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

```
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

```python
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

```python
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

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

We are interested in finding a point $\boldsymbol{x} = (x_1, \ldots, x_n)^T$ 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 $c_1, \ldots, c_n$ 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 $\min\{t|x \leq t, -x \leq t\}$.

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

$$\boldsymbol{a}^T[\boldsymbol{x}_1 + \lambda(\boldsymbol{x}_2 - \boldsymbol{x}_1)] = (1-\lambda)\boldsymbol{a}^T\boldsymbol{x}_1 + \lambda\boldsymbol{a}^T\boldsymbol{x_2} \leq (1-\lambda)b + \lambda b = b.$$

□

Definition 1.22. *Given an LP, we call* ***polytope*** *the set of feasible points.* → notebook
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 $\boldsymbol{x_1}, \boldsymbol{x_2}$ with $n-1$ common activating constraints; we say that the two nodes $\boldsymbol{x_1}, \boldsymbol{x_2}$ are* ***neighbors****.*

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.

→ notebook

```
1 def simplex(polytope, f, x):
2   for y in neighbors(x, polytope):
3     if f(y) > f(x):
4       return simplex(polytope, f, y)
5   return x
```

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. □

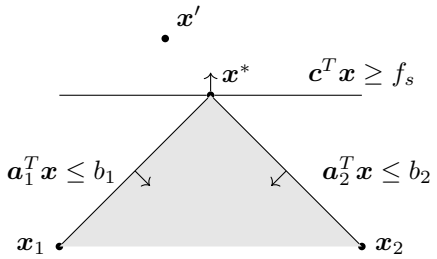

Figure 1.27: Illustration of Theorem 1.26. The neighbors of $\boldsymbol{x}^*$ are x_1 and x_2.

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:*

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

Remarks:

- Unlike LPs, no efficient algorithm solving ILPs is known.
- It is tempting to relax the constraints $x_i \in \{0, 1\}$ to $0 \leq x_i \leq 1$, 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 $0 \leq x_i \leq 1$.*

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

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

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*

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

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

*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:*

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

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:*

$$residual(u, v) = c(u, v) - f(u, v) + f(v, u)$$

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

```
1 def find_augmenting_path(u, t, G, flow, visited):
2   visited.insert(u)
3   for v in G.neighbors(u):
4     if v is not in visited and residual[u, v] > 0:
5       path = find_augmenting_path(v, t, G, flow, visited)
6       if len(path) > 0 or v == t:
7         path.append((u, v))
8         return path
9   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

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.