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?

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:

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 (+, −, ×, ÷, 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, ignor-
ing constant factors. Such an asymptotic time complexity can be
expressed by the “big \( \mathcal{O} \)” notation.

**Definition 1.6** (\( \mathcal{O} \)-notation). The \( \mathcal{O} \)-notation is used to denote a set of func-
tions with similar asymptotic growth. More precisely,
\[
\mathcal{O}(f(n)) = \{ g(n) \mid \lim_{n \to \infty} \frac{g(n)}{f(n)} < \infty \}.
\]

**Remarks:**
• In other words, \( \mathcal{O}(f(n)) \) is the set of functions \( g(n) \) that asymptoti-
cally 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
\( \omega(), \omega(), \Theta() \), 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 ba-
sic 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, \ldots , 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 num-
ber of items. Even if there were only \( n = 100 \) items to be evalu-
ated, the currently fastest supercomputer in the world would take
\( 2^{200} \text{ops}/(114 \cdot 10^{15} \text{ops/s}) \approx 273\,000 \text{yrs} \) 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?!
def knapsack(items, capacity):
    items.sort(key=lambda item: -item.value/item.weight)

    for c in range(capacity+1):
        V[i+1][c] = max(V[i][c-item.weight] + item.value, V[i][c])

    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 pseudo-code, 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 - 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 $O(n \cdot c)$. 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 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 $O(n \cdot c)$.
- 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 $V[0][c] = [0, \ldots, 0]$ 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 $c \in [0, \ldots, c - item.weight)$), 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.
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.

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

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 \( V[i][c] \) once we have computed both \( V[i+1][c] \) 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.
- 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 unsplitable 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 \( \rightarrow \) tanksize and a preparation time per liter. Given 1 hour 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?

Remarks:
- Top-down DP is inheriting the best of recursion and bottom-up DP. Consequently, the time complexity of Algorithm 1.16 is \( \mathcal{O}(2^n \cdot \text{capacity}) \).
- 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.

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

\[
\begin{align*}
& a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\
& a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\
& \vdots \hspace{1cm} \vdots \hspace{1cm} \vdots \hspace{1cm} \vdots \\
& a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m
\end{align*}
\]

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

\[
f(x) = c_1x_1 + c_2x_2 + \cdots + c_nx_n
\]

where \( a_{ij}, b_i, \) and \( c_i \) are given real-valued parameters. We call the point \( x \) an optimum of the LP.
Remarks:

- There is also a short hand notation using linear algebra
  \[ \max \{ c^T x \mid Ax \leq b, x \geq 0 \}. \]
  where \( A \) is the matrix with entries \( a_{ij} \) and \( b \) and \( c \) 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 \( a^T x = b \) can be rewritten as a combination of \( a^T x \leq b \) and \( a^T x \geq b \) which itself can be rewritten as \(-a^T x \leq -b \). Also, minimizing a linear function with coefficients \( c_1, \ldots, c_n \) is the same as maximizing a linear function with coefficients \(-c_1, \ldots, -c_n \).

- 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 \( \min \{ |x| \mid x \leq 1, -x \leq 1 \} \).

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 \( x_1 \) and \( x_2 \), any point in the line segment joining them can be written as \( x = \lambda x_1 + (1 - \lambda) x_2 \) for \( \lambda \in [0,1] \). For any constraint \( a^T x \leq b \), we compute
  \[ a^T [\lambda x_1 + (1 - \lambda) x_2] = (1 - \lambda) a^T x_1 + \lambda a^T x_2 \leq (1 - \lambda) b + \lambda b = b. \]

Definition 1.22. Given an LP, we call polytope the set of feasible points.

A constraint \( a^T x \leq b \) is tight at \( x \) if \( a^T x = b \). 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 \( x_1, x_2 \) with \( n - 1 \) common activating constraints, we say that the two nodes \( x_1, x_2 \) are neighbors.
Definition 1.25 (Local Optimum). A feasible node \( x \) is a local optimum if 
\( f(x) \geq f(y) \) for any neighboring node \( y \).

Remarks:
- In contrast to a local optimum, an optimum from Definition 1.18 is called a 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 \( x^* \) returned by the simplex algorithm is an optimum.

Proof. Let us consider the hyperplane \( c^T x = f^* \), where \( f^* = c^T x^* \). We know that all the neighbors of node \( x^* \) are on the side \( c^T x \leq f^* \). 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 \( c^T x > f^* \), and hence the node \( x^* \) is a global optimum.

![Figure 1.27: Illustration of Theorem 1.26. The neighbors of \( x^* \) are \( x_1 \) and \( x_2 \).](image)

1.6. LINEAR RELAXATION

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 \( A x = b \) or \( y \) is satisfied when \( x \) is optimal in the phase 1 LP. Since \( \max(-1^T y) = \min(\text{sum}(y)) \) is optimal for \( y = 0 \), node \( x \) is optimal in the phase 1 LP. With Theorem 1.26, we know that the phase 1 LP will find such a node \( x \).

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 `scipy.optimize` module.

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

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

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.
1.7 FLOWS

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?

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

\[
\begin{align*}
\text{maximize} & \quad \sum_{i,j} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{j} x_{ij} = 1 \quad \text{for each customer } i \\
& \quad \sum_{i} x_{ij} \leq 1 \quad \text{for each cab } j \\
& \quad x_{ij} \in \{0,1\}
\end{align*}
\]

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 \( \text{in}(u) \) (resp. \( \text{out}(u) \)) 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 \( \omega(e) \) to 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 \( f : E \to \mathbb{R}_+ \) such that:

\[
\begin{align*}
f(u,v) & \leq c(u,v) \quad \text{for all } (u,v) \in E \quad \text{(capacity constraints)} \\
\sum_{v \in \text{in}(u)} f(v,u) & = \sum_{v \in \text{out}(u)} f(u,v) \quad \text{for all } u \in V \setminus \{s,t\} \quad \text{(flow conservation)}
\end{align*}
\]

We call the total flow reaching \( t \) the \textit{value} of \( f \), i.e., \( |f| = \sum_{u,v \in E} f(u,v) \).
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-integer flow. If there is a path from s to t with every edge being non-integer, we can increase the flow on that path, so our original flow was not optimal. Hence, there cannot be a non-integer path from s to t.

Let u be a node adjacent to an edge e with non-integer flow. Then u needs at least another edge e' with non-integer flow because of flow conservation at node u. We can follow these non-integer edges until we find a path from s to t with non-integer flow. Then u needs another edge e'' with non-integer flow because of flow conservation at node u. We can follow these non-integer edges until we find a path from s to t with non-integer flow. Now we have one edge less with non-integer flow. If there is still an edge with non-integer 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:

\[
\begin{align*}
\text{maximize} & \quad \sum_{e=(u,v) \in E} x_{e} \\
\text{subject to:} & \quad x_{e} \leq c_{e} \\
& \quad x_{e} \geq 0 \\
& \quad \sum_{e \in \text{in}(u)} x_{e} - \sum_{e \in \text{out}(u)} x_{e} = 0 \quad \text{for all } u \in V \setminus \{s,t\}
\end{align*}
\]

1.7 FLOWS

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 \( s \to t \) 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 of an edge is defined as

\[
\text{residual}(u,v) = c(u,v) - f(u,v) + f(v,u)
\]

Remarks:

- We can find an augmenting path in linear time, using a recursive algorithm!
- Instead of using the residual capacity defined above, we can 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 \( c(u,v) = f(u,v) + f(v,u) \) and we can use the former check for finding edges with non-zero residual capacity.

Algorithm 1.43: Find augmenting path

```python
def find_augmenting_path(u, t, G, flow, visited):
    visited.insert(u)
    for v in G.neighbors(u):
        if v 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
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

Chapter Notes

The word algorithm is derived from the name of Muhammad ibn Masur 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

    for solving divide-and-conquer recurrences. SIGACT News, 12(3):36-44,
    September 1980.
    berg, 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.
    8:399-404, 1956.