. *A commitment scheme can either be perfectly binding or perfectly hiding but not both.*

Proof. Assume a commitment scheme is perfectly binding. Then Alice cannot change her commitment and open another value, even if she is computationally unbounded. This can be the case if and only if there is only a unique value m that can be committed to c. But this means that for a computationally unbounded Bob, he can simply generate all possible commitments and find the value m. Thus the commitment scheme is not perfectly hiding. □

Remarks:

- Commitment schemes have important applications in several cryptographic protocols, such as zero-knowledge proofs, and multiparty computation.

3.11 Threshold Secret Sharing

How does a company share its vault passcode among its board of directors so that at least half of them have to agree to opening the vault?

Definition 3.69 (Threshold Secret Sharing)**.** *Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t,n)-threshold secret sharing scheme.*

→ notebook

```
# $s$ = secret real number to be shared
# $t$ = threshold number of participants to recover the secret
# $n$ = total number of participants

def distribute($s,t,n$):
    Generate $t-1$ random $a_1,\dots,a_{t-1} \in R$
    Obtain a polynomial $f(x) = s + a_1x + \ldots + a_{t-1}x^{t-1}$
    Generate $n$ distinct $x_1,\dots,x_n \in R \setminus \{0\}$
    Send $(x_i, f(x_i))$ to participant $P_i$

def recover($x = [x_0,x_1,\dots,x_t], y = [f(x_0), f(x_1), \dots, f(x_t)]$):
    $f$ = lagrange($x,y$)
    return $f(0)$
```

Algorithm 3.70: Shamir's (t,n) Secret Sharing Scheme

Theorem 3.71. *Algorithm 3.70 is correct.*

Proof. Any t shares will result in the reconstruction of the same polynomial, hence the secret will be revealed. □

Theorem 3.72. *Algorithm 3.70 has perfect security.*

Proof. A polynomial of degree $t-1$ can be defined only by t or more points. So, any subset $t-1$ of the n shares cannot reconstruct a polynomial of degree $t-1$. Given less than t shares, all polynomials of degree $t-1$ are equally likely; thus any adversary, even with unbounded computational resources, cannot deduce any information about the secret if they have less than t shares. □

Remarks:

- Note that for numerical reasons, in practice modulo p arithmetic is used instead of real numbers. → notebook
- What happens if a participant is malicious? Suppose during recovery, one of the t contributing participants publishes a wrong share $(x_i', f(x_i'))$. The $t-1$ honest participants are blocked from the secret while the malicious participant is able to reconstruct it. To prevent this, we employ *verifiable* secret sharing schemes.

Definition 3.73 (Verifiable Secret Sharing or VSS)**.** *An algorithm that achieves threshold secret sharing and ensures that the secret can be reconstructed even if a participant is malicious is called verifiable secret sharing.*

Remarks:

- Typically, a secret sharing scheme is verifiable if auxiliary information is included that allows participants to verify their shares as consistent.
- VSS protocols guarantee the secret's reconstruction even if the distributor of the secret (the dealer) is malicious.
- So far, we assumed a dealer knows the secret and all the shares. However, we want to avoid trusted third parties and distribute trust. A strong cryptographic notion towards this direction is multiparty computation.

3.12 Multiparty Computation

Alice, Bob, and Carol are interested in computing the sum of their income without revealing to each other their individual income.

```
# $a, b, c$ = Alice's, Bob's and Carol's income

def Sum_MPC():
    Alice picks a large random number $r$
    Alice sends to Bob $m_1 = a + r$
    Bob sends to Carol $m_2 = b + m_1$
    Carol sends to Alice $m_3 = c + m_2$
    Alice computes $s = m_3 - r$
    Alice shares $s$ with Bob and Carol
```

Algorithm 3.74: Computation of the Sum of 3 Parties' Income

Theorem 3.75. *Algorithm 3.74 is correct, meaning the output is the desired sum.*

Proof. The output of the algorithm is $m_3 - r = c + m_2 - r = c + b + m_1 - r = c + b + a + r - r = a + b + c$. □

Theorem 3.76. *Algorithm 3.74 keeps the inputs secret.*

Proof. Bob receives $r + a$, hence no information is revealed concerning Alice's income as long as r is large enough. In addition, both Carol and Alice cannot deduce any information about the individual incomes as they are obfuscated. □

Remarks:

- Algorithm 3.74 is an example of secure 3-party computation.
- The generalization of this problem to multiple parties is known as multiparty computation.

Definition 3.77 (Multiparty Computation or MPC)**.** *An algorithm that allows n parties to jointly compute a function $f(x_1, x_2, \ldots, x_n)$ over their inputs $x_1, x_2, \ldots, x_n$ while keeping these inputs secret achieves secure multiparty computation.*

Remarks:

- Formal security proofs in MPC protocols are conducted in the *real/ideal world paradigm.*

Definition 3.78. *The real/ideal world paradigm states two worlds: In the ideal world, there is an incorruptible trusted third party who gathers the participants' inputs, computes the function, and returns the appropriate outputs. In contrast, in the real world, the parties exchange messages with each other. A protocol is secure if one can learn no more about each participant's private inputs in the real world than one could learn in the ideal world.*

Remarks:

- In Algorithm 3.74, we assume all participants are honest. But what if some participants are malicious?
- In MPC, the computation is often based on secret sharing of all the inputs and zero-knowledge proofs for a potentially malicious participant. Then, the majority of honest parties can assure that bad behavior is detected and the computation continues with the dishonest party eliminated or her input revealed.

Chapter Notes

In 1974, Ralph Merkle designed Merkle Puzzles [15], the first key exchange scheme which works over an insecure channel. In Merkle Puzzles, the eavesdropper Eve's computation power can be at most quadric to Alice's and Bob's computational power. This quadratic difference is not enough to guarantee security in practical cryptographic applications. In 1976, Diffie and Hellman introduced a practically secure key exchange scheme over an insecure channel [6].

Diffie Hellman key exchange [6], Schnorr zero-knowledge proofs [19], ElGamal signature and encryption schemes [7] all rely on the hardness of the discrete logarithm problem [3]. So far we have been conveniently vague in our choice of a group, but the discrete logarithm problem is solvable in polynomial-time when we choose an inappropriate group. To avoid this, we can select a group that contains a large subgroup. For example, if $p = 2q + 1$ and q is prime, there is a subgroup of size q, called the quadratic residues of p, which is often used in practice.

Another frequently employed hard problem is integer factorization [12]. The RSA cryptosystem [18], developed in 1977 at MIT by Ron Rivest, Adi Shamir, and Leonard Adleman, depends on integer factorization. RSA was also the first public-key encryption scheme that could both encrypt and sign messages.

A trapdoor one-way function is a function that is easy to compute, difficult to invert without the trapdoor (some extra information), and easy to invert with a trapdoor [6, 25]. The factorization of a product of two large primes, used in

RSA, is a trapdoor function. While selecting and verifying two large primes and multiplying them is easy, factoring the resulting product is (as far as known) difficult. However, if one of the prime numbers is given as a trapdoor, then it is easy to compute the other prime number. There are no known trapdoor one-way functions based on the difficulty of discrete logarithms (either modulo a prime or in a group defined over an elliptic curve), because there is no known "trapdoor" information about the group that enables the efficient computation of discrete logarithms. In general, a digital signature scheme can be built by any trapdoor one-way function in the random oracle model [14].

A random oracle [1] is a function that produces a random output for each query it receives. It must be consistent with its replies: if a query is repeated, the random oracle must return the same answer. Hash functions are often modeled in cryptographic proofs as random oracles. If a scheme is secure assuming the adversary views some hash function as a random oracle, it is said to be secure in the random oracle model.

Secure digital signature schemes are unforgeable. There are several versions of unforgeability. For instance, Schnorr signatures, a modification of ElGamal signatures, are existentially unforgeable against adaptively chosen message attacks (EUF-CMA) [20]. In the adaptively chosen message attack, the adversary wants to forge a signature for a particular public key (without access to the corresponding secret key) and has access to a signing oracle, which receives messages and returns valid signatures under the public key in question. The proof that Schnorr digital signatures are EUF-CMA is based on the proof that the Schnorr zero-knowledge proof is sound.

Zero-knowledge proofs are a complex cryptographic primitive; formally defining zero-knowledge proofs was a delicate task that took 15 years of research [2, 10]. One key application for zero-knowledge proofs is in user identification schemes. Another recent one is in cryptocurrencies, such as Monero [23].

The concept of information-theoretically secure communication was introduced in 1949 by American mathematician Claude Shannon, the inventor of information theory, who used it to prove that the one-time pad system was secure [22]. Secret sharing schemes are information theoretically secure. Verifiable secret sharing was first introduced in 1985 by Benny Chor, Shafi Goldwasser, Silvio Micali and Baruch Awerbuch [5]. Thereafter, Feldman introduced a practical verifiable secret sharing protocol [9] which is based on Shamir's secret sharing scheme [21] combined with a homomorphic encryption scheme. Verifiable secret sharing is important for secure multiparty computation to handle active adversaries.

Multiparty computation (MPC) was formally introduced as secure two-party computation (2PC) in 1982 for the so-called Millionaires' Problem, a specific problem which is a Boolean predicate, and in general, for any feasible computation, in 1986 by Andrew Yao [24, 26]. MPC protocols often employ a cryptographic primitive called oblivious transfer.

An oblivious transfer protocol, originally introduced by Rabin in 1981 [17], allows a sender to transfer one of potentially many pieces of information to a receiver, while remaining oblivious as to what piece of information (if any) has been transferred. Oblivious transfer is complete for MPC [11], that is, given an implementation of oblivious transfer it is possible to securely evaluate any polynomial time computable function without any additional primitive! An "1-out-of-n" oblivious transfer protocol [8, 16, 13] is a generalization of oblivious transfer where a receiver gets exactly one database element without the server (sender) getting to know which element was queried, and without the receiver knowing anything about the other elements that were not retrieved. A weaker version of "1-out-of-n" oblivious transfer, where only the sender should not know which element was retrieved, is known as Private Information Retrieval [4].

This chapter was written in collaboration with Zeta Avarikioti, Klaus-Tycho Foerster, Ard Kastrati and Tejaswi Nadahalli.

Bibliography

[1] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73, 1993.

[2] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[3] Dan Boneh. The decision diffie-hellman problem. In *International Algorithmic Number Theory Symposium*, pages 48–63. Springer, 1998.

[4] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[5] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395. IEEE, 1985.

[6] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[7] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[8] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[9] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity and a methodology of cryptographic protocol design. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 285–306. 2019.

[11] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer–efficiently. In *Annual international cryptology conference*, pages 572–591. Springer, 2008.

[12] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K Lenstra, Emmanuel Thomé, Joppe W Bos, Pierrick Gaudry, Alexander Kruppa, Peter L Montgomery, Dag Arne Osvik, et al. Factorization of a 768-bit rsa modulus. In *Annual Cryptology Conference*, pages 333–350. Springer, 2010.

[13] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829, 2016.

[14] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, 1979.

[15] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[16] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 35(5):1254–1281, 2006.

[17] Michael O Rabin. How to exchange secrets with oblivious transfer.

[18] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[20] Yannick Seurin. On the exact security of schnorr-type signatures in the random oracle model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 554–571. Springer, 2012.

[21] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[22] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[23] Nicolas van Saberhagen. Monero whitepaper. Technical report, 2013.

[24] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[25] Andrew C Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*, pages 80–91. IEEE, 1982.

[26] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

Chapter 4

Databases

A computer does more than just computation. In particular, a computer can also store and retrieve large amounts of data efficiently. In this chapter, we want to understand some of the key ingredients of databases.

4.1 Dictionary

We manage a library and want to be able to quickly tell whether we carry a given book or not. We need the capability to *insert*, *delete*, and *search* books.

Definition 4.1 (Dictionary)**.** *A **dictionary** is a data structure that manages a set of **objects**. Each object is uniquely identified by its **key**. The relevant operations are* → notebook

- `search`*: find an object with a given key*
- `insert`*: put an object into the set*
- `delete`*: remove an object from the set*

Remarks:

- There are alternative names for dictionary, e.g. key-value store, associative array, map, or just set.
- If the dictionary only offers `search`, it is called *static*; if it also offers `insert` and `delete`, it is *dynamic.*
- When discussing the algorithms, we will often ignore that we actually have a set of objects, each of which is identified by a unique key, and just talk about the set of keys. With regard to the library example, books are globally uniquely identified by a key called `ISBN`. Whenever we say we insert/delete/search a key, we can just drag the key's object along.
- The classic data structure for dictionaries is a binary search tree.

Definition 4.2 (Binary search tree)**.** *A **binary search tree** is a rooted tree, where each node stores a key. Additionally, each node may have a pointer to a left and/or right child tree. For all nodes, if existing, the nodes in the left child tree store smaller keys, and those in the right child tree store larger keys.*

```
def search(self, key): # self is current node, initially root
    if key < self.key:
      if self.left is None: return None
      else: return self.left.search(key)
    elif key > self.key:
      if self.right is None: return None
      else: return self.right.search(key)
    return self.val
```

→ notebook

Algorithm 4.3: Search Tree: Search

Remarks:

- The cost of searching in a binary search tree is proportional to the *depth* of the key, which is the distance between the node with the key and the root.
- There are search trees called *splay trees* that keep frequently searched keys close to the root for quick access. On the other hand, there may be rarely accessed keys deep in a splay tree.
- Using *balanced search trees*, we can maintain a dictionary with worst-case logarithmic depth for all keys, and thus worst-case logarithmic cost per insert/delete/search operation.
- Is there a way to build a dictionary with less than logarithmic cost and with keys that cannot be ordered?

4.2 Hashing

In this section we use hashing to implement an efficient dictionary.

Definition 4.4 (Universe, Key Set, Hash Table, Buckets)**.** *We consider a **universe** U containing all possible keys. We want to maintain a subset of this universe, the **key set** $N \subseteq U$ with $|N| =: n$, where $|N| \ll |U|$. We will use a **hash table** M, i.e. an array M with m **buckets** $M[0], M[1], \ldots, M[m-1]$.*

Remarks:

- The standard library of almost every widely used programming language provides hash tables, sometimes by another name. In C++, they are called `unordered_map`, in Python `dictionary`, in Java `HashMap`.

- The translation from virtual memory to physical memory uses a piece of hardware called *translation lookaside buffer* (TLB), which is a hardware implementation of a hash table. It has a fixed size and acts like a cache for frequently looked up virtual addresses.
- Compilers make use of hash tables to manage the symbol table.

Definition 4.5 (Hash Function)**.** *Given a universe U and a hash table M, a* ***hash function*** *is a function $h : U \to M$. Given some key $k \in U$, we call $h(k)$ the* ***hash*** *of k.*

Remarks:

- A hash function should be fast to compute and distribute hashes nicely, e.g. $h(k) = k \bmod m$ for a key $k \in \mathbb{N}$; in contrast to Chapter 3, we do not care whether a hash function is one-way.
- If we use `ISBN` mod m as our library hash function, can we insert/delete/search books in constant time?!
- What if two keys $k \neq k'$ have $h(k) = h(k')$?

Definition 4.6 (Collision)**.** *Given a hash function $h : U \to M$, two distinct keys $k, k' \in U$ produce a* ***collision*** *if $h(k) = h(k')$.*

Remarks:

- Since keys may experience collisions, the key must be stored in the bucket.
- There are competing objectives we want to optimize for when hashing. On the one hand, we want to make the hash table small since we want to save memory. On the other hand, small tables will have more collisions. How likely is it to get a collision for a given n and m?

Theorem 4.7 (Birthday Problem)**.** *If we throw a fair m-sided dice $n \leq m$ times, let D be the event that all throws show different numbers. Then D satisfies*

$$\mathbb{P}[D] \leq \exp\left(-\frac{n(n-1)}{2m}\right).$$

Proof. We have that

$$\begin{aligned}\mathbb{P}[D] &= \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-(n-1)}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m} \\ &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right)\end{aligned}$$

We can use that $\ln(1+x) \leq x$ for all $x > -1$ and the monotonicity of e^x:

$$\mathbb{P}[D] = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right) \leq \exp\left(\sum_{i=0}^{n-1} -\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right)$$

□

Remarks:

- Theorem 4.7 is called the "birthday problem" since traditionally, people use birthdays for illustration: In order to have a chance of at least 50% that two people in a group share a birthday, we only need a group of 23 people.
- If we insert more than roughly $n \approx \sqrt{m}$ keys into a hash table, the probability of a collision approaches 1 quickly. In other words, unless we are willing to use at least $m \approx n^2$ space for our hash table, we will need a good strategy for resolving collisions.
- Theorem 4.7 assumes totally random hash functions — for non-random distributions of hashes, we might have more collisions. In particular, if we fix a hash function, then we can always end up with a key set N that suffers from many collisions. E.g., if many books have an `ISBN` that ends in 000, then `ISBN` mod 1000 is a terrible hash function.
- Maybe we can use modulo, but with a different m?
- In general, several efficient ways to deal with collisions are known, e.g., hashing with chaining, hashing with probing, static hashing, cuckoo hashing. We do not discuss these advanced methods in this class.
- Universal hashing is a particularly intriguing technique, as it guarantees that a random hash function from a larger family as good as it gets.

4.3 Key-Value Databases

Definition 4.8 (Key-Value Database System)**.** *The concept of dictionaries is used in key-value database systems. The server maintains the dictionary and clients can insert and query the stored data using the keys.*

Remarks:

- Popular key-value databases are Redis and Memcached. They are often used for caching in web services. Dynamically generated documents or results of queries to other databases can be stored temporarily to allow fast access to often requested data.
- The data is often kept in main memory to speed up the access and only duplicated to disk to recover the database in case of a system failure.
- Depending on the used database, different data types can be stored in the value. This can be an integer, a string, or even an array.
- Document databases are an extension of simple key-value database systems. The value has to be in a format that the database understands, such as a JSON or XML document. These databases allow queries on the content of the documents. MongoDB and CouchDB are popular document databases.

4.4 Relational Databases

However, most databases offer queries beyond simple key searches. Questions like “What is the movie with the largest cast?” or “How many directors have directed more than ten movies?” should be answered without first writing a new program. Relational databases can store large amounts of *structured* data and answer possibly complex questions about it.

Definition 4.9 (Table, Row, Column, Database)**.** *A **table** consists of **rows**, so that each row (data record) contains the same **fields**, i.e., kinds of entries. When the rows of a table are written line by line, the fields form the **columns** of the table. Each column is referred to by a descriptive name, and is associated with the type of the respective field, e.g., integer, floating point, string, or a date. A **database** is a collection of tables.*

Remarks:

- In the database context, tables are also called *relations*, because the entries in each row are related to each other, namely by belonging to the same row.

movies

title	director	year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966

⋮

Figure 4.10: A database containing a single table called “movies” storing the title, director, and year of release for each movie.

Remarks:

- Databases as we study them are accessed using the so-called *structured query language* (SQL). Thus they are referred to as SQL or *relational* databases.

- MySQL and PostgreSQL are two popular open source SQL database systems.

- SQL database systems typically run as a daemon process on some server. Client applications connect to the server and authenticate themselves via username and password. Therefore, multiple users accessing the same database may result in concurrency issues. Some form of concurrency control is necessary!

- Other database systems are tailored to single-user processing. They relieve developers from the burden of implementing efficient data structures for relational data. SQLite is one such example, and is used, e.g., in Firefox, Chrome, Android, Adobe Lightroom, and Windows 10.

4.5 SQL Basics

Definition 4.11 (SQL Data Types)**.** *SQL defines the following types of columns.*

- CHARACTER(m) *and* CHARACTER VARYING(m) *for fixed and variable length strings of (maximum) length m,*
- BIT(m) *and* BIT VARYING(m) *for fixed and variable length bit strings of (maximum) length m,*
- NUMERIC, DECIMAL, INTEGER*, and* SMALLINT *for fixed point and integer numbers,*
- FLOAT, REAL*, and* DOUBLE PRECISION *for floating point numbers,*
- DATE, TIME*, and* TIMESTAMP *for points in time, or*
- INTERVAL *for ranges of time.*

Remarks:

- The range of each type includes the special value NULL. Note that NULL is different from the string 'NULL', the empty string, and from the number 0 (zero). NULL indicates that the row has *no value* for the corresponding field.
- Many database systems implement more types, e.g., geographic coordinates, IP addresses, geometric objects, or large integers.
- All SQL statements end with a semicolon. The SQL language is case insensitive, but by convention keywords are often typed in upper case.
- The SQL-92 specification is over 600 pages long, newer versions of the standard even longer. To add insult to injury there are lots of vendor specific "SQL dialects", i.e., modifications and extensions. However, the basic set of commands for creating, manipulating, and querying tables are largely the same across database implementations.

CREATE DATABASE *database-name*; → notebook

Additional parameters allow to set database-specific options, e.g., user-based permissions, or default character sets for text strings. How a database is opened depends on the implementation.

CREATE TABLE *table-name* (*field-name type, field-name type, . . .*);

To enforce that all rows have a value for a particular field, one can add NOT NULL to the type when creating the table. Fields have a default value, which is NULL if not specified by adding DEFAULT *value* to the type description.

Remarks:

- There are also GUI and web-based client applications (that execute locally or on an http-server, respectively) and offer access to the database in a more intuitive manner than the classic command line tools. Examples for PostgreSQL are pgAdmin, DataGrip and DBeaver.
- Such tools are especially helpful for creating the databases and tables and often support multiple database systems. They also feature importing data from various formats, e.g., CSV files, instead of using SQL statements to populate the tables.

INSERT INTO *table-name* (*field-name*, . . .) VALUES (*value*, . . .); → notebook

Values must be listed in the same order as the corresponding field names. When a field name (and thus its value) is omitted the field's default value is assumed. When the list of field names is omitted the field's values must be listed in the same order that was used when creating the table. To insert more than one row in one statement, multiple rows may be separated by commas.

→ notebook

```sql
SELECT * FROM movies;
SELECT * FROM movies WHERE director = 'Spielberg, Steven';
SELECT title FROM movies WHERE year BETWEEN 1990 AND 1999;
SELECT * FROM movies WHERE title IS NULL OR director IS NULL;
SELECT title, director FROM movies WHERE title LIKE '%the%';
```

Listing 4.12: Querying the movies table.

SELECT *field-name*, . . . FROM *table-name* WHERE *condition*;

Lists all specified fields of all rows in the table that fulfill the condition. The special field * lists all fields. The WHERE condition may be omitted to list the whole table. A condition can include comparisons (<, >, =, <>) between fields constants. The special value NULL can be tested with IS NULL. Conditions can be joined using parenthesis and logic operators like AND, OR, and NOT. Strings can be matched with patterns using ***field-name* LIKE *pattern***. In the pattern, an underscore (_) matches a single character, whereas % matches arbitrarily many.

→ notebook

```sql
SELECT MIN(year) FROM movies;
SELECT AVG(year) FROM movies WHERE director='Lumet, Sidney';
SELECT COUNT(*) FROM movies;
SELECT COUNT(DISTINCT director) FROM movies;
```

Listing 4.13: Aggregation with SQL.

SELECT *aggregate*, . . . ;

Functions for aggregation include AVG to compute the average of a certain field, MIN and MAX for the minimum and maximum value, SUM for the sum of a field, and COUNT to count the number of occurrences. In an aggregation, the keyword DISTINCT indicates that only distinct values should be considered.

→ notebook

```sql
SELECT director, COUNT(title) FROM movies GROUP BY director;
SELECT director, COUNT(title) FROM movies GROUP BY director
HAVING COUNT(title) > 10;
```

```sql
SELECT year, director, COUNT(title) FROM movies
GROUP BY director, year
ORDER BY year DESC, director ASC;
```

Listing 4.14: Grouping and sorting.

SELECT *field-name*|*aggregate*, . . . GROUP BY *field-name*,. . . ;

Aggregations may be partitioned using the group-by clause. Similar to before, the query result can only include aggregates and fields by which the result is partitioned.

Since WHERE clauses are applied before GROUP BY the result of aggregations cannot appear in them. When the result should be conditioned on the result of an aggregation, a HAVING clause can be used.

SELECT . . . ORDER BY *field-name*,. . . ;

After each field-name, the keyword ASC or DESC can be used to determine ascending or descending sorting order, respectively.

```sql
UPDATE movies SET title = 'Star Wars Episode IV: A New Hope'
WHERE title = 'Star Wars';
DELETE FROM movies WHERE title = '';
```

Listing 4.15: Updating and removing rows.

UPDATE *table* SET *field-name* = *value*,. . . WHERE *condition*;

Updates the specified fields in all rows fulfilling the condition.

DELETE FROM *table-name* WHERE *condition*;

Removes all rows fulfilling the condition from the table.

4.6 Modeling

The way our example table from Figure 4.10 is designed results in lots of duplicate data—the director's name is stored anew for each row, and two directors with the same name cannot be distinguished. The situation worsens when we want to store the cast of each movie. In other words, the way we modeled our data can be improved. *Entity-Relationship* (ER) diagrams are a tool to find good representations for data.

Definition 4.16 (Entity-Relationship Diagram)**.** *Rectangles denote* ***entities*** *(tables), and diamonds with edges to entities indicate* ***relations*** *between those entities. On such an edge, the number* 1 *or the letter* n *denotes whether the corresponding entity takes part once or arbitrarily many times in the relation. Entities and relations can have* ***attributes*** *(columns) with a name, drawn as ellipses. Italicised attributes are* ***key attributes*** *which must be unique for each such entity.*

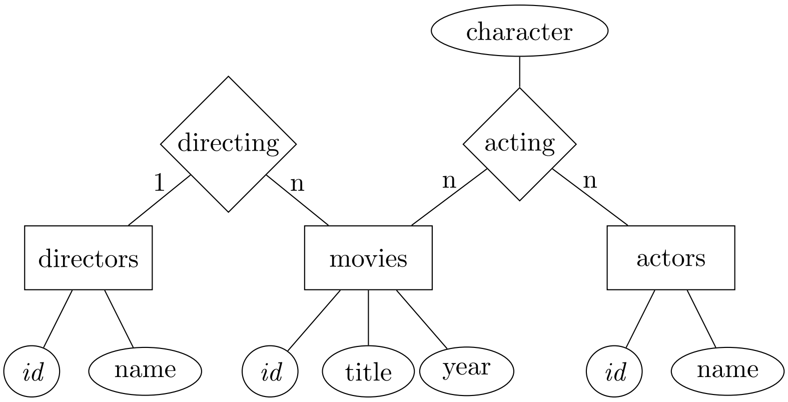


Figure 4.17: Model for a movie database. Movies and directors are in a 1-to-n relation: Each movie is directed by 1 director, and a director may work on many movies. Movies and actors are in a n-to-n relation, which has an additional attribute: An actor may appear in many movies, and each appearance is associated with a character in that movie, played by that actor.

Remarks:

- It is standard practice to assign a so-called *key attribute*, often named *id*, to every entity.
- What do ER diagrams have to do with SQL? Primarily, ER diagrams are for conceptually modeling the kind of data and relations one wishes to store. They can be translated into databases. Each entity corresponds to a table with the corresponding attributes as columns. An n-to-n relation is represented by a table with columns for each attribute, and a column for the key attribute of each entity in the relation.
- A close relative of the ER diagram is the Unified Modeling Language (UML). UML is used to represent the tables of a database (or classes of object oriented software) accurately, with detailed information, e.g. fields.

	actor
id	name
1	Harrison Ford
2	Tom Cruise
	⋮

	acting	
actor_id	character	movie_id
1	Indy	2
2	Ray Ferrier	3
	⋮	

Figure 4.18: The actor table and a table capturing the acting relation.

Remarks:

- The same scheme can be used for 1-to-1 and 1-to-n relations. However, one may also include the relation in the table storing the entity on the 1-side.

directors

id	name
1	Sidney Lumet
2	Steven Spielberg
3	Harold P. Warren
	⋮

movies

id	title	year	director_id
1	12 Angry Men	1957	1
2	Raiders of the Lost Ark	1981	2
3	War of the Worlds	2005	2
4	Manos: The Hands of Fate	1966	3
	⋮		

Figure 4.19: The movie and director tables using the new database layout. The director table simply maps ids to director names. Since the directing relationship is 1-to-n, it can be represented by adding a column to the movies table that stores the director for each movie.

Remarks:

- Similarly, a 1-to-1 relation can be turned into an attribute of one of the entities.
- Tables dedicated to capturing relations are often called *join* tables.

4.7 Keys & Constraints

What is stopping us from inserting a row in the acting table that contains an actor_id or a movie_id that does not exist? Or from creating a director with a duplicate id?

Definition 4.20 (Key)**.** *In a table, a column (or set of columns) is a* ***unique key*** *if the corresponding values uniquely identify the rows within the table. The* ***primary key*** *of a table is a designated unique key. A* ***foreign key*** *is a column (or set of columns) that references the primary key of another table.*

Remarks:

- SQL databases can automatically enforce these constraints. For example, a row containing a foreign key can only be inserted if it references an existing primary key. Vice versa, a row may only be removed if its primary key is not referenced by any foreign key.

ALTER TABLE *table*
ADD CONSTRAINT UNIQUE (*field-name*,. . .); → notebook

Any two rows must differ in at least one of the specified fields.

ALTER TABLE *table* ADD PRIMARY KEY (*field-name*,. . .);

Sets the specified fields as the primary key for the table. Any two rows must differ in at least one of the specified fields. The entries in these fields must not be NULL.

ALTER TABLE *left-table* ADD FOREIGN KEY (*field-name*,. . .)
REFERENCES right-table;

Ensures that the values in the specified fields in the left table are the primary key of a row in the right table.

Remarks:

- Constraints for new tables can also be set using CREATE TABLE.
- Other ALTER TABLE queries add different constraints (e.g., checking that an integer field contains only certain values), remove constraints, and change the name, type or default value of fields.
- To ensure that checking constraints and searching for data is fast, database systems rely on *index* data structures.

4.8 Joins

How can we access the data, which is now scattered across multiple tables?

```
SELECT movie.title, director.name AS director, movie.year
FROM movie
INNER JOIN director ON movie.director_id = director.id;
```

→ notebook

Listing 4.21: Example query that returns the table depicted in Figure 4.22.

SELECT . . .
FROM *left-table* INNER JOIN *right-table* ON *condition*;
Returns all rows that can be formed from a row in the left-table and a row in the right-table that satisfy the specified condition.

movie.title	director	movie.year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966

⋮

Figure 4.22: The result returned by the query in Listing 4.21.

Remarks:

- In a query, one can create aliases for field and table names using the AS keyword, see Listing 4.21.
- The result of a JOIN clause can be ordered, fields can be aggregated and grouped, and conditions can be added using WHERE clauses.
- For example, we can combine joins and aggregations to answer our initial question of which movie has the largest cast.

→ notebook

```
SELECT movie.title, COUNT(*) AS cast_size
FROM acting INNER JOIN movie ON acting.movie_id = movie.id
GROUP BY movie.id ORDER BY cast_size DESC LIMIT 10;
```

Listing 4.23: Finding the 10 movies with the largest cast.

Remarks:

- The query from Listing 4.23 uses a LIMIT clause to return only the ten first entries of the sorted results.
- An INNER JOIN where the condition is TRUE returns the Cartesian product of both tables. This special case can also be obtained with the CROSS JOIN clause.
- An inner join will only return those rows of one table that have a matching row (that satisfies the condition) in the other table. For example, in Listing 4.21, a director with id 5 would not appear in the result if there are no movies which have director_id=5.
- If you want unmatched rows to appear in the result, you need to use an OUTER JOIN.

→ notebook

```
SELECT movie.title, director.name AS director, movie.year
FROM movie
RIGHT OUTER JOIN director ON movie.director_id = director.id;
```

Listing 4.24: Example query that returns the table depicted in Figure 4.25.

movie.title	director	movie.year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966
NULL	Jon Doe	*NULL*

⋮

Figure 4.25: The result returned by the query in Algorithm 4.24. The right outer join includes all rows from the inner join (see Figure 4.22) and, additionally, all entries from the directors table for which there is no matching entry in the movies table. In our example, "director" Jon Doe has not directed any movies, hence the movie title and year column are filled with *NULL* values.

SELECT ...

FROM *left-table* LEFT|RIGHT|FULL OUTER JOIN *right-table* ON *condition*;

Returns all rows from the inner join. In addition, a LEFT or RIGHT

OUTER JOIN also returns all rows from the left or right table that have no matching row on the opposite table, respectively. The fields in unmatched rows that cannot be filled from the other table are filled with NULL values. A FULL OUTER JOIN returns both of the above.

Remarks:

- A LEFT OUTER JOIN in Listing 4.24 would include the movies with no director instead of the directors who have not directed any movie.
- Queries may use more than one JOIN clause.

→ notebook

```sql
SELECT movie.title
FROM actor INNER JOIN acting
ON acting.actor_id = actor.id AND actor.name = 'Ford, Harrison'
RIGHT OUTER JOIN movie ON acting.movie_id = movie.id
WHERE acting.actor_id IS NULL;
```

Listing 4.26: Finding all movies that Harrison Ford did not appear in.

Remarks:

- The conditions for the first join in Listing 4.26 ensure that only movies with Harrison Ford are taken into account for the second OUTER JOIN. That second join in turn delivers all movies that cannot be matched, yielding a NULL entry for the actor_id for movies without Harrison Ford.

Chapter Notes

Dictionaries based on search trees are useful for providing additional operations such as nearest neighbor queries or range queries, where we want to find all keys in a certain range. Binary search trees were first published by three independent groups in 1960 and 1962 (for references, see Knuth [13]). The first instance of a self-balancing search tree that guarantees logarithmic cost for insert/search/delete is the AVL-tree, named so after its inventors Adelson-Velski and Landis [1]. For multidimensional keys, e.g. geometric data or images, there are specialized tree structures such as kd-trees [2] or BK-trees [4].

Hashing has a long history and was initially used and validated based on empirical results. One of the first publications was Peterson's 1957 article [14] where he defined an idealized version of probing and empirically analyzed linear probing. Universal hashing was introduced two decades later by Carter and Wegman in 1979 [5]. Perfect static hashing was invented in 1984 by Fredman et al. [10] and is sometimes also referred to as FKS hashing after its inventors. Its dynamization by Dietzfelbinger et al. took another decade until 1994 [9].

In 1970, Edgar F. Codd proposed the relational database model [8] while working at IBM research. Later in the 70s, another group at IBM developed SQL's predecessor SEQUEL (Structured English QUEry Language) [6]. After being renamed SQL due to trademark issues, it was standardized by the ISO in 1987 and later revised [11]. Other companies started developing relational database systems, and nowadays there are many SQL databases implementing different feature sets to choose from.

Around the same time, ER diagrams were conceived as a modeling tool [3, 7]. The Unified Modeling Language (UML), first standardized by the ISO in 1995 [12] and revised in 2012, also includes diagrams that model databases.

This chapter was written in collaboration with Georg Bachmeier and Jochen Seidel.

Bibliography

[1] M Adelson-Velskii and Evgenii Mikhailovich Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] A. P. G. Brown. Modelling a real world system and designing a schema to represent it. In *IFIP TC-2 Special Working Conference on Data Base Description*, 1975.

[4] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[5] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[6] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74. ACM, 1974.

[7] Peter Pin-Shan Chen. The entity-relationship model&mdash;toward a unified view of data. *ACM Trans. Database Syst.*, 1976.

[8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 1970.

[9] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[10] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[11] International Organization for Standardization. Information technology – Database languages – SQL – part 1: Framework (SQL/Framework), 2011. ISO/IEC 9075-1.

[12] International Organization for Standardization. Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure, 2012. ISO/IEC 19505-1.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[14] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1(2):130–146, 1957.

Chapter 5

Machine Learning

So far, we told the computer exactly what to do: every problem was solved by a specific algorithm. However, in the real world, we might have to deal with messy data in order to understand its underlying function. In other words, it may be difficult to separate function from noise. Maybe a computer can do part of the job, and learn some of the parameters of a function? Welcome to machine learning!

More honestly: Changing random stuff to make your program work better is “hacky” and “bad coding practice”. But if you do it fast enough it is “machine learning” and pays high salaries.

5.1 Linear Regression

Definition 5.1 (Dataset, Input, Output)**.** *A **dataset** D is a set of n tuples (x, y) sampled from an unknown function*

$$f : x \mapsto y$$

*We call $x \in X$ an **input** and $y \in Y$ the corresponding **output** of f.*

Remarks:

- We want to learn a function $\hat{f}$ such that $\hat{f}(x) \approx f(x)$.
- Since we learn $\hat{f}$ from the dataset D, we may write $\hat{f}_D$.
- For example, if the dataset consists of points on a line, we choose $\hat{f}(x) = w_1 x + w_0$ and determine the parameters $w_i \in \mathbb{R}$.
- If the points in D do not line up perfectly, a linear approximation function $\hat{f}$ will have some error. Even if f is truly linear, there can still be some random noise that introduces error, e.g., some measurement error. → notebook

Definition 5.2 (Approximation Error)**.** *The **approximation error** $Err(x)$ denotes the deviation of $\hat{f}$ from the unknown function f at some input x:*

$$Err(x) = f(x) - \hat{f}(x).$$

Remarks:

- For our linear function this amounts to $Err(x) = y - (w_1 x + w_0)$.
- We want to minimize this error over the entire dataset D. Hence, we choose $\hat{f}$ according to an objective function as follows.

Definition 5.3 (Squared Error Loss)**.** *The **loss function** is used to determine the real-valued parameters $\boldsymbol{w} = (w_0, w_1, ...)^T$ according to the dataset D. There are several options, the most common being the **squared error loss** function:*

$$L(\hat{f}, D) = \sum_{(x,y)\in D} Err(x)^2.$$

Remarks:

- By squaring the error, we ensure each term is positive. Squaring also weighs large errors more highly.
- Another natural choice for a loss function is the absolute error: → notebook
$$L_{abs}(\hat{f}, D) = \sum_{(x,y)\in D} |Err(x)|.$$
- However, the squared error loss is often preferred as it has both a closed-form solution and is everywhere differentiable. How do we find such a solution for our linear function?

Lemma 5.4. *Let $\bar{x} = \frac{1}{n}\sum_D x$ and $\bar{y} = \frac{1}{n}\sum_D y$ be the average input and output of the dataset D. For a linear function $\hat{f}(x) = w_1 x + w_0$, the squared error loss is minimal for*

$$w_1^* = \frac{\sum_{(x,y)\in D}(x-\bar{x})(y-\bar{y})}{\sum_{(x,y)\in D}(x-\bar{x})^2}, \; w_0^* = \bar{y} - w_1^* \bar{x}.$$

*We call these weights the **ordinary least-square** (OLS) estimates, as they minimize the squared error loss.*

Proof. For our linear function, the squared error loss amounts to

$$L(\hat{f}, D) = \sum_{(x,y)\in D} \left(y - (w_1 x + w_0)\right)^2.$$

We find the minimum loss by differentiating $L(\hat{f}, D)$ with respect to $\boldsymbol{w}$:

$$\begin{aligned}
\frac{\partial L}{\partial w_0} &= \sum_{(x,y)\in D} -2\left(y - (w_1 x + w_0)\right) \overset{!}{=} 0 \\
\iff & \sum_{(x,y)\in D} (y - w_1 x - w_0) = 0 \\
\iff & \sum_{(x,y)\in D} (y - w_1 x) = n w_0 \\
\iff & w_0 = \frac{1}{n} \sum_{(x,y)\in D} y - w_1 \frac{1}{n} \sum_{(x,y)\in D} x \\
\iff & w_0 = \bar{y} - w_1 \bar{x}
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial L}{\partial w_1} &= \sum_{(x,y)\in D} -2x\left(y-(w_1x+w_0)\right) \overset{!}{=} 0\\
\iff & \sum_{(x,y)\in D} x(y-w_1x-\bar{y}+w_1\bar{x}) = 0\\
\iff & \sum_{(x,y)\in D} x(y-\bar{y}) = w_1 \sum_{(x,y)\in D} x(x-\bar{x}).
\end{aligned}
$$

Note that $\sum_D y = n\bar{y} = \sum_D \bar{y}$, so $\sum_D \bar{x}(y-\bar{y}) = 0$ and similarly we get $\sum_D \bar{x}(x-\bar{x}) = 0$. After subtracting "0" from both sides, we obtain

$$
\begin{aligned}
&\sum_{(x,y)\in D} (x-\bar{x})(y-\bar{y}) = w_1 \sum_{(x,y)\in D} (x-\bar{x})(x-\bar{x})\\
\iff & w_1 = \frac{\sum(x-\bar{x})(y-\bar{y})}{\sum(x-\bar{x})^2}.
\end{aligned}
$$

□

Remarks:

- This also works if the input is not a single scalar x but a whole vector $\boldsymbol{x}$. This is known as linear regression.

Definition 5.5 (Linear Regression, Features, Weights)**.** *With* ***linear regression****, we search for a function $\hat{f}$ of the form*

$$\hat{f}(\boldsymbol{x}) = \sum_{i=1}^{d-1} w_i x_i + w_0,$$

where the $x_i \in \mathbb{R}$ are the ***features*** *of the input. The parameters $w_i \in \mathbb{R}$ are called the* ***weights*** *and need to be determined. We can write this in vector form as*

$$\hat{f}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x},$$

where $\boldsymbol{x} = (1, x_1, x_2, ..., x_{d-1})^T$ with an additional 1 *to incorporate the weight w_0.*

Remarks:

- Let us find the weights $\boldsymbol{w}$. In order to describe the closed-form solution concisely we use matrix notation, where $\mathbf{X}$ is a matrix of the input features (the so-called *design matrix*). $\mathbf{X}$ has n rows, each row represents a transposed feature vector $\boldsymbol{x}^T = (1, x_1, x_2, ..., x_{d-1})$. The outputs are given as a vector $\boldsymbol{y}$ of length n, where each value has a corresponding row in $\mathbf{X}$.

Theorem 5.6. *The ordinary least-square (OLS) estimates for the weight parameters $\boldsymbol{w}$ of a linear regression model are given by*

$$\boldsymbol{w}^* = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\boldsymbol{y}.$$

Proof. The squared error loss is $L(\hat{f}, D) = \sum_{(\boldsymbol{x},y)\in D} \left(y - \boldsymbol{w}^T\boldsymbol{x}\right)^2$ which can be rewritten in matrix form as

$$L(\hat{f}, D) = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}).$$

Again, we can differentiate with respect to $\boldsymbol{w}$ to find the optimal weights:

$$\begin{aligned}
\frac{\partial L}{\partial \boldsymbol{w}} &= - \left(\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w})\right)^T - (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T\mathbf{X} \overset{!}{=} \mathbf{0}^T \\
\iff & -\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) - \mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) = \mathbf{0} \\
\iff & \mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w}) = \mathbf{0} \\
\iff & \mathbf{X}^T\boldsymbol{y} = \left(\mathbf{X}^T\mathbf{X}\right)\boldsymbol{w} \\
\iff & \boldsymbol{w} = (\mathbf{X}^T\boldsymbol{X})^{-1}\mathbf{X}^T\boldsymbol{y}.
\end{aligned}$$

□

Remarks:

- We must be careful when differentiating in matrix form, as we are differentiating sums.
- We assume in this proof that $\mathbf{X}^T\mathbf{X}$ is invertible. This is for example the case when $\mathbf{X}$ has full column rank, but might not be the case in general.
- We could have derived the result in Lemma 5.4 in matrix form as well. To see that the results are the same, just expand $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\boldsymbol{y}$. Alternatively, if you consider the training data to be normalized ($\bar{x} = \bar{y} = 0$), then $w_0 = 0$ and $w_1 = \frac{\sum xy}{\sum x^2} = (\sum xx)^{-1}\sum xy$.
- What if the relation between $\boldsymbol{x}$ and y is not just linear?

5.2 Feature Modeling

Definition 5.7 (Feature)**.** *A **feature** of the input can be any real-valued term depending on the input variables.*

Remarks:

- Note that we previously used the feature vector $\boldsymbol{x} = (1, x_1, x_2, ..., x_{d-1})^T$. However, a feature can be more complex. For example, if we know in advance that the quantity $\sin x_1\sqrt{x_2}$ is a good term to approximate $f(\boldsymbol{x})$, then we can include this term as a feature in our linear regression model. Also splines, radial basis functions or wavelets work out of the box.
- When modeling a problem, we often start with an "educated guess" about which family of functions $\mathcal{F}$ is well-suited to model the unknown function f. We then restrict ourselves to find the best $\hat{f} \in \mathcal{F}$.

Definition 5.8 (Model)**.** *We call the family of functions $\mathcal{F}$ chosen to approximate f a **model**. The function $\hat{f} \in \mathcal{F}$ is found by fitting the parameters of the model to best represent the dataset D.*

Remarks:

- For instance, we might want to restrict $\hat{f} \in \mathcal{F}$ to polynomials (of degree m), yielding → notebook

$$\hat{f}(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + ... + w_m x^m.$$

- This is called *polynomial regression*, even though it is just a special case of linear regression (and is solved the same way).
- Note that the linear regression method can handle any model $\mathcal{F}$, as long as the (unknown) parameters w_i are linear coefficients.
- For multi-dimensional input we can add all combinations of the variables up to the m^{th} power. For polynomial regression with degree m and input dimension d, this gives $\left(\!\binom{d}{m}\!\right) = \binom{d+m-1}{m}$ features! E.g. for $m = 2$, $d = 3$, we have the 6 features $\{1, x_1, x_2, x_1^2, x_2^2, x_1 x_2\}$.
- What if some input variables are not continuous? For example, we might have a categorical input variable, such as a city name. Should we encode the city as a single variable taking values 1, 2 and 3? This would establish an inherent ordering and scaling between the cities, which would be unreasonable in many cases. One way to deal with such inputs is to use so-called one-hot encoding.

Definition 5.9 (One-Hot Encoding)**.** *A* ***one-hot encoding*** *of a categorical input variable taking k values is a vector representation of length k consisting of* 0*'s and a single* 1 *indicating the corresponding category.*

Remarks:

- Note that using a one-hot encoding may increase the dimension of the model significantly as each category gets its own coefficient w_j.
- See Figure 5.10 for an example.

x_i
London
Budapest
Zurich
n/a
London

$\longrightarrow$

x_{i1}	x_{i2}	x_{i3}
0	1	0
1	0	0
0	0	1
0	0	0
0	1	0

Figure 5.10: One-hot encoding of a categorical variable that can take 3 values (as well as n/a).

5.3 Generalization & Overfitting

What if we include more and more features? For instance, we can add more input features, higher degree terms and other engineered features. Eventually we might have more features than data points, and a linear model will be able to fit the training data perfectly.

Lemma 5.11. *Given $\mathbf{X} \in \mathbb{R}^{n\times d}$ with $d \geq n$, there is at least one solution $\boldsymbol{w}$ to $\mathbf{X}\boldsymbol{w} = \boldsymbol{y}$ for all $\boldsymbol{y} \in \mathbb{R}^n$, if and only if $\mathbf{X}$ has rank n.*

Proof. This is a standard result from Linear Algebra. The proof goes along the lines of: $\mathbf{X}$ has rank $n \iff$ there are n linearly independent columns of $\mathbf{X}$, but n linearly independent columns in $\mathbb{R}^{n\times d}$ form a basis of $\mathbb{R}^{n\times n}$ so (setting w for all other columns to 0) $\mathbf{X}\boldsymbol{w} = \boldsymbol{y}$ has a unique solution for all $\boldsymbol{y} \in \mathbb{R}^n$. □

Remarks:

- For example powers of a feature, $\{1, x_i, x_i^2, x_i^3, ...\}$ are linearly independent, so polynomial regression with high enough degree will always be able to fit training data perfectly. This is called *polynomial interpolation.*
- What is the problem with adding too many features? Overfitting. The fitted model will not generalize well to unseen data. Ultimately, our goal is to minimize the expected error over *all possible* data.

Definition 5.12 (Expected Loss)**.** *We assume all our data (including any unseen data) comes from some unknown distribution $(\boldsymbol{x}, y) \sim P(X, Y)$, where X denotes the entire input space and Y the output space. Then the **expected loss** is defined as*

$$L(\hat{f}) = \mathbb{E}_{\boldsymbol{x},y}\left[L(\hat{f}, \boldsymbol{x})\right],$$

where $L(\hat{f}, \boldsymbol{x})$ is the loss incurred by $\hat{f}$ at data point $(\boldsymbol{x}, y)$.

Remarks:

- Expected loss is also referred to as *risk.*
- Unfortunately, we cannot calculate the expected loss as we do not know the probability distribution P. Thus we also cannot directly minimize it.
- So far, we have instead minimized the loss on our dataset D. This is called the empirical loss (or empirical risk).

Definition 5.13 (Empirical Loss)**.** *We estimate the expected loss by the empirical loss on a dataset D, given by*

$$\hat{L}_D(\hat{f}) = \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} L(\hat{f}, \boldsymbol{x}).$$

Remarks:

- The empirical squared error loss is exactly the (normalized) total loss from Definition 5.3.
- By the law of large numbers $\hat{L}_D(\hat{f}) \to L(\hat{f})$ for any fixed $\hat{f}$ almost surely as $n \to \infty$. Therefore the more data we have, the closer the empirical loss will be to the expected loss and so, the closer $\hat{f}$ can be to the true function f by minimizing the empirical loss.

- But how do we know how well our $\hat{f}$ performs on the rest of the domain X?
- We could take the empirical loss as our estimate. Unfortunately, since we fit our model to this data, it will inherently underestimate the expected loss.
- We could find new data for evaluation, but we usually just have one dataset to work with. However, we can sample a subset D_t from our dataset D and only train our model with this subset, while reserving the rest for evaluation.

Definition 5.14 (Train-Evaluation Split)**.** *Partitioning a dataset D into two disjoint subsets D_t and D_e (typically 80% to 20%) is called a* ***train-evaluation split****.*

Remarks:

- This is also called a *train-test* split. For ease of notation we will stick with evaluation.

Definition 5.15 (Training Loss, Evaluation Loss)**.** *We define the* ***training loss*** *as*

$$\hat{L}_t(\hat{f}) = \frac{1}{|D_t|} \sum_{(\boldsymbol{x},y)\in D_t} L(\hat{f}, \boldsymbol{x}).$$

Similarly, we define the ***evaluation loss*** *as*

$$\hat{L}_e(\hat{f}) = \frac{1}{|D_e|} \sum_{(\boldsymbol{x},y)\in D_e} L(\hat{f}, \boldsymbol{x}).$$

Remarks:

- Note that $\hat{f}$ depends on the training data, so $\hat{f} = \hat{f}_{D_t}$.
- We can use the evaluation dataset D_e to estimate how well the function $\hat{f}_{D_t}$ generalizes to new data.

Definition 5.16 (Overfitting, Underfitting)**.** *A model is* ***overfitting*** *when it fits the training dataset D_t too well, learning random patterns/noise that will not be present in new unseen data D_e. The model will not generalize well. Conversely, a model is* ***underfitting*** *when it is not expressive enough to approximate f. A more complex model $\mathcal{F}'$ should be tried.* → notebook

Remarks:

- $\hat{L}_e(\boldsymbol{w}) \gg \hat{L}_t(\boldsymbol{w})$ is a clear indication of overfitting.
- See Figure 5.17 for examples of overfitting and underfitting.

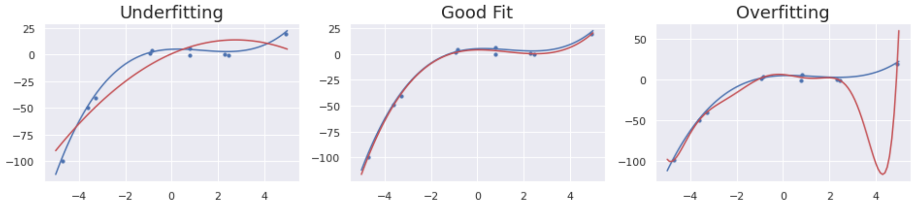


Figure 5.17: Examples of underfitting and overfitting

Remarks:

- There are several ways we can counter overfitting. We could try a simpler model, gather more training data, or introduce *regularization*.
- High training and evaluation errors could be a sign of underfitting. However, high errors could also indicate that the data is inherently noisy/random and cannot be fitted well.
- By repeating the train-evaluation split process multiple times we can get a more accurate evaluation of how well our model generalizes, and whether or not it is overfitting or underfitting. This is known as *cross validation*.

Definition 5.18 (Cross Validation). *In k-fold cross validation, we randomly partition D into k equal sized subsets. We train a model on the union of $k-1$ of these subsets. Then we evaluate our model on the last, withheld subset. This is repeated k times, with each subset used $k-1$ times as part of the training data and once as evaluation data. This gives k evaluation scores, which are averaged to produce a single evaluation metric.*

Remarks:

- There are other types of cross validation. For example in *Monte Carlo cross validation*, D_e is randomly sampled from D each time.
- Cross validation can also be used for *model selection*: We first do a train-evaluation split and then use cross validation on the training set D_t to select our model $\mathcal{F}$. The best model is then evaluated on the as yet unseen evaluation data.
- The mean and variance of the error across splits can tell us more about the sources of error in our model.

5.4 Bias-Variance Tradeoff

Definition 5.19 (Bias, Variance). ***Bias** is the expected error of a model, that is, how much the model $\mathcal{F}$ deviates from the target value on average. Formally, with $\bar{f}(\boldsymbol{x}) = \mathbb{E}_D\left[\hat{f}(\boldsymbol{x})\right]$:*

$$Bias^2[\mathcal{F}] = \mathbb{E}_{\boldsymbol{x}}\left[\left(f(\boldsymbol{x}) - \bar{f}(\boldsymbol{x})\right)^2\right],$$

where we have two expectations, one taken over $\boldsymbol{x}$ *and one over* D. *The* ***variance*** *of a model is the variance in its predictions when training on different random datasets. Formally:*

$$Var[\mathcal{F}] = \mathbb{E}_{\boldsymbol{x},D}\left[\left(\hat{f}(\boldsymbol{x}) - \bar{f}(\boldsymbol{x})\right)^2\right].$$

Remarks:

- If the bias is large, we know that our model $\mathcal{F}$ cannot approximate f well and we should consider a more expressive model, i.e., a more general family of functions $\mathcal{F}$. If on the other hand $f \in \mathcal{F}$, then the bias will be zero.
- Note that random noise will not contribute to the bias. On average (taking the expectation over D), the effects due to noise will cancel out. The bias really only covers systematic errors because our model is not able to match f exactly (even with the best weights). See Figure 5.20 (left).
- The variance tells us how sensitive our model is to the specific D. If the model is very sensitive, then $\boldsymbol{w}$ and ultimately the predictions will vary greatly depending on the dataset, in other words it will overfit. More noise will increase the variance. See Figure 5.20 (right).
- The simplest possible model is a constant, $\hat{f} = c$. This always has zero variance. Its predictions never change.

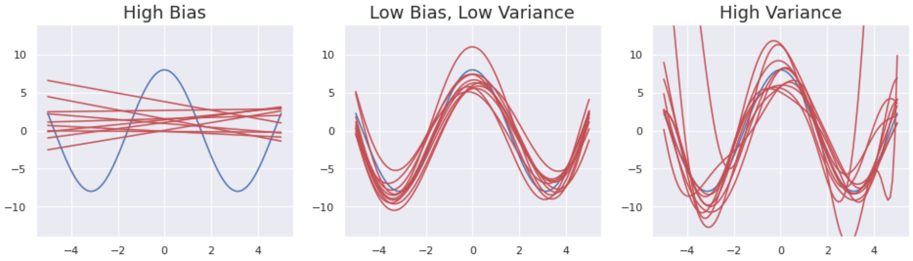


Figure 5.20: Fitting polynomials of increasing degree to the cosine function within the range $[-5, 5]$. The first order model (left) is too simple and has a high bias. On the other hand the 11th order model (right) is too complex and has a high variance. The 6th order model (middle) has low bias and low variance. In each iteration $f(x) = 8\cos(x)$, 25 inputs were sampled uniformly from $[-7, 7]$ and y was calculated with additional Gaussian noise. Predictions were then plotted on the interval $[-5, 5]$.

Theorem 5.21 (Bias-Variance Decomposition)**.** *We can decompose the mean squared error (*MSE*) of a model into its squared bias and its variance:*

$$\text{MSE}(\mathcal{F}) = Bias^2[\mathcal{F}] + Var[\mathcal{F}]$$

where the mean squared error (or expected squared error) equals:

$$\text{MSE}(\mathcal{F}) = \mathbb{E}_{\boldsymbol{x},D}\left[\left(y - \hat{f}(\boldsymbol{x})\right)^2\right]$$

Remarks:

- In general our dataset will not cover the whole input space and the data will contain some noise. This means bias or variance or both will generally be greater than zero.

- However, there is an inherent tradeoff between bias and variance. A more complex model will be able to approximate f better, giving lower bias. But this also allows it to fit noise better leading to higher variance, since the noise it fits is random. On the other hand a simpler model will generally have higher bias and lower variance.

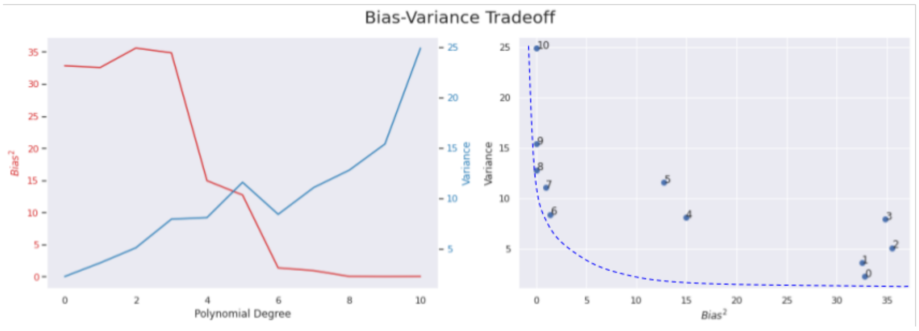


Figure 5.22: Fitting polynomials of increasing degree to the cosine function within the range $[-5, 5]$. On the left we see that as the degree (complexity) of the model increases, the bias decreases and the variance increases. The plot on the right shows the bias variance tradeoff frontier (dashed line). The frontier has been added to visualize the tradeoff trend, it is not a hard boundary that cannot be crossed.

- If we have high variance, regularization is one way of reducing the total error. Regularization focuses on decreasing the variance at the potential expense of increasing the bias.

5.5 Regularization

How about including all the features we may possibly want, but charging a cost for each non-zero weight w_i? This should help us reduce overfitting to noise, but still allows us to fit f well. This is what Lasso does.

Definition 5.23 (Lasso Regression)**.** *Lasso regression minimizes*

$$\min_{\boldsymbol{w}} \left\{ \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} (y - \boldsymbol{w}^T\boldsymbol{x})^2 + \lambda \sum_{i=0}^{d-1} |w_i| \right\}$$

Remarks:

- The regularization parameter λ weighs the parameter cost versus the MSE loss. It is a so-called *hyperparameter* that can be tuned.

Definition 5.24 (Hyperparameter)**.** *A **hyperparameter** is a parameter that controls the training process and whose value has to be chosen in advance.*

Remarks:

- The degree m in polynomial regression was also a hyperparameter.
- Cross validation can be used for hyperparameter tuning.
- For each hyperparameter you can define a set of values. Exhaustively iterating through all combinations of these values is called *grid search*.

Remarks:

- Lasso introduces bias into the regression solution by guiding the weights to be close to zero. This can reduce variance considerably relative to the OLS solution, see Figure 5.25.

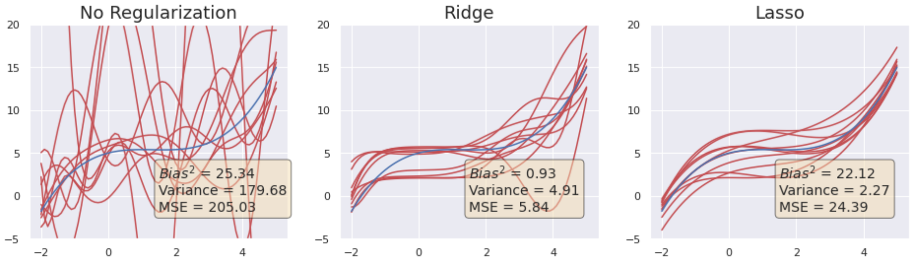


Figure 5.25: Fitting a polynomial of high degree to a cubic function with and without regularization. We see that variance and total error drops when regularization is added.

Remarks:

- When using regularization, features should be normalized (subtract mean, divide by standard deviation). This ensures that the coefficient of a feature is not influenced by its magnitude.
- The target vector $\boldsymbol{y}$ should also be centered around 0, so that the intercept w_0 does not count towards the total cost.
- Ridge has the same objective, but Ridge uses the L^2 norm for the cost of the weights, whereas Lasso uses the L^1 norm. In other words, we replace $|w_i|$ with w_i^2.
- Ridge has a closed form solution, just like OLS. However Lasso does not, so we have to solve it differently, for example by *gradient descent*.

5.6 Gradient Descent

The OLS method had a closed form solution (Lemma 5.4 and Theorem 5.6), but often we do not have a closed form solution to our optimization problem. An alternative is to minimize the loss function using gradient based methods. If we can calculate the gradient of the loss function with respect to the weights, then we can perturb the weights in the right direction to decrease the loss. We can repeat this process until reaching a minimum.

Definition 5.26 (Gradient Descent)**.** *Given a loss function $L(\hat{f}(\boldsymbol{w}), D)$, repeatedly perform the update*

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} L(\hat{f}, D)$$

simultaneously for $j = 0, 1, ..., d$, where hyperparameter α is the ***learning rate****.*

Remarks:

- The partial derivative tells us which weights to decrease or increase. If $\frac{\partial}{\partial w_j} L(\hat{f}, D)$ is positive, then the loss is increasing in w_j, so we decrease w_j to lower the loss.
- The gradient, i.e., the vector of partial derivatives, $\nabla L(\hat{f}(\boldsymbol{w}), D)$ points in the direction of steepest ascent, with the negation pointing in the direction of steepest descent.
- A (loss) function can have multiple minima, and gradient descent may converge to a local minimum rather than finding the global minimum. The minimum reached depends on the initial starting point and the learning rate α. See Figure 5.27.

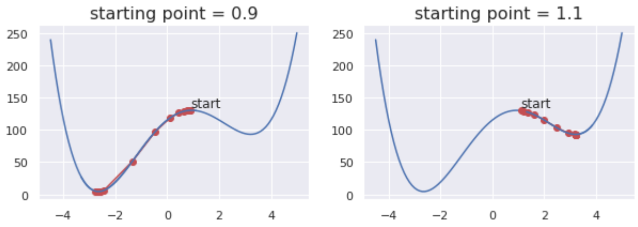


Figure 5.27: Gradient descent with different initialization points. On the (left) the initialization is favorable and we reach the global minimum of the polynomial loss function. However on the (right) gradient descent gets stuck in a local minimum.

Remarks:

- The *learning rate* α is a hyperparameter that controls the size of each update step. If the learning rate is too high, the algorithm might jump beyond the optimum $\boldsymbol{w}$; if it is too low the algorithm will be very slow to converge. See Figure 5.28 for examples. Often a decaying learning rate is used for efficiency at the beginning and accuracy towards the end of training. → notebook

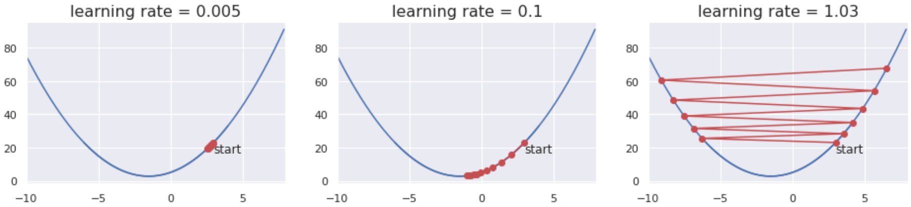


Figure 5.28: Gradient descent with different learning rates for finding the minimum of a quadratic loss function. On the (left) the learning rate is very low so convergence is slow and on the (right) the learning rate is too high so gradient descent diverges.

Remarks:

- Gradient descent is very similar to Newton's method for optimization, but in Newton's method the learning rate is not a hyperparameter, but 1 over the second derivative of the loss function, $\frac{\partial^2}{\partial w^2} L(\hat{f}, D)$. Newton's method often converges in fewer steps, but unfortunately the second derivative is usually hard to calculate.

- Any differentiable loss function can be optimized with gradient descent. If the loss function is convex (see Definition **??**), then the global minimum will eventually be reached (with a suitable learning rate).

- When the training dataset is large, it is often too costly to calculate $\frac{\partial}{\partial w_j} L(\hat{f}, D)$ over the whole dataset D just to make a single update step. In practice the training data is often shuffled and split into minibatches D_i (subsets of equal size) and $\frac{\partial}{\partial w_j} L(\hat{f}, D_i)$ is used in the update step. This is called *stochastic gradient descent* (SGD).

- Even though we have a closed form solution for minimizing the loss in linear regression, let's derive the corresponding update rule to see how this works.

Theorem 5.29 (LMS Rule)**.** *The **Least Mean Squares** (LMS) update rule for linear regression is given by*

$$w_j := w_j + \alpha \left(y - \boldsymbol{w}^T \boldsymbol{x} \right) x_j$$

And the batch update rule is given by

$$w_j := w_j + \alpha \sum_{(\boldsymbol{x}, y) \in D} \left(y - \boldsymbol{w}^T \boldsymbol{x} \right) x_j$$

Proof. We derive the batch update rule. Recall the squared error loss function

$$L(\hat{f}, D) = (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w}).$$

And recall from Theorem 5.6 that the derivative with respect to $\boldsymbol{w}$ is given by

$$\frac{\partial L}{\partial \boldsymbol{w}} = -2\mathbf{X}^T (\boldsymbol{y} - \mathbf{X}\boldsymbol{w})$$

Substituting into the gradient descent formula, we get the update rule

$$\begin{aligned} w_j &:= w_j + 2\alpha \left(\mathbf{X}^T(\boldsymbol{y} - \mathbf{X}\boldsymbol{w})\right)_j \\ &= w_j + 2\alpha \sum_{(\boldsymbol{x},y)\in D} \left(y - \boldsymbol{w}^T\boldsymbol{x}\right) x_j \end{aligned}$$

If we scale the learning rate by a half we get the LMS batch update rule. And taking a single training sample for D we get the single update rule. □

5.7 Logistic Regression

So far we have discussed how to learn a function $f : x \mapsto y$. when $y \in \mathbb{R}$. In this section we introduce a method for binary classification, where y is either 0 or 1, i.e., $y \in \{0, 1\}$. Similar to linear regression, we learn a linear function $\boldsymbol{w}^T\boldsymbol{x}$, but now we classify all samples with $\boldsymbol{w}^T\boldsymbol{x} > 0$ as 1, all samples with $\boldsymbol{w}^T\boldsymbol{x} < 0$ as 0. In other words we find a hyperplane to separate the classes. There are different approaches to do so, we present a statistically motivated approach called logistic regression. The key idea is simple: We take the output of the linear function and squash it into the range $[0, 1]$. We treat this squashed output as a probability. The more sample $\boldsymbol{x}$ is in the direction of vector $\boldsymbol{w}$, the higher the probability that sample $\boldsymbol{x}$ belongs to class 1.

Definition 5.30 (Binary Logistic Regression). *We want to find an approximation $\hat{f}(\boldsymbol{x}) \approx f(\boldsymbol{x}) = y \in \{0, 1\}$. We choose the following form for $\hat{f}$:* → notebook

$$\hat{f}(\boldsymbol{x}) = \psi(\boldsymbol{w}^T\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{w}^T\boldsymbol{x})}$$

*where ψ is called the **logistic** or **sigmoid** function.*

Remarks:

- The logistic function ψ squashes inputs from $[-\infty, \infty]$ into the range $[0, 1]$. Other functions, such as the *probit link function*, could be used instead.

- $\hat{f}(\boldsymbol{x})$ can be seen as an estimate of the probability that $y = 1$. Therefore, we usually classify $\boldsymbol{x}$ as positive (1) if $\hat{f}(\boldsymbol{x}) > 0.5$ and as negative (0) if $\hat{f}(\boldsymbol{x}) < 0.5$.

- This defines a *decision boundary* $\psi(\boldsymbol{w}^T\boldsymbol{x}) = 0.5$, or $\boldsymbol{w}^T\boldsymbol{x} = 0$, where everything on one side of the boundary is classified as positive and everything on the other side as negative. Samples on the boundary can be classified arbitrarily as positive or negative. The decision boundary is linear in the features.

- As in linear regression, we again can add higher order features, e.g., x^2 or $sin(x)$, to learn non-linear decision boundaries.

- How do we find the optimal values for $\boldsymbol{w}$? We could minimize the linear regression squared error loss (Definition 5.3). However, this would give predictions beyond $[0, 1]$.

- Instead we apply the logistic function, interpret the output as a probability and choose the model that maximizes the likelihood of generating exactly the labels in our training data.

Definition 5.31 (Bernoulli Likelihood function)**.**

$$\mathcal{L}(\boldsymbol{w}) = \prod_{(\boldsymbol{x},y)\in D} P(y \mid \boldsymbol{x}, \boldsymbol{w}) = \prod_{(\boldsymbol{x},1)\in D} \psi(\boldsymbol{w}^T\boldsymbol{x}) \prod_{(\boldsymbol{x},0)\in D} (1 - \psi(\boldsymbol{w}^T\boldsymbol{x}))$$

Remarks:

- $\mathcal{L}(\boldsymbol{w})$ is the probability of observing the vector of outputs $\boldsymbol{y}$ given input data $\mathbf{X}$ and parameters $\boldsymbol{w}$. Intuitively, we want to choose $\boldsymbol{w}$ such that we maximize this probability, i.e., we want to choose the parameter values that make our observations the most likely to occur. This is called Maximum Likelihood Estimation (MLE).

Lemma 5.32 (Logistic Regression Loss Function or Log Loss)**.** *Assuming the y's in $\boldsymbol{y}$ are independent and identically distributed Bernoulli with parameters $p = \hat{f}(\boldsymbol{x}) = \psi(\boldsymbol{w}^T\boldsymbol{x})$, maximizing $\mathcal{L}(\boldsymbol{w})$ is equivalent to minimizing the following loss function:*

$$L(\hat{f}, D) = -\frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \Big[y \log(\hat{f}(\boldsymbol{x})) + (1-y)\log(1-\hat{f}(\boldsymbol{x})) \Big]$$

Proof.

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}) &= \prod_{(\boldsymbol{x},1)\in D} \psi(\boldsymbol{w}^T\boldsymbol{x}) \prod_{(\boldsymbol{x},0)\in D} (1 - \psi(\boldsymbol{w}^T\boldsymbol{x})) \\
&= \prod_{(\boldsymbol{x},y)\in D} (\psi(\boldsymbol{w}^T\boldsymbol{x}))^y (1 - \psi(\boldsymbol{w}^T\boldsymbol{x}))^{1-y} \\
&= \prod_{(\boldsymbol{x},y)\in D} \hat{f}(\boldsymbol{x})^y (1 - \hat{f}(\boldsymbol{x}))^{1-y}
\end{aligned}$$

In practice we maximize the logarithm of the likelihood function because the product simplifies to a sum, which prevents numerical problems and makes differentiation simpler. Taking the logarithm does not change the optimal $\boldsymbol{w}$:

$$\begin{aligned}
\log \mathcal{L}(\boldsymbol{w}) &= \sum_{(\boldsymbol{x},y)\in D} \Big[y \log(\hat{f}(\boldsymbol{x})) + (1-y)\log(1-\hat{f}(\boldsymbol{x})) \Big] \\
&= -n \cdot L(\hat{f}, D)
\end{aligned}$$

Therefore $\operatorname{argmax}_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \operatorname{argmin}_{\boldsymbol{w}} L(\hat{f}, D)$.

□

Remarks:

- The scaling factor n does not change the optimum $\boldsymbol{w}$, but it averages the value of the loss across samples, hence allowing for better comparison across models.
- If $y = 1$ and $\hat{f} = 1$, then $L = -\log(\hat{f}) = -\log(1) = 0$, i.e, our classifier is correct with perfect "confidence", and hence the loss is zero. If on the other hand $y = 1$ and $\hat{f} = 0$, then $L = -\log(\hat{f}) = -\log(0) \approx \infty$, i.e., our classifier is wrong with perfect "confidence", which incurs very high cost. Similarly if $y = 0$ and $\hat{f} = 0$, then $L = 0$, and if $y = 0$ and $\hat{f} = 1$, then $L = \infty$.
- Unfortunately, there is in general no closed form solution for logistic regression. However, we can use gradient descent (Definition 5.26) to minimize $L(\hat{f}, D)$. But first we need to calculate the gradient.

Lemma 5.33 (Gradient of the Log Loss). *The gradient of $L(\hat{f}, D)$ from Lemma 5.32 with respect to w_j is given by*

$$\frac{\partial L}{\partial w_j} = \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \left[\hat{f}(\boldsymbol{x}) - y\right] \cdot x_j$$

Proof. First we calculate the derivative of the logistic function:

$$\begin{aligned}
\frac{d}{dz}\psi(z) &= \frac{d}{dz}\left[\frac{1}{1+e^{-z}}\right] \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\
&= \psi(z) \cdot (1 - \psi(z))
\end{aligned}$$

Using this we can calculate the derivative of the loss for a single training sample, $L(\hat{f}, \boldsymbol{x})$, with respect to w_j:

$$\begin{aligned}
&\frac{\partial}{\partial w_j} L(\hat{f}, \boldsymbol{x}) = -\frac{\partial}{\partial w_j}\left[y\log(\hat{f}(\boldsymbol{x})) + (1-y)\log(1-\hat{f}(\boldsymbol{x}))\right] \\
&= -\frac{\partial}{\partial w_j}\left[y\log(\psi(\boldsymbol{w}^T\boldsymbol{x})) + (1-y)\log(1-\psi(\boldsymbol{w}^T\boldsymbol{x}))\right] \\
&= -\left[\frac{y}{\psi(\boldsymbol{w}^T\boldsymbol{x})} - \frac{1-y}{1-\psi(\boldsymbol{w}^T\boldsymbol{x})}\right] \cdot \frac{\partial}{\partial w_j}\psi(\boldsymbol{w}^T\boldsymbol{x}) && \text{(chain rule)} \\
&= -\left[\frac{y}{\psi(\boldsymbol{w}^T\boldsymbol{x})} - \frac{1-y}{1-\psi(\boldsymbol{w}^T\boldsymbol{x})}\right] \cdot x_j \cdot \psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x})) && \text{(chain rule)} \\
&= -\left[\frac{y-\psi(\boldsymbol{w}^T\boldsymbol{x})}{\psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x}))}\right] \cdot x_j \cdot \psi(\boldsymbol{w}^T\boldsymbol{x})(1-\psi(\boldsymbol{w}^T\boldsymbol{x})) \\
&= -\left[y - \psi(\boldsymbol{w}^T\boldsymbol{x})\right] \cdot x_j \\
&= \left[\hat{f}(\boldsymbol{x}) - y\right] \cdot x_j
\end{aligned}$$

And since differentiation and finite summation are interchangeable, i.e.,

$$\frac{d}{dx}\sum g(x) = \sum \frac{d}{dx} g(x)$$

we get the gradient for the total loss, $L(\hat{f}, D)$, as:

$$\frac{\partial L(\hat{f}, D)}{\partial w_j} = \frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \left[\hat{f}(\boldsymbol{x}) - y\right] \cdot x_j$$

□

Remarks:

- We can then use gradient descent (Def. 5.26) to update the model parameters.
- We can also add lasso or ridge regularization to prevent our model from overfitting (Def. 5.23).
- What if y can take on more than two values? A straightforward way to extend binary logistic regression to $k > 2$ classes is to train k separate logistic regression models, one for each class. We then choose the class with the highest probability score. This is a general method for extending binary classifiers to multinomial problems, and is referred to as One versus Rest (OvR).
- There are other ways to extend logistic regression to the multinomial case. For example, we can use the softmax function instead of the sigmoid function.

Definition 5.34 (Softmax Regression)**.** ***Softmax regression*** *with $k \geq 2$ classes chooses the following functional form for $\hat{f}$:*

$$\hat{f}(\boldsymbol{x})_i = \sigma(\boldsymbol{w}_{(1)}^T\boldsymbol{x}, \boldsymbol{w}_{(2)}^T\boldsymbol{x}, ..., , \boldsymbol{w}_{(k)}^T\boldsymbol{x})_i = \frac{\exp(\boldsymbol{w}_{(i)}^T\boldsymbol{x})}{\sum_{j=1}^k \exp(\boldsymbol{w}_{(j)}^T\boldsymbol{x})} \qquad \text{for } i = 1, \dots, k$$

where σ is called the ***softmax function****.*

Remarks:

- $\hat{f}(\boldsymbol{x})$ is a vector of length k, with elements summing up to 1. It can be seen as a vector of probabilities, with $\hat{f}(\boldsymbol{x})_i \approx \mathbb{P}(f(\boldsymbol{x}) = i)$.
- A different set of linear weights, $\boldsymbol{w}_{(i)}$, is learned for each class i. Since the probabilities sum up to 1, one set of weights is redundant.
- For $k = 2$, softmax regression reduces exactly to binary logistic regression (Definition 5.30) with $\boldsymbol{w} = \boldsymbol{w}_{(2)} - \boldsymbol{w}_{(1)}$.
- We also get the same form for the loss function:

Lemma 5.35 (Softmax Regression Loss Function)**.** *Assuming the y's in Y are independent and identically distributed categorical random variables with parameters $p_i = \sigma(\boldsymbol{w}_{(i)}^T \boldsymbol{x})$ for $i = 1, \dots, k$, then maximizing the likelihood $\mathcal{L}(\boldsymbol{w})$ is equivalent to minimizing the following loss function:*

$$L(\hat{f}, D) = -\frac{1}{n} \sum_{(\boldsymbol{x},y)\in D} \sum_{i=1}^{k} \left[\mathbb{1}\{y = i\} \log(\hat{f}(\boldsymbol{x})_i)\right]$$

where $\boldsymbol{w}$ is the set of all weights $\{\boldsymbol{w}_{(i)}\}_{i=1,\dots k}$ and $\mathbb{1}\{\cdot\}$ is the indicator function.

Remarks:

- Note that for $k = 2$ this loss is identical to the loss for logistic regression in Lemma 5.32, only the notation has been changed to use the indicator function.

- In order to learn non-linear decision boundaries with logistic regression, we need to do feature engineering. This can be challenging and time consuming, and it also makes the resulting model more difficult to interpret. In the following we introduce a different type of model that addresses these issues: Decision Trees.

5.8 Decision Trees

So far we have considered regression models based on the form $\boldsymbol{w}^T\boldsymbol{x}$ where the output is essentially a weighted sum of the input features, potentially squashed by a sigmoid function. Binary decision trees introduce hierarchy: We keep partitioning the input space into smaller regions. In order to make a prediction $\hat{f}(\boldsymbol{x})$, we simply start at the root of the decision tree and apply the decision rules until we reach a leaf, which then determines the output value.

```
1 def predict(self, x): # self is current node, initially root
2     if self.is_leaf():
3         return self.value
4     if x[self.feature] ≤ self.threshold:
5         self.left.predict(x)
6     else:
7         self.right.predict(x)
```

Algorithm 5.36: Decision tree algorithm.

Remarks:

- For classification, in Line 3 we return the majority class of the leaf. For regression, we return the average value of the leaf.

- Decision trees can learn non-linear decision boundaries and are easy to interpret and visualize. → notebook

- Decision trees are binary trees that contain nodes V, where each internal (non-leaf) node has an associated splitting rule. Every node v corresponds to a subset $D_v \subseteq D$.
- How are splitting rules defined?

Definition 5.37 (Decision Tree Splitting Rule)**.** *A **splitting rule** of an internal node v is given by a tuple (i, t), where i is the index of a feature and t is a threshold value. A split defines **left** and **right** subsets*

$$D_{v,l} = \{\boldsymbol{x} \mid x_i \leq t\}, \quad D_{v,r} = \{\boldsymbol{x} \mid x_i > t\}$$

Remarks:

- Unlike everything so far, a splitting rule here is based on a single feature value and not a linear combination of features. As such all splits are axis-aligned.
- Now we know how to use a decision tree to make predictions, but how do we build a decision tree in the first place? We do so by finding good splitting rules. And we find good splitting rules by minimizing a loss function of course!

Definition 5.38 (Regression Tree Loss: MSE)**.** *For node v with samples D_v the mean squared error (*MSE*) loss is defined as:*

$$L(D_v) = \frac{1}{|D_v|} \sum_{y \in D_v} (y - \bar{y})^2$$

where the prediction $\bar{y}$ of a node is the average of the target values of all samples in D_v.

Remarks:

- The MSE is the variance of the target value. The aim is to create splits that lower the total variance in the leaves.
- For classification the loss function is a measure of purity. We call a node with all samples belonging to the same class perfectly pure. We aim to find splits that successively increase the purity of the nodes. The most common measures of purity are *entropy* and *Gini impurity.*
- We now have a measure of loss for each node, but we need to combine the loss of the left and right subsets to decide what the next split should be. One could consider many things here: minimizing the total loss, minimizing the maximum loss, making balanced splits, etc. One natural choice is to minimize the weighted average loss.

Definition 5.39 (CART loss function)**.** *To find the best split (i^*, t^*) at node v, CART minimizes the following loss function:*

$$L(i, t) = \frac{|D_{v,l}|}{|D_v|} L(D_{v,l}) + \frac{|D_{v,r}|}{|D_v|} L(D_{v,r})$$

where the loss $L(\cdot)$ measures the impurity or error of the resulting left and right subsets.

Remarks:

- The weights ensure that all training samples have equal contribution. For example, for regression trees the weighted MSE loss reduces to the mean squared error over D_v. This is not the same as $L(D_v)$, since we use the two new mean values $\bar{y}_l$ and $\bar{y}_r$ to calculate the errors.
- CART trees are built recursively using binary splits, with every split minimizing above loss function.
- One could keep splitting until all the leaves contain single samples, like a binary search tree (Definition 4.2). This would give 100% accuracy on the training data, but would not generalize well to new data. Stopping criteria can be used to halt splitting. Typical stopping criteria are: maximum tree depth, minimum number of samples in a node, minimum decrease in the value of the loss function.
- Instead of stopping criteria, it is generally better to grow a large tree and then prune it back. This way we might add some useful additional splits and have the chance to remove less useful splits. Even if all additional splits are not helpful, we can still remove them when pruning. In essence pruning allows us to see some steps into the future, before finalizing our tree.
- CART is a greedy algorithm that finds good solutions, but is unlikely to find the optimal solution. Finding the optimal tree (e.g. a minimum depth decision tree) is NP-hard.
- One big advantage of decision trees is their simple interpretation. They suffer from high variance and overfit easily. A common technique to improve decision trees is to use many trees in an ensemble.

Definition 5.40 (Bootstrap Sample)**.** *A bootstrap sample D_b is obtained by drawing n samples from dataset D uniformly with replacement.*

Remarks:

- In expectation, every D_b contains $1 - 1/e = 63.2\%$ samples from D.
- What happens if we draw many bootstrap samples and use each to train a separate model?

Definition 5.41 (Bootstrap Aggregating or Bagging)**.** *Bagging is an ensemble algorithm where q models are trained on q bootstrap samples D_b. The models are combined either by averaging the outputs (regression) or by majority vote (classification).*

Remarks:

- Bagging ensembles can be built with any base learners, including decision trees. In general, an ensemble method can turn many low-bias high-variance base learners into a single low-bias low-variance model.
- The use of bootstrap samples de-correlates the individual learners, i.e., they make different mistakes which will average out.

- If we use decision trees as our base learners we can do even better. What if we add even more randomness to bagging by subsampling the features that can be chosen for finding the best splits?

Definition 5.42 (Random Forest)**.** *A random forest is a bagging ensemble of q decision trees. The learners $\hat{f}_b$ are trained in such a way that at every node only a random subset of the features can be used for finding the best split.*

Remarks:

- The random subsampling of features for each split further de-correlates the individual trees, making random forests powerful models that can achieve both relatively low bias and low variance.

5.9 Evaluation

How can we know whether our models are performing well? For regression we can simply measure the models MSE (or absolute error) on the evaluation set (Definition 5.15). However, for classification, the value of the loss function is not intuitive; it does not directly tell us how good our model is at classifying samples. To get a sense of the performance of a classifier, the most important tool is the confusion matrix.

Definition 5.43 (Confusion Matrix)**.** *A confusion matrix, also known as error matrix, visualizes the performance of a classifier on a given dataset. Rows represent the actual class labels, and columns contain the predictions of the classifier.*

		Predicted label	
		p	**n**
True label	$\mathbf{p}'$	True Positive TP	False Negative FN
	$\mathbf{n}'$	False Positive FP	True Negative TN

Remarks:

- From the confusion matrix we can derive different metrics to summarize the performance of a classifier:

accuracy	$\mathsf{ACC} = \frac{\mathsf{TP+TN}}{\mathsf{P+N}}$
positive predictive value (precision)	$\mathsf{PPV} = \frac{\mathsf{TP}}{\mathsf{TP+FP}}$
true positive rate (recall)	$\mathsf{TPR} = \frac{\mathsf{TP}}{\mathsf{TP+FN}}$
false positive rate	$\mathsf{FPR} = \frac{\mathsf{FP}}{\mathsf{FP+TN}}$
F1 score	$F_1 = 2 \cdot \frac{\mathsf{PPV}\cdot\mathsf{TPR}}{\mathsf{PPV+TPR}}$

Remarks:

- Confusion matrices and the derived metrics can be used with more than two classes. However, the derived metrics can then be computed in several different ways. The *macro* method first computes the metrics for each class and then averages these values. The *micro* method aggregates the predictions from all samples and computes the metrics directly.
- Another very common way of evaluating binary classifiers ($k = 2$) is the Receiver Operator Characterstic (ROC) Curve and the derived Area Under Curve (AUC) metric.

Definition 5.44 (Receiver Operator Characteristic (ROC) Curve)**.** *The ROC Curve plots the TPR against the FPR. Given a binary classifier $\hat{f}$, one can order the data points by $\hat{f}(\boldsymbol{x})$, and then plot the TPR and FPR for every possible classification threshold $\tau_{pr} \in [0, 1]$. The resulting graph is called the ROC Curve.*

Remarks:

- τ_{pr} is the threshold for which if $\hat{f}(\boldsymbol{x}) \geq \tau_{pr}$ we classify a sample as positive, and otherwise as negative.
- Trade-off between TPR and FPR: If we want 100% TPR (recall), we can simply classify every sample as positive. However, we would then also (wrongly) classify every negative sample as positive, leading to a high FPR. An increase in the TPR generally leads to an increase in the FPR.
- A perfect classifier would achieve $TPR = 1$ at $FPR = 0$, i.e., the curve would “hug” the top left corner.
- The ROC curve of a random classifier follows the diagonal, i.e., $TPR = FPR$.
- The area under the ROC curve (AUC) is often used as a convenient summary of a classifier’s performance
- Intuitively, the AUC metric tells us the following: Given a random negative sample $\boldsymbol{x}_N$ and a random positive sample $\boldsymbol{x}_P$, what is the probability that $\hat{p}(\boldsymbol{x}_P) > \hat{p}(\boldsymbol{x}_N)$, i.e, the classifier will assign higher probability to the random positive sample than to the negative sample.

Chapter Notes

Ordinary least squares (OLS) was one of the first statistical methods to be developed, circa 1800 [5]. There is still controversy over who first applied it, Gauss or Legendre. Subsequently, weighted least squares, minimization of other norms (e.g., L1), multivariate minimization, regularization (e.g., Ridge, Lasso) and many other tools were developed. The term "ordinary" was added to least squares only after many alternative methods were suggested.

Gradient descent is generally attributed to Cauchy, who first suggested it in 1847 [4]. Hadamard independently proposed a similar method in 1907 [3]. Its convergence properties for non-linear optimization problems were first studied by Haskell Curry in 1944 [2], with the method becoming increasingly well-studied and used in the following decades, also often called steepest descent.

The logistic function was developed as a model of population growth and named "logistic" by Pierre François Verhulst in the 1830s and 1840s [1]. The logistic model was likely first used as an alternative to the probit model by Edwin Bidwell Wilson and Jane Worcester in 1943 [6]. The probit model is similar to logistic regression, but models the output as the cumulative distribution of a normal distribution centered around $\boldsymbol{w}^T\boldsymbol{x}$. The logit model was initially dismissed as inferior to the probit model, but gradually achieved an equal footing before surpassing it. Its popularity is credited to its computational simplicity, mathematical properties, and generality, allowing its use in varied fields.

This chapter was written in collaboration with Gino Brunner and Béni Egressy.

Bibliography

[1] J.S. Cramer. The origins of logistic regression. *Tinbergen Institute, Tinbergen Institute Discussion Papers*, 01 2002.

[2] Haskell B Curry. The method of steepest descent for non-linear minimization problems. *Quarterly of Applied Mathematics*, 2(3):258–261, 1944.

[3] Jacques Hadamard. *Mémoire sur le problème d'analyse relatif à l'équilibre des plaques élastiques encastrées*, volume 33. Imprimerie nationale, 1908.

[4] Claude Lemaréchal. Cauchy and the gradient method. *Doc Math Extra*, 251(254):10, 2012.

[5] Stephen M. Stigler. Gauss and the Invention of Least Squares. *The Annals of Statistics*, 9(3):465 – 474, 1981.

[6] E. B. Wilson and J Worcester. The determination of l.d.50 and its sampling error in bio-assay. *Proceedings of the National Academy of Sciences of the United States of America*, 29 2:79–85, 1943.

Chapter 6

Neural Networks

Computers are better than humans at playing Chess, Go, Poker, Dota, or Starcraft. They compose pop songs, write fiction stories, draw paintings, replace actors in movies, and drive vehicles. Whenever a computer does something mind-boggling, you can bet that a neural network is involved. Neural networks have become fascinating function approximators. How so? At their core, neural networks are based on simple linear mappings, combined with non-linear activation functions and gradient descent. So conceptually neural networks are not so different from our discussions in Chapter 5. But size matters! The biggest neural networks have up to 530 billion weights. So training needs data, hardware and patience.

6.1 Nodes and Networks

Definition 6.1 (Node)**.** *A node (or neuron) is a computing unit v that produces an activation value y. The node v first calculates an affine transformation on $\boldsymbol{x} \in \mathbb{R}^d$, then applies an activation function $\sigma : \mathbb{R} \to \mathbb{R}$:*

$$y = \sigma(\boldsymbol{w}^T \boldsymbol{x}),$$

where $\boldsymbol{x}$ is an input vector and $\boldsymbol{w} \in \mathbb{R}^d$ are learned weights. We call $z = \boldsymbol{w}^T \boldsymbol{x}$ the pre-activation value. Like in Chapter 5 (Definitions 5.5 and 5.30), we assume that w_0 is integrated into $\boldsymbol{w}$, i.e., $\boldsymbol{w} = (w_0, w_1, \ldots, w_{d-1})^T$, and $\boldsymbol{x}$ includes an additional constant 1, i.e., $\boldsymbol{x} = (1, x_1, \ldots, x_{d-1})^T$.

Remarks:

- In the literature, the intercept w_0 is sometimes referred to as “bias” b, and kept separate from $\boldsymbol{w}$, i.e., $y = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$. This naming complicates the vector notation; it may also be confusing since we used the term bias for a model property in Section 5.4.

- The activation function σ can take many different forms. Some nodes may simply use the identity as activation function, i.e., $\sigma(z) = z$. Most nodes apply non-linear activation functions in order to allow the model to approximate non-linear functions, e.g. the sigmoid function $\psi(\boldsymbol{w}^T \boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{w}^T \boldsymbol{x})}$ of Definition 5.30.

- In order to allow for gradient-based training, the activation function must be differentiable.

- We combine many neural nodes into a network:

Definition 6.2 (Neural Network)**.** *A neural network is a directed acyclic graph (DAG) formed by a set of nodes V that are connected by a set of directed edges E. The input $\boldsymbol{x}$ of the network is stored by the n input nodes V_i (with no incoming edges). The output $\boldsymbol{y}$ of the network will be computed by the m output nodes V_o (with no outgoing edges). All other nodes (with incoming and outgoing edges) are called hidden nodes V_h. We have $V_i + V_h + V_o = V$. Note that we use the letters x and y to refer to both, the input and output of the whole network as well as the input and output of a single node. We will use subscripts if the usage is not clear from the context.*

In neural networks the function is computed in the forward direction: The input $\boldsymbol{x}_v$ of each node v is the vector of computed outputs $\boldsymbol{y}$ of its DAG predecessor nodes. Then, v computes its own output as $y_v = \sigma(\boldsymbol{w}_v^T \boldsymbol{x}_v)$. Hence the nodes must be processed in DAG order.

Given an input $\boldsymbol{x} \in \mathbb{R}^n$, a neural network as a whole then approximates a function $f(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ by calculating $\hat{f}(\boldsymbol{x}) \approx f(\boldsymbol{x})$.

```
# $V = V_i \cup V_h \cup V_o$ = network
# $v.x$ = $v$'s input, the output of $v$'s DAG input-nodes
def forward($V$):
    for v in $V_h \cup V_o$ (in DAG order):
        v.y = v.$\sigma$(v.w,v.x)  # Definition 6.1
    return $V_o$
```

Algorithm 6.3: Feed-forward computation in DAG.

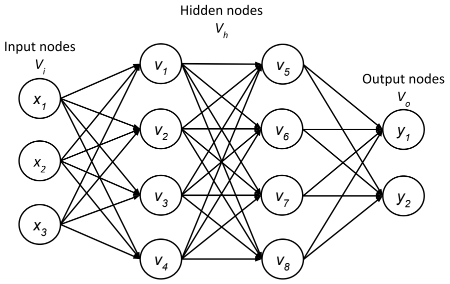


Figure 6.4: Example of a neural network for an input $\boldsymbol{x}$ and an output $\boldsymbol{y}$ with three input nodes, eight hidden nodes and two output nodes.

Remarks:

- There exist cyclic neural networks as well, see Definition 6.22.

- In neural networks, nodes generally follow a structure that can be represented with layers.

Definition 6.5 (Multi-Layer Perceptron or MLP). *The nodes are often organized in layers. The first layer are the input nodes V_i, the last layer the output nodes V_o. Each hidden node is in a layer l, and all nodes of layer l have the same input, namely the outputs y_v of all nodes v of layer $l-1$. Layered networks are known as Multi-Layer Perceptrons, where perceptron is an older name for node.*

Remarks:

- Layering will help us to speed up computation, as the pre-activation of a whole layer can be computed with a single matrix-vector multiplication $\boldsymbol{z} = \mathbf{W} \cdot \boldsymbol{x}$, where the matrix $\mathbf{W}$ is composed of the weight rows $\boldsymbol{w}$, $\boldsymbol{z}$ is the vector of pre-activation values in the nodes and $\boldsymbol{x}$ are the output values of the previous layer (with the additional 1).
- The number of layers determines the depth of the network. A "deep" network is a neural network with multiple layers.
- Can a neural network compute/approximate any function?

6.2 Universal Approximation

Theorem 6.6 (Universal Approximation Theorem). *Given a continuous function $f : \mathbb{R}_+ \to \mathbb{R}$ (for simplicity, input $x \geq 0$), there exists a neural network $\hat{f}$ with one hidden layer that approximates f arbitrarily well. That is*

$$|\hat{f}(x) - f(x)| < \varepsilon \textit{ for all } x \geq 0 \textit{ and } \varepsilon > 0.$$

Proof. We construct a neural network with sigmoid non-linearities in the hidden nodes v_i $(i > 0)$ and a single linear output node v_o.

The single output node v_o computes function $\hat{f}(x) = \boldsymbol{w}^T \boldsymbol{y}$, where $\boldsymbol{w}$ are v_o's weights, and y_i are the outputs produced by the hidden nodes. As usual, $\boldsymbol{w}$ includes an additional value w_0, and $\boldsymbol{y}$ starts with a additional constant 1. We choose $w_0 = f(0)$, such that $\hat{f}(0) = f(0)$ even without any hidden nodes. Every hidden node v_i computes $y_i = g_i(x) = \psi(\kappa \cdot x + b_i)$, where $\kappa \to \infty$ is a large constant. While the sigmoid function $\psi(z) = \frac{1}{1+\exp(-z)}$ from Definition 5.30 is a smooth step function, $\kappa \to \infty$ will make that step sharp.

The construction is inductive. We start out with $x = 0$, hence $\hat{f}(x) = f(x)$. As long as the difference between $\hat{f}(x)$ and $f(x)$ is less than ε we keep growing x. As soon as $|f(x) - \hat{f}(x)| \geq \varepsilon$, we introduce a new hidden node v_i. The value b_i of the hidden node v_i is representing the current position x, as $b_i = -\kappa \cdot x$. This makes sure that v_i will introduce a new step in $\hat{f}$ right at the current x. The weight w_i of v_o for the new input y_i is set as $w_i = f(x) - \hat{f}(x)$. This corrects the output of $\hat{f}$ for the newly accumulated error. If $f(x)$ was increasing, then a correcting $+\varepsilon$ step is added, if $f(x)$ was decreasing, a correcting $-\varepsilon$ step is added. In both cases, we again get $\hat{f}(x) \approx f(x)$. Figure 6.7 visualizes the effects of the parameters b_i and w_i. □

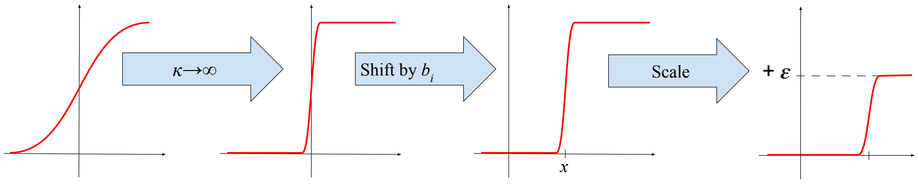


Figure 6.7: Effect of weights on the sigmoid function.

Remarks:

- Our proof is a simplified version of the original. The full theorem is more general, also applicable to continuous functions with inputs from a multi-dimensional compact set. There exist also versions using other activation functions than sigmoid, and multidimensional version, etc.
- The theoretical construction given in the proof is not used in practice, as it would lead to numerical instabilities ($\kappa \to \infty$).
- However, the promise of a neural network is not only to approximate any function, but rather to *learn* how to approximate any function. It can be shown via various reductions (Definition 2.8) that learning is NP-hard (Definition 2.20).

6.3 Training Neural Networks

During training, neural networks learn to automatically extract features from the raw input: In the forward computation the representation of the data held by the network becomes progressively closer to the value of the approximated function. Therefore, neural networks are effectively feature extractors.

Definition 6.8 (Feature Extractor ϕ)**.** *A feature extractor ϕ is a function that transforms raw input data $\boldsymbol{x}$ into features. These features represent the initial data in a way that simplifies approximating a function f.*

Remarks:

- In Section 5.2 we manually engineered ϕ. In Sections 5.3 and 5.4 we then learned how to tell whether we did a good job.
- Neural networks on the other hand automatically learn ϕ and thus, no manual feature extraction is needed. The idea is that every additional hidden layer of the network represents the data more abstractly than the previous one. With every layer, the representation of the data is less like input $\boldsymbol{x}$ and more like output $\boldsymbol{y}$.
- Why and how exactly this works is not well understood – this is the *mystique* of deep neural networks.
- Training a neural network is similar to training a linear regression node with gradient descent (Chapter 5): We calculate the gradient of the loss with respect to the network parameters and adjust the parameters accordingly. This is called backpropagation.

Definition 6.9 (Backpropagation)**.** *Backpropagation is an algorithm that computes the gradient of the loss L with respect to the parameters W of the neural network. In the DAG representation of a neural network, each node computes its pre-activation value z as a weighted sum over its input values $\boldsymbol{x}$. By the chain rule, we can calculate the error of the output nodes with respect to z as*

$$\frac{\partial L(\hat{f}, D)}{\partial z} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial z}.$$

Given this gradient, and using $\boldsymbol{z} = \boldsymbol{w}^T \boldsymbol{x}$, we can calculate the error with respect to the weights w_i and the error with respect to the node's inputs x_i as

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial z} \cdot x_i, \text{ and } \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \cdot w_i.$$

The gradients with respect to the weights $\boldsymbol{w}$ can then be used for updating the weights, while the gradient with respect to the inputs can be aggregated to pass the gradient $\frac{\partial L}{\partial y}$ to the preceding nodes in the network. Concretely, each node in the network adjusts its own weights $\boldsymbol{w}$ based on the error signal $\frac{\partial L}{\partial w}$ and tells its input nodes x_i to adjust by backpropagating $\frac{\partial L}{\partial x_i}$. The backpropagation algorithm is given in Algorithm 6.10.

```
# v.x = v's input, stored during forward computation
# v.z = pre-activation value, stored during forward computation
# v.err = initial error of the node, set to $\frac{\partial L}{\partial y}$ for output nodes
# v.err = 0 initially for hidden and input nodes
# v.w_grad = gradient vector of node v
# v.prev = list of indices of v's input nodes
def backward(V):
    for v in V (in reversed DAG order):
        v.z_err = v.err * $\frac{\partial v.y}{\partial v.z}$  # $\frac{\partial y}{\partial z}$ is the gradient of $\sigma(z)$
        v.w_grad = v.z_err * v.x  # v.w_grad and v.x are vectors
        for v_i in v.prev:
            v_i.err += v.z_err * v.w[v_i]
    return [v.w_grad for v in V]
```

Algorithm 6.10: Backpropagation Algorithm

Remarks:

- Many libraries backpropagate gradients with a simple function call. → notebook
- Memory may be problem, since we need to memorize $\boldsymbol{x}$ and z at every node.
- Backpropagation is only the method for computing the gradient, while another algorithm, such as stochastic gradient descent (Definition 5.26), is used to perform the parameter update using this gradient.

- Backpropagating gradients, i.e., applying the chain rule, means performing a number of multiplications. For deep neural networks this can lead to numerical issues.

Definition 6.11 (Vanishing Gradients)**.** *Gradient descent updates can stagnate due to vanishing gradients, i.e., gradients that are close to 0. These can occur during backpropagation for different reasons:*

- *The activation function $\sigma(\cdot)$ saturates, i.e. the gradient $\frac{\partial y}{\partial z} \approx 0$. In this case, the gradient $\frac{\partial L}{\partial w_i}$ of all weights w_i and the backpropagated gradients $\frac{\partial L}{\partial x_i}$ of the node will also be close to zero. The node stops learning.*
- *The summation in Line 12 of Algorithm 6.10 can accidentally become 0 (when terms cancel each other).*

Definition 6.12 (Exploding Gradients)**.** *The gradient calculations with backpropagation can also lead to devastatingly large gradients, called exploding gradients. Note that gradient descent only converges for sufficiently small gradient steps and too large gradients can lead to divergence. Reasons are:*

- *Some large weight $|w_i| \gg 1$ boosts the backpropagated error.*
- *The summation in Line 12 of Algorithm 6.10 can accidentally become large because of many positive (or negative) terms.*

Remarks:

- Vanishing or exploding gradients can propagate in a network, i.e. nodes with vanishing or exploding gradients can backpropagate the problem to their predecessor nodes.
- Simple yet generally effective solutions include reducing the number of layers or clipping the gradients (for the exploding case). A related solution is to normalize the activation values, as done in techniques such as *batch normalization* or *layer normalization*.
- Another effective solution for the vanishing gradient problem is the introduction of skip connections (connect nodes which are not in neighboring layers) which provide an additional path for the flow of information. During backpropagation this helps the gradients to continuously flow backwards, even if they vanish in certain points of the network.
- An additional option for mitigating the vanishing gradients problem is to use activation functions that do not saturate in both directions (positive and negative). The best-known of such activation functions is the Rectified Linear Unit activation (ReLU).

Definition 6.13 (ReLU)**.** *The Rectified Linear Unit activation is defined as:*

$$\sigma(x) = \max(0, x)$$

Remarks:

- The ReLU activation function introduces a nonlinear transformation that remains very close to linear (piecewise linear with only two pieces) and does not saturate in the positive direction.

- Remarkably, ReLU is non-differentiable, which violates Definition 6.1. In fact, for gradient-based learning it is enough if the subderivatives of the function exist (and for ReLU they do).

- The gradient $\frac{\partial y}{\partial z}$ of the ReLU activation is 0 when $x \leq 0$, 1 otherwise. This simplicity speeds up the computation of backpropagation.

- ReLU activation is the current default choice for neural networks. However, a large number of related activation functions exist, which modify some aspects of the function, e.g., leakyReLU has a non-zero slope for values smaller than 0. Some other functions are: PReLU, GeLU, SeLU, Maxout, etc.

- Another design choice that impacts the performance of the model is the initialization scheme.

Definition 6.14 (Initialization scheme)**.** *Rule that determines the initial parameter values W of a neural network, i.e., the values before training starts.*

Remarks:

- As seen in Figure 5.27, when the loss function is non-convex (has multiple local minima), starting the learning process at different points can lead to different solutions.

- Stochastic initialization is a good default for initializing the parameters of a neural network. These schemes give random initial values (with some constraints) to the parameters of the network in order to “break the symmetry”, i.e., to prevent that nodes with the same input and same activation converge to the same values during optimization.

- The loss landscape of neural networks is complex, with a large number of local minima. Surprisingly, converging to a local minimum during training is good enough for a neural network to perform well usually.

6.4 Practical Considerations

The complex loss landscape of neural networks makes the learning process significantly more complicated than in classical machine learning models. Therefore, sophisticated learning algorithms (also called optimizers) that build on top of stochastic gradient descent (Section 5.6) are used in practice. There is no consensus on which of the existing algorithms is best.

Remarks:

- Adam optimizer is currently considered a good default. It belongs to the family of adaptive learning rate algorithms, which adapt the learning rate for each parameter individually throughout the course of learning.
- Other popular optimizers include SGD with Momentum, RMSProp, and linear learning rate decay.
- The learning scheme/rate is probably the most important hyperparameter in neural networks. Finding an appropriate learning rate can produce a dramatic improvement in the performance of the network.
- Neural networks often have a remarkably large amount of hyperparameters, which do have a strong impact on the performance of the model. Although tuning hyperparameters is more an art than a science there are automatic hyperparameter optimization algorithms that can help in this process.

Definition 6.15 (Hyperparameter Optimization Algorithm)**.** *A hyperparameter optimization algorithm is an algorithm that wraps the learning algorithm of a model and chooses its hyperparameters, hiding this choice from the user.*

Remarks:

- When there are few hyperparameters to set, a common approach is Grid Search as discussed in Definition 5.24. The main problem of Grid Search is that the computational cost grows exponentially with the number of hyperparameters, which makes it expensive for large neural networks.
- An alternative is Random Search: the hyperparameter values are samples from a uniform distribution in a certain interval. Random search converges faster to an optimum.
- A large number of hyperparameter optimization algorithms exist using techniques such as evolutionary algorithms, Bayesian optimization or population-based-training.
- Hyperparameter optimization algorithms often have their own hyperparameters, such as the range of values that have to be explored. Fortunately, these secondary hyperparameters are easier to set in the sense that similar secondary hyperparameters can lead to acceptable performance in a wide range of tasks.

6.5 Regularization

Everything discussed so far portraits neural networks as powerful function approximators. Neural networks can approximate any continuous functions and even functions of high complexity, e.g., functions from a family with high VC dimension. The issue with this is that neural networks tend to overfit. To give an intuitive explanation why this is the case, recall the bias-variance trade-off from the previous chapter. There, we saw that polynomials of too high degree yield a high variance which leads to a bad generalization performance. Now, the universal approximation theorem states that a sufficiently large neural network can approximate any continuous function. Hence, a sufficiently large neural network can also approximate any polynomial. Without any restrictions, the variance of a neural network can be very high and the generalization performance very poor.

- To prevent it, classical parameter norm penalty can be applied, like the L2 (ridge) and L1 (lasso) penalties seen in Definition 5.23. These penalties are applied by including the penalty term in the loss function of the model, exactly the same as in Section 5.5.

- Furthermore, there are some other regularization techniques specific to neural networks.

Definition 6.16 (Dropout)**.** *Dropout is a regularization technique: for each sample at each training iteration, we set the output y of each node to zero with probability p. After training has completed we do not drop nodes anymore as we want to use the full capacity of the network. However, we multiply each activation where dropout was applied with $1 - p$. This is done to keep the activation on the same level as it was in expectation during training.*

Remarks:

- Effectively, dropout trains a different model at each iteration, where all models share the non-zeroed parameters. For large networks there is no risk that dropout breaks the information flow between the input and the output of the network.

- Dropout reduces the inter-dependencies between nodes in the network, which helps the model to learn more robust features, and also reduces overfitting.

- Dropout is computationally cheap and can be applied to any model that uses distributed representations and that is trained with gradient descent, i.e., any neural network.

- Dropout reduces overfitting but does not completely eliminate the problem. Luckily, dropout can easily be combined with other regularization strategies, for example, with early stopping.

Definition 6.17 (Early Stopping)**.** *Early stopping is a regularization strategy that returns to the parameter setting that produces the lowest validation error. In early stopping, training terminates when the best recorded validation error does not improve for a predefined number of epochs; this number is called patience.*

Remarks:

- Early stopping can be understood as an efficient algorithm for selecting the number of training steps, which is a hyperparameter.
- The cost of early stopping in terms of computation is that the validation needs to be run periodically after each epoch and that at least one copy of the parameters needs to be stored in memory. These costs are however small and generally do not cause any limitation.
- Early stopping does not affect the learning dynamics, can be used in conjunction with other regularization strategies, and is easy to implement.

6.6 Advanced Layers

While neural networks can theoretically learn any function, large networks (with too many weights) often struggle to converge to good solutions. We can use knowledge about the underlying problem to reduce the number of weights substantially. Some early successes of neural networks were achieved in image processing, where methods from classical computer vision were adapted to neural networks in the form of convolutions.

Definition 6.18 (Convolutional Neural Network or CNN)**.** *A convolutional neural network is a neural network layer that works on structured input data such as the pixels of an image. A CNN applies the same function (the same weights) to all neighborhoods of the input layer.*

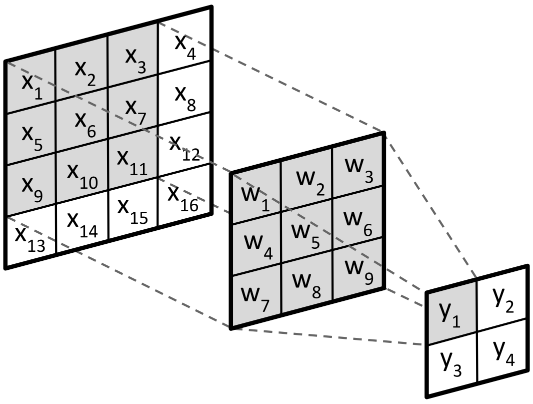


Figure 6.19: Convolution operation for a 4 × 4 input image $\boldsymbol{X}$, a 3 × 3 filter $\boldsymbol{W}$, and a 2 × 2 output $\boldsymbol{Y}$. The filter $\boldsymbol{W}$ slides over the image and for each position of the filter, a value y_i is calculated as the dot-product of the filter and the sub-matrix of $\boldsymbol{X}$ that it covers, e.g., $y_1 = w_1x_1 + w_2x_2 + \cdots + w_9x_{11}$.

Example 6.20. *We want to detect vertical edges in images. Vertical edges can be found calculating the convolution (with symbol $\circledast$) of the image and a vertical*

Sobel filter, given by the matrix $\boldsymbol{W}$:

$$\boldsymbol{W} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

As an example, consider the 3×5 *greyscale image* $\boldsymbol{X}$ *where each pixel takes a value from* 0 *to* 255 *represented by the following matrix:*

$$\boldsymbol{X} = \begin{pmatrix} 80 & 92 & 163 & 234 & 230 \\ 85 & 98 & 237 & 233 & 232 \\ 83 & 96 & 236 & 235 & 231 \end{pmatrix}$$

To compute the convolution operation, the filter $\boldsymbol{W}$ *slides over the image* $\boldsymbol{X}$*. Generally, we require that the output* $\boldsymbol{Y} = \boldsymbol{X} \circledast \boldsymbol{W}$ *is of the same size as the input* $\boldsymbol{X}$*, for this, we need to pad the matrix* $\boldsymbol{X}$*. To prevent the padding from creating artificial edges, we extend the image by copying the border pixels* $\boldsymbol{X'}$*:*

$$\boldsymbol{X'} = \begin{pmatrix} 80 & 80 & 92 & 163 & 234 & 230 & 230 \\ 80 & 80 & 92 & 163 & 234 & 230 & 230 \\ 85 & 85 & 98 & 237 & 233 & 232 & 232 \\ 83 & 83 & 96 & 236 & 235 & 231 & 231 \\ 83 & 83 & 96 & 236 & 235 & 231 & 231 \end{pmatrix}$$

In a convolution, the filter $\boldsymbol{W}$ *slides over the image* $\boldsymbol{X}$*. For each position of the filter on the padded image (see Figure 6.19), one element of* $\boldsymbol{Y}$ *is calculated as the sum of the element-wise multiplication of the overlap. As the filter slides horizontally and vertically, this operation is repeated to obtain the values of each element of* $\boldsymbol{Y}$*. The resulting matrix* $\boldsymbol{Y}$ *is:*

$$\boldsymbol{Y} = \begin{pmatrix} -49 & -401 & -561 & -196 & 13 \\ -51 & -540 & -551 & -52 & 10 \\ -52 & -611 & -552 & 20 & 13 \end{pmatrix}$$

The large values in the second and third column of matrix $\boldsymbol{Y}$ *indicate that there is a strong gradient (variation) between those columns in the image, i.e., the image has a vertical dark/light edge. This edge is less pronounced in the first row.*

Remarks:

- In this example the filter $\boldsymbol{W}$ was given. A CNN does not know these weights, but only the information that $\boldsymbol{W}$ is a 3×3 convolution filter between two layers. With appropriate training data, the CNN will learn the weights of $\boldsymbol{W}$ by using backpropagation.

- Bias (w_0) and non-linearity (e.g. ReLU activation) are omitted in our examples for improved clarity.

- Compared to a fully connected network with the same input size, a CNN has a significantly lower number of weights.

- Sometimes these learned patterns correspond to those that we as humans consider meaningful, e.g., edges. However, usually the patterns extracted by CNNs are not understandable. The power of CNNs reside in learning complex patterns that humans would not be able to design.

- Discrete convolutions can be performed beyond two dimensions. For example, an audio signal can be represented by the signal intensity at discrete time steps. In that case, a 1-dimensional convolution can be applied over the time dimension to filter the signal.

- In general, to apply convolutions to a given input, the input has to be structured as a tensor.

Definition 6.21 (Tensor)**.** *A tensor of order d is a d-dimensional array. A tensor generalizes vectors (1-dimensional) and matrices (2-dimensional). The shape of a tensor is a list of d integers defining the size of each dimension of the tensor.*

Remarks:

- An image, represented by its RGB (red, green, blue) pixel values can be naturally represented as a tensor of order 3 and shape [3, height, width]. An index $(0, x, y)$ into this tensor yields the intensity of red in the pixel at location (x, y). → notebook

- Note that the memory requirements of higher order tensors can be high. E.g., an order 5 tensor with 100 values in each dimension stores 100^5 values, which, given a floating point precision of 32 bits, requires 40 GB of memory.

- We have seen that CNNs exploit translation invariance in the structure of the data. Are there other such structural biases we can exploit? For example, what if we want a neural network to remember important features over several time steps of a sequential input?

Definition 6.22 (Recurrent Neural Network or RNN)**.** *In contrast to feed-forward neural networks, a recurrent neural network operates with time steps t. Each time step t gets an input $\boldsymbol{x}_t$ and a state $\boldsymbol{s}_t$. It outputs an updated state $\boldsymbol{s}_{t+1}$ and an output $\boldsymbol{y}_t$. More formally, we define the mappings*

$$\boldsymbol{y}_t = \hat{g}(\boldsymbol{x}_t, \boldsymbol{s}_t)$$

$$\boldsymbol{s}_{t+1} = \hat{h}(\boldsymbol{x}_t, \boldsymbol{s}_t)$$

where $\{\boldsymbol{x}_t\}_{t=0}^{\tau}$ and $\{\boldsymbol{y}_t\}_{t=0}^{\tau}$ are the input and corresponding output sequence of length τ and $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$ are differentiable functions with learnable parameters. The initial state $\boldsymbol{s}_0$ can be a vector of learnable parameters, or simply initialized to $\mathbf{0}$.

Remarks:

- There are several ways of how to define $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$, from simple linear projections to complex combinations of operations to combine the given inputs.

Example 6.23. *Consider that we want to solve the multi-path problem of wireless transmission, i.e., given a signal we wish to filter delayed copies from the signal. To do this online, i.e., while the signal is received, we have to remember the current input signal to filter a similar pattern later. We therefore seek to train an RNN to remember a given input for a few time steps and then reproduce it for filtering purposes. E.g., given the input signal* $[5, 10, 0, 1.5, 3.5, 1]$ *we want the RNN to output* $[5, 10, 0, 0, 0, 0]$. *This can be achieved if we initialize the state* $\boldsymbol{s}_0$ *to* $\mathbf{0}$ *and parametrize* $\boldsymbol{s}_{t+1} = \hat{h}(x_t, \boldsymbol{s}_t) = \boldsymbol{W}\boldsymbol{s}_t + \boldsymbol{w}_h \cdot x_t$ *with* $\boldsymbol{W}$ *and* $\boldsymbol{w}_h$ *as*

$$\boldsymbol{W} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -0.1 & -0.3 & 0 & 0 \end{pmatrix} \qquad \boldsymbol{w}_h = [0, 0, 0, 1]^T$$

where $\boldsymbol{W}$ *shifts the state and filters new inputs and* $\boldsymbol{w}_h$ *reads in the new symbol. The readout is given by* $y_t = \hat{g}(x_t, \boldsymbol{s}_t) = \boldsymbol{w}_g^T \boldsymbol{s}_t + x_t$ *with*

$$\boldsymbol{w}_g = [-0.1, -0.3, 0, 0]^T$$

Note that this parametrization simply reads the input symbol into the state $\boldsymbol{s}_t$, *propagates the symbol for some time steps and then subtracts it from a later input.*

Remarks:

- Instead of these given weights, neural networks will learn $\hat{g}$ $(\boldsymbol{W}, \boldsymbol{w}_g)$ and $\hat{h}$ $(\boldsymbol{w}_h)$ when trained on real world signals. This is particularly useful if the input is a vector of multiple correlated noisy signals and a simple remember-and-reproduce solution is sub-optimal.

- Earlier we discussed that neural networks are directed acyclic graphs (DAGs). But RNNs are cyclic, as the state from step t gets fed back to the network in step $t + 1$. How can we train such a network?

- The solution is to copy the network τ times, i.e., unroll the cycle (see Figure 6.24). This yields one long DAG where the state $\boldsymbol{s}_t$, calculated as intermediate output of one copy, is fed into the next copy. The calculated gradients for each copy are then summed to update the parameters. This is called *backpropagation through time (BPTT)*. Note that this can lead to vanishing/exploding gradients as we are essentially trying to train a network of depth τ.

- RNNs that are commonly used today are *Gated Recurrent Units (GRUs)* and *Long Short Term Memories (LSTMs)*. These address the issue of vanishing/exploding gradients in their definition of $\hat{g}$ and $\hat{h}$. The resulting architectures implement ideas similar to that of skip connections in feed-forward neural networks, albeit historically GRUs and LSTMs came long before people started talking about skip connections in MLPs and CNNs.

- As apparent from the equations in the definition above, RNNs are inherently sequential. They can process one input only after the previous input has been processed. This is slower than approaches that can process the whole sequence in parallel (such as CNNs).

- Both, CNNs and RNNs take advantage of *weight sharing*. In CNNs, the same weights (filters) are applied to all locations of the image. In RNNs, the same functions $\hat{g}(\cdot)$ and $\hat{h}(\cdot)$ (with the same learnable weights) are applied to all time steps t.

- What if we do not want to apply the same function everywhere? More specifically, what if only a selection of the input is of interest? Can we design an architecture that favors solutions which select features from the input instead of using the whole input? Can we index the input in a differentiable way?

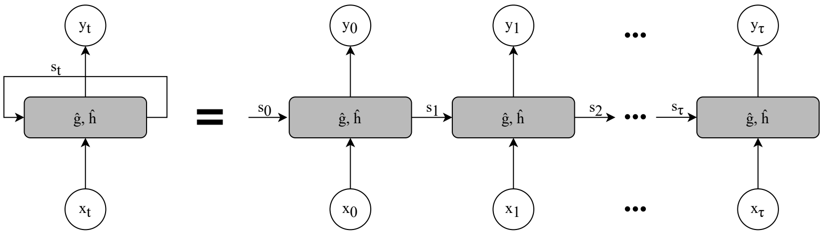


Figure 6.24: Unrolling the RNN through time. At each time step, $\hat{g}$ and $\hat{h}$ compute output $\boldsymbol{y}_t$ and state $\boldsymbol{s}_{t+1}$ respectively, given input $\boldsymbol{x}_t$ and the previous state $\boldsymbol{s}_t$. BPTT propagates the gradients through the unrolled network.

Definition 6.25 (Attention)**.** *Attention is a method to aggregate inputs in a selective manner. Given n input vectors $\{\boldsymbol{x}_i\}_{i=0}^{n-1} \in \mathbb{R}^d$, each input vector is projected into a key vector $\boldsymbol{k}_i \in \mathbb{R}^{d_k}$ and a value vector $\boldsymbol{v}_i \in \mathbb{R}^d_v$. The projection is done by two learned matrices $\boldsymbol{W}_k \in \mathbb{R}^{d \times d_k}$ and $\boldsymbol{W}_v \in \mathbb{R}^{d \times d}$. Additionally, attention is given a query vector $\boldsymbol{q} \in \mathbb{R}^{d_k}$. For each input vector an attention score s_i is calculated as the dot-product between the query $\boldsymbol{q}$ and the key vector $\boldsymbol{k}_i$:*

$$s_i = \boldsymbol{q} \cdot \boldsymbol{W}_k \boldsymbol{x}_i = \boldsymbol{q} \cdot \boldsymbol{k}_i$$

The attention scores are normalized by a softmax (Definition 5.34) operation and each normalized score is multiplied by its corresponding value vector $\boldsymbol{v}_i = \boldsymbol{W}_v \boldsymbol{x}_i$. The results are added to produce the attention output $\boldsymbol{y} \in \mathbb{R}^d$:

$$\boldsymbol{y} = \frac{\sum_{i=0}^{n-1} \exp(s_i) \cdot \boldsymbol{v}_i}{\sum_{j=0}^{n-1} \exp(s_j)}$$

Remarks:

- The attention mechanism described here is called dot-product attention. This is the most common type of attention, but other variants exist as well.

Example 6.26. *Consider that we want to find in what position a sequence of $n = 5$ integers contains the value "3". Our input is $\boldsymbol{x} = [7, -5, -2, 3, 4]^T$. We use a ReLU for the keys: $\boldsymbol{k}_i = \text{ReLU}\left[x_i - 3, 3 - x_i\right]^T$. And we use a one-hot encoding for the values $\boldsymbol{v}_i$. For example, $x_1 = 7$ gets key and value*

$$\boldsymbol{k}_1 = [4, 0]^T, \boldsymbol{v}_1 = [1, 0, 0, 0, 0]^T.$$

Using $\boldsymbol{q} = [-1, -1]^T$ we get the scores $\boldsymbol{s} = [-4, -8, -5, 0, -1]^T$. Plugging the scores and values into the softmax gives

$$\boldsymbol{y} = [0.013, 0.000, 0.004, 0.718, 0.264]^T.$$

Because of the softmax, the output is not quite as clean as one might hope, since "$4 \approx 3$".

Remarks:

- Thanks to weight sharing, attention can process input vectors of any length.
- A neural network can consist of multiple attention aggregations to select multiple (potentially different) inputs.
- If the scores are calculated based on the inputs, i.e., $s_i = f_i(\{\boldsymbol{x}_i\}_{i=0}^{k-1})$ for some functions f_i, attention is also referred to as *self-attention*, as the input "attends" to itself.
- Attention architectures yield the state-of-the-art performance in natural language processing tasks, as most of the time some words are more important than others to understand a sentence.
- All architectures presented in this section, i.e., CNNs/RNNs and Attention, take some domain knowledge to tailor the neural network to a specific purpose. This is also referred to as an *inductive bias.*

6.7 Architectures

In the previous section we introduced several building blocks that give different inductive biases. Let us now see how different losses and architectures can be combined to solve advanced computational challenges.

Definition 6.27 (Autoencoder)**.** *An autoencoder is a neural architecture formed by two neural networks: an encoder $f_{enc}(\cdot)$, which encodes the input $\boldsymbol{x} \in \mathbb{R}^n$ into a representation $\boldsymbol{z} \in \mathbb{R}^m$ called latent code; and a decoder $f_{dec}(\cdot)$, which decodes the latent code back into an approximation of the original input, i.e., $f_{dec}(f_{enc}(\boldsymbol{x})) \approx \boldsymbol{x}$. The representation is often designed to compress information by setting $m \ll n$. Autoencoders minimize a loss term named "reconstruction loss", that represents the difference between the output and the input.*

Remarks:

- The encoder and the decoder can be any type of neural network, e.g., MLPs, CNNs, RNNs or a combination of them.
- An example of reconstruction loss is the L^2 error:

$$L = \frac{1}{|D|} \sum_{\boldsymbol{x} \in D} ||\boldsymbol{x} - f_{dec}(f_{enc}(\boldsymbol{x}))||_2^2$$

- A schematic depiction of an autoencoder is shown in Figure 6.28.

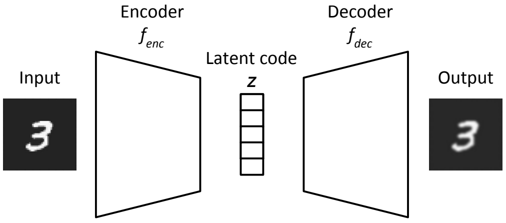


Figure 6.28: Schematic view of an autoencoder.

Remarks:

- The requirement $m \ll n$ is not a necessity. We can also design autoencoders to have a specific structure in the latent code, or a latent code tailored to a given purpose through an additional loss.
- The main advantage of autoencoders is that they learn the latent representation (or code) in an unsupervised manner, i.e., without labels. This makes them a versatile architecture that can be used in a wide range of problems, such as dimensionality reduction, compression, data denoising or unsupervised feature extraction.
- There exist many different types of autoencoders, with different losses, architectural elements or with additional inductive biases.
- Besides compression and encoding, neural networks can also be used for data generation.

Definition 6.29 (Generative Adversarial Network or GAN)**.** *GANs are a class of deep generative models in which two neural networks are trained simultaneously, while competing in a two-player minimax game. The generator's $f_{gen}(\boldsymbol{r})$ tries to produce realistic synthetic samples from a random input $\boldsymbol{r}$. The binary classifier called discriminator $f_{dis}(\boldsymbol{x})$ estimates whether a sample $\boldsymbol{x}$ is real or synthetic. The goal of the generator is to maximize the probability that the discriminator makes a mistake on $f_{dis}(f_{gen}(\boldsymbol{r}))$.*

Remarks:

- As in the case of autoencoders, the discriminator and generator can be any type of neural network.
- During training, the discriminator improves its ability to recognize synthetic samples while the generator learns to produce increasingly realistic samples to deceive the discriminator. In this adversarial setting, the equilibrium is reached when the generator produces realistic samples such that the discriminator cannot distinguish whether they are real or synthetic.
- The architecture of a vanilla GAN is shown in Figure 6.32.
- GANs achieved remarkable results in image generation. In particular, they can generate realistic-looking pictures and videos, which has raised concerns about malicious uses of these models to generate deepfakes.
- A GAN is a fully automated Turing Test with generator = testee, and discriminator = tester.

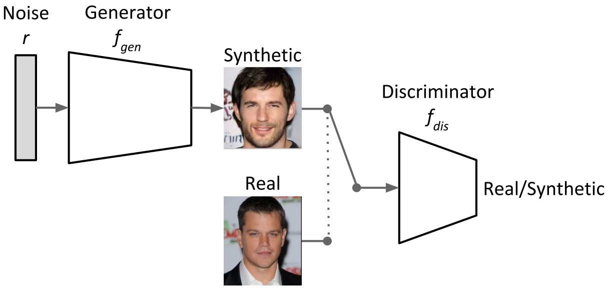


Figure 6.30: GAN architecture for generation of synthetic images of celebrities. For each sample, the discriminator needs to decide whether its input corresponds to a real or to a synthetic celebrity.

Example 6.31. *Faceswap-GAN is a popular implementation of a model trained for deepfake generation. At high level, this model uses an autoencoder as generator. Given an image of a human face, it produces a segmentation mask as well as the reconstructed input image. A segmentation mask is a representation of an image that delineates the most important objects in the image; in the case of human faces, these are the eyes, nose, ears, etc. Roughly speaking, an arbitrary image of a face A can be combined with the segmentation mask of another face B in order to generate an image that replaces the features of image B with those of A, i.e., a deepfake. This is what the model does at inference time.*

During training, Faceswap-GAN used a discriminator that determines whether an input image is a real face or a deepfake, as well as some other advanced methods such as a perceptual loss that improve image quality.

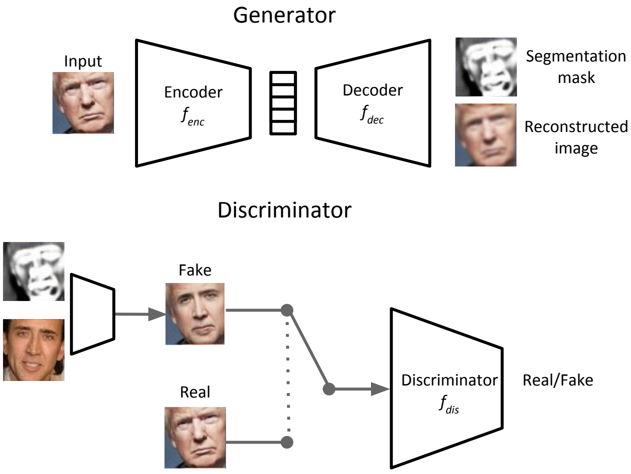


Figure 6.32: Architecture of Faceswap-GAN.

Remarks:

- Although originally conceived for generative tasks such as denoising, reconstruction or data generation, GANs have proved useful in other domains such as supervised learning, semi-supervised learning or reinforcement learning.

- Deep learning is applied in many different areas and consequently, there is a wide range of architectures. We list some promising architectures in Table 6.33.

Name	Description	Purpose
Transformer	A family of sequence-to-sequence models based on the self-attention operation.	State of the art in natural language processing (NLP) and gaining relevance in other areas. Recently received a lot of media attention with models like BERT and GPT-3.
Variational Auto-Encoder (VAE)	A generative model that tries to match the data distribution by enforcing a simplified posterior distribution in the latent space of an autoencoder.	Data generation and posterior approximation in inherently stochastic models. Disentangled representation learning.
Contrastive Learning	The same network applied to different inputs, trained to recover a notion of similarity in the output representation.	Unsupervised representation learning (e.g., recovering an approximately metric space), authentication, hashing and matching
Graph Neural Network	Replicating a network on all nodes of a graph and incorporating operations for message exchanges with neighbors on the graph.	Graph/node/edge classification, community detection in social network and predicting protein/molecular interactions
Implicit Network	Training a neural network to recover a value from an index.	Data compression, super-resolution, image in-painting
Hyper Network	A neural network that outputs the weights of another neural network.	Mode abstraction and meta learning

Table 6.33: A glossary of promising architectures.

6.8 Reinforcement Learning

A neural network and its corresponding loss have to be end-to-end differentiable in order to apply gradient descent. So what if a problem is not differentiable? What if we want to find an optimum in a sequential setting, like an optimal sequence of decisions to reach a desired goal in an environment?

Definition 6.34 (Markov Decision Process or MDP)**.** *A Markov decision process formally defines an environment. An MDP is a 5-tuple* (S, A, T, R, s_0)*, where* S *is a set of states,* A *is a set of possible actions,* $T : S \times A \to S$ *is a state transition function, which describes the next state based on current state and action. However,* T *could also be probabilistic, i.e.,* $T : S \times A \to S \times [0, 1]$*.* $s_0 \in S$ *(or* $s_0 : S \to [0, 1]$*) is an initial state (distribution). Finally,* R *is a reward function. Rewards can be given when reaching certain states (*$R : S \to \mathbb{R}$*), or when taking the right action in a state,* $R : S \times A \to \mathbb{R}$*.*

Problem 6.36. *Figure 6.35 shows a simple example of an MDP with 6 states* $S = \{u_0, u_1, u_2, u_3, \mathit{fail}, \mathit{pass}\}$ *and two possible actions* $A = \{\mathit{study}, \mathit{party}\}$*. The*

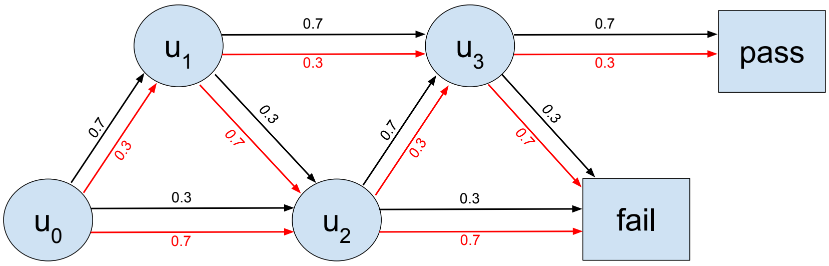


Figure 6.35: An example Markov decision process to figure out whether and when one should study (black) or party (red) in front of an exam.

states fail and pass are terminal states, and represent whether the agent fails or passes the exam, respectively. The agent starts in state u_0, i.e., $s_0 = u_0$. The transition probabilities for choosing the 'party' action (red) or 'study' action (black) are shown in the figure. Every time the agent chooses to party, it receives a reward of +1. If the agent chooses to study, it receives a reward of -1 (studying is painful). At the terminal states, the agent gets a reward of +10 for passing the exam and -10 for failing. Intuitively, the states u_1 and u_3 represent states where the agent has learned something. These states are more likely to be reached when studying rather than partying, and from these states the agent is more likely to pass the exam. How should the agent act? We can calculate the solution backward from the terminal states by filling in a 2×4 matrix Q giving the quality of each action in each of the non-terminal states. In u_3 taking the action 'study' will yield an expected reward of $Q[u_3, study] = -1+0.7\cdot 10+0.3\cdot(-10) = 3$. Similarly, choosing 'party' in this state yields an expected reward of $Q[u_3, party] = +1+0.3\cdot 10+0.7\cdot(-10) = -3$. In all earlier states we can assume that we take the action with higher expected reward in later states and thereby calculate the remaining values recursively as given in Table 6.37.

State	study	party
u_3	3	-3
u_2	-1.9	-5.1
u_1	0.53	0.57
u_0	-1.171	-0.159

Table 6.37: Expected reward Q for each action in each non-terminal state. It's best to party in states u_0 and u_1, and best to study in states u_2 and u_3. Note that filling in this table is dynamic programming (Definition 1.11).

Definition 6.38 (Policy)**.** *A policy $\pi : S \times A \to [0,1]$ describes how the agent acts in the environment, i.e., how likely it will take an action $a \in A$ in a given state $s \in S$.*

Remarks:

- Given an MDP and a policy, the state distribution of the agent after τ steps, i.e., how likely it is that the agent is in a given state after τ actions can be calculated.
- The goal in reinforcement learning is to find a good policy π, that is, a policy that accumulates positive rewards.
- More formally, we want to find the optimal policy π^* that maximizes the expected cumulative γ-discounted reward:

$$\pi^* = \max_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

  where $\gamma \in [0, 1]$ is a discount factor that weighs immediate returns relative to future returns, where s_t is the state at time step t, respectively. The expectation is taken over actions sampled from the policy and states sampled from the transition distribution $P(\cdot|s, a)$ given the state s and action a.
- To find such a policy, we need to know how valuable each state is to a given policy.

Definition 6.39 (Value Function)**.** *A value function $V_\pi : S \to \mathbb{R}$ is a policy specific function that given a state returns the expected cumulative discounted reward of the policy starting in state s_t.*

$$V_\pi(s_t) = \mathbb{E}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} R(s_\tau)\right]$$

Definition 6.40 (Action-Value Function or Q-Function)**.** *An action-value function or Q(uality)-function $Q_\pi : S \times A \to \mathbb{R}$ is a policy specific function that given a state s and an action a returns the expected cumulative discounted reward of taking action a in state s and following policy π thereafter. That is*

$$Q_\pi(s, a) = R(s) + \mathbb{E}\left[\gamma V_\pi(s')\right]$$

where the expectation is over states s' sampled according to $T(s'|a, s)$.

Remarks:

- The value function can also be defined in terms of the Q function as

$$V_\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q_\pi(s, a)\right].$$

- Given the general definition of value and action-value function, we can get a better understanding of what we want to find: the optimal policy π^*
- Note that V_{π^*} gives us for each state $s \in S$ the maximal expected cumulative reward that can be achieved when starting in state s.

- Further, if we are given Q_{π^*} it is easy to derive the optimal policy by simply taking the action that maximizes Q_{π^*}, i.e.

$$\pi^*(a|s) = \begin{cases} 1 \text{ if } a = \operatorname{argmax}_{a'} Q^{\pi^*}(a', s) \\ 0 \text{ else} \end{cases}$$

  In other words, the optimal policy in an MDP is deterministic!

- Knowing that we can derive the optimal policy from a quantity which requires the optimal policy might be a bit "recursive", but we can nevertheless try:

```
1  # S = states
2  # A = actions
3  # T = transitions
4  # R = rewards
5  def value_iteration(S, A, R, T):
6      V = zero vector of size len(S)
7      Q = zero matrix of size len(S) × len(A)
8      while Q not converged:
9          for s in S:
10             for a in A:
11                 Q[s][a] = R(s) + $\gamma\sum_{s'\in S}$ T(s'|a,s) * V[s']
12             V[s] = max(Q[s])  # max over a in A
13     return Q
```

Algorithm 6.41: Value iteration

Remarks:

- If the MDP has cycles and $\gamma \to 1$, then this algorithm may not converge.

Lemma 6.42. *If we are given an MDP that can be represented as a DAG, value iteration converges in one iteration if we process the states $s \in S$ in reversed DAG order.*

Proof. Reversed DAG order means we will start at the terminal states and propagate the cumulative rewards back to the initial states. By induction, we always propagate the maximal achievable value by the max operation in Line 12 of Algorithm 6.41. Therefore, after one iteration, all states have the optimal value V_{π^*} assigned. □

Remarks:

- However, in many real world applications, the state space is just hilariously large. The game Go for instance has $3^{19\times 19} \approx 10^{172}$ possible states, so it is infeasible to compute or even store the whole Q-table. We can however train a neural network to approximate how likely a given position is to lead to a win. In combination with a bit of look-ahead planning (recursion) a neural network was able to beat the world champion.

Chapter Notes

While the beginnings of artificial neural networks go back to the 1940s [5], deep learning only became widely adapted and efficient in recent years with the use of GPUs to run computations in parallel. However, many of the theoretical investigations and architectures presented here have been known for quite some time by now. The universal approximation capability of neural networks was first shown for sigmoid non-linearities [1] and later generalized to other non-linearities [3]. Even before it was shown that learning various functions is NP-complete [4, 6]. This is still an active research area, e.g. [2]. The VC Dimension discussion is even older [7]. We summarize further milestones achieved by neural networks in the table below.

Year	Name	Milestone
1989	MNIST	Handwritten digit classification
2005	DARPA	Self-driving car challenge: 212km in 7h
2012	AlexNet	Image classification breakthrough
2014	Deepface	Human level performance in face recognition
2014	DQN	Superhuman performance in many Atari games
2016	AlphaGo	Beats Champion in Go
2017	Waymo	Fully autonomous self-driving on public roads
2018	Obvious	Sells art generated by a GAN for $432,500
2020	OpenAI	GPT-3 model can create poetry, code, etc.
2020	AlphaFold	Achieves 90% in CASP protein folding

Table 6.43: Neural Network Milestones

This chapter was written in collaboration with Damian Pascual and Oliver Richter.

Bibliography

[1] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

[2] Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 relu networks. 2021.

[3] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[4] Stephen Judd. On the complexity of loading shallow neural networks. *Journal of Complexity*, 4(3):177 – 192, 1988.

[5] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[6] Nimrod Megiddo. On the complexity of polyhedral separability. *Discrete Computational Geometry*, 3:325–337, 1988.

[7] V. N. Vapnik and A. Ya. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

Chapter 7

Computability

Computability was pioneered by Alan Turing and Kurt Gödel. Turing probably committed suicide by eating an apple he poisoned with cyanide. Gödel on the other hand had an obsessive fear of being poisoned with food; when his wife was hospitalized, he refused to eat, and eventually starved to death. Now it's your turn to study this intoxicating subject.

7.1 The Halting Problem

In the previous chapters, we have analyzed various computational tasks. While some functions were easy to compute efficiently, others were difficult; in these cases, we have focused on approximations or heuristics. However, given enough time and resources, could we always find the solution to any problem?

Problem 7.1 (Halting problem)**.** *Given a program P and an input x to P, does $P(x)$ halt (stop running) after a finite amount of time?*

Remarks:

- Can we write a Python program that solves Problem 7.1? Somewhat surprisingly, the input of our program is also a program (plus an input parameter of this program).
- Naturally, we must somehow encode the input program. There are various ways to do this. We can, for example, consider the whole code of the program as a long string of text, encoding each character in this string with a byte.
- The halting problem is sometimes easy to solve, for example, in case of the simple programs in Algorithm 7.2. But is it always easy?

```
def P1(x):
    print("Hello, world!")
    return

```

```
def $P_2(x)$:
    while x > 0:
        x += 1
    return
```

Algorithm 7.2: Example programs; P_1 is halting, P_2 depends on the input.

Definition 7.3 (Undecidable)**.** *We say that a problem is undecidable if no algorithm solves the problem in finite time for every possible finite input.*

Remarks:

- This should be surprising! Definition 7.3 does not say that the runtime increase will be exponential or even double exponential in the input size, but that the problem really cannot be solved in *any* finite amount of time.

Theorem 7.4. *The halting problem is undecidable.*

Proof. Assume for the sake of contradiction that there exists a program $P_H(P, x)$ that solves the halting problem in finite time for any input. We design a new program $P_T(x)$ that takes a bitstring as an input and calls P_H as a subroutine:

```
1 def $P_T(x)$:
2     if $P_H(x, x)$ == True:
3         while True: pass    #loop forever
4     else: return
```

Algorithm 7.5: Test program $P_T(x)$.

P_T interprets its input x as a program encoding, and calls the halting solution P_H on program x with input x. Since we assumed that P_H can always solve this problem in finite time, Line 2 is evaluated in finite time for any x.

Since P_T is also a program, it has a bitstring encoding τ according to our encoding scheme. What happens when we call $P_T(\tau)$? Note that this means that we are calling $P_H(\tau, \tau)$ as a subroutine, i.e., querying whether the program described by τ (that is, P_T) halts on input τ.

- If $P_H(\tau, \tau)$ is true, then P_T goes in to an infinite loop according to our code, and never halts. Hence $P_H(\tau, \tau)$ should be false instead!

- If $P_H(\tau, \tau)$ is false, then P_T immediately terminates according to our code, so $P_H(\tau, \tau)$ should be true!

We get a contradiction in both cases, so the halting problem is undecidable. □

Remarks:

- We assumed that the encodings of a program and its input are simply bitstrings. This is also close to practice. What if a bitstring is an invalid program, not respecting the syntax of programs? We could argue that in this case, the program simply halts (with an error).
- Of course Theorem 7.4 only holds in general: we cannot solve the halting problem correctly in finite time for *every* algorithm-input pair. Some specific algorithm-input pairs, e.g. the simple examples in Algorithm 7.2, can be decided easily.
- Also, a program P that actually halts is easy as well: we just run/simulate P, which will eventually halt. The halting problem is only undecidable because of programs P that do not halt. In this case it is difficult to distinguish if P is still running because it did not reach the halting point yet, or because it is never going to halt.
- There is a name for this weaker kind of decidability that is only required to work in one of the two cases (unlike our original concept of decidability in Definition 2.2).

Definition 7.6 (Semi-decidable)**.** *We say that a problem is semi-decidable if there exists an algorithm $\mathcal{A}$ such that*

- *if the answer is* `True`*, then $\mathcal{A}$ outputs* `True` *in finite time,*
- *if the answer is* `False`*, then $\mathcal{A}$ either outputs* `False` *in finite time or keeps running indefinitely.*

Theorem 7.7. *The halting problem is semi-decidable.*

Remarks:

- Given the undecidability of halting, is there an easy way to show that some other problems are also undecidable? Yes, we can use reductions again (Definition 2.8). Given a problem Π, we can show that if Π was decidable, then the halting problem would also be decidable. This implies that Π is also undecidable. One slight difference from Definition 2.8, however, is that for this argument, we do not need the reductions to run in polynomial time.
- Here is an example for such a reduction.

Problem 7.8 (Mortality problem)**.** *Given a program P, is it true that $P(x)$ halts for any possible input x?*

Remarks:

- This is different from the halting problem, but only so much: instead of a specific input, we now want to know if P halts on *every* input.

Theorem 7.9. *The mortality problem is undecidable.*

Proof. Assume that we have a program P_M that takes a program description as an input, and solves the mortality problem in finite time. Then given a specific input for the halting problem (a program P and an input x), consider the following program:

```
def $P_T(y)$:
    if $y$ == $x$:
        run program $P(x)$
    else:
        return
```

Algorithm 7.10: Another testing program.

Now let us run our mortality solution P_M on the encoding of P_T. We know that P_T certainly terminates in finite time for any input different from x. This implies that $P_M(P_T)$ is true if and only if $P(x)$ halts; hence a solution for the mortality problem allows us to solve the halting problem on P and x. Since halting is undecidable, such a solution cannot exist, so we have a contradiction. □

Remarks:

- Well, that was not so surprising; after all, mortality is a close relative of the halting problem.
- How about problems that are a far cry from halting? We need a clean and simple theoretical definition of what we mean by a program, algorithm or function. Let us make a brief detour into abstract machine models and computation theory.

Definition 7.11 (Model of Computation)**.** *A model of computation defines the rules how the output of a mathematical function is computed from a given input.*

7.2 The Turing Machine

What kind of building blocks do we need to obtain a simple theoretical model of a machine that can, intuitively speaking, do the same computations as a real computer?

Remarks:

- First we need some abstract *states* that represent the current state of our program, and we need to describe the transitions between these states. Such a set of states with predefined transition rules is known as a finite automaton.
- We also need some memory to store data. We usually assume that a memory consists of cells; our program can read data from these cells, write data into these cells, and move between these cells to be able to access any of them. We will now consider a *tape* of cells that is infinite in both directions, i.e. the cells can be enumerated by integers $\ldots, -2, -1, 0, 1, 2, \ldots$ (from $-\infty$ to ∞).

- We also have a *tape pointer* that points to a specific tape cell at each point in time, indicating that this is the tape cell that we can currently read/write. Initially the tape pointer points to cell 0.
- What kind of data can we write onto this tape? We assume that we have an *alphabet* Σ of possible symbols, and we can write exactly one symbol into each cell. In the simplest case, this alphabet can be binary, i.e. $\Sigma = \{0, 1\}$. We usually use some extra symbol, e.g. $\perp$ for the cells that we consider empty, and we use $\Sigma := \Sigma \cup \{\perp\}$.
- These building blocks define a famous theoretical model of computation.

Definition 7.12 (Turing Machine or TM)**.** *A Turing Machine has a finite set of states S, and a two-way infinite tape. Initially the machine is in a specified starting state $s_0 \in S$, the tape has some symbols on it (the input), and the tape pointer points to cell 0 of the tape.*

In each discrete time step, depending on the current state s and the current tape cell content σ, the machine executes the following steps:

- *change to another state $s' \in S$,*
- *write the tape, i.e. change the content σ of the current tape cell to any symbol $\sigma' \in \Sigma$,*
- *possibly move the tape pointer one step to the left or one step to the right.*

Formally a TM is defined by a function $T : (S, \Sigma) \rightarrow (S, \Sigma, m)$, where $m \in \{left, right, stay\}$ indicates the movement of the tape pointer.

With a TM we usually also select a specific halting (accepting) state $s_h \in S$. We say that the TM accepts an input if, when executed on this input, the TM eventually enters state s_h.

Remarks:

- We assume that computation is over whenever the halting state s_h is reached: the machine does not do anything (i.e. never changes the state/tape content/tape pointer) after this point, and the current tape content is considered to be the *output* of the computation.
- While a TM acts as a model of computation, we can also interpret it as a function: it converts an input (the initial content of the tape) to an output (final content of the tape).
- However, the function is not complete: for some inputs, the output may be undefined, since just like a Python program, a TM can easily run forever and never halt.
- How can we do actual computations in this abstract setting? Let us see an example for a simple operation: incrementing an integer.

Example 7.13 (Incrementation with a TM)**.** *Given a positive integer input x, our task is to increment x by 1. We assume that the input x is given in a binary representation, starting with least significant bit (LSB) first at cell 0, and going until cell $\lfloor \log_2 x \rfloor$. The tape pointer starts at cell 0, and empty cells of the tape are marked with a $\perp$.*

Lemma 7.14. *Example 7.13 can be solved on a simple TM with 2 states.*

Proof. The formal process of incrementing a binary number is as follows. We start going from the LSB to the most significant bit (MSB) until we encounter a 0, and we change every 1 to a 0 during the process. When we first find a 0, we change it to a 1 (or if we have left the MSB, we add an extra 1 to the front), and the incrementation is done. We have to translate this process to states and transitions.

This can be done with two states s_0 and s_h. The starting state is s_0, this is where the execution begins; s_h is a halting state where none of the transitions do anything. The transitions from s_0 are defined as follows:

Transitions from state s_0

Read		Write	Pointer	Next state
1	$\longrightarrow$	0	right	s_0
0 or $\perp$	$\longrightarrow$	1	stay	s_h

This ensures that the machine enters the halting state exactly when the incrementation is finished, i.e. when the tape contains $x + 1$ in the same binary representation. □

Remarks:

- Describing a TM for more complex computations can be some work, since it usually requires a higher number of states and transitions. Even in case of our incrementation example, if the number is in a reversed representation (i.e. starting with MSB), we already need an extra state to first move to the end of the input and then start processing the input from the other direction.
- The definition of the halting problem on TMs is as follows: given a description of a TM and an input (initial tape content), decide if this TM ever goes into the halting state. Note that this is only a reformulation of our original halting problem, so the same proof shows that this problem is undecidable.
- There also various other versions of TMs, e.g. a TM that has multiple tapes, and it can read/write these tapes simultaneously in every step. One can show that this is equivalent to the single-tape setting in terms of computability.
- There is one important concept in computation that this basic machine model cannot capture: randomization. In order to model that, we need to slightly extend the machine model.

Definition 7.15 (Randomized TM or RTM)**.** *In a Randomized Turing Machine, each transition is replaced by a set of available transitions, and a probability distribution over these transitions. In each step, a transition is chosen at random according to this probability distribution.*

Remarks:

- For example, a program on an RTM might have a state where it moves left on the tape with 50% probability, and moves right with 50% probability.
- Another formulation of RTMs is to take a deterministic TM, and add an extra tape of infinite random bits to the machine. The TM then reads bits from this extra tape, and (possibly) executes different transitions based on the next random bit.
- Our argument uses an important assumption: that we can use a TM to simulate the execution of another TM. We also need this property when expressing the proof of Theorem 7.4 in a TM-based context.
- Luckily, this is possible:

Theorem 7.16 (Universal TM)**.** *There exists a Universal Turing Machine which receives the encoding of another TM T (i.e., a program encoded as a string) and an input x to T on its tape, and simulates the behavior of T on x.*

Remarks:

- This is somewhat similar to a real-world Von Neumann computer architecture, where source code, constants and inputs of a computation are stored in the same memory.
- So how close are TMs to real computers? The fact that our program moves between a finite number of states is pretty realistic. What is unusual, however, is that we can only move in memory one step at a time.
- More realistic machine models do exist:

Definition 7.17 (RAM Machine)**.** *A RAM Machine is a model of computation that has explicit registers (instead of only cells) which can store integer values. The machine is capable of addressing these registers indirectly through pointers (instead of moving only sequentially between them).*

Remarks:

- While RAM machines seem more expressive, they are in fact equivalent to TMs: any program on a RAM machine can also be simulated on a TM.
- Since TMs are simpler, we usually stick to TMs. We say that a problem Π is computable if a TM can compute Π. This means that we can essentially use TMs to define the general notion of an algorithm.

7.3 Computing on Grids

Definition 7.18 (Tile)**.** *A tile is a 1×1 square. Each tile has a color, and each side of the tile (left, top, right, and bottom) has a specific code (e.g., a letter or a number). We assume that we are not allowed to rotate or flip tiles.*

Definition 7.19 (Correct Tiling)**.** *Two tiles can be placed next to each other if their touching side has the same code. E.g., if tile t_1 has code A on its top and tile t_2 has code A on its bottom, then we can place t_2 directly above t_1.*

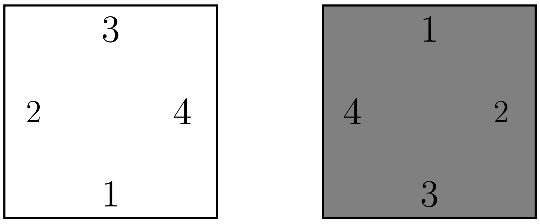


Figure 7.20: Example of a tile set consisting of two 2 tiles that tile the infinite plane in a chessboard pattern

Remarks:

- Our goal is to tile a given pattern (e.g., a chessboard) or shape (e.g., a 4 × 5 rectangle) with a set of tiles such that the tiling is correct.

Problem 7.21 (Tiling)**.** *Assume we are given a set of n tiles, and we can take an arbitrary number of copies of each of these tiles. Does there exist a correct tiling of the entire infinite plane with our given set of tiles?*

Remarks:

- At first glance, this problem seems to have no connection to TMs or the halting problem.
- The most simple correct tiling one can imagine is a periodic tiling, when the same pattern of tiles keep repeating.

Definition 7.22 (Periodic Tiling)**.** *We say that a tiling of the plane is periodic if there exist positive integers w, h such that for every pair of coordinates $i, j \in \mathbb{Z}$, grid square (i, j) has the same tile as grid squares $(i + w, j)$ and $(i, j + h)$.*

Remarks:

- In a periodic tiling, we tile a $w \times h$ rectangle R such that the tiling within rectangle R is correct, and it is also correct to place two such rectangles R next to each other top/bottom or left/right. Then we can cover the entire plane with copies of R.

Lemma 7.23. *If we know that a periodic tiling exists, we can find it in finite time.*

Proof. We take every possible rectangle size $w \times h$, in an increasing ordering according to the sum $w + h$: first 1×1, then 1×2 and 2×1, then 1×3, 2×2 and 3×1, and so on. For each such size, we can try all possible tilings in each of these rectangles, and check their correctness.

If there exists a periodic tiling with a rectangle of size $w \times h$, then we try at most $(w + h)$ total sizes, and thus at most $(w + h)^2$ rectangle shapes before reaching $w \times h$. Each such shape has at most $n^{(w+h)^2}$ possible tilings. Since $(w + h)^2 \cdot n^{(w+h)^2}$ is a finite number, the algorithm indeed terminates in finite time. □

Remarks:

- Unfortunately, this does not answer the question whether the tiling problem is decidable in general. There are sets of tiles where a tiling of the entire plane is possible, but only in a fashion that is not periodic.
- To settle the question of decidability, we show that these tilings are in fact a surprisingly expressive model: we can use them to simulate any TM. This property will allow a reduction to the halting problem.

Theorem 7.24. *Some tile sets are Turing-complete: tilings can simulate the run of any TM on any input x.*

Proof. To outline the main idea of the proof, we will assume a slightly simpler setting: that we only need to tile the bottom half of the plane, i.e. below the origin. This is only for convenience; with further tricks, the same proof method can be extended to the entire plane.

The main idea of the proof is that each row describes the complete state of the tape of a TM in a specific time step, with the top row corresponding to time step 0, the row immediately below corresponding to time step 1, and so on. We can design the tiles carefully such that given a specific row (i.e. current configuration of the TM), the only possible tiling of the row directly below is the next configuration of the TM. For simplicity, we assume all tiles have no color.

For each tape symbol σ, we can create a tile that has code σ on top and bottom, and a special empty code on the left and right. This already allows us to automatically copy the content of the tape into the row below. Furthermore, we use special tiles to keep track of the tape pointer and the current state: the tape cell with the pointer will also be marked with the current state. If, for example, we have a transition from state q_1 to q_2 that also replaces a 0 by a 1 on the tape and moves the tape pointer one step to the right, then we can describe this behavior with the tiles shown in Figure 7.25.

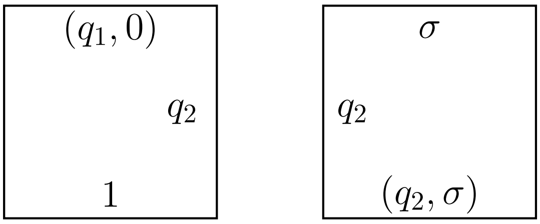


Figure 7.25: Example tiles to simulate a transition of the TM.

The number of tiles we require altogether is only a function of the tape alphabet size and the number of states and transitions in the TM, and thus it is a finite number. By defining some special tiles for the first row, one can also ensure that the only possible tiling of the first row is to have the input x on the tape, the tape pointer at position 0 and the TM in its initial state s_0 (note that this is a significant technical step that we do not discuss here).

With such a set of tiles, if the TM halts on x in k steps, then this allows us to tile the first k rows of the plane, and then no tiling will be possible for the

$(k+1)^{\text{th}}$ row (since there is no following configuration of the TM). On the other hand, if the TM runs forever, then there is always a next configuration, so the tile set allows us to tile the entire lower half of the plane. □

Remarks:

- So we can do actual computations with a set of tiles!

Theorem 7.26. *The tiling problem is undecidable.*

Proof. Consider an instance of the halting problem with TM T and input x. As we have seen in Theorem 7.24, we can create a set of tiles that correspond to running T on x, and a tiling with this set is possible if and only if T does not halt on x. Thus solving this tiling problem allows us to decide whether T halts on x; however, the halting problem is undecidable, so the tiling problem must be undecidable, too. □

Remarks:

- There are other models of computation that work on grids. A popular example is Game of Life.

Definition 7.27 (Game of Life or GoL)**.** *In Game of Life, each cell has two states, black (alive) and white (dead), and the update rule is as follows:*

- *If the cell is black: if it has exactly 2 or 3 black neighbors among its 8 neighboring cells, it remains black, otherwise it becomes white.*
- *If the cell is white: if it has exactly 3 black neighbors among its 8 neighboring cells, it becomes black, otherwise it remains white.*

Remarks:

- These simple rules create a surprisingly wide range of patterns. There are stable configurations which keep their shape without changing; there are oscillators that keep repeating a few specific configurations periodically; there are "gliders" that exhibit a similar periodicity but also slowly move through the grid in the meantime. There are more complex patterns that repeatedly create smaller oscillators or gliders.
- These constructions can then be used to form gadgets on a higher abstraction level: we can create logical AND and OR gates, and ultimately a finite automaton. These tools then allow us to simulate the behavior of a TM in GoL, similarly to the tiles before.

Theorem 7.28. *Game of Life is Turing-complete.*

Remarks:

- As a result, we can also formulate some undecidable problems in this model.

Problem 7.29 (GoL Reachability)**.** *Given an initial configuration c, the task is to decide if another configuration c' will ever occur.*

Theorem 7.30. *GoL Reachability is undecidable.*

Remarks:

- GoL is a in fact a special case of a widespread model of computation on grids called Cellular Automaton.

Definition 7.31 (Cellular Automaton or CA). *A Cellular Automaton consists of a (two-dimensional) grid of cells, where each cell is in a specific state. In each iteration, every cell (concurrently and independently) changes its state based on the current states of the cells in its immediate neighborhood.*

Remarks:

- If we denote the set of states by S, then a CA is essentially described by a function $f : S \times S^N \to S$ (with N denoting the size of the neighborhood). Each cell executes this function in each round to obtain the state in the next round.

- Since GoL is a special case of Cellular Automata, CAs in general are also Turing-complete.

- CAs can model various processes in natural sciences, ranging from Physics to Biology, with the cells of the automaton representing anything from chemical molecules to actual (biological) cells.

- There are many other areas (beyond halting and grids) where we can find undecidable problems. To mention another surprising example: Given a couple of $k \times k$ matrices with integer entries, it is undecidable if they can be multiplied in some order, possibly with repetitions, such that we obtain the zero matrix as a result.

7.4 Post Correspondence Problem

Finally, we discuss some variants of the so-called Post Correspondence Problem. This problem is an interesting conclusion to our whole lecture: it demonstrates that seemingly similar problems can easily have a completely different complexity.

Problem 7.32 (PCP). *We have a set of dominoes, where each domino (α, β) has two words written on the domino: one word α on the top, and one word β on the bottom. Can we make a sequence of these dominoes, possibly with repetitions, such that the concatenation of words on top is the same as the concatenation of words on the bottom?*

Remarks:

- Given a finite alphabet of symbols Σ, a *word* is a finite string formed from these letters, possibly with repetitions. A concatenation of words $\alpha_1, \alpha_2, \ldots$ is the word obtained by writing these words after each other in this order.

Theorem 7.34. *PCP is undecidable.*

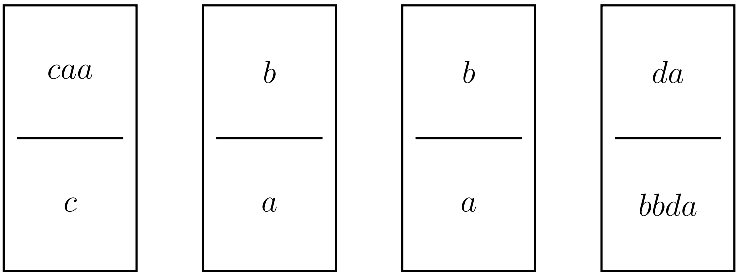


Figure 7.33: Example solution of a PCP: both the top and the bottom string is *caabbda*. The sequence consists of 4 dominoes, using one of the dominoes twice.

Proof. The proof is quite technical, so we only outline the main idea. Similarly to tiles, we can use dominoes to simulate the running of a TM. The concatenated string will describe the history of the run of the TM as a list of subsequent configurations. The bottom string is always "one step ahead" the top string in this computation; thus by defining an appropriate domino for each possible transition in the TM, we can ensure that the next configuration is always a valid follow-up to the current configuration. If the TM reaches a terminal state, then some extra dominoes ensure that the top string can catch up to the bottom string; this way the two strings become identical, and thus we have a valid sequence of dominoes that solves the PCP problem.

Since such a PCP solution exists if and only if the TM halts. Since the halting problem is undecidable, the PCP problem is also undecidable. □

Remarks:

- However, there is a simple algorithm that terminates in finite time if the answer is yes.

Theorem 7.35. *PCP is semi-decidable.*

Proof. We can enumerate all possible domino sequences based on their length in increasing order. Then if there exists a solution with a domino sequence of length k for some finite number k, then until reaching this sequence, we check at most

$$n + n^2 + ... + n^k \leq k \cdot n^k$$

possible sequences. Since each such check takes at most $O(k)$ time, we can find the solution in finite time. □

Remarks:

- The PCP problem is often used in reductions when analyzing problems related to formal languages.
- In terms of the number of dominoes used, the best known method to simulate a TM requires 5 different dominoes. This shows that if we have the correct 5 dominoes, then the problem is undecidable. However, what happens if we restrict the problem to less than 5 dominoes?

Theorem 7.36. *With only 1 domino, the PCP problem is decidable in polynomial time.*

Proof. In this case, any sequence consists of a specific number of repetitions of our single domino. In this case, the top and bottom strings are only identical if our single domino has the same word on the top and bottom side (and in this case, a single instance of the domino already provides a solution). We can easily check this in linear time: we only need to read the two words and compare them. □

Theorem 7.37. *With only 2 dominoes, the PCP problem is decidable.*

Proof. The proof of this claim is quite involved, so we do not discuss it here. □

Remarks:

- With 3 or 4 dominoes, it is still an open question whether PCP is undecidable or not.
- Another possible modification is to restrict the size of the alphabet. More specifically we need an alphabet of at least $|\Sigma| \geq 2$ letters for undecidability. If Σ only consists of a single character, then the problem becomes decidable.

Theorem 7.38. *PCP with* $|\Sigma| = 1$ *is solvable in polynomial time.*

Proof. In this case, we only need to make sure that the top and bottom words have the same length, i.e. the same number of occurrences of our single character. This means that for each available domino, we only need to consider the difference of length between the top and bottom words, which gives us a (not necessarily positive) integer. The task then reduces to analyzing this set of integers, and selecting a subset of them (with possible repetitions) that sums up to 0.

Solving this is rather easy in polynomial time. If one of the integers is 0, then this already forms a valid sequence on its own. If not, then we need to check if there is at least one positive number $x_i > 0$ and at least one negative number $x_j < 0$ among our integers: then a sequence consisting of x_i copies of the number x_j and $|x_j|$ copies of the number x_i also sums up to 0. Otherwise, all the numbers are positive (or negative); in this case, the sum of any sequence is also positive (or negative, respectively). □

Remarks:

- For another variant, we can also restrict the size of the allowed domino sequence.

Problem 7.39 (Bounded PCP)**.** *In Bounded PCP, the input also contains an integer k, and we only accept domino sequences that have length at most k.*

Theorem 7.40. *Bounded PCP is decidable but NP-hard.*

Proof. With n dominoes, we only have n^k possible domino sequences. By enumerating and checking all these possibilities, the problem is clearly decidable in finite time.

The proof of NP-hardness can be shown through a reduction from the longest common substring problem; we do not discuss it here. □

Chapter Notes

The Turing Machine was developed by Alan Turing in 1936, shortly after the Gödel machine motivated the mechanisation of reasoning, but long before the invention of modern day computers [16]. Turing has specifically defined the model in order to study the halting problem, and prove its incomputability. The halting problem (and its different variants) has kept its central place in the area; the majority of known incomputability results are shown through a reduction that comes either directly or indirectly from this problem.

Turing submitted his paper to the London Mathematical Society in May 1936. Emil Post, working independently from the United States, submitted a manuscript detailing a machine almost identical to that of Turing in September of the same year. Turing's paper, however, was published first, in early 1937. Even though he was only four months late, Post's work was overlooked and only dug up by historians years later.

A very similar line of thought and proof technique to the halting problem's incomputability has also appeared in the work of Kurt Gödel, who was studying incompleteness theorems and the axiomatization of natural numbers at about the same time [14]. The general message of these two results has caused a large surprise (even shock) in the scientific community, where the general belief (based on Hilbert's conjectures) was that, intuitively speaking, every well-defined question can be answered. The results have shown that this is not the case, which had far-reaching philosophical consequences for the beliefs about the nature of reasoning at the time. The modern-day interpretation of Gödel's incompleteness theorems, however, is that they are a minor technical difficulty subject to the use of first order predicate calculus for proofs with finitely many steps.

The Turing Machine has also remained the most approachable model to study computations ever since. The closely related concepts (e.g. Universal Turing Machine, Turing-completeness, or different formulations of the Church-Turing thesis) have been gradually developed and refined in the following decades. This rapidly developing area was studied by some of the most important mathematicians of the 20th century, including John von Neumann, Alonzo Church, or Stephen Kleene.

In 1960s, Kurt Gödel strongly attacked the increasing belief that the Church-Turing thesis "holds", and his points still stand today. Roger Penrose led another, more recent high-profile attack in [12].

The tiling problem was first discussed by Wang in 1961 [17]. However, in his first work, Wang conjectured that whenever a tiling exists, a periodic tiling also exists. A few years later his student Berger showed that some tile sets only allow an aperiodic tiling, and that the problem is undecidable due to its connection to the halting problem [3].

Game of Life was devised by John Conway in 1970 [8], and has been analyzed in numerous papers and books since then [2]. There are many simulators online where you can create different patterns and follow their development through the rounds [1].

As for Cellular Automata in general, there is an immense literature discussing different aspects of the topic. Different variants of automata have been used in a very wide range of applications, e.g. generating pseudo-random numbers in computer science [15], modeling the crystallization of snowflakes [4], modeling the geometric patterns on seashells [6] or modeling the flow of traffic on the freeway [10].

The PCP problem was introduced by Emil Post in 1946 [13]. A long line of works have followed that tried to reduce the number of dominoes required for undecidability, going down to set of 7 dominoes in 1996 [9], and then finally 5 tiles in the work of Neary in 2015 [11]. The decidability for 2 dominoes was proven by Ehrenfeucht, Karhumäki and Rozenberg [7], while the NP-hardness of Bounded PCP was first discussed in [5]. In [18] Zhao presents some strategies for solving the PCP problem and the hardest known instances for 5 subclasses of the PCP consisting of few tiles and short words.

This chapter was written in collaboration with Pál András Papp and Peter Belcak.

Bibliography

[1] John Conway's Game of Life Online. `https://playgameoflife.com`.

[2] Andrew Adamatzky. *Game of life cellular automata*, volume 1. Springer, 2010.

[3] Robert Berger. *The undecidability of the domino problem.* Number 66. American Mathematical Soc., 1966.

[4] Charles D Brummitt, Hannah Delventhal, and Michael Retzlaff. Packard snowflakes on the von neumann neighborhood. *Journal of Cellular Automata*, 3(1), 2008.

[5] Robert L Constable, Harry B Hunt III, and Sartaj Sahni. On the computational complexity of scheme equivalence. Technical report, Cornell University, 1974.

[6] Stephen Coombes. The geometry and pigmentation of seashells. *Nottingham: Department of Mathematical Sciences, University of Nottingham*, 2009.

[7] Andrzej Ehrenfeucht, Juhani Karhumäki, and Grzegorz Rozenberg. The (generalized) post correspondence problem with lists consisting of two words is decidable. *Theoretical Computer Science*, 21(2):119–144, 1982.

[8] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[9] Yuri Matiyasevich and Geraud Senizergues. Decision problems for semi-thue systems with a few rules. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 523–531. IEEE, 1996.

[10] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de physique I*, 2(12):2221–2229, 1992.

[11] Turlough Neary. Undecidability in Binary Tag Systems and the Post Correspondence Problem for Five Pairs of Words. In *32nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 30 of *LIPIcs*, pages 649–661, Dagstuhl, Germany, 2015.

[12] Roger Penrose and N David Mermin. The emperor's new mind: Concerning computers, minds, and the laws of physics, 1990.

[13] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[14] Panu Raatikainen. Gödel's Incompleteness Theorems. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2020 edition, 2020.

[15] Marco Tomassini, Moshe Sipper, and Mathieu Perrenoud. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Transactions on computers*, 49(10):1146–1151, 2000.

[16] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[17] Hao Wang. Proving theorems by pattern recognition—ii. *Bell system technical journal*, 40(1):1–41, 1961.

[18] Ling Zhao. Tackling post's correspondence problem. In *International Conference on Computers and Games*, pages 326–344. Springer, 2002.