Algorithmic Discrete Mathematics

lecture notes, summer term 2024

Prof. Dr. Yann Disser

July 16, 2024

Contents

1 Algorithms and their Analysis

This lecture is intended as an introduction to the mathematical design and analysis of algorithms. Abstractly, an algorithm is nothing more than a formal specification of a systematic way to solve a computational problem. For example, at school we learn a basic algorithm to multiply any pair of numbers using only basic multiplications and additions, e.g.,

	7	3	·	5	8
		3	5		
+			1	5	
+			5	6	
+		$_1$		2	4
=		4	2	3	4

This method uses 4 multiplications in this example, and n^2 multiplications in general for two numbers with n digits each. It is a natural question whether we can do with fewer multiplications. Indeed, we can compute the same result by only using the 3 multiplications

$$P_1 = 7 \cdot 5 = 35, \quad P_2 = 3 \cdot 8 = 24, \quad P_3 = (7-3)(8-5) = 12,$$

as follows:

	7	3	·	5	8
		3	5		
+			3	5	
+			2	4	
+				2	4
+		$_1$	1_1	2	
=		4	2	3	4

This works, since we compute

$$\begin{aligned} 100P_1 + 10P_1 + 10P_2 + P_2 + 10P_3 &= 70 \cdot 50 + 10 \cdot (P_1 + P_2 + P_3) + 3 \cdot 8 \\ &= 70 \cdot 50 + 10 \cdot (3 \cdot 5 + 7 \cdot 8) + 3 \cdot 8 \\ &= 70 \cdot 50 + 3 \cdot 50 + 70 \cdot 8 + 3 \cdot 8. \end{aligned}$$

This method can be generalized to only take roughly $\alpha n^{1.58}$ multiplications, and even better algorithms are known that take less than $\alpha n^{1.01}$ multiplications (for very large values of αn and some constant α).

In this lecture, we will be concerned with algorithmic questions of this type. In particular, we will investigate how to systematically solve algorithmic problems and compare different algorithms in terms of efficiency of computation.

Algorithmic problems and solutions permeate all aspects of our modern lives. Here is a (very) short list of questions that we commonly delegate to algorithms:

- How to search large collections of data like the Internet?
- How to find a shortest route in a transportation network?
- How to assign and route delivery trucks to ship online orders?
- How to schedule lectures in order to minimize overlaps?
- How to position cellular network towers for good coverage?
- How to layout large buildings to allow for fast evacuation?
- How to render complex 3D-scenes to a 2D-display?
- How to distinguish photos of cats and dogs?

To illustrate our approach, we will begin by informally considering a simple problem. We will later formalize our analysis and apply it to more challenging problems.

1.1 Example 1: The Celebrity Problem

Consider a group P of n people where each person $p \in P$ knows a subset $P_p \subseteq P$ of the group. In this setting, we call a person $p \in P$ a *celebrity* if everybody knows them, but they know nobody, i.e., if $p \in P_{p'}$ for all $p' \in P \setminus \{p\}$ and $P_p = \emptyset$. Our problem is to determine whether there is a celebrity in P and, if yes, who the celebrity is (obviously, there can be at most one celebrity in P). To do this, we may ask questions of the form "Does $a \in P$ know $b \in P$?". How can we systematically solve our problem using the smallest number of questions possible?

We can solve the problem simply by asking every pair (a, b) of (distinct) persons whether a knows b and check explicitly if there is a celebrity. This naive algorithm takes $n \cdot (n-1)$ questions. Can we do better?

Observe that even if we knew that either $a \in P$ is the celebrity or there is none, we still need at least $2(n-1)$ questions to check whether a is the celebrity. This means that we *always* need at least $2(n-1)$ questions at the very least. On the other hand, if we are lucky and guess $a \in P$ correctly, we may only need these $2(n-1)$ questions.

However, in the theory of algorithms it is customary to demand a guarantee regarding the efficiency of an algorithm. This means that we want to bound the maximum number of questions our algorithm may need in the *worst case*. Throughout the lecture, we will focus on this perspective and analyze the *worst-case performance* of algorithms. So can we do better than asking all $n \cdot (n-1)$ potential questions *in the worst case*?

Mathematical induction motivates an important paradigm of algorithm design: *recursion.* The idea is simple: Assume we already knew an algorithm A_{n-1} to solve our problem efficiently for groups of at most $n-1$ people. We can apply this algorithm on a subset $P' = P \setminus \{p\}$ for some arbitrarily chosen person $p \in P$ to efficiently check whether P' has a celebrity. As described above, we can invest $2(n-1)$ additional questions to check whether p is the celebrity in P. If not and if P' has a celebrity p', the answers to the two questions involving p and p', which we already asked, tell us whether p' is a celebrity in P. If not, there is no celebrity.

We trivially know the best-possible algorithm A_1 for groups of size 1. We can therefore recursively define A_i to use A_{i-1} for $i \in \{2, \ldots, n\}$ as described above. It is easy to see that we still ask all possible questions: If $f(i)$

denotes the number of questions asked by algorithm A_i for $i \in \{1, \dots, n\}$, we obtain

$$\begin{aligned} f(n) &= f(n-1) + 2(n-1) \\ &= f(n-2) + 2(n-2) + 2(n-1) \\ &= \dots \\ &= 2\sum_{i=1}^{n-1}(n-i) \\ &= n(n-1). \end{aligned}$$

How can we avoid some of these questions?

The main source of questions in our algorithm comes from the fact that we need to explicitly check whether the person $p \in P$ arbitrarily excluded from P is the celebrity. We can save a lot of questions by ensuring that we exclude a person that cannot be the celebrity. To do this, we can simply ask a single question for any pair (a, b) of distinct persons. Either a knows b, then a cannot be the celebrity; or a does not know b, then b cannot be the celebrity. In either case, we can make sure to exclude a person p that cannot be the celebrity.

Our algorithm can formally be stated using *pseudocode*:

Algorithm: FINDCELEBRITY(P)

input: set of people P
output: celebrity $p^\star \in P$ if one exists, otherwise $\emptyset$

if $|P| = 1$**:**
  return unique person $p \in P$

take any $\{a, b\} \subseteq P$
if *"Does a know b?"***:**
  $p \leftarrow a$
else
  $p \leftarrow b$

$p' \leftarrow$ FINDCELEBRITY$(P \setminus \{p\})$
if $p' \neq \emptyset$ **and** *"Does p know p'?"* **and not** *"Does p' know p?"***:**
  return p'
else
  return $\emptyset$

The number of questions $f(n)$ asked by this algorithm reduces to

$$f(n) = \begin{cases} 0 & \text{if } n = 1, \\ 1 + f(n-1) + 2 & \text{if } n > 1. \end{cases}$$

To solve such recurrences, we always proceed in the same manner: We first guess the solution by telescoping, i.e.,

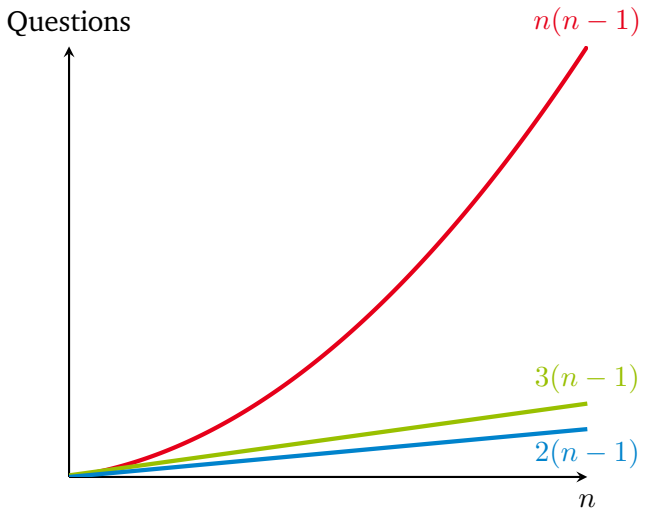
$$f(n) = 3 + f(n-1) = 3 + 3 + f(n-2) = \dots = 3(n-1) + 0 = 3(n-1).$$

We can then formally prove our hypothesis that the number of questions asked is $f(n) = 3(n-1)$ by induction. The induction basis holds since $f(1) = 0$. Now assume that $f(i-1) = 3(i-2)$ holds. Then,

$$f(i) = 3 + f(i-1) = 3(i-1),$$

which concludes the proof.

Obviously, we have improved our algorithm, at least for $n > 3$, since then $3(n-1) < n(n-1)$. But how significant is this improvement? The theory of algorithms is mainly concerned with the *asymptotic* efficiency, i.e., the behavior for large n. The following plot illustrates that, asymptotically, our *linear* number of questions $3(n-1)$ is close to the *linear* lower bound of $2(n-1)$ and much lower than the quadratic number achieved by the naive algorithm. We will soon make this intuition precise.



1.2 Running Times of Algorithms

In the previous section we measured the efficiency of our algorithms by counting the number of questions they ask relative to the size of the group we are given. To generalize this to arbitrary problems and algorithms, we first have to provide formal definitions of the notion of a problem, of the size of a given instance of such a problem, and what it means to solve a problem.

Definition 1.1. An *algorithmic problem* is given by a tuple $(\mathcal{I}, (\mathcal{S}_I)_{I \in \mathcal{I}})$ of a set of *instances/inputs* $\mathcal{I}$ and a family of sets of *solutions* $(\mathcal{S}_I)_{I \in \mathcal{I}}$. Every instance $I \in \mathcal{I}$ is associated with an *input size* $|I|$.

Example. The set of instances of the multiplication problem we considered initially are all pairs of natural numbers, the size of an instance is given by the number of digits of the larger number. An instance of the celebrity problem is a set P together with the sets P_p for all $p \in P$, and the size of an instance is the cardinality of P.

Definition 1.2. An algorithm A *solves* an algorithmic problem $(\mathcal{I}, (\mathcal{S}_I)_{I \in \mathcal{I}})$ if, given any instance $I \in \mathcal{I}$, it terminates after a finite number of steps and returns a solution $A(I) \in \mathcal{S}_I$.

Of course, we can no longer count the number of "questions" when speaking about problems other than the celebrity problem. Instead, we count the number of *elementary operations* like accessing variables, performing simple arithmetic, comparing, copying and storing numbers or words, evaluating conditional statements, etc. We adopt the *unit cost model* where we assume that each elementary operation takes one time step, independent of the size of the numbers involved. Additionally, we assume that we can iterate over a set X in

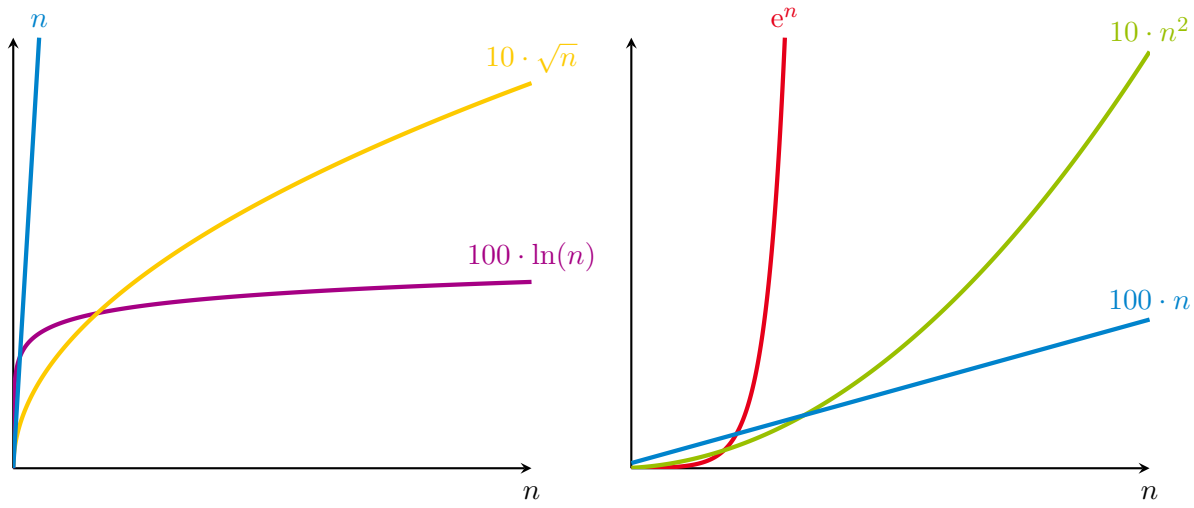
$|X|$ steps and that we can infer $|X|$ in one step. The unit cost model significantly simplifies our mathematical analysis of algorithms and allow us to focus on their qualitative behavior, without being distracted by technical details that depend on how an algorithm is implemented in practice.

Definition 1.3. The (worst-case) *running time* of an algorithm A for an algorithmic problem $(\mathcal{I}, (\mathcal{S}_I)_{I\in\mathcal{I}})$ is given by the function $f\colon \mathbb{N} \to \mathbb{N}$ with

$$f(n) := \max_{I\in\mathcal{I}, |I|\leq n} \{\text{number of elementary operations needed by } A \text{ to compute } A(I)\}\,.$$

1.3 Asymptotic Growth

We mentioned that we will compare algorithms according to the asymptotic growth of their running times for large input sizes n. Asymptotically, functions that only differ by constant factors or offsets exhibit a similar growth, as we saw above in the illustration for the polynomials $3(n-1)$ and $2(n-1)$ compared with $n(n-1) = n^2 - n$. In addition, since we already made the simplifying assumption that all elementary operations have the same cost, we should really not distinguish between running times that are only a constant factor apart. The following examples illustrate the irrelevance of constant factors when comparing the asymptotic growth of functions:



Intuitively, it seems reasonable to say that n^2 "grows faster than" n and $\sqrt{n}$ "grows faster than" $\ln(n)$. These examples also indicate that we can focus on the fastest growing term when analyzing the asymptotic growth of a function. In other words, $n^2+10n+100\sqrt{n}$ essentially behaves like n^2 for large n. We can capture this intuition formally by observing that $n^2 < n^2 + 10n + 100\sqrt{n} < 2n^2$ for $n > 29$. Together with our earlier observation that we can ignore constant factors, this indicates that n^2 "grows similarly fast as" $n^2 + 10n + 100\sqrt{n}$.

We now introduce notation that formally captures this intuition. We begin by formalizing the set of functions that "grow no faster than" a function g.

Definition 1.4. The set of functions that *asymptotically grow no faster than* $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ is defined as

$$\mathcal{O}(g) := \{\, f\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0, n_0 \in \mathbb{N}\, \forall n \geq n_0\colon\ f(n) \leq cg(n)\}\,.$$

Example. As intended, we have $n^2 + 10n + 100\sqrt{n} \in \mathcal{O}(n^2)$, e.g., by choosing $c = 2$ and $n_0 = 29$.

We can now easily extend this notation to other asymptotic relationships between functions.

Definition 1.5. Let $g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. We define the set of functions that *asymptotically grow*

(i) *at least as fast as g:* $\Omega(g) := \{ f\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \mid g \in \mathcal{O}(f)\}$,

(ii) *as fast as g:* $\Theta(g) := \mathcal{O}(g) \cap \Omega(g)$,

(iii) *(strictly) slower than g:* $o(g) := \mathcal{O}(g) \setminus \Theta(g)$,

(iv) *(strictly) faster than g:* $\omega(g) := \Omega(g) \setminus \Theta(g)$.

The symbols $\mathcal{O}, \Omega, \Theta, o, \omega$ are called *(Bachmann-)Landau symbols* after their creators, or *(big-)O notation*.

Remark 1.6. It is often convenient to use Landau symbols within arithmetic expressions and derivations, e.g.,

$$n^2 + 4n = n^2 + \mathcal{O}(n) = n^2 + o(n^2).$$

In this context, we interpret operations and relations element-wise, i.e., $f + \Xi(g) := \{ f + h \mid h \in \Xi(g)\}$, and we write $f = \Xi(g)$ for $f \in \Xi(g)$ and $\Xi(g) = \Xi'(h)$ for $\Xi(g) \subseteq \Xi'(h)$, where Ξ, Ξ' denote arbitrary (expressions containing) Landau symbols.

Proposition 1.7. Let $f, g, h\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$. The Landau symbols $\Xi \in \{\mathcal{O}, \Omega, \Theta, o, \omega\}$ have the following properties:

(i) Transitivity, i.e., if $f \in \Xi(g)$ and $g \in \Xi(h)$, then $f \in \Xi(h)$.

(ii) If $f \in \mathcal{O}(g)$, then $f + g \in \Theta(g)$.

(iii) For all constants $a, b > 1$, we have $\Xi(\log_a n) = \Xi(\log_b n)$, which justifies the notation $\Xi(\log n)$.

(iv) We have $f \in \mathcal{O}(1)$ if and only if $c \in \mathbb{N}$ exists with $f(n) \leq c$.

(v) For all constants $0 < a < b$ and $1 < \alpha < \beta$, we have

$$\mathcal{O}(1) \subsetneq \mathcal{O}(\log n) \subsetneq \mathcal{O}(n^a) \subsetneq \mathcal{O}(n^b) \subsetneq \mathcal{O}(\alpha^n) \subsetneq \mathcal{O}(\beta^n) \subsetneq \mathcal{O}(n!) \subsetneq \mathcal{O}(n^n).$$

Equipped with this notation, we can now formally state our result for the celebrity problem. Recall that every algorithm needs at least $2(n-1) = \Omega(n)$ steps, and our algorithm FindCelebrity only takes $3(n-1) = \mathcal{O}(n)$ steps. This means that we have found a tight bound, which we can express using Θ-notation:

Theorem 1.8. The best possible running time for solving the celebrity problem is $\Theta(n)$.

1.4 Example 2: Matrix Multiplication

Similarly to our initial example of multiplying two numbers, we can ask how to efficiently compute a matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B with $n = 2^k$ for $k \in \mathbb{N}$. Note that we can add rows and

columns of 0's to bring A and B into this form, increasing both dimensions by a factor of at most 2. Of course, we can explicitly compute $C = (c_{ij})$ via

$$c_{ij} = \sum_{\ell=1}^{n} a_{i\ell} b_{\ell j},$$

which takes n multiplications and $n-1$ additions, i.e., $\Theta(n)$ operations, per entry, for a total of $\Theta(n^3)$ operations. Can we do better?

Consider the following recursive idea: We subdivide the matrices into four parts

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix},$$

of size $n/2 \times n/2$, and compute C via

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2}. \end{aligned}$$

This takes 8 multiplications of matrices of roughly half dimension. If $f(n)$ denotes the number of elementary multiplications we need for matrices of dimension n, we get

$$f(n) = 8f(n/2) = \cdots = 8^{\log_2 n} = 8^{\log_8 n / \log_8 2} = n^{1/\log_8 2} = n^3,$$

where we used that two 1×1 matrices can be multiplied with a single elementary multiplication. This means, that we have not improved our running time yet.

However, we can save one matrix multiplication by computing the auxiliary matrices

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}), \end{aligned}$$

and then

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6. \end{aligned}$$

This way, we can reduce the number of multiplications to

$$f(n) = 7f(n/2) = 7^{\log_2 n} = 7^{\log_7 n / \log_7 2} = n^{\log_2 7} = n^{2.81}.$$

We can easily see that the number of multiplications dominates our running time, by observing that the number of matrix additions in each recursive level is bounded by a constant times the number of matrix multiplications. We get the following result.

Theorem 1.9. Two $n \times n$ matrices can be multiplied in time $\mathcal{O}(n^{2.81})$.

Remark 1.10. The above algorithm is called *Strassen's algorithm*. The best known matrix multiplication algorithm has running time $\mathcal{O}(n^{2.373})$ and was found by François Le Gall in 2014.

2 Searching and Sorting

In this chapter we consider the problem of sorting a collection of elements and of finding a specific element in an already sorted collection. To that end, suppose that we are given as input a *list* of elements $L = (L_1, \dots L_n) \in X^n$ over a ground set X, together with a total order on the elements of X. We assume that comparing two elements is an elementary operation and we use n as a measure for the input size.

2.1 Binary Search

We first consider the search problem, where we want to determine whether element x occurs in the list $L = (L_1, \dots, L_n)$ and, if yes, find an index $i \in \{1, \dots, n\}$ with $L_i = x$. Of course, we can always solve this problem simply by explicitly comparing each element of L with x.

Algorithm: LINEARSEARCH(L, x)

input: list $L = (L_1, \dots, L_n)$, element $x \in X$
output: index $\min\big(\{i \in \{1, \dots, n\} | L_i = x\} \cup \{\infty\}\big)$

```
for i ← 1, ..., n:
    if L_i = x:
        return i
return ∞
```

In general, this algorithm obviously has linear running time $\Theta(n)$ in the worst-case, and we cannot hope to do better.

However, if we know that L is sorted, i.e., $L_1 \leq L_2 \leq \dots \leq L_n$, we can improve on this significantly. The idea is simple: If we compare first to the element (roughly) in the middle of the list, we can discard half of the list, depending on the result of the comparison. We can then repeat the same idea recursively on the remaining half. This algorithm is commonly called *binary search*.

The following pseudocode shows an iterative implementation of binary search. Note that, in case x does not occur in L, the algorithm below outputs the index corresponding to the position at which x would need to be inserted in order to maintain a sorted list. This convention will be useful later.

Algorithm: BINARYSEARCH(L, x)

input: sorted list $L = (L_1, \ldots, L_n)$, element $x \in X$
output: index $\min\big(\{i \in \{1, \ldots, n\} | L_i \geq x\} \cup \{n+1\}\big)$

```
l, r ← 1, n
while l ≤ r:
    m ← ⌊(l + r)/2⌋
    if L_m < x:
        l ← m + 1
    else
        r ← m − 1
return l
```

Theorem 2.1. BINARYSEARCH solves the search problem for sorted lists in time $\Theta(\log n)$.

Proof. To show correctness, let $i^\star := \min\big(\{i \in \{1, \ldots, n\} | L_i \geq x\} \cup \{n+1\}\big)$ be the index we need to find. We claim that the following invariant holds throughout the course of the algorithm: $i^\star \in \{l, \ldots, r+1\}$. This is true initially, and it is easy to see that it remains true whenever l or r are updated (since L is sorted). The algorithm terminates, since at least the index m is removed from the set $\{l, \ldots, r+1\}$ in each step. In the last iteration, either $r = m$ and l is set to $l = m + 1$, or $l = m$ and r is set to $r = m - 1$, i.e., $l = r + 1$ in either case. By our invariant, we know that at that point $i^\star \in \{l\}$, hence the algorithm returns the correct result.

The claimed running time follows from the fact that the number of iterations of the loop lies between $\lfloor \log_2 n \rfloor$ and $\lceil \log_2 n \rceil + 1$ and all other operations are elementary. □

Note that binary search can also be implemented elegantly using recursion:

Algorithm: BINARYSEARCH'(L, x)

input: sorted list $L = (L_1, \ldots, L_n)$, element $x \in X$
output: index $\min\big(\{i \in \{1, \ldots, n\} | L_i \geq x\} \cup \{n+1\}\big)$

```
if n = 0:
    return 1
else
    m ← ⌊(n + 1)/2⌋
    if L_m < x:
        return m + BINARYSEARCH'(L_{m+1,...,n}, x)
    else
        return BINARYSEARCH'(L_{1,...,m−1}, x)
```

2.2 Basic Sorting Algorithms

In the previous section we have seen why it can be useful to sort collections of data. Suppose, again, that we are given this data in the form of a list $L = (L_1, \ldots, L_n)$. How can we sort L efficiently?

Sorting Problem
input: list $L = (L_1, \ldots, L_n) \in X^n$ with X totally ordered
problem: reorder L such that $L_i \leq L_j$ for all $i \leq j$

First observe that L is sorted if and only if $L_i \leq L_{i+1}$ for all $i \in \{1, \ldots, n-1\}$, i.e., if all pairs of consecutive elements are in the correct relative order. This observation suggests the following naive algorithm: Go over the list repeatedly from front to back and swap pairs of consecutive elements that are in the wrong relative order until no such pairs are found anymore. This algorithm is called *bubble sort*, since it makes elements "bubble" towards the end of the list (cf. Figure 2.1). In particular, after the i-th pass over the list, the last i elements have reached their correct positions. This means that we need at most $n-1$ passes over the list and we may stop each pass one element sooner than the previous one. This allows us to slightly improve the algorithm:

Algorithm: BUBBLESORT(L)
input: list $L = (L_1, \ldots, L_n)$
output: sorted version of L

```
for i ← 1, ..., n − 1 :
    for j ← 1, ..., n − i :
        if L_j > L_{j+1} :
            (L_{j+1}, L_j) ← (L_j, L_{j+1})
return L
```

Theorem 2.2. BUBBLESORT solves the sorting problem in time $\Theta(n^2)$.

Proof. Correctness follows from the fact that, after the i-th iteration of the outer loop, element L_{n-i+1} is at the correct position.

We can obtain a lower bound on the running time by only considering the first $\lfloor n/2 \rfloor$ iterations of the outer loop and, for each such iteration, only the first $\lfloor n/2 \rfloor$ iterations of the inner loop. This gives a total of $\lfloor n/2 \rfloor^2 = \Omega(n^2)$ iterations and all other operations are elementary. Similarly, we obtain an upper bound by pretending that both loops always run n times. Even with this generous estimation, we have a total of $n^2 = \mathcal{O}(n^2)$ elementary operations.

Since the actual number of elementary operations must lie between both bounds, we obtain the claimed running time. □

We observed that in each iteration of BUBBLESORT one more element is guaranteed to have reached its final position in the list. Since this property of the algorithm is sufficient to sort the list in $n-1$ iterations, we can further improve the algorithm by concentrating on finding and repositioning the i-th element (in sorted order) in iteration i, while forgoing any other swapping operations. The resulting algorithm is called *selection sort*, since we select one item in each iteration (cf. Figure 2.1):

Algorithm: SELECTIONSORT(L)

input: list $L = (L_1, \ldots, L_n)$
output: sorted version of L

for $i \leftarrow 1, \ldots, n-1$**:**
  $j \leftarrow \arg\min_{\ell \in \{i,\ldots,n\}} \{L_\ell\}$
  $(L_i, L_j) \leftarrow (L_j, L_i)$
return L

While selection sort only performs a linear number of element moves, it still needs a quadratic number of comparisons.

Theorem 2.3. SELECTIONSORT solves the sorting problem in time $\Theta(n^2)$.

Proof. By definition of SELECTIONSORT, the first i items are in their correct positions at the end of iteration i. This implies that the list is correctly sorted upon termination of the algorithm. Since it takes linear time in the worst case to locate the smallest item in a list, an analogous estimation as for BUBBLESORT still yields a quadratic running time. □

Yet another natural sorting algorithm that we often use when playing card games is *insertion sort*. The idea of this algorithm is to add elements one at a time to a presorted sublist. We start with a sublist containing the first element and insert each successive element at its correct relative position to keep the sublist sorted (cf. Figure 2.1). We already know a good algorithm to find this correct relative position, namely BINARYSEARCH. This reduces the number of element comparisons to subquadratic. However, in order to make room for the element inserted in each iteration, we may need to shift the positions of all other elements in the sorted sublist, so that we still get a quadratic numer of moves in the worst case.

Algorithm: INSERTIONSORT(L)

input: list $L = (L_1, \ldots, L_n)$
output: sorted version of L

for $i \leftarrow 2, \ldots, n$**:**
  $j \leftarrow$ BINARYSEARCH$(L_{1,\ldots,i-1}, L_i)$
  $(L_j, L_{j+1}, \ldots, L_i) \leftarrow (L_i, L_j, \ldots, L_{i-1})$
return L

Theorem 2.4. INSERTIONSORT solves the sorting problem in time $\Theta(n^2)$.

Proof. Again, INSERTIONSORT maintains the invariant that the first i items are in their correct order at the end of iteration i, which implies correctness of the algorithm.

It is obvious that INSERTIONSORT has running time $\mathcal{O}(n^2)$, since the time needed for binary search on a list of at most $n-1$ numbers is $\mathcal{O}(\log n)$ (by Theorem 2.1), and the time to relocate at most n numbers is $\mathcal{O}(n)$. To see the lower bound, consider a list initially in decreasing order and consider only the iterations for $i \geq \lfloor n/2 \rfloor$. In each of these $\lceil n/2 \rceil$ iterations the time needed for BINARYSEARCH is $\Omega(\log \lfloor n/2 \rfloor) = \Omega(\log n)$

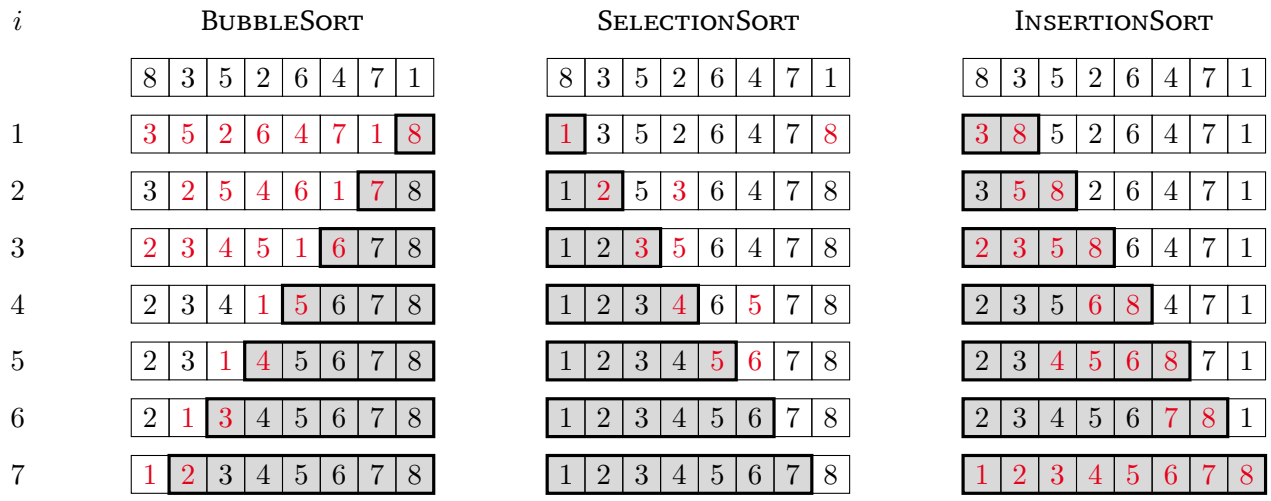


Figure 2.1: Illustration of the basic sorting algorithms. The grey sublists are guaranteed to be sorted; the elements changing position in each iteration are marked in red.

(by Theorem 2.1) and the time to relocate exactly $i \geq \lfloor n/2 \rfloor$ elements is $\Omega(\lfloor n/2 \rfloor) = \Omega(n)$. This means that the time needed in the worst case is at least $\lceil n/2 \rceil \cdot (\Omega(\log n) + \Omega(n)) = \Omega(n^2)$. Overall, the (worst-case) running time thus is $\Theta(n^2)$, as claimed. □

2.3 Mergesort

There are several sorting algorithms that asymptotically have a better running time than $\Theta(n^2)$. We will see a simple recursive idea that uses the paradigm of *Divide and Conquer*. The name of this paradigm expresses the general idea of splitting problem instances into two (roughly) equal parts, solving each of the parts separately, and then combining results. The solution to the following recurrence illustrates why this approach is promising to achieve subquadratic running times, provided that combining results takes linear time.

Proposition 2.5. Let $f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ be such that

$$f(n) = \begin{cases} 1 & \text{if } n = 1, \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + n & \text{else.} \end{cases} \tag{2.1}$$

Then, $f(n) = \Theta(n \log n)$.

Proof. First, consider the case that $n = 2^k$ for some $k \in \mathbb{N}$. By telescoping, we obtain

$$f(n) = 2 \cdot f(n/2) + n = 2 \cdot 2 \cdot f(n/4) + n + n = \cdots = 2^{\log_2 n} + n \cdot \log_2 n = n \cdot (1 + \log_2 n). \tag{2.2}$$

We can prove this by induction: Equation (2.2) trivially holds for $n = 1$, since $f(1) = 1(1 + 0) = 1$. Inductively, if (2.2) holds for $n/2$, we obtain

$$\begin{aligned} f(n) &= 2f(n/2) + n \\ &= 2 \cdot n/2 \cdot (1 + \log_2 n/2) + n \\ &= n \cdot (1 + \log_2 n), \end{aligned}$$

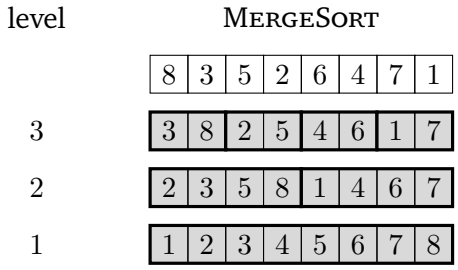


Figure 2.2: Illustration of MergeSort. The grey sublists are guaranteed to be sorted. The number of recursive levels to sort 8 numbers is $3 = \log_2 8$, as expected.

which proves (2.2).

Now, consider the remaining case that $2^{k-1} < n < 2^k$ for some $k = \lceil \log_2 n \rceil$. By (2.1), f is monotone and we have

$$f(n) < f(2^k) = 2^k \cdot (1 + k) = \mathcal{O}(n \log n),$$
$$f(n) > f(2^{k-1}) = 2^{k-1} \cdot k = \Omega(n \log n),$$

which implies $f(n) = \Theta(n \log n)$. □

With this motivation, *merge sort* is the most natural divide and conquer algorithm: We simply sort both halves of the list recursively, and then combine results in linear time (cf. Figure 2.2).

Algorithm: MergeSort(L)

input: list $L = (L_1, \ldots, L_n)$
output: sorted version of L

```
if n > 1:
    A ← MergeSort(L_{1,...,⌊n/2⌋})
    B ← MergeSort(L_{⌊n/2⌋+1,...,n})
    for i ← 1, ..., n:
        if A = ∅ or (A ≠ ∅ and B ≠ ∅ and B_1 < A_1):
            rename A, B to B, A
        L_i ← A_1; A ← A_{2,...,|A|}
return L
```

Theorem 2.6. MergeSort solves the sorting problem in time $\Theta(n \log n)$.

Proof. Correctness of MergeSort for lists of length n follows inductively if we assume correctness for lists of length up to $\lceil n/2 \rceil$.

To bound the running time, we need to bound the total number of iterations of the loop over all levels of the recursion for $n > 1$ and the number of times that MergeSort is invoked for $n = 1$. This number is given by equation (2.1), and thus application of Proposition (2.5) concludes the proof. □

3 Basic Graph Theory

Many algorithmic problems arise on structures formed by entities and their relationships, e.g., social and data networks, road maps and other infrastructures, processes and dependencies. We introduce the abstract notion of a *graph* to formally capture the underlying structure of these settings, independently of problem specific features.

Definition 3.1. A *graph* is a pair $G = (V, E)$ of a set of *vertices* V and a set of *edges* E with

- $E \subseteq \{\{u, v\} \subseteq V \mid u \neq v\}$ if G is *undirected*, and
- $E \subseteq \left\{(u, v) \in V^2 \middle| u \neq v\right\}$ if G is *directed*.

The edges of a directed graph are called *arcs*. In order to unify the following definitions, we write $[u, v] \equiv \{u, v\}$ for undirected and $[u, v] \equiv (u, v)$ for directed graphs.

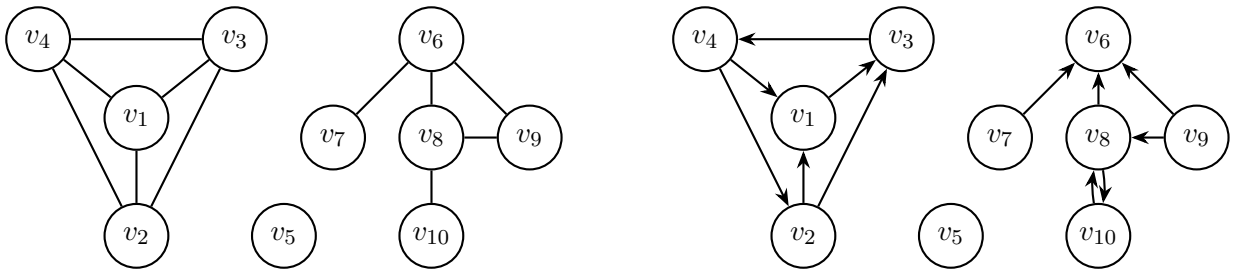
Remark 3.2. Our definition of graphs demands that there is only at most one edge between every pair of vertices, and that there cannot be an edge between a vertex and itself. This significantly simplifies our notation. Definitions that relax these requirements are usually called *multigraphs* and *graphs with self-loops* in the literature.

Example. An instance of the celebrity problem can be interpreted as a directed graph $G = (V, E)$ with vertices $V = P$ and arcs $E = \left\{(p, p') \in V^2 \middle| p' \in P_p\right\}$.

We can visualize graphs very naturally by drawing them in the plane, such that vertices correspond to points in $\mathbb{R}^2$ and edges are lines or arrows between the corresponding points. We may label vertices or edges with additional information associated with the graph. For example, for $V = \{v_1, \ldots, v_{10}\}$ and

$$E_1 = \{\{v_2, v_1\}, \{v_1, v_3\}, \{v_4, v_1\}, \{v_2, v_3\}, \{v_4, v_2\}, \{v_3, v_4\}, \{v_7, v_6\}, \{v_8, v_6\}, \{v_8, v_9\}, \{v_9, v_6\}, \{v_8, v_{10}\}\},$$
$$E_2 = \{(v_2, v_1), (v_1, v_3), (v_4, v_1), (v_2, v_3), (v_4, v_2), (v_3, v_4), (v_7, v_6), (v_8, v_6), (v_9, v_6), (v_8, v_{10}), (v_9, v_8), (v_{10}, v_8)\},$$

the undirected graph $G_1 = (V, E_1)$ and the directed graph $G_2 = (V, E_2)$ can be drawn as follows:



Note that such a drawing is not unique, and not all graphs can be embedded in the plane without edges crossing. Graphs for which this is possible are called *planar*.

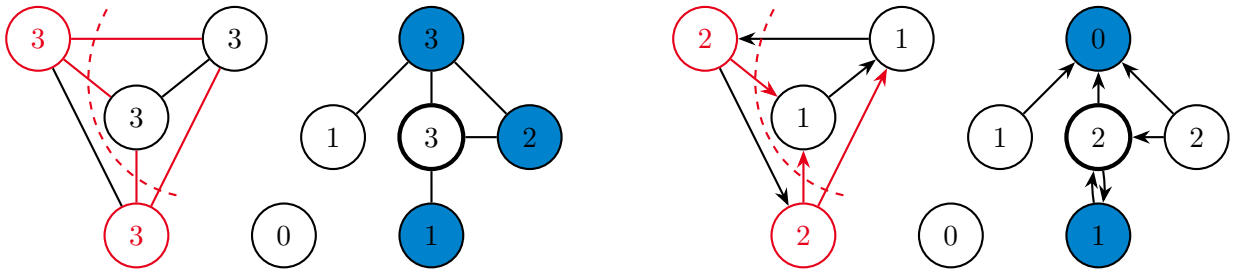


Figure 3.1: Illustration of Definition 3.4 for an undirected (left) and a directed graph (right). Vertex labels specify degrees. The neighborhood of the highlighted vertex is indicated in blue. Edges of the dashed cut induced by the red vertices are highlighted.

For convenience, we often identify graphs with their set of vertices or edges, in particular in conjunction with set notation. In particular, for a graph $G = (V, E)$, we write $G \pm v$ as a shorthand for $(V \pm \{v\}, E)$ and $G \pm e$ for $(V, E \pm \{e\})$. In addition, we treat tuples (in particular, arcs) like sets of size two when applying set operations. The following definition provides two different notions of subgraphs.

Definition 3.3. A graph $G' = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$, and we write $G' \subseteq G$. The subgraph *induced* by $V' \subseteq V$ is $G[V'] := (V', \{\, e \in E \mid e \subseteq V' \,\})$.

We introduce further notation regarding the relationships between (sets of) vertices and edges in a graph (cf. Figure 3.1).

Definition 3.4. Let $G = (V, E)$ be a graph.

- The *neighborhood* of $u \in V$ is $\Gamma_G(u) := \{\, v \in V \mid [u, v] \in E \,\}$, the neighborhood of $U \subseteq V$ is $\Gamma_G(U) := \bigcup_{u \in U} \Gamma_G(u) \setminus U$. We say that $v \in V$ is *adjacent* to $u \in V$ if $v \in \Gamma_G(u)$.
- The set of edges *incident* to $u \in V$ is $\delta_G(u) := \{\, [u, v] \in E \mid v \in V \,\}$, and the set of edges incident to $U \subseteq V$ is $\delta_G(U) := \{\, [u, v] \in E \mid u \in U, v \in V \setminus U \,\}$. For directed graphs, we write $\delta^+_G(U) := \delta_G(U)$ and $\delta^-_G(U) := \delta_G(V \setminus U)$ to distinguish outgoing and incoming arcs.
- The *degree* of $u \in V$ is $\deg_G(u) := |\Gamma_G(u)|$.
- A *cut* is a set of edges $S \subseteq E$ for which $\emptyset \neq U \subset V$ exists, such that $S = \delta_G(U)$. We say that U *induces* S.

In the above notations, we omit the index whenever the graph is clear from the context.

One of the central structural features of a graph is encoded in the way that local relationships extend into global connections between vertices. These connections are captured by the following definition (cf. Figure 3.2).

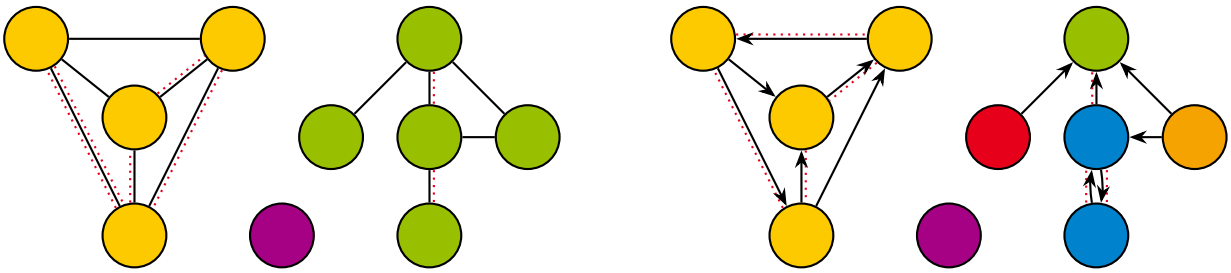


Figure 3.2: Illustration of connectivity in undirected (left) and directed (right) graphs. Vertex colors indicate connected components. Dotted edges form a closed walk, a path, a cycle, and a walk (from left to right).

Definition 3.5. A *walk* from v_0 to v_ℓ in a graph $G = (V, E)$ is a sequence

$$W = (v_0, e_1, v_1, e_2, v_2, \dots, e_\ell, v_\ell)$$

with $e_i = [v_{i-1}, v_i] \in E$ for $i \in \{1, \dots, \ell\}$. We often identify a walk with the sequence of its edges or vertices, or with the subgraph $(\bigcup_{i=0}^{\ell} \{v_i\}, \bigcup_{i=1}^{\ell} \{e_i\})$. We say

- W has *length* $|W| := \ell$,
- $v_1, \dots, v_{\ell-1}$ are the *internal* vertices of W,
- W is *closed* if $v_0 = v_\ell$,
- W is a *(v_0-v_ℓ-)path* if all vertices of W are distinct,
- W is a *cycle* if it is closed, has length $\ell \geq 1$, and no vertex or edge appears more than once in $(e_1, v_1, e_2, \dots, e_\ell, v_\ell)$. We call G *acyclic* if it does not contain any cycles.

We note the following important relationship between walks and cuts in a graph.

Observation 3.6. Let $G = (V, E)$ be a graph and let $U \subset V$. Then, every walk from $u \in U$ to $v \in V \setminus U$ in G contains an edge of $\delta(U)$.

We capture the intuitive fact that a walk is nothing else than a path with detours. Note that, in directed graphs, closed walks of length two are cycles as well, but this is not the case in undirected graphs.

Definition 3.7. For two walks $W = (u_0, \dots, u_\ell), W' = (v_0, \dots, v_{\ell'})$ with $u_\ell = v_0$, we write $W \oplus W' := (u_0, \dots, u_\ell = v_0, \dots, v_{\ell'})$. If C is a closed walk starting (and ending) at $u_\ell = v_0$, we say that $W \oplus C \oplus W'$ is a *union* of $(W \oplus W')$ and C.

Proposition 3.8. Let W be a walk from $u \in V$ to $v \in V$ in a graph $G = (V, E)$. Then, W is a union of a u-v-path and a set of cycles and of closed walks of length two.

Proof. We proceed by induction over the length ℓ of the walk W. For $\ell \leq 1$, the claim holds since W must be a path. Now consider a walk $W = (v_0, e_1, \dots, e_\ell, v_\ell)$ with $\ell \geq 2$ and assume that the claim holds for walks

of length at most $\ell - 1$. If W is a path, a cycle, or a closed walk of length two, the claim trivially holds. Otherwise, we must have $\ell > 2$ and we can choose $i \in \{0, \dots, \ell - 1\}$ and $j \in \{i + 1, \dots, \ell\}$ such that $v_i = v_j$ and $(i, j) \neq (0, \ell)$. By induction, $(v_i, e_{i+1}, \dots, v_j = v_i)$ must be the union of the (trivial) v_i-v_j-path (v_i) and a set C of cycles and closed walks of length two. Also, the walk $(v_0, e_1, \dots, v_i, e_{j+1}, \dots, v_\ell)$ must be the union of a u-v-path p and a set C' of cycles and closed walks of length two. But then, W is the union of the u-v-path p and a set $C \cup C'$ of cycles and closed walks of length two. □

Using the notion of walks, we can make the global connectivity of vertices more precise (cf. Figure 3.2).

Definition 3.9. A vertex $v \in V$ is *reachable* from a vertex $u \in V$ in a graph $G = (V, E)$ if there exists a path from u to v. The graph G is *(strongly) connected* if every vertex $v \in V$ is reachable from every other vertex $u \in V$. The *connected component* of $u \in V$ is the inclusion maximal set $C_u \subseteq V$ such that $u \in C_u$ and $G[C_u]$ is strongly connected, i.e., $C_u := \{v \in V \mid u, v \text{ are reachable from each other}\}$.

Observation 3.10. Every connected component C of an undirected graph induces an empty cut $\delta(C) = \emptyset$. This is not true for directed graphs.

We often encounter graphs that have all possible edges, which justifies the following terminology.

Definition 3.11. The *complement* of a graph $G = (V, E)$ is the graph $\bar{G} = (V, \bar{E})$ with $\bar{E} = \{[u, v] \notin E \mid u, v \in V\}$. We say that G is *complete* if it is the complement of the empty graph, i.e., if $\bar{G} = (V, \emptyset)$.

3.1 Graph Traversal

We now turn to our first basic graph algorithms. In order to efficiently operate on the relational structure defined by a graph, it makes sense to assume that we can efficiently enumerate the edges incident to a given vertex. More specifically, we assume that accessing all $\deg(v)$ edges incident to a vertex v requires $\deg(v)$ elementary operations.

This assumption allows us to efficiently traverse a graph, i.e., to systematically visit all of its vertices by using edge-traversals. Graph traversals lie at the heart of many algorithms operating on graphs. For example, the problems of deciding whether a graph is connected or of finding a specific vertex of a graph (e.g., the exit of a maze) can be reduced to traversing the graph systematically.

There are two obvious approaches for graph traversal: depth-first search and breadth-first search. Depth-first search (DFS) pursues a single path for as long as possible and only backtracks when there are no more unvisited vertices in the neighborhood. Intuitively, this is the algorithm by which most people would explore a maze if we allow them to leave markings on the walls. Breadth-first search (BFS) visits a vertex closest to the starting point in each step. Intuitively, this is the algorithm most people would use when exploring a maze as a team. Figure 3.3 gives an example for both algorithms.

Observe that the following formulations of both algorithms are extremely similar, and only differ in their usage of the lists of vertices S and Q whose edges have not been explored yet. In DFS we always extract the vertex from S that has been inserted most recently, while in BFS we extract the vertex that has been inserted the earliest. We assume that both these operations are elementary. A data structure that efficiently supports the last-in-first-out (LIFO) scheme of DFS is called a *stack*, a data structure that supports first-in-first-out (FIFO) scheme of BFS is called a *queue*.

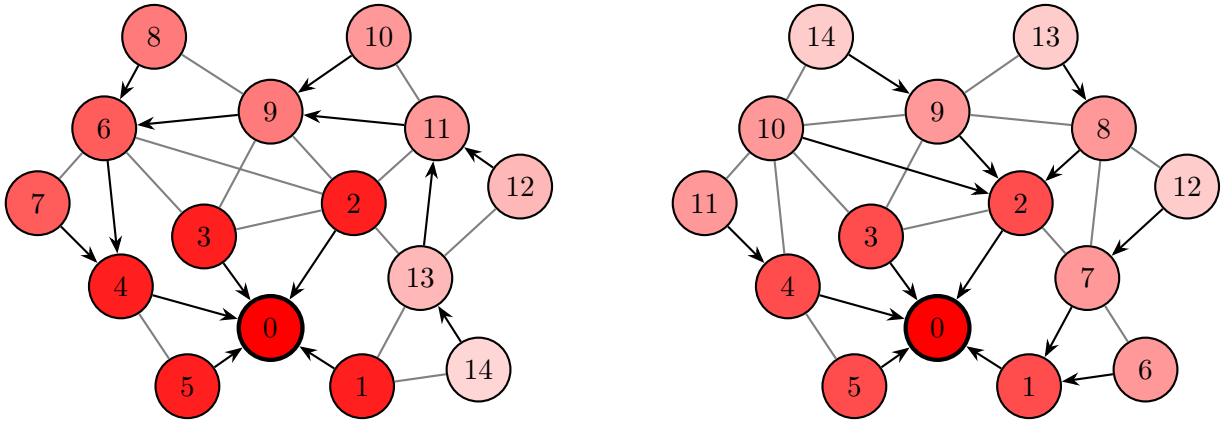


Figure 3.3: Execution of DFS (left) and BFS (right) on an undirected graph. Numbers and colors specify the order in which vertices are visited; arrows indicate parents. Note that the outcome of the algorithms depends on the ordering of the neighborhoods.

Algorithm: DFS(G, r)

input: graph $G = (V, E)$, vertex $r \in V$
output: parents $(p_v \in V)_{v \in V}$,
depths $(d_v \in \mathbb{N})_{v \in V}$

$S \leftarrow (r)$
$p_r \leftarrow r$; $d_r \leftarrow 0$
visit r
while $S \neq \emptyset$**:**
  $u \leftarrow S_1$; $S \leftarrow S_{2,\dots,|S|}$
  for each $v \in \Gamma(u)$**:**
    if v **not visited:**
      $S \leftarrow (v) \oplus S$
      $p_v \leftarrow u$; $d_v \leftarrow d_u + 1$
      visit v
return $(p_v)_{v \in V}$, $(d_v)_{v \in V}$

Algorithm: BFS(G, r)

input: graph $G = (V, E)$, vertex $r \in V$
output: parents $(p_v \in V)_{v \in V}$,
depths $(d_v \in \mathbb{N})_{v \in V}$

$Q \leftarrow (r)$
$p_r \leftarrow r$; $d_r \leftarrow 0$
visit r
while $Q \neq \emptyset$**:**
  $u \leftarrow Q_1$; $Q \leftarrow Q_{2,\dots,|Q|}$
  for each $v \in \Gamma(u)$**:**
    if v **not visited:**
      $Q \leftarrow Q \oplus (v)$
      $p_v \leftarrow u$; $d_v \leftarrow d_u + 1$
      visit v
return $(p_v)_{v \in V}$, $(d_v)_{v \in V}$

Algorithm: DFS-Traversal(G)

input: graph $G = (V, E)$

for each $v \in V$**:**
  if v **not visited:**
    DFS(G, v)

Algorithm: BFS-Traversal(G)

input: graph $G = (V, E)$

for each $v \in V$**:**
  if v **not visited:**
    BFS(G, v)

We first observe that both DFS and BFS behave as intended.

Lemma 3.12. DFS(G, r) and BFS(G, r) visit every vertex of G reachable from r exactly once.

Proof. First observe that every vertex is visited at most once, since only unvisited vertices are ever inserted

into S or Q and they are immediately marked as visited after being inserted.

It remains to show that, for every $\ell \in \mathbb{N}$, every vertex of every r-v-path p of length ℓ in G is eventually visited. We show this by induction over ℓ. For $\ell = 0$, the only possible path is $p = (r)$, and the statement is trivial. Now assume the statement holds for paths of length $\ell - 1$ and consider any path $p = (v_0 = r, \dots v_{\ell-1}, v_\ell)$ in G. By induction, $v_{\ell-1}$ is eventually visited, which means that it gets added to S or Q. Since the algorithm only terminates once S or Q is empty, $v_{\ell-1}$ will be extracted from S or Q at some point. Since $v_\ell \in \Gamma(v_{\ell-1})$, either v_ℓ is already visited at this point, or it will be visited. □

In addition, we establish the meaning of the parents and depths computed by DFS and BFS.

Proposition 3.13. Let $(p_v)_{v \in V}$ and $(d_v)_{v \in V}$ be the parents and depths computed by DFS(G, r) or BFS(G, r). Then, for every vertex v reachable from r in G, the sequence $(v_0 = r, v_1, \dots, v_{d_v} = v)$ with $v_{i-1} = p_{v_i}$ for $i \in \{1, \dots, d_v\}$ is an r-v-path.

Proof. We prove the statement for all vertices v reachable from r by induction over d_v. For $d_v = 0$ we have $v = r$ and the statement is trivial. Now consider a vertex $v \in V$ with $d_v \geq 1$ and assume that the statement holds for all $v' \in V$ with $d_{v'} < d_v$. By Lemma 3.12, every vertex reachable from r is visited exactly once, hence the parents and depths do not change once they are set by DFS or BFS. Since d_v is set to $d_{p_v} + 1$, we thus know that $d_{p_v} < d_v$ and hence, by induction, $(v_0 = r, \dots, v_{d_v - 1} = p_v)$ with $v_{i-1} = p_{v_i}$ for $i \in \{1, \dots, d_v - 1\}$ is an r-p_v-path. In addition, every vertex v_i of this path has $d_{v_i} = i$ by induction, which means that the path cannot contain v. But then $(v_0, \dots, v_{d_v - 1}, v_{d_v} = v)$ with $v_{i-1} = p_{v_i}$ for $i \in \{1, \dots, d_v\}$ is a path, as claimed. □

Before we can bound the running times of DFS and BFS, we have to specify a definition of the input size and what we consider to be elementary operations. Obviously, the number of objects involved in the description of a graph $G = (V, E)$ is $\Theta(n + m)$, where $n := |V|$ and $m := |E|$. In many applications it makes sense to account for the contributions of n and m to the running time separately, since the costs of visiting vertices and traversing edges may differ significantly (e.g., crossing an intersection vs. traveling along a segment of a highway). From now on, whenever we are dealing with an algorithmic problem that has a graph as its input, we will therefore follow the convention that running times are expressed as functions of n and m. Throughout, we will use n and m to denote the number of vertices and edges of the input graph, respectively. In terms of elementary operations, we assume that, given $u \in V$, we have direct access to the sets $\delta(u)$ and $\Gamma(u)$, i.e., in particular, we have access to $\deg(u)$ and can iterate over all $e \in \delta(u)$ and $v \in \Gamma(u)$ in time $\Theta(\deg(u))$.

It is now easy to see that both DFS and BFS can be used to solve the connectivity problem on undirected graphs as efficiently as possible.

Proposition 3.14. DFS or BFS can be used to determine whether an undirected graph is connected in time $\Theta(n + m)$.

Proof. To solve the connectivity problem for a given undirected graph $G = (V, E)$, we simply invoke DFS or BFS on any vertex $v \in V$ and check afterwards whether all vertices have been visited. This algorithm is correct by Lemma 3.12, since every reachable vertex is visited. The lemma also guarantees that every vertex is visited at most once, which means that the inner loop iterates at most $\sum_{v \in V} \deg(v) = 2m$ times in total. Overall, the running time is bounded by $\mathcal{O}(n + m)$. To prove the matching lower bound, observe that we may need to inspect all vertices and edges in order to conclude that a graph is not connected. □

The algorithm described in the proof of Proposition 3.14 can be repeated on unvisited vertices in order to visit all vertices of the graph. The corresponding algorithms are called DFS-Traversal and BFS-Traversal. For

example, these algorithms allow us to determine all connected components of an undirected graph, if we keep track of the sets of vertices that are visited in each invocation of DFS/BFS.

Observe that Proposition 3.14 does not hold for directed graphs, nor does its application to determine all strongly connected components. However, it is still true that DFS-Traversal and BFS-Traversal systematically visit all vertices, even in directed graphs.

Theorem 3.15. DFS-Traversal and BFS-Traversal visit every vertex exactly once and have running time $\Theta(n+m)$.

Proof. The statement follows from the fact that vertices are inserted exactly once into S or Q and that $\sum_{v \in V} \deg(v) \leq 2m$ (m for directed graphs), as in the proofs of Lemma 3.12 and Proposition 3.14. □

Proposition 3.16. DFS-Traversal or BFS-Traversal can be used to determine whether an undirected graph is acyclic in time $\Theta(n+m)$.

Proof. We simply execute either algorithm and check whether we ever encounter a vertex $v \in \Gamma(u) \setminus \{p_u\}$ that is already visited during DFS or BFS. The running time follows from Theorem 3.15. □

So what are the advantages / disadvantages of DFS and BFS, or when should we prefer one over the other? Both algorithms have many applications where the specific order in which they visit vertices are important (see Theorems 3.18 and 3.25 below). In terms of pure graph traversal, DFS has the advantage that it can be implemented very elegantly using recursion (note that the traversal order of this recursive implementation differs from the iterative implementation above, see Figure 3.4):

Algorithm: DFS′(G, u)

input: graph $G = (V, E)$, vertex $u \in V$

visit u

for each $v \in \Gamma(u)$**:**
  if v **not visited:**
    $p_v \leftarrow u$; $d_v \leftarrow d_u + 1$
    DFS′(G, v)

In addition, DFS often profits from "locality", i.e., the fact that vertices that are visited consecutively tend to be close to each other. This not only is an advantage when implementing DFS in a physical system where we actually have to travel through a graph-like environment, but also for implementations on computing systems that benefit from memory locality, i.e., architectures with a memory hierarchy (e.g., hard drive, main memory, caches, registers) where memory is transferred block-wise from level to level. On the other hand, BFS can be parallelized more easily. This is important in physical implementations where we have multiple agents traversing the environment, or for implementations on multi-core systems.

A crucial feature of the order in which BFS visits vertices lies in the fact that it prefers short paths and that each vertex gets its smallest possible depth assigned to it. To make this precise, we introduce the following notion.

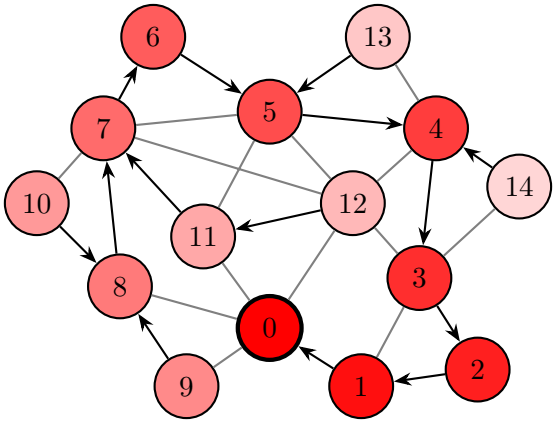


Figure 3.4: Execution of DFS′ on the example of Figure 3.3. Again, numbers and colors specify the order in which vertices are visited, arrows indicate parents, and the outcome depends on the ordering of the neighborhoods.

Definition 3.17. The *distance* $d(u, v) \in \mathbb{N} \cup \{\infty\}$ between vertices u and v of a graph G is the shortest length among all u-v-paths in G, or ∞ if no such path exists. An *(unweighted) shortest path* in G is a u-v-path P with $|P| = d(u, v)$.

With this, we can show that BFS computes shortest paths to all reachable vertices. Observe that this is not true for DFS.

Theorem 3.18. BFS(G, r) computes shortest paths from r to all vertices v reachable from r in G in time $\Theta(m)$.

Proof. Using Proposition 3.13, it remains to show that the depths computed by BFS satisfy $d_v = d(r, v)$ for all vertices v reachable from r in G. We show this claim by induction on $d(r, v)$, and, additionally, that v is added to Q before any vertex v' with $d(r, v') > d(r, v)$. The case $d(r, v) = 0$ is trivial, since it implies $v = r$. Now consider $v \in V$ with $d(r, v) \geq 1$ and assume that both claims hold for all vertices of distance at most $d(r, v) - 1$ from r. By Lemma 3.12, every vertex reachable from r is visited exactly once, hence the parents and depths do not change once they are set by BFS. Let $(r, \ldots, u, v)$ be any r-v-path of length $d(r, v)$ in G. Then, $d(r, u) \leq d(r, v) - 1$ (even equality holds) and, by induction, $d_u = d(r, u) < d(r, v)$. Consider the iteration of the outer loop in which p_v is extracted from Q.

At this point, either $p_v = u$, or u was not yet extracted, since otherwise v would already have been visited. Since Q is first-in-first-out, this means p_v was added to Q no later than u. Hence, by induction, $d(r, p_v) \leq d(r, u)$ and $d_{p_v} = d(r, p_v)$, which means that BFS sets $d_v = d_{p_v} + 1 = d(r, p_v) + 1 \leq d(r, u) + 1 \leq d(r, v)$. Because $v \in \Gamma(p_v)$, we also have $d(r, v) \leq d(r, p_v) + 1 = d_v$, thus equality holds.

Since, by induction, p_v was added to Q before any vertex v' with $d(r, v') > d(r, p_v)$, and Q is first-in-first-out, we know that only vertices v' with $d(r, v') \leq d(r, p_v) + 1 = d(r, v)$ can have been added to Q until p_v is extracted and v is added.

The running time follows from the fact that exactly those vertices reachable from r are added to Q exactly once, by Lemma 3.12. This implies that the number of iterations of the inner loop is at most $\sum_{v \in V} \deg(v) = \mathcal{O}(m)$, and at least $\Omega(m)$ for connected graphs. □

3.2 Trees

A particularly important class of graphs are *trees*. Abstractly, trees capture hierarchical structures and branching processes.

Definition 3.19. A *forest* is an acyclic, undirected graph. A *tree* is a connected forest. The vertices of a forest are called *nodes* and nodes of degree at most one are called *leaves*.

Trees naturally arise in many contexts. They are extremal graphs in multiple senses. For example they are sparsest possible graphs in that they have the fewest number of edges and paths possible for a connected graph. The following theorem gives various characterizations of trees.

Theorem 3.20. Let $G = (V, E)$ be an undirected graph with $n \geq 1$ vertices and m edges. The following are equivalent:

(i) G is a tree.

(ii) G is acyclic and $m = n - 1$.

(iii) G is connected and $m = n - 1$.

(iv) G is connected and $G - e$ is not for every $e \in E$.

(v) G is acyclic and $G + e$ is not for every $e \in \binom{V}{2} \setminus E$.

(vi) G contains a unique u-v-path for all $u, v \in V$.

For inductive proofs over trees, the following property is essential. It allows to remove a leaf to apply induction for the resulting subtree.

Proposition 3.21. Every tree with n nodes has at least $\min\{n, 2\}$ leaves.

Proof. We use induction over n. The statement is trivial for $n = 1$. Now consider any tree $T = (V, E)$ with $n \geq 2$ and assume the statement holds for trees with fewer than n nodes. By Theorem 3.20, removing any edge $e \in E$ yields a graph with two connected components C_1 and C_2. Then, $T[C_1]$ and $T[C_2]$ are still trees and, by induction, they have $\min\{|C_1|, 2\}$ and $\min\{|C_2|, 2\}$ leaves. This means that at least one node of C_1 and one node of C_2 must be leaves of T. □

While trees are undirected by definition, they often describe hierarchical structures where edges correspond to asymmetrical relationships between a predecessor and one or many successors. Rather than orienting the edges of such a tree, it is sufficient to specify a *root* node at the top of the hierarchy to implicitly assign an orientation to all edges (cf. Figure 3.5).

Definition 3.22. A *rooted tree* is a tree $T = (V, E)$ together with a distinguished *root* $r \in V$. The *depth* of a node $v \in V$ is $d(r, v)$ and the *height* of T is $\max_{v \in V} d(r, v)$. The *parent* p_v of a node $v \in V \setminus \{r\}$ is the unique node in $\Gamma(v)$ with $d(r, p_v) < d(r, v)$, and the *children* of $u \in V$ are the nodes in $\{v \in V \mid p_v = u\}$. The *subtree* T_v of T rooted at $v \in V$ is the subgraph of T induced by the set of vertices whose unique paths to the root contain v.

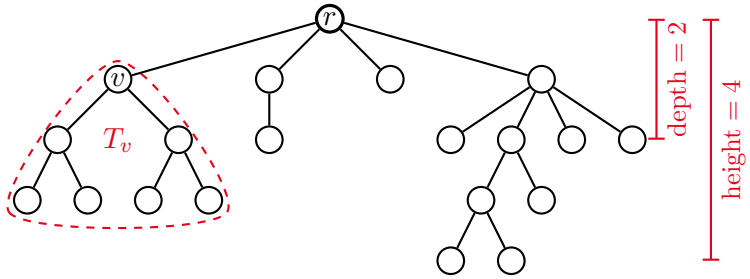


Figure 3.5: Illustration for Definition 3.22.

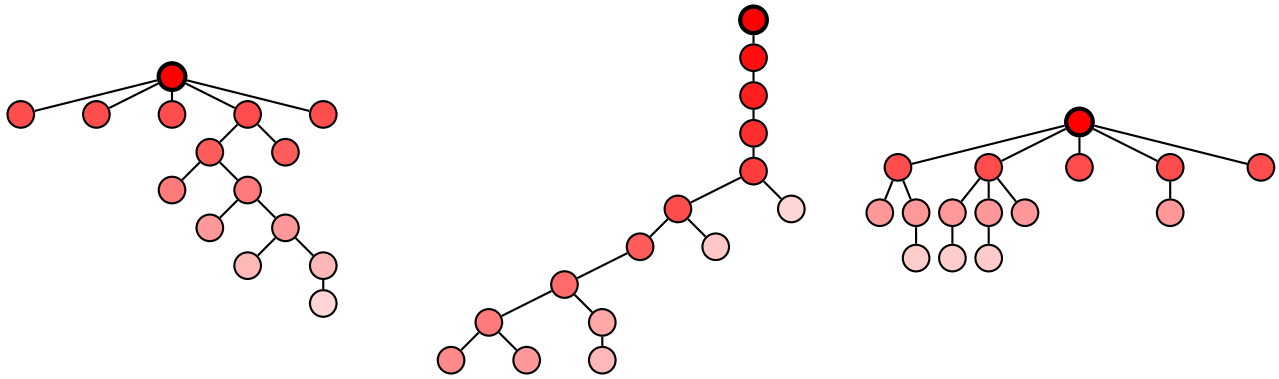


Figure 3.6: The DFS-, DFS′- and BFS-trees for the example of Figures 3.3 and 3.4.

For example, DFS and BFS naturally induce a rooted tree in the following sense (see Figure 3.6).

Proposition 3.23. The vertices p_v computed by DFS(G, r) (or DFS′(G, r)) and BFS(G, r) uniquely determine trees $(V, \{\{v, p_v\} | \, v \in V \setminus \{r\}\})$ rooted at r that contains all vertices reachable from r. These trees are called *DFS-tree* resp. *BFS-tree* of G rooted at r.

Proof. Without loss of generality, assume that all vertices are reachable from r. Then, all vertices $v \in V \setminus \{r\}$ have $p_v \neq v$ after execution of DFS/BFS. We need to show that the undirected graph $T = (V, E_T)$ with $E_T = \{\{v, p_v\} | \, v \in V \setminus \{r\}\}$ is a tree. We have $|E_T| = n - 1$, thus, by Theorem 3.20 it suffices to show that T is acyclic.

Let $(d_v)_{v \in V}$ be the depths computed by DFS or BFS. Observe that every edge $\{u, v\} \in E_T$ satisfies $|d_u - d_v| = 1$ by definition of either algorithm, i.e., $d_u \neq d_v$. Furthermore, every node v of T has at most one neighbor $u = p_v$ in T with $d_u < d_v$. For the sake of contradiction, assume that there was a cycle C in T and let v be the vertex of C that maximizes d_v. But then the two neighbors of v in C must have smaller depth, which is a contradiction. □

With this, we are ready to give an example where the order in which depth-first search visits vertices is crucial. We start by relating strong connectivity to the order in which vertices are visited by DFS'. Note that the following lemma holds just the same for DFS and BFS.

Lemma 3.24. Let $G = (V, E)$ be a directed graph and $r \in V$. Then, G is strongly connected if and only if DFS'(G, r) visits all vertices and from every vertex $v \in V \setminus \{r\}$ some vertex v' can be reached that was visited before v by DFS'.

Proof. First observe that reachability is transitive by Proposition 3.8. This means that G is strongly connected if and only if every vertex can be reached from r, and r can be reached from every vertex.

$\Rightarrow$: If G is strongly connected, by the above observation and since Lemma 3.12 applies to DFS' as well, DFS' visits all vertices of G. Also, since we can reach r from every vertex and r is visited first, the second part of the claim holds.

$\Leftarrow$: Note that if DFS' visits all vertices of G, then every vertex can be reached from r (Proposition 3.13). It remains to show that r can be reached from every vertex. Consider the vertices in the order $v_1, \dots, v_n$ in which they are visited. We show by induction on i that r can be reached from v_i. This trivially holds for $i = 1$ since $v_1 = r$. For $i \geq 2$, we know that there is an index $j < i$ such that v_j is reachable from v_i. By induction, r can be reached from v_j. By transitivity, r can be reached from v_i. □

We can turn this lemma into an efficient algorithm to check whether a directed graph is strongly connected. Crucially, we will see that DFS' traversal order allows to keep track of the reachable vertices visited earliest whenever a vertex is being visited.

Theorem 3.25. The problem of determining whether a directed graph is strongly connected can be solved in time $\Theta(n + m)$.

Proof. Consider an execution of DFS′(G, r) on a directed graph $G = (V, E)$ for $r \in V$ and let T denote the resulting DFS-tree. We define two times for each vertex u: The time t_u is the position of u in the order that vertices are visited by DFS′, and t_u^{reach} is the smallest time t_v among all vertices $v \in V_u \cup \Gamma_G(V_u)$, where $T_u = (V_u, E_u)$ denotes the subtree of T rooted at u. We can adapt DFS′ to compute these times as follows.

Algorithm: StrongConnection(G, u)

input: graph $G = (V, E)$, vertex $u \in V$, global time $\mathbf{t}$

```
t_u ← t; t ← t + 1
visit u
t_u^reach ← t_u
for each v ∈ Γ(u):
    if v not visited:
        StrongConnection(G, v)
        t_u^reach ← min{t_u^reach, t_v^reach}
    else
        t_u^reach ← min{t_u^reach, t_v}
```

Clearly, if t_v^{reach} is correctly computed for all $v \in V$ with $t_v > t_u$, then t_u^{reach} is correctly computed as well. Hence, by induction, StrongConnection correctly computes t_u^{reach} for all $u \in V$. Note that it is crucial for this inductive argument that we use the recursive implementation DFS', and that it is not clear how to compute t_u^{reach} using BFS.

Now, observe that all vertices in $\Gamma(V_u)$ are visited before u, since T_u contains all vertices reachable from u that are not visited before u. This means that $t_u^{\text{reach}} < t_u$ if and only if from u we can reach a vertex visited before u. By Lemma 3.24, we only need to check whether all vertices are visited and $t_u^{\text{reach}} < t_u$ for all $u \in V \setminus \{r\}$ to decide whether a graph G is strongly connected. Overall, the computation takes time $\Theta(n + m)$, which

is best possible, since we may need to inspect every vertex and every arc to conclude that G is not strongly connected. □

Finally, we illustrate the usefulness of trees when carrying out formal arguments, by showing that mergesort has a best-possible running time. As an exercise, try to carry out the following proof without using the notion of a tree.

Theorem 3.26. Every sorting algorithm has running time $\Omega(n \log n)$.

Proof. Consider the rooted decision tree of any search algorithm for a list L of mutually distinct entries. Every node of the tree corresponds to a comparison during the course of the algorithm, and every path from the root to a possible sequence of comparisons. The result of the algorithm, i.e., the resulting permutation of L, can only depend on the outcomes of the comparisons along the corresponding path. Since there are $n!$ different total orders of the elements in L that must yield different permutations in the end, the decision tree must have at least $n!$ leaves. The number of comparisons that the algorithm needs in the worst case is equal to the height h of the tree. By definition, every node of the tree has at most two children, hence the number of leaves of the tree is at most 2^h. To allow for $n!$ leaves, the tree thus needs height

$$h \geq \log_2(n!) = \log_2(\prod_{i=1}^{n} i) = \sum_{i=1}^{n} \log_2(i) \geq \frac{n}{2} \log_2\left(\frac{n}{2}\right) = \Omega(n \log n).$$ □

4 Minimum Spanning Trees

In this chapter we focus on the problem of connecting a set of vertices as cheaply as possible by choosing a subset of the available edges. Formally, we are looking for a spanning tree of an undirected graph.

Definition 4.1. A *spanning tree* of an undirected graph $G = (V, E)$ is a tree $T \subseteq G$ with set of nodes V.

Observe that we have already seen a way of computing such a spanning tree.

Proposition 4.2. If the input is an undirected, connected graph, DFS and BFS compute a spanning tree in time $\Theta(m)$.

Proof. This follows immediately from Lemma 3.12 and Proposition 3.23. □

Corollary 4.3. An undirected graph is connected if and only if it contains a spanning tree.

Proof. This follows by Proposition 4.2 and the fact that every graph that contains a spanning tree is connected. □

In many applications we are not just looking for some spanning tree of a graph, but the *best* spanning tree in terms of a cost function associated with the edges of a graph. For example, consider the problem of connecting all major cities of a country by optical high-speed data connections. The cost of connecting cities can vary significantly due to their relative geographic locations and other factors. We will often encounter settings where edges have costs/utilities associated with them, and we introduce the abstract notion of *weighted* graphs to model such settings.

Definition 4.4. An *(edge-)weighted* graph is a graph $G = (V, E)$ together with a function $c\colon E \to \mathbb{R}$. The *weight* of a set of edges $E' \subseteq E$ or the corresponding subgraph of G is given by $c(E') := \sum_{e \in E'} c(e)$.

We consider the following algorithmic problem.

Minimum Spanning Tree (MST) Problem	
input:	undirected, connected graph $G = (V, E)$; edge weights $c\colon E \to \mathbb{R}_{\geq 0}$
problem:	find spanning tree of minimum weight

A spanning tree of minimum weight is simply called *minimum spanning tree* from now on. The algorithms that we will see in this chapter rely on the following properties of minimum spanning trees (cf. Figure 4.1).

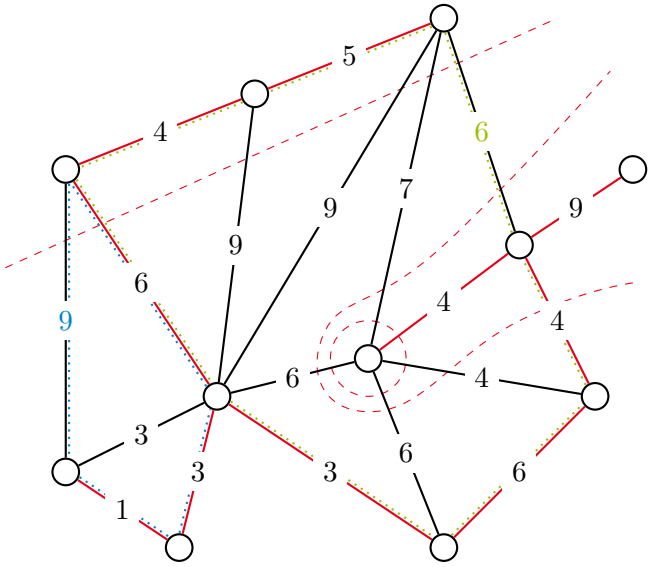


Figure 4.1: Example of a minimum spanning tree (red). Every edge of the tree induces a unique cut in the underlying graph, and is cheapest on this cut. Every edge not in the tree induces a unique cycle, and is most expensive on this cycle.

Observation 4.5. Let $G = (V, E)$ be an undirected, connected graph and $T = (V, E_T)$ be a minimum spanning tree of G with respect to edge weights $c\colon E \to \mathbb{R}_{\geq 0}$. Then, the following statements hold.

(i) Let $e \in E_T$ and let C be one of the two components of $T - e$ (Theorem 3.20 (iv)). Then $c(e) = \min_{e' \in \delta_G(C)} c(e')$, i.e., e has minimum weight in the (unique) cut $\delta_G(C)$ induced by e.

(ii) Let $e \in E \setminus E_T$ and let K be the (unique) cycle in $T + e$ (Theorem 3.20 (v)). Then $c(e) = \max_{e' \in K} c(e')$, i.e., e has maximum weight on the (unique) cycle K induced by e.

Proof. If there was an edge e' with $c(e') < c(e)$ in the cut induced by $e \in E_T$, we could obtain a spanning tree of smaller weight than T by replacing e by e'. Similarly, if there was an edge e' with $c(e') > c(e)$ on the cycle induced by $e \in E \setminus E_T$, we could obtain a spanning tree of smaller weight by replacing e' by e. Uniqueness of the induced cut and cycle follows by Proposition 3.20 (vi). □

So how can we find a tree with these properties? A natural approach to solve the MST problem is to consider edges one after the other and include or exclude edges from our solution as described below.[1]

[1] For simplicity, we use $\arg\min_{x \in X} f(x)$ to denote a (single) element $x \in X$ with $f(x) = \min_{x' \in X} f(x')$, and similarly for $\arg\max$. We use an arbitrary but fixed total order of X for tie-breaking in case $|\{x \in X | \min_{x' \in X} f(x')\}| > 1$.

Algorithm: MST-Scheme(G, c)

input: undirected, connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T \leftarrow (V, \emptyset)$; $G' \leftarrow (V, E)$
while $T \neq G'$**:**
  apply one of the following:
  rule 1**:** (add lightest edge over a cut)
    $C \leftarrow$ any component of T
    $e \leftarrow \arg\min_{e' \in \delta_{G'}(C)} c(e')$
    $T \leftarrow T + e$
  rule 2**:** (remove heaviest edge on a cycle)
    $K \leftarrow$ any cycle in G'
    $e \leftarrow \arg\max_{e' \in K \setminus T} c(e')$
    $G' \leftarrow G' - e$
return T

We show that every algorithm that proceeds according to the above scheme is guaranteed to eventually compute a minimum spanning tree.

Theorem 4.6. Every implementation of the MST-Scheme computes a minimum spanning tree.

Proof. We first argue that, as long as $T \neq G'$, at least one of the two rules can be applied. Observe that $T \subseteq G'$ throughout, since we only consider edges of G' in rule 1 and only edges outside of T in rule 2. If $T \neq G'$, we can therefore find $e = \{u, v\} \in G' \setminus T$. If u and v lie in different connected components $C_u \neq C_v$ of T, we can apply rule 1 with $C = C_u$, since $e \in \delta_{G'}(C_u) \neq \emptyset$. If u and v lie in the same connected component of T, then $T + e$ contains a cycle and we can apply rule 2.

We now prove that the scheme yields a minimum spanning tree. To that end, we show that it maintains the invariant that G contains a minimum spanning tree that uses all edges in T and no edge outside G'. Since we have $T = G'$ in the end, this concludes the proof.

Let $T^\star$ be the minimum spanning tree guaranteed by our invariant before an application of rule 1 for some connected component C of T and some $e = \{u, v\} \in \arg\min_{e' \in \delta_{G'}(C)} c(e')$. If $e \in T^\star$, the invariant is obviously maintained. Otherwise, the (unique) u-v-path P in $T^\star$ must use another edge $e^\star \in \delta_{G'}(C)$ (by Observation 3.6). By definition of C, we have $e^\star \notin T$ and, by our choice of e, we have $c(e) \leq c(e^\star)$. Additionally, u and v lie in different connected components of $T^\star - e^\star$, since P is unique by Theorem 3.20 (vi). Therefore the spanning tree $T^\star - e^\star + e$ satisfies our invariant.

Now let $T^\star$ be the minimum spanning tree guaranteed by our invariant before an application of rule 2 for come cycle K in G' and some $e = \{u, v\} \in \arg\max_{e' \in K \setminus T} c(e')$. If $e \notin T^\star$, the invariant is obviously maintained. Otherwise, u and v are in different components $C_u \neq C_v$ of $T^\star - e$ (by Theorem 3.20 (vi)). By Observation 3.6, the u-v-path $K - e$ must contain an edge $e' \in \delta_{G'}(C_u) \setminus \{e\}$. By definition of C_u, we have that $\delta_{T^\star}(C_u) = \{e\}$, thus $e' \notin T^\star$. By our choice of e, we have $c(e') \leq c(e)$ and $e \notin T$. Therefore the spanning tree $T^\star - e + e'$ satisfies our invariant. □

There are many ways of implementing the general scheme outlined above, differing by the order in which edges are considered. We will see two straight-forward approaches in the following.

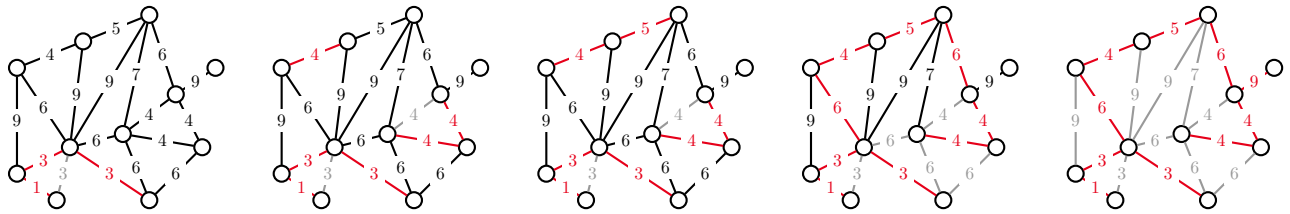


Figure 4.2: Some intermediate steps of an execution of KRUSKAL on the example of Figure 4.1. Red edges are included in T and grey edges are excluded from G', in terms of MST-SCHEME.

4.1 Kruskal's Algorithm

A natural order in which to consider the edges is by increasing weights. The edge with lowest weight can safely be included in the spanning tree, since it is a lightest edge on every cut it is part of. Every subsequent edge that does not close a cycle with the edges we already included in the spanning tree cannot be the unique edge of maximum weight on any cycle in the graph G' of the MST-SCHEME. This means that the scheme can never exclude this edge, so it must eventually be included. But then we can safely include it immediately when we consider it.

Analogously, we can consider edges by decreasing weights and discard edges that do not disconnect our solution, since these edges must be heaviest on a cycle. The two corresponding algorithms are listed below.

Algorithm: KRUSKAL(G, c)
input: undir., connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T \leftarrow (V, \emptyset)$
$(e_1, \dots, e_m) \leftarrow$ sort E s.t. $c(e_1) \leq c(e_2) \leq \dots$
for $e \leftarrow e_1, e_2, \dots, e_m$ **:**
  if $T + e$ *is acyclic* **:**
    $T \leftarrow T + e$ (rule 1)
  else
    discard e (rule 2)
return T

Algorithm: KRUSKAL-DUAL(G, c)
input: undir., connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T \leftarrow G$
$(e_1, \dots, e_m) \leftarrow$ sort E s.t. $c(e_1) \leq c(e_2) \leq \dots$
for $e \leftarrow e_m, \dots, e_2, e_1$ **:**
  if $T - e$ *is connected* **:**
    $T \leftarrow T - e$ (rule 2)
  else
    keep e (rule 1)
return T

An example for Kruskal's algorithm is given in Figure 4.2. Correctness of both algorithms immediately follows from Theorem 4.6. Their running times depend on a specific implementation.

Proposition 4.7. A naive implementation of KRUSKAL or KRUSKAL-DUAL computes a minimum spanning tree in time $\mathcal{O}(m^2)$.

Proof. The running time is dominated by sorting the edges and checking whether $T \pm e_i$ is acyclic/connected for $i \in \{1, \dots, m\}$. Sorting can be done in time $\mathcal{O}(m \log m)$ by using mergesort (Theorem 2.6). Observe that $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$ since $m < n^2$. Checking whether $T \pm e_i$ is acyclic/connected can be done in time $\mathcal{O}(n + m)$ using DFS or BFS (Propositions 3.14 and 3.16). Overall, this gives a running time of $\mathcal{O}(m \log n) + m \cdot \mathcal{O}(n + m) = \mathcal{O}(m^2)$, since $n \leq m + 1$ for connected graphs. □

We can improve this running time for KRUSKAL by storing which component every vertex belongs to. To do this, we define variables $(\mathrm{comp}_v \in V)_{v \in V}$ that are initially set to $\mathrm{comp}_v = v$ for all $v \in V$. This allows us to check in constant time whether $T + e$ is acyclic for $e = \{u, v\}$, simply by checking whether $\mathrm{comp}_u \neq \mathrm{comp}_v$. Whenever we add an edge $e = \{u, v\}$ to T, we can merge the components of u and v by going over all vertices and setting $\mathrm{comp}_{u'} \leftarrow \mathrm{comp}_u$ for all $u' \in \{v' \in V \mid \mathrm{comp}_{v'} = \mathrm{comp}_v\}$. Each such update takes linear time, and we can merge components at most $n - 1$ times until a single component remains. We obtain a total running time of $\mathcal{O}(m \log n + n^2)$.

We can further improve the implementation to achieve the best-possible running time.

Theorem 4.8. KRUSKAL can be implemented to find a minimum spanning tree in time $\Theta(m \log n)$.

Proof. Observe that the method to store components described above represents each component by a rooted tree of height one if we interpret the variables $(\mathrm{comp}_v \in V)_{v \in V}$ as the parents of each vertex. We can improve the running time of merging components if we allow trees to have larger heights. The idea is to balance the running time for merging components with the running time for checking whether two vertices share the same component, which essentially corresponds to the time needed to determine the component of a vertex. We begin by reformulating the algorithm:

Algorithm: KRUSKAL(G, c) (union-find)

input: undir., connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T \leftarrow (V, \emptyset)$
for $v \in V$**:**
  $\mathrm{comp}_v \leftarrow v$; $h_v \leftarrow 0$
$(e_1, \ldots, e_m) \leftarrow$ sort E s.t. $c(e_1) \leq c(e_2) \leq \ldots$
for $e = \{u, v\} \leftarrow e_1, e_2, \ldots, e_m$**:**
  $C_u \leftarrow$ FIND(u)
  $C_v \leftarrow$ FIND(v)
  if $C_u \neq C_v$**:**
    UNION(u, v)
    $T \leftarrow T + e$
return T

Algorithm: FIND(u)

while *comp*$_u \neq u$**:**
  $u \leftarrow \mathrm{comp}_u$
return u

Algorithm: UNION(u, v)

$u' \leftarrow$ FIND(u)
$v' \leftarrow$ FIND(v)
if $h_{u'} < h_{v'}$**:**
  $(u', v') \leftarrow (v', u')$
$\mathrm{comp}_{v'} \leftarrow u'$
$h_{u'} \leftarrow \max\{h_{u'}, h_{v'} + 1\}$

We can make the merge operation (UNION) cheaper by simply attaching the root of one of the trees to the other root. Both to determine whether two vertices are in the same component or to merge components then amounts to finding the roots of the corresponding trees (FIND). This needs time linear in the heights of the trees. To make the process efficient, we thus need a way to limit the heights of the trees representing components.

A simple trick to achieve this is to make sure that we always attach trees of smaller heights to trees of larger heights. If we do this, the height of a tree can only increase if it is merged with a tree of the same height. With this, a simple induction over the height shows that every tree of height h contains at least 2^h nodes. Since no tree can contain more than all vertices of G, we have $2^h \leq n$ and thus $h \leq \log_2 n$. The running time becomes $\mathcal{O}(m \log n) + m \cdot \mathcal{O}(\log n) = \mathcal{O}(m \log n)$, as claimed.

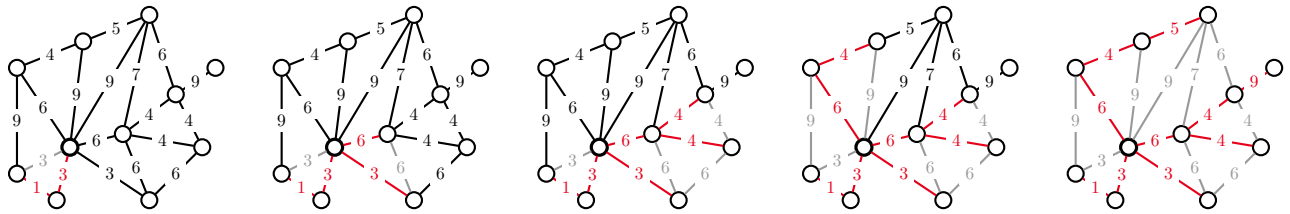


Figure 4.3: Some intermediate steps of an execution of PRIM on the example of Figure 4.1. Red edges are included in T and grey edges are excluded from G', in terms of MST-SCHEME.

The lower bound of $\Omega(m \log n)$ follows from Theorem 3.26 and $m \geq n - 1$ (since G is connected). □

Remark 4.9. A data structure that supports the operations described in the proof of Theorem 4.8 is called a *union-find* data structure.

4.2 Prim's Algorithm

While KRUSKAL takes the globally lightest edges first, we can take a local approach by growing a connected component starting with a single vertex. In every step, the connected component C induces a cut $\delta_G(C)$ and it is safe to include an edge of minimum weight on this cut and repeat with the same reasoning. The resulting algorithm is Prim's algorithm. If we simultaneously grow all connected components, we obtain Borůvka's algorithm.

Algorithm: PRIM(G, c)
input: undir., connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T = (V_T, E_T) \leftarrow (\{u\}, \emptyset)$ for any $u \in V$
while $V_T \neq V$ **:**
  $e \leftarrow \arg\min_{e' \in \delta_G(V_T)} c(e')$
  $T \leftarrow (V_T \cup e, E_T \cup \{e\})$ (rule 1)
  discard edges in $G[V_T] \setminus T$ (rule 2)
return T

Algorithm: BORŮVKA(G, c)
input: undir., connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

$T \leftarrow (V, \emptyset)$
while T *not connected* **:**
  $\mathcal{C} \leftarrow$ connected components of T
  $E_\delta \leftarrow \{\arg\min_{e' \in \delta_G(C)} c(e') \mid C \in \mathcal{C}\}$
  $T \leftarrow T + E_\delta$ (rule 1)
  discard edges in $G[C] \setminus T \, \forall C \in \mathcal{C}$ (rule 2)
return T

An example for Prim's algorithm is given in Figure 4.3. Again, correctness follows immediately from Theorem 4.6, and running times depend on a specific implementation. Note that consistent tie-breaking is essential for BORŮVKA (see footnote 1 on page 32).

Proposition 4.10. A naive implementation of PRIM or BORŮVKA computes a minimum spanning tree in time $\Theta(nm)$.

Proof. We can obtain the claimed running time by using DFS$(T \cup \delta_G(T), u)$ or BFS$(T \cup \delta_G(T), u)$ to find an edge of minimum weight in the cut $\delta_G(T)$. This algorithm runs in time $\mathcal{O}(m)$, and every time we grow a connected component. This means that we are done after $n - 1$ steps and obtain the claimed running time. □

We can improve this running time for PRIM by using a data structure that keeps track of the best edge to reach every vertex in $V \setminus T$ from T. With this, finding an edge of minimum weight in $\delta_G(T)$ can be accomplished in time $\mathcal{O}(n)$. Since we need to do this at most $n-1$ times, the total running time for finding edges of minimum weight is $\mathcal{O}(n^2)$. Every edge may be the best edge to reach a vertex in $V \setminus T$ from T, so the total cost for updating our data structure is $\mathcal{O}(m)$. Overall, we get an improved running time of $\mathcal{O}(n^2) + \mathcal{O}(m) = \mathcal{O}(n^2)$.

The data structure we described above is a so-called *priority queue*. It supports the following operations:

- INSERT(S): Inserts all elements in S into the queue and assigns value ∞ to them.
- VALUE(s): Returns the value assigned to element s.
- DECREASEVALUE(s, x): The value assigned to s is *decreased* to x.
- EXTRACTMIN: Returns an object of lowest value and removes it from the queue.

We can use a priority queue to store the smallest weight of an edge to reach every vertex $v \notin V_T$ from a vertex in V_T. This leads to the following reformulation of PRIM:

Algorithm: PRIM(G, c) (priority queue)

input: undirected, connected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$
output: minimum spanning tree

```
take any u ∈ V
T = (V_T, E_T) ← ({u}, ∅)
INSERT(V \ {u})
while V_T ≠ V:
    for v ∈ Γ_G(u) \ Γ_T(u):
        if c({u, v}) < VALUE(v):
            e_v ← {u, v}
            DECREASEVALUE(v, c(e_v))
    u ← EXTRACTMIN
    T ← T + u + e_u
return T
```

We use the fact that priority queues can be implemented more efficiently without proof (see *Combinatorial Optimization*).

Theorem 4.11. It is possible to implement a priority queue that takes time $\mathcal{O}(m + n \log n)$ to insert $\mathcal{O}(n)$ elements and perform $\mathcal{O}(m)$ other operations.

With this, we obtain an even better running time for Prim's algorithm.

Theorem 4.12. PRIM can be implemented to find a minimum spanning tree in time $\mathcal{O}(m + n \log n)$.

Proof. For every vertex, PRIM has to iterate over all neighbors, which means that the for-loop iterates at most $2m$ times overall. Consequently, we insert n elements into the priority queue and perform $\mathcal{O}(m)$ other operations on it. The running time thus follows immediately by Theorem 4.11. □

Remark 4.13. Observe that this running time is better than the best running time achievable with Kruskal's algorithm. The best known running time of $\mathcal{O}(m \cdot \alpha(m))$ for the minimum spanning tree problem was achieved by Chazelle. Here the inverse Ackermann function $\alpha(m)$ is an extremely slowly growing function (e.g., $\alpha(9876!) = 5$).

5 Shortest Paths

We now turn to the problem of finding short paths in a directed graph. This problem has obvious applications for routing and navigation, but it also appears naturally as a subproblem in many other places (we will see one example in Chapter 6).

We already showed that BFS computes a path with the minimum number of arcs from a given root (Theorem 3.18). It is more challenging to find a good path in a directed graph $G = (V, E)$ with respect to given arc weights $c\colon E \to \mathbb{R}$. We will make this notion more precise in the following. For convenience, we write $c(u, v) := c((u, v))$ and set $c(u, u) = 0$ and $c(u, v) = \infty$ if $(u, v) \notin E$. For a walk W in G, we let $c(W) := \sum_{e \in W} c(e)$.

Definition 5.1. Let $G = (V, E)$ be a directed graph and $c\colon E \to \mathbb{R}$. We let $\mathcal{W}_{uv}$ denote the set of all walks from u to v in G and define

$$d^{(k)}_{c,G}(u, v) := \min\left\{\infty, \min\left\{ c(W) \middle| W = (v_0 = u, e_1, \dots, v_{k'} = v) \in \mathcal{W}_{uv} \text{ with } k' \leq k\right\}\right\},$$
$$d_{c,G}(u, v) := \lim_{k\to\infty} d^{(k)}_{c,G}(u, v).$$

We drop the index G and write $d_c(u, v) := d_{c,G}(u, v)$ and $d^{(k)}_c(u, v) := d^{(k)}_{c,G}(u, v)$ when the graph G is clear from the context.
If P is a u-v-path in G with $c(P) = d_c(u, v)$, we say that P is a *shortest path* in G with respect to c. If K is a cycle in G with $c(K) < 0$, we say that K is a *negative cycle* in G induced by c.

We formally show the intuitive fact that, if there are no negative cycles, detours can only increase the length of a walk.

Proposition 5.2. Let $G = (V, E)$ be a directed graph and let $c\colon E \to \mathbb{R}$ not induce negative cycles. Then, for all $u \in V$ and all $v \in V$ reachable from u, there exists a shortest u-v-path in G with respect to c and it holds that $d_c(u, v) = d^{(n-1)}_c(u, v)$.

Proof. For any $k \geq n$, let W be a walk from u to v in G with at most k arcs and such that $c(W) = d^{(k)}_c(u, v)$. By Proposition 3.8, we can decompose W into a u-v-path P and a set of cycles $\mathcal{K}$ (note that, in directed graphs, closed walks of length two are cycles). Since c does not induce negative cycles, we have $c(K) \geq 0$ for all $K \in \mathcal{K}$ and thus $c(W) = c(P) + \sum_{K\in\mathcal{K}} c(K) \geq c(P)$. Since P has at most k arcs, we have $c(P) \geq d^{(k)}_c(u, v) = c(W)$. Together this implies that $c(P) = d^{(k)}_c(u, v)$.

Now, P is a path and thus has at most $n - 1$ arcs, which means that $d^{(n-1)}_c(u, v) \leq c(P) = d^{(k)}_c(u, v)$, and thus $d^{(n-1)}_c(u, v) = d^{(k)}_c(u, v)$. Since our argument holds for all $k \geq n$, we conclude that $d_c(u, v) = d^{(n-1)}_c(u, v)$ and P is a shortest u-v-path in G. □

A key feature of shortest paths is that they have *optimum substructure*, i.e., any part of a shortest path is again a shortest path. We prove the following generalization of this statement.

Proposition 5.3. Let $G = (V, E)$ be a directed graph and $c\colon E \to \mathbb{R}$. If, for some $k \in \mathbb{N}$, we have $c(W) = d_c^{(k)}(v_0, v_{k'})$ for a walk $W = (v_0, e_1, \dots, v_{k'})$ in G with $k' \leq k$, then $c(W_{ij}) = d_c^{(k-k'+j-i)}(v_i, v_j)$ for every $W_{ij} = (v_i, e_{i+1}, \dots, v_j)$ with $0 \leq i < j \leq k'$.

Proof. By definition, if $c(W_{ij}) \neq d_c^{(k-k'+j-i)}(v_i, v_j)$, then there is a walk W'_{ij} from v_i to v_j with at most $k - k' + j - i$ arcs in G and with $c(W'_{ij}) < c(W_{ij})$. But then the walk $W' := (v_0, \dots, v_i) \oplus W'_{ij} \oplus (v_j, \dots, v_{k'})$ has at most k arcs and $c(W') < c(W)$, contradicting $c(W) = d_c^{(k)}(v_0, v_{k'})$. □

Propositions 5.2 and 5.3 immediately imply the following.

Corollary 5.4. Let $G = (V, E)$ be a directed graph and let $c\colon E \to \mathbb{R}$ not induce negative cycles. For all $s \in V$ and $v \in V \setminus \{s\}$ reachable from s, there exists a vertex $u \in V \setminus \{v\}$ and a shortest s-u-path P_{su} in $G - v$ such that $P_{su} \oplus (u, v)$ is a shortest s-v-path in G.

Corollary 5.4 yields an easy way to determine shortest paths once we have computed $d_c(u, v)$ for all $u, v \in V$: For every $v \in V$ reachable from $s \in V$, the predecessor of v on a shortest s-v-path is $u \in V$ with $v \in \Gamma(u)$ and $d_c(s, v) = d_c(s, u) + c(u, v)$. We can obtain a shortest s-v-path by backtracking from v along these predecessors until we reach s. Note that we have to deal differently with arcs of weight 0.

We are now ready to consider the algorithmic problem of finding a shortest path between vertices s and t. Observe that, in the worst case, the shortest path to vertex t visits all other vertices, which means that we have to compute shortest paths to all $v \in V$ on the way. It therefore makes sense to reformulate the problem and ask for shortest paths to all vertices reachable from s to begin with. As described above, it suffices to compute the values $d_c(s, v)$ for all $v \in V$, since we can find the corresponding paths via Corollary 5.4.

Shortest Paths Problem	
input:	directed graph $G = (V, E)$; arc weights $c\colon E \to \mathbb{R}$; vertex $s \in V$
problem:	find shortest paths in G with respect to c from s to all vertices reachable from s

5.1 Dijkstra's Algorithm

Corollary 5.4 suggests a *greedy* approach to computing the shortest paths from $s \in V$ if all arc weights are positive: We start with the set $S = \{s\}$ of G. Let $v \notin S$ be the closest vertex to s with respect to c. The shortest s-v-path P must lie in $G[S \cup \{v\}]$, since every path that leaves S elsewhere cannot be shorter than P already up to this point, and it can only be even longer overall, since arc weights are positive. This means that if we keep track of all shortest paths to vertices in S, we can find the closest vertex to s and add it to S in each step. We will see that this simple algorithm, listed below, still works if arc weights are non-negative.

Algorithm: DIJKSTRA(G, c, s)

input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

$$d_{sv} \leftarrow \begin{cases} 0, & \text{if } v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

$R \leftarrow V$ (INSERT)

while $R \neq \emptyset$**:**

  $u \leftarrow \arg\min_{v \in R} d_{sv}$

  $R \leftarrow R \setminus \{u\}$ (EXTRACTMIN)

  for $v \in \Gamma(u) \cap R$**:**

    $d_{sv} \leftarrow \min\{d_{sv}, d_{su} + c(u, v)\}$ (DECREASEVALUE)

return $(d_{sv})_{v \in V}$

We now give a formal argument for the correctness of the above formulation of Dijkstra's algorithm.

Theorem 5.5. DIJKSTRA is correct.

Proof. Denote $S := V \setminus R$ and $G_v := G[S \cup \{v\}]$ for $v \in V$ throughout the algorithm. We show that DIJKSTRA maintains the invariant that for all $v \in V$ we have (i) $d_{sv} = d_{c,G_v}(s, v)$, and (ii) if $v \in S$, then $d_{sv} = d_c(s, v)$. This invariant trivially holds in the beginning, and it implies correctness when the algorithm terminates with $S = V$ after n iterations. To prove the invariant, consider the beginning of an iteration where vertex $u \in R$ is extracted from R.

Let P be a shortest s-u-path in G and let (w, w') be the first arc along P that lies in the cut $\delta(S)$ (this arc exists by Observation 3.6). Since c is non-negative, Proposition 5.3 implies that $c(P) \geq d_c(s, w') = d_{c,G_{w'}}(s, w')$. Our invariant (i) guarantees $d_{c,G_{w'}}(s, w') = d_{sw'}$. Finally, by choice of u, we have $d_{sw'} \geq \min_{v \in R} d_{sv} = d_{su}$. Using invariant (i) again, we conclude that $d_c(s, u) = c(P) \geq d_{su} = d_{c,G_u}(s, u)$. Since $d_c(s, u) \leq d_{c,G_u}(s, u)$ by definition, equality holds and invariant (ii) is maintained when u is added to S.

Now, by invariants (i) and (ii), we have $d_{c,G_w}(s, w) = d_c(s, w)$ for all $w \in S$, i.e., there is a shortest s-w-path that lies entirely in G_w and, in particular, does not contain u. Consequently, for every $v \in \Gamma(u) \cap R$ there is a shortest s-v-path in $G_{uv} := G[S \cup \{u, v\}]$ that either does not go via u or uses the arc from u to v. By Corollary 5.4 and invariant (i), we obtain

$$\begin{aligned} d_{c,G_{uv}}(s, v) &= \min\{d_{c,G_u}(s, u) + c(u, v), d_{c,G_v}(s, v)\} \\ &= \min\{d_{su} + c(u, v), d_{sv}\}, \end{aligned}$$

which means that we update the values d_{sv} for $v \in \Gamma(v) \cap R$ such that invariant (i) is maintained when u is added to S. □

The similarity between DIJKSTRA and PRIM should be obvious. While PRIM adds the arc next that leads to a vertex closest to any $v \in S$, DIJKSTRA adds the arc next that leads to a vertex closest to s. Otherwise, both algorithms are identical, which means that we can use a very similar implementation using a priority queue.

Theorem 5.6. DIJKSTRA can be implemented with a running time of $\mathcal{O}(m + n \log n)$.

Proof. We can implement R as a priority queue. We initially insert n vertices into R and perform at most $n + m$ other operations on it. By Theorem 4.11, this can be accomplished with a total running time of $\mathcal{O}(m + n \log n)$. □

5.2 The Algorithm of Bellman-Ford(-Moore)

While Dijkstra's algorithm is conceptually simple, its efficient implementation is quite involved. Additionally, Dijktra's algorithm cannot cope with negative arc weights. We can circumvent both these issues by investigating the interdependency of the values $d_c^{(k)}(u, v)$.

Lemma 5.7. The following recurrence holds:

$$d_c^{(k)}(u,v) = \begin{cases} c(u,v) & \text{if } k = 1, \\ \min_{w \in V}\{d_c^{(k-1)}(u,w) + c(w,v)\} & \text{if } k \geq 2. \end{cases}$$

Proof. The statement is trivial for $k = 1$. For $k \geq 2$, let $W = (v_0 = u, \ldots, v_{k'} = v)$ be a walk with $k' \leq k$ and $c(W) = d_c^{(k)}(u, v)$. Then, by Proposition 5.3, $c(W') = d_c^{(k-1)}(u, v_{k'-1})$ for $W' = (v_0, \ldots, v_{k'-1})$. Hence, $d_c^{(k)}(u, v) = c(W) = c(W') + c(v_{k'-1}, v) = d_c^{(k-1)}(u, v_{k'-1}) + c(v_{k'-1}, v)$. It remains to show that there is no vertex w with $d_c^{(k-1)}(u, w) + c(w, v) < d_c^{(k)}(u, v)$. This is the case since, otherwise, we could construct a walk from u to v with at most k arcs that is shorter than W. □

Lemma 5.7 immediately suggests a recursive approach to calculating the values $d_c^{(k)}(u, v)$ for all $k \in \{1, \ldots, n-1\}$, listed below. Recall that it suffices to compute the values $d_c^{(n-1)}(s, v)$ for all $v \in V$ by Proposition 5.2, assuming that c does not induce negative cycles. However, it is easy to see that in this way we may consider all n^{n-1} possible sequences of n vertices starting at s in the process (for a complete graph), which leads to a horrendous running time.

The problem of the recursive approach is that we repeatedly recompute the same values (e.g., $d_{sv}^{(k)}(u, v)$ is computed up to n^{n-1-k} times). This is a typical issue arising with recursive implementations that absolutely needs to be avoided! A typical way to repair such an implementation without much effort uses *memoization*, i.e., the storage of precomputed partial solutions. The main disadvantage of memoization from a theoretical perspective lies in the difficulty to analyze a memoization solution (not a big issue in the simple implementation below). From a practical perspective, implementations relying on memoization often suffer from a lack of locality, i.e., data accesses are not grouped well together according to the corresponding locations in system memory.

Algorithm: $\textsc{Rec}(v, k)$

if $k = 1$**:**
  return $c(s, v)$
else
  return $\min_{u \in V}\{\textsc{Rec}(u, k-1) + c(u, v)\}$

Algorithm: $\textsc{RecursiveSP}(G, c, s)$

input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

for $v \in V$**:**
  $d_{sv} \leftarrow \textsc{Rec}(v, \max\{1, n-1\})$
return $(d_{sv})_{v \in V}$

Algorithm: $\textsc{Memo}(v, k)$

if $d_{sv}^{(k)} = -\infty$**:**
  $d_{sv}^{(k)} \leftarrow \min_{u \in V}\{\textsc{Memo}(u, k-1) + c(u, v)\}$
return $d_{sv}^{(k)}$

Algorithm: $\textsc{MemoizationSP}(G, c, s)$

input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

$$d_{sv}^{(k)} \leftarrow \begin{cases} c(s, v), & \text{if } k = 1, \\ -\infty, & \text{otherwise.} \end{cases}$$

for $v \in V$**:**
  $d_{sv}^{(n-1)} \leftarrow \textsc{Memo}(v, \max\{1, n-1\})$
return $(d_{sv}^{(n-1)})_{v \in V}$

A much more systematic paradigm is *dynamic programming*, where we compute partial solutions in a bottom-up fashion (listing below). Dynamic programs (DPs) are easier to analyze and to verify and can be extremely efficient in practice. As a rule of thumb, problems that admit optimum substructure lend themselves to dynamic programming solutions. Conceptually, it is often easiest to start with a recursive solution, reformulate it to use memoization, and finally reformulate it again using dynamic programming. In a nutshell, the description of every (!) dynamic program consists of the following ingredients:

(i) DP table and interpretation of the meaning of its entries (here: Definition 5.1)
(ii) dependencies between entries of the DP table (here: Lemma 5.7)
(iii) an order of computation of the entries that respects dependencies (here: by increasing values of k)
(iv) extraction of the final solution from the DP table (here: Proposition 5.2)

Algorithm: $\textsc{DynamicSP}(G, c, s)$

input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

$$d_{sv}^{(0)} \leftarrow \begin{cases} 0, & \text{if } v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

for $k \leftarrow 1, 2, \ldots, n-1$**:**
  for $v \in V$**:**
    $d_{sv}^{(k)} \leftarrow \min_{u \in V}\{d_{su}^{(k-1)} + c(u, v)\}$
return $\left(d_{sv}^{(n-1)}\right)_{v \in V}$

Indeed, $\textsc{DynamicSP}$ is easy to analyze: It's running time is $\Theta(n^3)$, which is drastically better than the purely recursive algorithm above.

We can further improve the algorithm by avoiding to iterate over pairs $u, v \in V$ with $v \notin \Gamma(u)$ and we can reduce the number of variables (i.e., the memory requirement) of the algorithm if we do not insist to keep

track of the best walks for every number of arcs. These changes lead us to the algorithm by Bellman-Ford (concurrently proposed by Moore):

Algorithm: BELLMANFORD(MOORE)(G, c, s)

input: directed graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}$, vertex $s \in V$
output: distances $d_{sv} = d_c(s, v)$ for all $v \in V$

$$d_{sv} \leftarrow \begin{cases} 0, & \text{if } v = s, \\ \infty, & \text{otherwise.} \end{cases}$$

for $k \leftarrow 1, 2, \ldots, n-1$**:**
  for $(u, v) \in E$**:**
    $d_{sv} \leftarrow \min\{d_{sv}, d_{su} + c(u, v)\}$

return $(d_{sv})_{v \in V}$

To show the following result, we mainly need to argue that using a single value d_{sv} instead of $d_{sv}^{(0)}, d_{sv}^{(1)}, \ldots, d_{sv}^{(n-1)}$ does not cause problems. Note that (for $m \in \omega(\log n)$) the running time is worse than for Dijkstra's algorithm, however, BELLMAN-FORD(-MOORE) is easy to implement (which also means that the hidden constant factor in the running time is small) and is very efficient in practice. Additionally, BELLMANFORD(MOORE) can work with negative arc weights, as long as their are no negative cycles, in which case shortest paths may not exist.

Theorem 5.8. BELLMANFORD(MOORE) solves the shortest path problem in time $\Theta(nm)$ on directed graphs without negative cycles.

Proof. We show that the algorithm maintains the invariant that, after the k-th iteration of the outer loop, $d_{sv} \leq d_c^{(k)}(s, v)$ for all $v \in V$ and that an s-v-path of length at most d_{sv} exists if $d_{sv} < \infty$ (with an arbitrary number of arcs). The invariant obviously holds in the beginning, and it implies that the result of the algorithm is correct, since $d_c(s, v) = d_c^{(n-1)}(s, v)$ by Proposition 5.2. It remains to show that the invariant is maintained. Consider an update of d_{sv} for arc (u, v) in iteration k of the outer loop. If the value of d_{sv} changes, then a walk of the claimed length exists via u. Since there are no negative cycles, we can use Proposition 3.8 to obtain a path of length at most d_{sv}. By Lemma 5.7, there is a vertex $w \in V$ with $d_c^{(k)}(s, v) = d_c^{(k-1)}(s, w) + c(w, v)$. By our invariant at the end of iteration $k-1$, we have $d_{sw} \leq d_c^{(k-1)}(s, w)$. In the iteration of the inner loop for (w, v), we set

$$d_{sv} \leftarrow \min\{d_{sv}, d_{sw} + c(w, v)\} \leq d_{sw} + c(w, v) \leq d_c^{(k-1)}(s, w) + c(w, v) = d_c^{(k)}(s, v),$$

hence our invariant is maintained. The running time of the algorithm is obvious. □

6 Network Flows

In this chapter we consider network flows as an abstract framework for throughput maximization in graphs. This framework captures flows of many types, including traffic flows, resource flows, data flows, currency flows, and many more. In addition, structural results for network flows are applicable in other domains, as we will see in Chapter 7. We start by introducing the basic framework.

Definition 6.1. A *network* is a 4-tuple (G, μ, s, t) where $G = (V, E)$ is a directed graph, $\mu \colon E \to \mathbb{R}_{>0}$ are arc *capacities*, $s \in V$ is the *source*, and $t \in V$ is the *sink*. To simplify notation, we additionally require that $\{(u, v), (v, u)\} \not\subseteq E$ for all $u, v \in V$.

Definition 6.2. A function $f \colon E \to \mathbb{R}_{\geq 0}$ is an *s-t-flow* in the network $(G = (V, E), \mu, s, t)$ if it satisfies:

(i) (capacity constraints) $f(e) \leq \mu(e)$ for all arcs $e \in E$,

(ii) (flow conservation) $\mathrm{ex}_f(v) = 0$ for all vertices $v \in V \setminus \{s, t\}$,

(iii) (positivity) $|f| \geq 0$,

where the *excess* of f at $v \in V$ is defined as $\mathrm{ex}_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$, and the *value* of f is defined as $|f| := \mathrm{ex}_f(t)$.

With this, we are ready to formally state the algorithmic problem that we will be concerned with in this chapter.

Maximum Flow Problem	
input:	network $(G = (V, E), \mu, s, t)$
problem:	find s-t-flow in G of maximum value

For the remainder of this chapter, we fix a network $(G = (V, E), \mu, s, t)$, an s-t-flow f in this network, and a flow $f^\star$ of maximum value. We will see later that such a flow always exists (Corollary 6.14) and say that $f^\star$ is a *maximum flow*.

The following notion will play a crucial role to bound the value of a maximum flow.

Definition 6.3. An *s-t-cut* is a cut $\delta^+(S)$ with $s \in S \subseteq V \setminus \{t\}$. Its *capacity* is $\mu(S) := \sum_{e \in \delta^+(S)} \mu(e)$.

Clearly, flow conservation implies $|f| = -\mathrm{ex}_f(s)$. We now show a stronger statement.

Proposition 6.4. For every s-t-cut $\delta^+(S)$ in G, it holds that

$$\sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) = |f| \leq \mu(S).$$

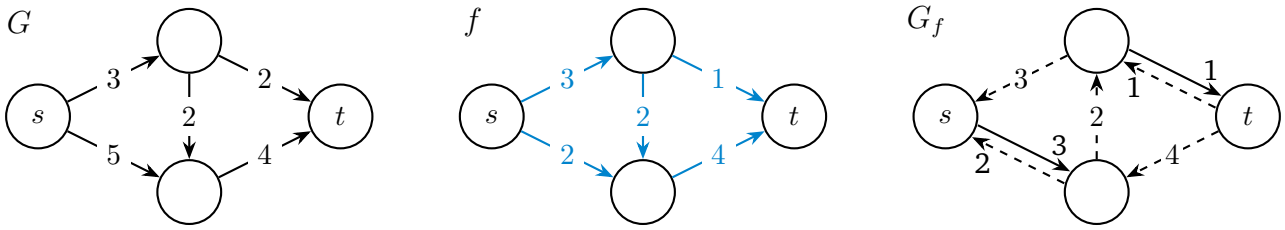


Figure 6.1: From left to right: network (G, μ, s, t) with arcs labeled by μ, graph G with arcs labeled by a flow f of value $|f| = 5$, residual graph G_f with arcs labeled by residual capacities.

Proof. For $T := V \setminus S$ and using capacity constraints (CC), and flow conservation (FC), we obtain

$$
\begin{aligned}
|f| &= \mathrm{ex}_f(t) \\
&= \sum_{e \in \delta^-(t)} f(e) - \sum_{e \in \delta^+(t)} f(e) \\
&\overset{\text{(FC)}}{=} \sum_{v \in T} \left(\sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) \right) \\
&= \sum_{v \in T} \sum_{e \in \delta^-(v)} f(e) - \sum_{v \in T} \sum_{e \in \delta^+(v)} f(e) \\
&= \sum_{v \in T} \sum_{e \in \delta^-(v) \setminus T^2} f(e) - \sum_{v \in T} \sum_{e \in \delta^+(v) \setminus T^2} f(e) \\
&= \sum_{e \in \delta^-(T)} f(e) - \sum_{e \in \delta^+(T)} f(e) \\
&= \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e) \\
&\leq \sum_{e \in \delta^+(S)} f(e) \\
&\overset{\text{(CC)}}{\leq} \sum_{e \in \delta^+(S)} \mu(e),
\end{aligned}
$$

which concludes the proof. □

A flow may not *saturate* all arcs, i.e., there may be arcs $e \in E$ with $f(e) < \mu(e)$, which means that there might be potential for increasing the flow value. To do this, we may need to reroute some of the flow (cf. Figure 6.1 (center)). It makes sense to consider the graph that encodes all potential flow increases and changes in order to use graph theoretic tools in the design of algorithms. This *residual graph* is defined below (cf. Figure 6.1 (right)).

Definition 6.5. For every $e = (u, v) \in V \times V$, we define the *reverse arc* of e as $\bar{e} := (v, u)$ and set $\bar{G} = (V, \bar{E})$ with $\bar{E} := \{\bar{e} \mid e \in E\}$. The *residual capacity* of $e \in E \cup \bar{E}$ with respect to f is defined by

$$\mu_f(e) := \begin{cases} \mu(e) - f(e) & \text{if } e \in E, \\ f(\bar{e}) & \text{if } e \in \bar{E}. \end{cases}$$

The *residual graph* of G with respect to f is

$$G_f := \big(V, \{\, e \in E \cup \bar{E} \,\big|\, \mu_f(e) > 0\}\big).$$

Remark 6.6. Observe that residual capacities are well-defined since $E \cap \bar{E} = \emptyset$ by our additional requirement for networks that arcs may not be present in both directions. Without this requirement, the residual graph needs to be defined as a multigraph (see Remark 3.2), which would complicate notation. Alternatively, we can avoid multigraphs if we subdivide each arc in E by inserting an additional vertex, which ensures that $E \cap \bar{E} = \emptyset$. However, this increases the number of vertices to $\Theta(m)$, which distorts the running times we achieve below.

We can now make the notion precise that a flow has the potential to be increased.

Definition 6.7. An *augmenting path* is an s-t-path in the residual graph G_f.

Theorem 6.8. The flow f is a maximum flow if and only if there is no augmenting path in G_f.

Proof. If there is an s-t-path P in G_f, we can increase flow values along this path by $\min_{e \in P} \mu_f(e)$ without violating capacity constraints (or flow conservation) – hence, f cannot be a maximum flow.

If there is no s-t-path in G_f, then the set $S \subseteq V$ of all vertices reachable from s in G_f induces an empty cut $\delta^+_{G_f}(S) = \emptyset$ in G_f. For every arc $e \in \delta^+_G(S) \cup \delta^+_{\bar{G}}(S)$ we thus have $\mu_f(e) = 0$. With this, we obtain

$$\begin{aligned} 0 &= \sum_{e \in \delta^+_G(S)} \mu_f(e) + \sum_{e \in \delta^+_{\bar{G}}(S)} \mu_f(e) \\ &= \sum_{e \in \delta^+_G(S)} \mu_f(e) + \sum_{e \in \delta^-_G(S)} \mu_f(\bar{e}) \\ &= \sum_{e \in \delta^+_G(S)} \big(\mu(e) - f(e)\big) + \sum_{e \in \delta^-_G(S)} f(e), \end{aligned}$$

which, by Proposition 6.4, implies

$$|f| = \sum_{e \in \delta^+_G(S)} f(e) - \sum_{e \in \delta^-_G(S)} f(e) = \sum_{e \in \delta^+_G(S)} \mu(e).$$

By Proposition 6.4 there cannot be any s-t-flow of larger value. It follows that f is a maximum s-t-flow. □

6.1 The Ford-Fulkerson Method

Theorem 6.8 suggests a straight forward algorithm to find a maximum flow: While there is an augmenting path P in G_f, augment the flow by the largest amount possible along P. This algorithm is called the Ford-Fulkerson

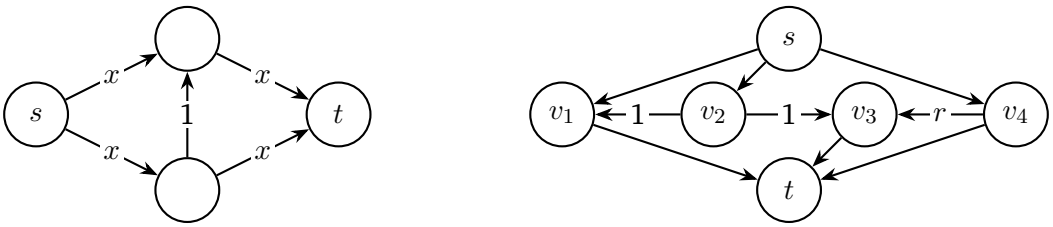


Figure 6.2: Left: a network where FORDFULKERSON needs $\Omega(|f^\star|)$ iterations. Right: a network where FORDFULKERSON may not terminate (for $r = (\sqrt{5}-1)/2$).

method, where the designation as a *method* is due to the fact that the algorithm can be implemented in different ways, since it does not specify which path exactly to choose in each step.

Algorithm: FORDFULKERSON(G, μ, s, t)

input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$f \leftarrow 0$
while $\exists$ ***any*** *s-t-path P in G_f* **:**
  $\delta \leftarrow \min_{e \in P} \mu_f(e)$
  AUGMENT(P, δ)
return f

Algorithm: AUGMENT(P, δ)

for $e \in P$**:**
  if $e \in E$**:**
    $f(e) \leftarrow f(e) + \delta$
  else
    $f(\bar{e}) \leftarrow f(\bar{e}) - \delta$

For integral capacities (i.e., $\mu \in \mathbb{N}$), we can show that this method *always* works, independent of which augmenting path we choose in each step. Note that each augmenting path can be found in time $\mathcal{O}(m)$ by using DFS/BFS, so that the following theorem yields a running time of $\mathcal{O}(|f^\star| \cdot m)$.

Theorem 6.9. Every implementation of FORDFULKERSON finds a maximum flow in $\mathcal{O}(|f^\star|)$ augmentations for a network with integral capacities. For some choices of augmenting paths, FORDFULKERSON needs $\Omega(|f^\star|)$ augmentations.

Proof. By Theorem 6.8, the algorithm is correct if it terminates. Since all capacities are integral, we increase the flow value by at least 1 in each iteration. The total number of iterations can thus be at most $|f^\star|$. For the lower bound, consider the example in Figure 6.2 (left). This network has $|f^\star| = 2x$ and needs $2x$ augmentations if we decide to augment along paths of length 3 in each iteration. □

Corollary 6.10. Every network with integral capacities admits an integral maximum flow.

In contrast, it can be shown that, in general, FordFulkerson does not even terminate if augmenting paths are chosen in an unfortunate way (see Figure 6.2 (right)). The proof of the following statement is left as an exercise.

Proposition 6.11. FordFulkerson may not always terminate for real-valued capacities.

6.2 The Edmonds-Karp Algorithm

Inspecting the examples in Figure 6.2, a natural idea to improve the Ford-Fulkerson method is to ensure that short augmenting paths are favored. The corresponding algorithm that always takes a shortest path in G_f is called the Edmonds-Karp algorithm.

Algorithm: EdmondsKarp(G, μ, s, t)
input: network $(G = (V, E), \mu, s, t)$
output: maximum s-t-flow

$f \leftarrow 0$
while $\exists s$-t-*path in* G_f **:**
  $P \leftarrow$ shortest s-t-path in G_f (unweighted)
  $\delta \leftarrow \min_{e \in P} \mu_f(e)$
  Augment(P, δ)
return f

The following lemma establishes that EdmondsKarp makes progress in some sense, between increasing flow along an arc and rerouting it later.

Lemma 6.12. Let $P_1, P_2, \dots$ denote the augmenting paths chosen by EdmondsKarp in this order. Then, for every two such paths P_k, P_ℓ with $k \leq \ell$ we have

$$|P_\ell| \geq |P_k| + 2 \cdot \min\{1, m_{k\ell}\},$$

where $m_{k\ell} := |\{ e \in E | \, e, \bar{e} \in P_k \cup P_\ell \}|$.

Proof. We may assume that there is no path P_i with $i \in \{k+1, \dots, \ell-1\}$ that contains an arc e with $\bar{e} \in P_\ell$. Otherwise, by induction on $\ell - k$, we immediately have $|P_\ell| \geq |P_i| + 2 \geq |P_k| + 2 \geq |P_k| + 2 \cdot \min\{1, m_{kl}\}$. Let $\bar{E}_{k\ell} := \left\{ e \in E \cup \bar{E} \middle| \, e, \bar{e} \in P_k \cup P_\ell \right\}$ and let f_k denote the flow before augmenting along P_k in the k-th iteration. We consider the graphs $H_k = (V, E_k) := (V, P_k \setminus \bar{E}_{k\ell})$, $H_\ell = (V, E_\ell) := (V, P_\ell \setminus \bar{E}_{k\ell})$, and $H_{k\ell} := (V, (P_k \cup P_\ell) \setminus \bar{E}_{k\ell})$. Since the paths $P_{k+1}, \dots, P_{\ell-1}$ do not contain any reverse arcs to arcs in P_ℓ, we have $P_\ell \setminus \bar{E}_{k\ell} \subseteq G_{f_k}$ and hence $H_k, H_\ell, H_{k\ell} \subseteq G_{f_k}$. Moreover, we have $|\delta^+_{H_k}(v)| + |\delta^+_{H_\ell}(v)| = |\delta^-_{H_k}(v)| + |\delta^-_{H_\ell}(v)|$ for all $v \in V \setminus \{s, t\}$. This allows us to construct a new walk W_1 in $H_{k\ell}$ as follows: Start at s and repeatedly take an unused arc in H_k or H_ℓ, until we finally reach t. Because of $|\delta^+_{H_k}(s)| + |\delta^+_{H_\ell}(s)| = 2$ and $|\delta^-_{H_k}(s)| + |\delta^-_{H_\ell}(s)| = 0$ we can repeat this process once to obtain an additional walk W_2 in $H_{k\ell}$ using different arcs of H_k and H_ℓ (in $H_{k\ell}$ some arcs may be used twice). By Proposition 3.8, we can obtain two s-t-paths $Q_1 \subseteq W_1$ and $Q_2 \subseteq W_2$ in $H_{k\ell} \subseteq G_{f_k}$. Since P_k is an (unweighted) shortest s-t-path in G_{f_k}, we must have

$$2\,|P_k| \leq |Q_1| + |Q_2| \leq |W_1| + |W_2| \leq |E_k| + |E_\ell| = |P_k| + |P_\ell| - 2m_{kl} \leq |P_k| + |P_\ell| - 2 \cdot \min\{1, m_{kl}\}. \quad \square$$

With this, we can bound the running time of the Edmonds-Karp algorithm even for general capacities.

Theorem 6.13. EDMONDSKARP can be implemented to find a maximum flow in time $\mathcal{O}(m^2 n)$.

Proof. Consider an arc $e \in E \cup \bar{E}$ and all augmenting paths $P_1, P_2, \ldots, P_k$ that saturate arc e. For $i \in \{1, \ldots, k-1\}$, between the augmentations along P_i and P_{i+1} there must have been an augmenting path P utilizing $\bar{e}$. According to Lemma 6.12 we have

$$|P_{i+1}| \geq |P| + 2 \geq |P_i| + 4.$$

With $|P_k| \leq n-1$ and $|P_1| \geq 1$, this implies $k \leq (n-1)/4 = \mathcal{O}(n)$.

Since every augmenting path saturates at least one arc and there are at most $\mathcal{O}(n)$ augmenting paths saturating the same arc, the total number of augmenting paths can be at most $\left|E \cup \bar{E}\right| \cdot \mathcal{O}(n) = \mathcal{O}(mn)$. We can find (unweighted) shortest paths using BFS in time $\mathcal{O}(m)$ (Theorem 3.18). Overall this gives a running time of $\mathcal{O}(m^2 n)$. □

In particular, we have found an algorithmic existence proof of maximum flows in finite networks.

Corollary 6.14. A maximum flow always exists.

6.3 Path Decomposition

Observe that Theorem 6.8 provides a straight-forward way of proving that a given flow in *not* a maximum flow. Conversely, by Proposition 6.4, we can prove that a flow is a maximum flow by finding a cut whos capacity is equal to the flow value. The following *max-flow min-cut theorem* guarantees that we can always find such a cut, and thus provides an elegant way of proving maximality. This structural insight is the most fundamental result for network flows.

Theorem 6.15 (max-flow min-cut)**.** The value of a maximum s-t-flow is equal to the minimum capacity over all s-t-cuts, i.e.,

$$|f^\star| = \min_{S \subseteq V \setminus \{t\} : s \in S} \mu(S).$$

Proof. Let $\delta^+(S^\star)$ be an s-t-cut of minimum capacity. By Proposition 6.4, we have

$$|f^\star| \leq \sum_{e \in \delta^+(S^\star)} \mu(e).$$

Conversely, by Theorem 6.8, there is no augmenting path in $G_{f^\star}$. We can therefore let $S \subseteq V \setminus \{t\}$ denote the set of vertices reachable from s in $G_{f^\star}$. Then, $\delta^+_{G_{f^\star}}(S) = \emptyset$. Hence, $\mu_{f^\star}(e) = 0$ for all $e \in \delta^+_G(S) \cup \delta^+_{\bar{G}}(S)$. With

Proposition 6.4, we obtain

$$\begin{aligned}
|f^\star| &= \sum_{e\in\delta_G^+(S)} f^\star(e) - \sum_{e\in\delta_G^-(S)} f^\star(e) \\
&= \sum_{e\in\delta_G^+(S)} f^\star(e) - \sum_{e\in\delta_G^-(S)} \mu_{f^\star}(\bar{e}) \\
&= \sum_{e\in\delta_G^+(S)} \big(\mu(e) - \mu_{f^\star}(e)\big) - \sum_{e\in\delta_{\bar{G}}^+(S)} \mu_{f^\star}(e) \\
&= \sum_{e\in\delta_G^+(S)} \mu(e) \\
&\geq \sum_{e\in\delta_G^+(S^\star)} \mu(e).
\end{aligned}$$

□

Another crucial and intuitive property of flows is that they can always be decomposed in the following sense.

Theorem 6.16 (path decomposition)**.** For every s-t-flow f there exists a family $\mathcal{F} = \mathcal{P} \cup \mathcal{K}$ with $|\mathcal{F}| \leq |E|$ of s-t-paths $\mathcal{P}$ and cycles $\mathcal{K}$ and weights $c\colon \mathcal{F} \to \mathbb{R}_{>0}$ with $f(e) = \sum_{F\in\mathcal{F}:e\in F} c(F)$ for all $e \in E$ and $|f| = \sum_{P\in\mathcal{P}} c(P)$.
If f is integral, all weights can be chosen to be integral.

Proof. We prove the statement by induction over $|E|$. For $|E| = 0$ or $f = 0$ the statement trivially holds with $\mathcal{F} = \emptyset$. Otherwise, we let $W = (v_0, e_1, v_1, \dots, e_k, v_k)$ be a walk of maximum length with $f(e_i) > 0$ for all $i \in \{1, \dots, k\}$ and $\{v_1, \dots, v_{k-1}\}$ are all distinct. Then, $k \geq 1$ since $f \neq 0$. By flow conservation and since W is maximal, we either have $v_0, v_k \in \{v_1, \dots, v_{k-1}\}$, or $v_0 = s$ and $v_k = t$. Accordingly, W contains a walk F that is a cycle or an s-t-path. Let $e_{\min} \in \arg\min_{e\in F} f(e)$. We define $G' = (V, E') := G - e_{\min}$ and a flow $f'\colon E' \to \mathbb{R}_{\geq 0}$ in G' with

$$f'(e) := \begin{cases} f(e) - f(e_{\min}) & \text{if } e \in F, \\ f(e) & \text{otherwise.} \end{cases}$$

Let $\mathcal{F}' = \mathcal{P}' \cup \mathcal{K}'$ be the family of paths and cycles that we obtain by induction for G' and f', and let $c'\colon E' \to \mathbb{R}_{\geq 0}$ be the corresponding weights. We set $\mathcal{F} := \mathcal{F}' \cup \{F\}$, $c(F) := f(e_{\min})$, and $c(F') = c'(F')$ for $F' \in \mathcal{F}'$. By induction, we have $|\mathcal{F}| = |\mathcal{F}'| + 1 \leq |E'| + 1 = |E|$. Let $\delta_e, \delta_F \in \{0, 1\}$ such that $\delta_e = 1$ if and only if $e \in F$ and $\delta_F = 1$ if and only if F is a path. With this, we obtain

$$\begin{aligned}
f(e) = f'(e) + \delta_e \cdot f(e_{\min}) &= \sum_{F'\in\mathcal{F}':e\in F'} c'(F') + \delta_e \cdot c(F) = \sum_{F\in\mathcal{F}:e\in F} c(F), \\
|f| = \big|f'\big| + \delta_F \cdot f(e_{\min}) &= \sum_{P'\in\mathcal{P}'} c'(P') + \delta_F \cdot c(F) = \sum_{P\in\mathcal{P}} c(P),
\end{aligned}$$

where $\mathcal{P}$ denotes the set of paths in $\mathcal{P}' \cup \{F\}$.

If f is integral, the flow f' is integral, and thus $c(F) = f(e_{\min})$ is integral. By induction, it follows that $c(F)$ is integral for all $F \in \mathcal{F}$. □

An important consequence of the above structural insights is Menger's theorem, which states that the number of disjoint paths equals the size of a separating set. This result holds for both vertices and edges of both directed and undirected graphs.

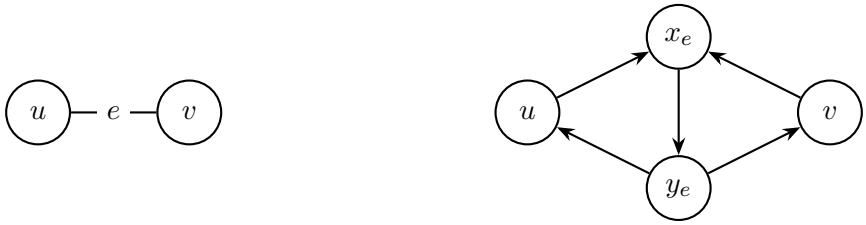


Figure 6.3: From left to right: Transformation for undirected graphs in the proof of Theorem 6.17.

Theorem 6.17 (Menger, edge version)**.** Let $G = (V, E)$ be a graph and $s, t \in V$. The maximum number $p_{\max}$ of edge-disjoint s-t-paths in G is equal to the minimum size $k_{\min}$ of a set $E' \subseteq E$ of edges for which t is not reachable from s in $(V, E \setminus E')$.

Proof. First consider directed graphs. We obviously have $k_{\min} \geq p_{\max}$, since we have to remove at least $p_{\max}$ edges to destroy a fixed set of disjoint paths. We introduce arc capacities $\mu = 1$ for every arc and let $f^\star$ be an integral (by Corollary 6.10) maximum flow in the network (G, μ, s, t). By Theorem 6.16, we can decompose $f^\star$ into a set $\mathcal{P}$ of s-t-paths and a set $\mathcal{K}$ of cycles, each with weight 1. It follows that no arc e can be part of more than one path in $\mathcal{P}$, since $1 = \mu(e) \geq f^\star(e) = \sum_{F \in \mathcal{P} \cup \mathcal{K} : e \in F} c(F)$. Hence, the maximum flow value is $|f^\star| = |\mathcal{P}| \leq p_{\max}$. For every s-t-cut $\delta^+(S)$ we have $|\delta^+(S)| \geq k_{\min}$, since we could remove the arcs of the cut to destroy all s-t-paths. By Theorem 6.15, we thus have $|f^\star| = |\delta^+(S^\star)| \geq k_{\min}$ for an s-t-cut $\delta^+(S^\star)$ of minimum capacity. With $|f^\star| \leq p_{\max}$ it follows that $k_{\min} \leq p_{\max}$.

For undirected graphs, we can replace every edge $e = \{u, v\}$ by arcs $(u, x_e), (x_e, y_e), (y_e, v), (y_e, u), (v, x_e)$, where we introduced additional vertices x_e, y_e (see Figure 6.3). In the resulting directed graph we can still destroy reachability between u and v by removing a single arc. On the other hand, edge-disjoint paths in the original undirected graph still correspond to edge-disjoint paths in the directed graph. □

Theorem 6.18 (Menger, vertex version)**.** Let $G = (V, E)$ be a graph and $t \in V$ be not adjacent to $s \in V$. The maximum number $p_{\max}$ of vertex-disjoint s-t-paths in G is equal to the minimum size $k_{\min}$ of a set $U \subseteq V \setminus \{s, t\}$ of vertices for which t is not reachable from s in $G[V \setminus U]$.

Proof. First consider directed graphs. We replace every vertex $v \in V$ in G by an arc $(v_{\text{in}}, v_{\text{out}})$ and replace every arc (u, v) by the arc $(u_{\text{out}}, v_{\text{in}})$ and call the resulting graph G' (see Figure 6.4). Now, there is a canonical bijection between s-t-paths in G and s_{out}-t_{in}-paths in G'. In particular, two s-t-paths in G are vertex disjoint if and only if the corresponding paths in G' are arc-disjoint. Furthermore, s and t can be separated in G by deleting k vertices if and only if s_{out} and t_{in} can be separated in G' by removing k arcs, since it is always sufficient to remove arcs of the form $(v_{\text{in}}, v_{\text{out}})$ in G', which corresponds to removing the vertex v in G. The statement thus follows for graph G by applying Theorem 6.17 to G'.

We can reduce the statement for undirected graphs to directed graphs with the same construction as in the proof of Theorem 6.17. Note that we need to require that s and t are not adjacent for $k_{\min}$ to be well-defined. □

Figure 6.4: From left to right: Transformation in the proof of Theorem 6.18.

7 Matchings

We now turn to simple assignment problems where objects have to be paired together, while respecting compatitibilies between objects. One example of such a pairing problem are kidney exchange programs, where people in need of an organ that have somebody willing to donate that is biologically incompatible need to find other people to perform cross-exchanges of organs. Another example is the matchmaking in sports or gaming, where teams or individuals have to be paired up against each other. Pairings of this type can be formally modeled as follows.

Definition 7.1. A *matching* in an undirected graph $G = (V, E)$ is a set $M \subseteq E$ of pairwise disjoint edges. A vertex $v \in V \setminus \bigcup_{e \in M} e$ is *(M-)exposed* and we write $v \notin M$. A vertex $v \in \bigcup_{e \in M} e$ is *(M-)covered* and we write $v \in M$. The matching M is

- *perfect* if all vertices of G are M-covered (i.e., if $|M| = \frac{1}{2}|V|$),
- *maximal* if there is no matching $M' \supsetneq M$ in G,
- *maximum* if it maximizes the number of edges in the matching, i.e., if $|M| = \max\{|M'| \mid M' \text{ is a matching in } G\}$.

The algorithmic problem pairing the largest number of objects can now be expressed as the problem of finding a maximum matching. This problem is sometimes also referred to as maximum *cardinality* matching problem, to emphasize the difference to the maximum *weight* matching problem on weighted graphs.

Maximum Matching Problem	
input:	undirected graph $G = (V, E)$
problem:	find maximum matching $M \subseteq E$ of G

We can define a notion of augmenting paths to obtain an optimality criterion very similar to Theorem 6.8 for network flows (see Figure 7.1).

Definition 7.2. Let M be a matching in an undirected graph G. A u-v-path P in G is *(M-)alternating* if $P \setminus M$ is a matching. If, additionally, $u, v \notin M$, then P is *(M-)augmenting*.

Theorem 7.3. Let M be a matching in an undirected graph G. Then, M is a maximum matching in G if and only if there is no M-augmenting path in G.

Proof. If there is an M-augmenting path P in G, then $M \triangle P := (M \cup P) \setminus (M \cap P)$ is a matching with $|M \triangle P| = |M| + 1$, hence M is not maximum.

Now, assume that the matching M is not maximum and let $M^\star$ denote any maximum matching in G. Every vertex of the subgraph $G' := M \cup M^\star$ has degree at most 2. Hence, G' is a disjoint union of paths and cycles. On each path of length greater than one and every cycle, the edges must belong alternatingly to M and $M^\star$. In particular, every cycle must contain the same number of edges from M and $M^\star$. From $|M| < |M^\star|$ it follows that there is a path with more edges from $M^\star$ than from M. This path must be M-augmenting in G. □

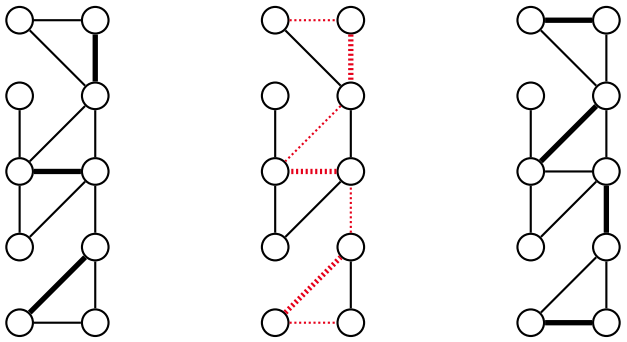


Figure 7.1: From left to right: maximal matching, augmenting path, maximum matching.

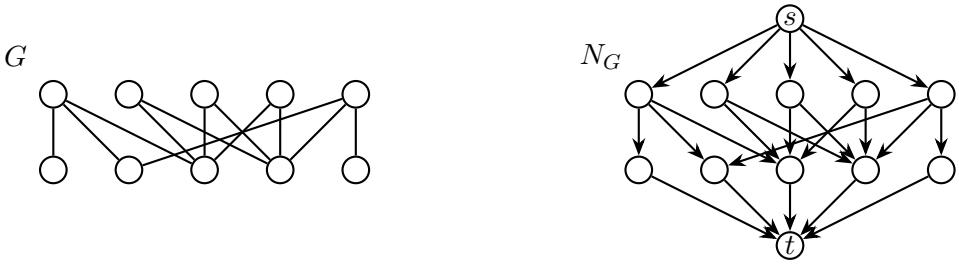


Figure 7.2: Left: bipartite graph G. Right: flow network N_G induced by G ($\mu = 1$).

7.1 Bipartite Matchings

In this lecture, we consider the maximum matching problem only on a restricted class of graphs whose vertices can be split into two sets such that every edge runs between these sets. These graphs arise whenever we are pairing objects of two types into mixed pairs, e.g., when assigning applicants to positions, men to women, operators to machines, etc.

Definition 7.4. A *bipartite* graph is an undirected graph $G = (U \cup V, E)$ with $U \cap V = \emptyset$ and $E = \delta(U)$.

We can reduce the problem of finding a maximum matching in a bipartite graph to the maximum flow problem on the following graph (see Figure 7.2).

Definition 7.5. Let $G = (U \cup V, E)$ be a bipartite graph with $s, t \notin (U \cup V)$. The *flow network* $N_G = (H, \mu, s, t)$ *induced by* G is given by $H = (V_H, E_H)$ with $V_H = U \cup V \cup \{s, t\}$ and $E_H = \{(u, v) \in U \times V \mid \{u, v\} \in E\} \cup (\{s\} \times U) \cup (V \times \{t\})$, and $\mu(e) = 1$ for all $e \in E_H$.

This construction allows us to rely on algorithms for computing maximum flows, rather than explicitly using Theorem 7.3 directly.

Theorem 7.6. In bipartite graphs, the maximum matching problem can be solved in time $\mathcal{O}(nm)$.

Proof. Observe that there is a bijection between matchings in a bipartite graph G and integral s-t-flows in N_G, such that every matching M is uniquely associated with a flow f with $|M| = |f|$. This implies that we can

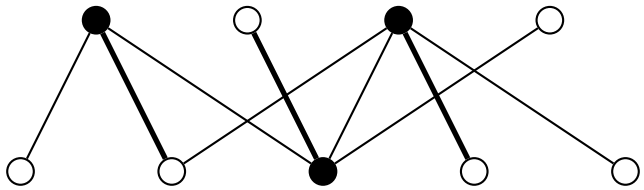


Figure 7.3: A bipartite graph with a minimum vertex cover of size three (black vertices) and a maximum matching of size three (thick edges). Observe that the vertices that are not part of the vertex cover violate the Hall condition.

find a maximum matching in G by finding a maximum s-t-flow in N_G. We can accomplish the latter using the Ford-Fulkerson method, as shown below.

Algorithm: FordFulkersonMatching(G)

input: bipartite graph $G = (U \cup V, E)$
output: maximum cardinality matching $M \subseteq E$ in G

$(H, \mu, s, t) \leftarrow N_G$
$f \leftarrow$ FordFulkerson(H, μ, s, t)
return $\{\{u, v\} \in E | (u, v) \in U \times V \wedge f((u, v)) = 1\}$

Since the network N_G has unit capacities, Theorem 6.9 implies that the above algorithm finds a maximum cardinality matching in time $\mathcal{O}(|f^\star|\, m)$, where $|f^\star|$ denotes the value of a maximum s-t-flow in N_G. The cut induced by the set $\{s\}$ has capacity $|U| \leq n$ in H, hence we have $|f^\star| \leq n$ (by Proposition 6.4), and the claimed running time of $\mathcal{O}(nm)$ follows. □

In addition to this algorithmic result, we can translate structural results from the theory of maximum flows to maximum matchings. This allows us to express the size of a maximum matching with respect to the following combinatorial object (see Figure 7.3).

Definition 7.7. A *vertex cover* of an undirected graph $G = (V, E)$ is a set $X \subseteq V$ with $\bigcup_{v \in X} \delta(v) = E$. A vertex cover is *minimum* if it has the smallest cardinality among all vertex covers.

We have already seen that the existence of an augmenting path proves that a matching is not maximum (Theorem 7.3). Conversely, we can prove that a matching is maximum by finding an s-t-cut of minimum cardinality in the flow network. This can be expressed more consisely via the following theorem.

Theorem 7.8 (Kőnig)**.** In a bipartite graph G, the cardinality of a maximum matching is equal to the cardinality of a minimum vertex cover.

Proof. Consider the flow network $N_G = (H, \mu, s, t)$ induced by G. The cardinality of a maximum matching in G is equal to the maximum number of vertex disjoint s-t-paths in H. On the other hand, the cardinality of a minimum vertex cover corresponds to the number of vertices (other than s or t) that we need to remove from H to destroy all s-t-paths. The statement of the theorem thus follows from Menger's Theorem for vertex-disjoint paths (Theorem 6.18). □

With this, we can derive a characterization of bipartite graphs that admit perfect matchings. We first give a characterization that focuses on one of the parts of the bipartite graph (see Figure 7.3 for an example). A symmetrical characterization will then follow directly.

Theorem 7.9 (Hall)**.** A bipartite graph $G = (U \cup V, E)$ has a matching covering all vertices of U if and only if

$$\big|\Gamma(U')\big| \geq \big|U'\big| \quad \text{for all } U' \subseteq U.$$

Proof. Clearly, if there is a matching covering all vertices of U, then the Hall condition $|\Gamma(U')| \geq |U'|$ holds for all $U' \subseteq U$.

Now, suppose that there is no matching covering all vertices of U and let $M^\star$ denote a maximum matching in G. Then, $|M^\star| < |U|$ and König's Theorem (Theorem 7.8) yields that the cardinality of a minimum vertex cover $X^\star$ is also bounded by $|X^\star| < |U|$. Let $U' \cup V' := X^\star$ with $U' \subseteq U$ and $V' \subseteq V$. By definition of $X^\star$, we have that $X^\star$ needs to cover all edges in $\delta(U \setminus U')$, hence $\Gamma(U \setminus U') \subseteq V'$. It follows that

$$|\Gamma(U \setminus U')| \leq |V'| = |X^\star| - |U'| < |U| - |U'| = |U \setminus U'|,$$

which means that the Hall condition is violated. □

Corollary 7.10 ("Marriage" Theorem)**.** A bipartite graph $G = (U \cup V, E)$ has a perfect matching if and only if $|U| = |V|$ and $|\Gamma(U')| \geq |U'|$ for all $U' \subseteq U$.

8 Complexity

In the previous chapters we developed efficient algorithms for various algorithmic problems. In many cases we repeatedly improved the running times of our algorithms, and for most problems even more efficient algorithms are known (see lecture "Combinatorial Optimization"). Ultimately, the goal is to find an "optimal" algorithm with best-possible running time for each problem. Unfortunately, proving lower bounds on achievable running times has proved extremely challenging, and very few super-linear lower bounds are known beyond the $\Omega(n \log n)$ lower bound for sorting (Theorem 3.26).

This lack of theoretical lower bounds is particularly problematic for problems where the best currently known solutions are extremely inefficient, since we do not know whether effcient algorithms are impossible or have simply not been found yet. We typically distinguish between efficiently solvable problems that admit polynomial running times and problems that require an exponential running time (in the worst-case). While a strict separation of problems according to their inherent difficulty (or "complexity") still eludes the mathematical community, complexity theory has developed an entire hierarchy of *complexity classes* to classify and compare algorithmic problems by difficulty. This hierarchy hinges on a fundamental conjecture relating problems with efficient solutions to problems where solutions can efficiently be verified.

We will now give an introduction into the fundamental concepts of complexity theory. These will provide us with a way of showing that a specific algorithmic problem is very unlikely to admit an efficient solution. To do this, it will prove convenient to restrict ourselves to problems that only have two possible solutions. In the following, the *binary representation* of an instance is defined by a (given) bijective map $b\colon \mathcal{I} \to \{0,1\}^*$.

Definition 8.1. An algorithmic problem $(\mathcal{I}, (S_I)_{I\in\mathcal{I}})$ is called a *decision problem* if $\emptyset \subset S_I \subset \{\text{'yes'}, \text{'no'}\}$ for all $I \in \mathcal{I}$. The input size $n := |I|$ of an instance $I \in \mathcal{I}$ of a decision problem is defined to be equal to the number of bits in the binary representation of I.

Example. The following are examples for decision problems:

MST Problem
input: undirected graph $G = (V, E)$, weights $c\colon E \to \mathbb{R}_{\geq 0}$, number $C \in \mathbb{R}_{\geq 0}$
problem: Is there a spanning tree T of G with $\sum_{e\in T} c(e) \leq C$?

VERTEXCOVER Problem
input: undirected graph $G = (V, E)$, integer $k \in \mathbb{N}$
problem: Is there a vertex cover $X \subseteq V$ in G with $|X| \leq k$?

MATCHING Problem
input: undirected graph $G = (V, E)$, integer $k \in \mathbb{N}$
problem: Is there a matching $M \subseteq E$ in G with $|M| \geq k$?

GENERALIZEDCHESS Problem
input: chess pieces on a $n \times n$ chessboard
problem: Can the active player guarantee victory?

Remark 8.2. Informally, we distinguish between decision problems, where we simply need to decide feasibility (e.g., is there a path of length at most L?), optimization problems, where we need to find the optimum value (e.g., what is the length of a shortest path?), and search problems, where we need to find an optimal object (e.g., find a shortest path!). We can often obtain a solution for the optimization problem from a solution for the decision problem by binary search for the best feasible value, and we can obtain a solution for the search problem from a solution for the optimization problem by backtracking or dynamic programming. In that sense, if we are only interested in the question whether a problem is polynomially solvable, we can often restrict ourselves to the decision problem without loss of generality.

We can now introduce our first complexity class that contains all efficiently solvable problems.

Definition 8.3. The complexity class P consists of all decision problems that can be solved in polynomial time.

Example. All problems we have considered in the previous chapters can be formulated as decision problems. For example, we can ask whether a spanning tree / path / flow / matching of a given weight / length / value / cardinality exists. As we have seen, all these problems are in P.

The second complexity class we will be interested in consists of all problems for which it can be proven efficiently that the solution for an instance is 'yes' (we do not require this for the 'no'-case). To formally define this class, we first need to make the notion of efficient proofs precise.

Definition 8.4. Let $\Pi = (\mathcal{I}, (S_I)_{I\in\mathcal{I}})$ and $\Pi' = (\mathcal{I}', (S'_{I'})_{I'\in\mathcal{I}'})$ be decision problems with $\mathcal{I}' = \mathcal{C} \times \mathcal{I}$ for some set $\mathcal{C}$. We say that Π' *certifies* Π if, for every $I \in \mathcal{I}$, we have $S_I = \{\text{'yes'}\}$ if and only if there exists $c \in \mathcal{C}$ with $S'_{(c,I)} = \{\text{'yes'}\}$. In that case, we call c a *certificate* for I.
We say that Π can be *polynomially verified* if Π' can be chosen such that $\Pi' \in$ P and there is a polynomial p such that every $I \in \mathcal{I}$ with $S_I = \{\text{'yes'}\}$ has a certificate of size at most $p(|I|)$.

With this, we can introduce the following complexity class.

Definition 8.5. The complexity class NP is the set of all decision problems that can be polynomially verified.

Example. The VertexCover problem is in NP. To see this, let $\mathcal{I}$ be the set of all instances of VertexCover and let $\mathcal{C} = 2^{\mathbb{N}}$. There is a polynomial algorithm that, given a set $c \in \mathcal{C}$, a graph $G = (V = \{v_1, \dots, v_n\}, E)$ and an integer $k \in \mathbb{N}$, decides whether $X_c := \{v_i \,|\, i \in c\}$ is a vertex cover in G of size at most k: Obviously, such an algorithm exists, since we can check in linear time whether every edge of G intersects X_c and whether $|X_c| \leq k$. Now, let $\mathcal{I}$ be the set of instances of VertexCover and let $\Pi' = (\mathcal{I}', (S')_{I'\in\mathcal{I}'})$ with $\mathcal{I}' = \mathcal{C} \times \mathcal{I}$ be the decision problem, such that, for all $I = (G, k) \in \mathcal{I}$, we have $S'_{(c,I)} = \{\text{'yes'}\}$ if and only if X_c is a vertex cover of size at most k in G. By definition, for $I = (G, k) \in \mathcal{I}$ there is a $c \in \mathcal{C}$ with $S'_{(c,I)} = \{\text{'yes'}\}$ if and only if G has a vertex cover of size k. Hence, Π' certifies Π. Moreover, we have $|c| \leq |I|$ and $\Pi' \in$ P, hence VertexCover can be polynomially verified and VertexCover $\in$ NP.

Remark 8.6. Almost always the certificate that proves inclusion of a problem in the class NP is simply given by a solution to the corresponding search problem. This is true in the example above, where a certificate is simply a vertex cover of the required size, and it will be true in all proofs below.

It is immediate that every problem in P is also in NP, since we can simply reuse its polynomial time algorithm to verify (both 'yes' and 'no') instances. We will see other problems below that fall into NP but are believed to lie outside P.

Observation 8.7. It holds that P $\subseteq$ NP.

Proof. This follows by definition, since every decision problem verifies itself (e.g., with $\mathcal{C} = \{\emptyset\}$). □

Remark 8.8. The name of the class NP stands for "**n**ondeterministic **p**olynomial", because it contains exactly those decision problems that can be solved in polynomial time by a nondeterministic algorithm that is allowed to "guess" what to do in every step and always guesses correctly. Intuitively, such an algorithm can simply guess the polynomial certificate of a "yes"-instance, and, conversely, we can use the outcomes of its (polynomially many) guesses as a certificate.

Remark 8.9. Note that there are (decision) problems that provably lie outside the class NP (for example GeneralizedChess). For a discussion of this domain of the complexity landscape, we refer to a class on complexity theory. Also note that there even are *undecidable* problems, for which no algorithm can exist at all that always terminates and yields the correct answer!

The distinction between P and NP lies at the heart of complexity theory. We will see how to show that a problem is "hardest possible" within NP, which is considered a (very) strong indication for a problem not being polynomial time solvable. However, it is still an open problem to prove (or disprove) that P $\neq$ NP. In fact this problem is part of the famous list of the millenium problems published by the Clay institute:

- Yang-Mills and Mass Gap
- Riemann Hypothesis
- **P vs NP Problem**
- Navier-Stokes Equation
- Hodge Conjecture
- Poincaré Conjecture (the only solved problem of this list)
- Birch and Swinnerton-Dyer Conjecture

8.1 NP-completeness

We have seen examples, where we were able to solve an algorithmic problem by reformulating it as a special case of another problem, e.g., we solved the maximum bipartite matching problem by formulating it as a flow problem. If this can be done with little overhead, it is justified to consider the former problem to not be harder than the latter. Since we are only interested in distinguishing polynomial time solvable problems from problems that cannot be solved in polynomial time, we can allow a polynomial effort for the transformation between problems. We now make this precise.

Definition 8.10. We say that an algorithmic problem $\Pi = (\mathcal{I}, (S_I)_{I \in \mathcal{I}})$ is *polynomial time reducible to* problem $\Pi' = (\mathcal{I}', (S'_{I'})_{I' \in \mathcal{I}'})$ if there is a function $R \colon \mathcal{I} \to \mathcal{I}'$ that satisfies $S_I = S'_{R(I)}$ for all $I \in \mathcal{I}$ and can be computed in time polynomial in $|I|$.

Observation 8.11. If a decision problem Π is polynomially reducible to $\Pi' \in$ P, then $\Pi \in$ P.

This notion gives us a partial ordering by difficulty between problems outside of P. In particular, we can introduce the class of problems that are "hardest" within the class NP.

Definition 8.12. An algorithmic problem Π is *NP-hard* if all $\Pi' \in$ NP are polynomial time reducible to Π. If, additionally, $\Pi \in$ NP, then Π is *NP-complete*.

The above definition seems very restrictive, and it is not clear a priori that any NP-complete problems exist. Surprisingly, it turns out that there are a lot of NP-complete problems. The first problem for which NP-completeness was established is the following:

Definition 8.13. Let $\mathcal{X} = \{x_i\}_{i=1,\dots,n}$ be a set of variables. We call $\Lambda = \{x, \bar{x} | \, x \in \mathcal{X}\}$ the set of *literals* over $\mathcal{X}$ and $C = \lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k$ with $\lambda_i \in \Lambda$ for $i \in \{1, \dots, k\}$ a *clause* over $\mathcal{X}$ of size $|C| = k$. A CNF (conjunctive normal form) formula is a Boolean formula of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

that is given by a set of clauses $\mathcal{C} = \{C_i\}_{i=1,\dots,m}$ over a set of variables $\mathcal{X}$. An *assignment* for a CNF formula is a function $\alpha \colon \mathcal{X} \to \{0,1\}$, and α is called *satisfying* if $\sum_{\lambda \in C} \alpha(\lambda) \geq 1$ for all $C \in \mathcal{C}$ with $\alpha(\lambda) := 1 - \alpha(\bar{\lambda})$ for $\lambda \in \Lambda \setminus \mathcal{X}$.

Satisfiability (SAT) Problem
input: CNF formula given by *clauses* $\mathcal{C} = \{C_i\}_{i=1,\dots,m}$ over *variables* $\mathcal{X} = \{x_i\}_{i=1,\dots,n}$
problem: Is there a *satisfying assignment* $\alpha \colon \mathcal{X} \to \{0,1\}$?

Example 8.14. The formula

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$$

has the satisfying assignment $\alpha(x_1) = 1$, $\alpha(x_2) = 1$, $\alpha(x_3) = 0$. The formula

$$(x_1 \vee x_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee \bar{x}_3)$$

is unsatisfiable.

It was a breakthrough and the birth of complexity theory when Cook (1971) showed that every problem in NP can be reduced to deciding whether a SAT formula admits a satisfying assignment. Roughly speaking, Cook showed that the outcome of an algorithm can be encoded as a single Boolean formula.

Theorem 8.15 (Cook)**.** SAT is NP-complete.

Corollary 8.16. There is no polynomial time algorithm for SAT, unless P = NP.

Knowing an NP-complete problem gives us a much simpler way of establishing NP-completeness for other problems: Instead of showing that *all* problems in NP can be reduced to the problem, it suffices to show that the NP-complete problem reduces to it.

Observation 8.17. If $\Pi \in$ NP is NP-complete and polynomial time reducible to $\Pi' \in$ NP, then Π' is NP-complete as well.

This simple oberservation has allowed to establish many (thousands of) additional NP-hard problems. The fact that not a single polynomial time algorithm has been found for any of these problems is seen as a strong indication that NP $\neq$ P, because of the following fact.

Observation 8.18. There is an NP-hard problem Π in P if and only if P = NP.

Based on this observation, it is customary to show NP-hardness (or -completeness) of a problem as a way of establishing that it is extremely unlikely that the problem admits a polynomial time algorithm. This justifies making compromises regarding running time, e.g., by using exponential time algorithms, and/or regarding exactness, e.g., by using heuristics or approximation algorithms (see lecture "Discrete Optimization").

8.2 Important NP-complete problems

We will now prove NP-completeness for some of the historically most important decision problems. Every proof of NP-completeness for a decision problem $\Pi' = (\mathcal{I}', (S'_{I'})_{I' \in \mathcal{I}'})$ that is based on Obeservation 8.17 needs to address the following aspects:

(i) Proof that $\Pi' \in$ NP.

(ii) Decision problem $\Pi = (\mathcal{I}, (S_I)_{I \in \mathcal{I}})$ that we are reducing from.

(iii) Construction of an instance $I' \in \mathcal{I}'$ for every instance $I \in \mathcal{I}$.

(iv) Equivalence of I' and I, i.e., $S_{I'} = \{$'yes'$\}$ if and only if $S_I = \{$'yes'$\}$.

(v) Proof that our construction can be carried out in time polynomial in $|I|$.

We first consider two important variants of SAT that are often useful for reductions.

3SAT Problem
input: instance $\mathcal{C} = \{C_i\}_{i=1,\dots,m}$, $\mathcal{X} = \{x_i\}_{i=1,\dots,n}$ of SAT with $|C| = 3$ for all $C \in \mathcal{C}$
problem: Is there a *satisfying assignment*?

Theorem 8.19. 3SAT is NP-complete.

Proof. From SAT $\in$ NP it follows that 3SAT $\in$ NP, since every instance of 3SAT is an instance of SAT.

We reduce SAT to 3SAT by replacing every clause $C = \bigvee_{i=1}^{k} \lambda_i$ of length $k \neq 3$ by an equivalent 3SAT formula Z of polynomial length (in k). To this end, we introduce new variables $\{y_i\}_{i=1,\dots,k+1}$. If $C = \lambda_1$, we set

$$Z = (\lambda_1 \vee y_1 \vee y_2) \wedge (\lambda_1 \vee \bar{y}_1 \vee y_2) \wedge (\lambda_1 \vee y_1 \vee \bar{y}_2) \wedge (\lambda_1 \vee \bar{y}_1 \vee \bar{y}_2),$$

and, if $C = \lambda_1 \vee \lambda_2$, we set

$$Z = (\lambda_1 \vee \lambda_2 \vee y_1) \wedge (\lambda_1 \vee \lambda_2 \vee \bar{y}_1).$$

It is easy to see that in either case Z is satisfied by an assignment $\alpha \colon \mathcal{X} \to \{0,1\}$ if and only if $\alpha(\lambda_i) = 1$ for some $i \in \{1, k\}$.

If $k > 3$, we set

$$Z = (\lambda_1 \vee \lambda_2 \vee y_1) \wedge (\bar{y}_1 \vee \lambda_3 \vee y_2) \wedge \cdots \wedge (\bar{y}_{k-4} \vee \lambda_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee \lambda_{k-1} \vee \lambda_k).$$

Now, Z can be satisfied by setting $\alpha(\lambda_i) = 1$ for any $i \in \{1, \dots, k\}$ and $\alpha(y_j) = 1$ for all $j \leq i-2$ and $\alpha(y_j) = 0$ for all $j \geq i-1$. On the other hand, we claim that there is no satisfying assignment α with $\alpha(\lambda_i) = 0$ for all $i \in \{1, \dots, k\}$. For the sake of contradiction, assume such an assignment exists. Then the first clause of Z forces $\alpha(y_1) = 1$, which means that the second clause forces $\alpha(y_2) = 1$, etc. The last clause cannot be satisfied anymore. □

We can use this variant of satisfiability to obtain NP-hardness for various fundamental graph problems. As a starting point, we consider the following problem.

Definition 8.20. A set of vertices $X \subseteq V$ is *stable* (or *independent*) in an undirected graph $G = (V, E)$ if $G[X] = (X, \emptyset)$.

StableSet Problem	
input:	undirected graph $G = (V, E)$, integer $k \in \mathbb{N}$
problem:	Is there a stable set $X \subseteq V$ in G with $\lvert X \rvert \geq k$?

Theorem 8.21. StableSet is NP-complete.

Proof. It is easy to check in polynomial time whether a given set of vertices is stable. This means that we can use the stable set X of size $|X| \geq k$ as a polynomial certificate for a 'yes'-instance of StableSet. Hence StableSet $\in$ NP.

We reduce 3Sat to StableSet. Let $Z = \bigwedge_{i=1}^{m} \bigvee_{j=1}^{3} \lambda_{ij}$ be a 3Sat formula. We construct a graph $G = (V, E)$ with $3m$ vertices that has a stable set of size m if and only if Z is satisfiable. We let $V = \{1, \dots, m\} \times \{1, 2, 3\}$ and introduce edges $\{(i, j), (i', j')\} \in E$ for every $\lambda_{ij} = \bar{\lambda}_{i'j'}$, as well as the edges $\{(i, 1), (i, 2)\}, \{(i, 2), (i, 3)\}, \{(i, 3), (i, 1)\} \in E$. Observe that our construction can be carried out in polynomial time.

Suppose there is a satisfying assignment α for Z and choose j_i such that $\alpha(\lambda_{ij_i}) = 1$ for all $i \in \{1, \dots, m\}$. Then, by construction, $\{(i, j_i)\}_{i \in \{1, \dots, m\}}$ is a stable set of size m in G.

Conversely, let X be a stable set of size m in G. Then, $|\{(i, 1), (i, 2), (i, 3)\} \cap X| \leq 1$ for all $i \in \{1, \dots, m\}$, since $\{(i, 1), (i, 2)\}, \{(i, 2), (i, 3)\}, \{(i, 3), (i, 1)\} \in E$. Since $|X| = m$, this implies $|\{(i, 1), (i, 2), (i, 3)\} \cap X| = 1$ for all $i \in \{1, \dots, m\}$. We can find an assignment α with $\alpha(\lambda_{ij}) = 1$ for all $(i, j) \in X$, since no two pairs (i, j), (i', j') corresponding to literals $\lambda_{ij} = \bar{\lambda}_{i'j'}$ can be in X by construction of G. This way, at least one literal of each clause is set to 1, and hence α is a satisfying assignment for Z. □

We have already seen that VertexCover is polynomial time solvable on bipartite graphs by Königs Theorem (Theorem 7.8 and Theorem 7.6). In contrast, the problem turns out to be NP-complete on general graphs (even though the matching problem remains polynomial time solvable, see lecture "Combinatorial Optimization").

Corollary 8.22. VertexCover is NP-complete.

Proof. It is easy to check in polynomial time whether a given set $X \subseteq V$ is a vertex cover, hence VertexCover $\in$ NP. We can trivially reduce StableSet to VertexCover, since X is a stable set in a graph $G = (V, E)$ if and only if $V - X$ is a vertex cover, i.e., there is a stable set of size at least k if and only if there is a vertex cover of size at most $n - k$. □

Another important covering problem where three disjoint sets have to be covered by selecting triples is the following:

3DMATCHING Problem
input: disjoint sets A, B, C, triples $T \subseteq A \times B \times C$
problem: Is there a set of disjoint triples $M \subseteq T$ with $\bigcup_{t \in M} t = A \cup B \cup C$?

Theorem 8.23. 3DMATCHING is NP-complete.

Proof. Obviously, 3DMATCHING $\in$ NP, since we can efficiently check whether $\bigcup_{t \in M} t = A \cup B \cup C$ and that all triples in M are disjoint for given $M \subseteq T$.

We reduce 3SAT to 3DMATCHING. Let Z be 3SAT formula over variables $\mathcal{X} = \{x_1, \dots, x_n\}$ and clauses $\mathcal{C} = \{C_1, \dots, C_m\}$, and let $k \in \mathbb{N}$ be the smallest number such that no literal appears more than k times in Z.

We describe sets A, B, C and triples $T \subseteq A \times B \times C$ that permit a disjoint covering of $A \cup B \cup C$ if and only if Z is satisfiable. We let $A := \{ x_{ij}, \bar{x}_{ij} | \, x_i \in \mathcal{X}, j \in \{0, \dots, k-1\}\}$.

For every variable $x_i \in \mathcal{X}$ and every $j \in \{0, \dots, k-1\}$, we introduce the elements $b_{ij} \in B$ and $c_{ij} \in C$ and the triples $\{(x_{ij}, b_{ij}, c_{ij}), (\bar{x}_{ij}, b_{ij}, c_{i,j+1 \bmod k})\} \subseteq T$. These will be the only triples containing b_{ij}, c_{ij}, hence every solution of 3DMATCHING must select exactly k of these triples. Depending on the selected triples, either all elements x_{ij} or all elements $\bar{x}_{ij}$ are not covered, and we interpret the corresponding choices as setting $\alpha(x_i) = 1$ an $\alpha(x_i) = 0$, respectively.

For every clause $C_\ell \in \mathcal{C}$ we introduce the elements $d_\ell \in B$ and $e_\ell \in C$, as well as three triples with these two elements: For every literal $\lambda \in C_\ell$ with $\lambda = x_i$ we introduce one of the triples (x_{ij}, d_ℓ, e_ℓ), such that every element x_{ij} is contained in at most one such triple (this is possible by definition of k). Similarly, for every literal $\lambda = \bar{x}_i$ we introduce one of the triples $(\bar{x}_{ij}, d_\ell, e_\ell)$, such that every element x_{ij} is contained in at most one such triple. The choice of one of the three triples of a clause in a solution corresponds to the decision which literal should satisfy the clause. Observe that, since every element of A may only be contained in exactly one triple in a solution to 3DMATCHING, this choice of a literal needs to be consistent with the variable assignment.

An assignment α satisfies Z if and only if the corresponding selection of triples leaves $kn - m$ elements of A uncovered. To complete the construction, we introduce additional pairs $(p_i, q_i) \in B, C$ for $i \in \{1, \dots, kn - m\}$ that ensure that all elements can be covered. For every element $a \in A$ and each of these pairs, we introduce all triples (a, p_i, q_i) for $i \in \{1, \dots, 2k - m\}$. Every solution to 3DMATCHING must cover all elements p_i, q_i, which can be accomplished exactly by coving $kn - m$ arbitrary elements of A.

Overall, the constructed instance of 3DMATCHING has a solution (i.e., the solution to the decision problem is {'yes'}) if and only if Z has a satisfying assignment α: Each such assignment uniquely fixes a choice of triples, and vice-versa. Furthermore, our construction can be carried out in polynomial time, since $|A| = |B| = |C| = 2kn$ and $|T| = 2kn + 3m + (kn - m) \cdot 2kn = \mathcal{O}(m^2 n^2)$. □

Finally, many optimization problems can be expressed as variants of the following problem. Intuitively, the complexity of this problem comes from its rigidity: we have to achieve an exact sum of numbers, and a solution that is off the target sum by very little may still be completely different from any exact solution.

SUBSETSUM Problem
input: numbers $A = (a_i \in \mathbb{N})_{i=1,\dots,n}$, integer $K \in \mathbb{N}$
problem: Is there a subset $S \subseteq \{1, \dots, n\}$ with $\sum_{i \in S} a_i = K$?

Theorem 8.24. SUBSETSUM is NP-complete.

Proof. Obviously SUBSETSUM $\in$ NP since we can check in polynomial time whether $\sum_{i \in I} a_i = K$ for a given set S.

We reduce 3DMATCHING to SUBSETSUM. Let (A, B, C, T) be an instance of 3DMATCHING with $\{u_0, \ldots, u_{m-1}\} := A \cup B \cup C$ and $\{T_1, T_2, \ldots, T_n\} := T$. We construct an instance of SUBSETSUM with

$$\begin{aligned} a_j &:= \sum_{u_i \in T_j} (n+1)^i, \\ K &:= \sum_{i=0}^{m-1} (n+1)^i. \end{aligned}$$

Represented in basis $(n+1)$, the number a_j has a 1 exactly in the digits that correspond to the elements in the triple T_j, all other digits are equal to 0. In this representation, all digits of K are exactly 1. Since there are only n different numbers a_j, there cannot be a carry-over in any digit with respect to basis $(n+1)$ when summing a subset of these numbers. Hence, every solution to our SUBSETSUM instance corresponds to a solution to the given 3DMATCHING instance, since all elements need to be covered and no element may be covered more than once. In the same way, every solution to the 3DMATCHING instance corresponds to a solution to the SUBSETSUM instance. Every number has $\mathcal{O}(m)$ digits in basis $(n+1)$, thus its binary representation has $\mathcal{O}(m \log n)$ bits. This means that our construction can be carried out in polynomial time. □