2 Algorithm design

Among the general algorithm design techniques, *dynamic programming* is the one most heavily used in this book. Since great parts of Chapters 4 and 6 are devoted to introducing this topic in a unified manner, we shall not introduce this technique here. However, a taste of this technique is already provided by the solution to Exercise 2.2 of this chapter, which asks for a simple one-dimensional instance.

We will now cover some basic primitives that are later implicitly assumed to be known.

2.1 Complexity analysis

The algorithms described in this book are typically analyzed for their *worst-case* running time and space complexity: complexity is expressed as a function of the input parameters on the worst possible input. For example, if the input is a string of length n from an alphabet of size σ, a *linear time* algorithm works in $O(n)$ time, where $O(\cdot)$ is the familiar big-O notation which hides constants. This means that the running time is upper bounded by cn elementary operations, where c is some constant.

We consider the alphabet size not to be a constant, that is, we use expressions of the form $O(n\sigma)$ and $O(n \log \sigma)$, which are not linear complexity bounds. For the space requirements, we frequently use notations such as $n \log \sigma (1 + o(1)) = n \log \sigma + o(n \log \sigma)$. Here $o(\cdot)$ denotes a function that grows asymptotically strictly slower than its argument. For example, $O(n \log \sigma / \log \log n)$ can be simplified to $o(n \log \sigma)$. Algorithms have an input and an output: by *working space* we mean the extra space required by an algorithm in addition to its input and output. Most of our algorithms are for processing DNA, where $\sigma = 4$, and it would seem that one could omit such a small constant in this context. However, we will later see a number of applications of these algorithms that are not on DNA, but on some derived string over an alphabet that can be as large as the input sequence length.

We often assume the alphabet of a string of length n to be $\Sigma = [1..\sigma]$, where $\sigma \leq n$. Observe that if a string is originally from an *ordered alphabet* $\Sigma \subseteq [1..u]$, with $\sigma = |\Sigma|$, it is easy to map the alphabet and the character occurrences in the string into alphabet $[1..\sigma]$ in $O(n \log \sigma)$ time using, for example, a *binary search tree*: see Section 3.1. Such mapping is not possible with an *unordered alphabet* Σ that supports only the operation $a = b?$ for characters $a, b \in \Sigma$: comparison $a < b?$ is not supported.

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

Yet still, the running time of some algorithms will be $O(dn^c)$, where c is a constant and d is some integer given in the input that can take arbitrarily large values, independently of the input size n. In this case we say that we have a *pseudo-polynomial* algorithm. More generally, a few algorithms will have running time $O(f(k)n^c)$, where c is a constant and $f(k)$ is an arbitrary function that does not depend on n but depends only on some other – typically smaller – parameter k of the input. In this case, we say that we have a *fixed-parameter tractable* algorithm, since, if k is small and $f(k)$ is sufficiently slow-growing, then this algorithm is efficient, thus the problem is *tractable*.

In addition to the notations $o(\cdot)$ and $O(\cdot)$, we will also use the notations $\Theta(\cdot)$, $\omega(\cdot)$, and $\Omega(\cdot)$ defined as follows. By $\Theta(\cdot)$ we denote a function that grows asymptotically as fast as its argument (up to a constant factor). Whenever we write $f(x) = \Theta(g(x))$, we mean that $f(x) = O(g(x))$ and $g(x) = O(f(x))$. The notions $\omega(\cdot)$ and $\Omega(\cdot)$ are the inverses of $o(\cdot)$ and $O(\cdot)$, respectively. Whenever we write $f(x) = \omega(g(x))$, we mean that $g(x) = o(f(x))$. Whenever we write $f(x) = \Omega(g(x))$, we mean that $g(x) = O(f(x))$.

None of the algorithms in this book will require any complex analysis technique in order to derive their time and space requirements. We mainly exploit a series of combinatorial properties, which we sometimes combine with a technique called *amortized analysis*. In some cases it is hard to bound the running time of a single step of the algorithm, but, by finer counting arguments, the total work can be shown to correspond to some other conceptual steps of the algorithm, whose total amount can be bounded. Stated differently, the time "costs" of the algorithm can all be "charged" to some other bounded resource, and hence "amortized". This technique deserves a concrete example, and one is given in Example 2.1.

Example 2.1 Amortized analysis – Knuth–Morris–Pratt

The *exact string matching* problem is that of locating the occurrences of a *pattern* string $P = p_1p_2\cdots p_m$ inside a *text* string $T = t_1t_2\cdots t_n$. The Morris–Pratt (MP) algorithm solves this problem in optimal linear time, and works as follows. Assume first that we already have available on P the values of a *failure function*, $\texttt{fail}(i)$, defined as the length of the longest prefix of $p_1p_2\cdots p_i$ that is also a suffix of $p_1p_2\cdots p_i$. That is, $\texttt{fail}(i) = i'$, where $i' \in [0\,..\,i-1]$ is the largest such that $p_1p_2\cdots p_{i'} = p_{i-i'+1}p_{i-i'+2}\cdots p_i$. The text T can be scanned as follows. Compare p_1 to t_1, p_2 to t_2, and so on, until $p_{i+1} \neq t_{j+1}$, where j is an index in T and equals i throughout the first iteration. Apply $i = \texttt{fail}(i)$ recursively until finding $p_{i+1} = t_{j+1}$, or $i = 0$; denote this operation by $\texttt{fail}^*(i, t_{j+1})$. Then, continue the comparisons, by again incrementing i and j, and again resetting $i = \texttt{fail}^*(\cdot,\cdot)$ in case of mismatches. An occurrence of the pattern can be reported when $i = |P| + 1$.

To analyze the running time, observe first that each character t_j of T is checked for equality with some character of P exactly once. Thus, we need only estimate the time needed for the recursive calls $\texttt{fail}^*(i, t_{j+1})$. However, without a deeper analysis, we can only conclude that each recursive call takes $O(m)$ time before one can increment j, which would give an $O(mn)$ bound for the time complexity of the algorithm. In other words, it is *not* enough to consider the worst-case running time of each step independently, and then just sum the results. However, we can show that the total

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

number of recursive calls is at most the length of T, thus each recursive call can be "charged" to each element of T.

Observe that if $\texttt{fail}^*(i, t_{j+1}) = i'$, then there are at most $i - i'$ recursive calls done for t_{j+1}. We "charge" these calls to the $i - i'$ characters $t_{j-i+1}t_{j-i+2}\cdots t_{j-i'}$ of T, characters which matched the prefix $p_1 \cdots p_{i-i'}$. We need only show that each t_j gets charged by at most one recursive call, which will allow us to conclude that the total number of recursive calls is bounded by n. To see this, observe that after setting $i' = \texttt{fail}^*(i, t_{j+1})$ we have $p_1 \cdots p_{i'} = t_{j-i'+1} \cdots t_j$, hence new recursive calls get charged to positions from $j - i' + 1$ onwards, since this is where a prefix of P again starts matching.

We still need to consider how to compute the values of the $\texttt{fail}(\cdot)$ function in $O(m)$ time; this is considered in Exercise 2.1.

The Knuth–Morris–Pratt (KMP) algorithm is a variant of the MP algorithm, where the $\texttt{fail}(\cdot)$ function is optimized to avoid some unnecessary comparison: see Exercise 2.2. We will use the KMP algorithm and its combinatorial properties in Chapter 12. For those developments, the concept of a *border* is useful: the suffix $B = P_{m-i+1..m}$ of $P[1..m]$ is called a border if it is also a prefix of P, that is, if $B = P_{1..i}$. The length of the *longest border* is given by $i = \texttt{fail}(m)$, and all other border lengths can be obtained by applying $\texttt{fail}(\cdot)$ recursively, where $\texttt{fail}(\cdot)$ is now referring to the MP algorithm.

The $\texttt{fail}(\cdot)$ function is often referred to as a *(K)MP automaton*.

The KMP algorithm from the above example can be useful only if the text in which the pattern is being sought is relatively short. The search time will be huge if one has to search for a sequence inside a fairly large genome. Thus, KMP cannot be used for genome-scale algorithms.

In order to support fast searches, it is necessary to have an *index* of the text. This will be in contrast to the KMP algorithm which, through the $\texttt{fail}(\cdot)$ function, indexed the pattern. In Chapter 8 we will study various indexing data structures, such as suffix trees and suffix arrays, which use $O(n)$ words of space for a text consisting of n characters, and which find occurrences of a pattern of length m in time $O(m)$. This is opposed to the KMP algorithm, which uses $O(m)$ words of space, but has a running time of $O(n)$. Still, for large genomes, another problem may arise, since the space usage $O(n)$ words might be too big to fit into the memory. In Chapter 9 we will introduce the concept of succinct text indexes, which can save an order of magnitude of space while still allowing fast queries. We introduce the concept of succinct data structures in the next section.

2.2 Data representations

We assume the reader is familiar with the elementary pointer-manipulation data structures such as *stacks* and *queues*, represented by doubly-linked lists, and with representations of *trees*. We also assume that the reader is familiar with bitvector manipulation routines (left/right shift, logical or/and) that give the possibility of allocating *arrays*

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

with fixed-length *bit-fields*. If this is not the case, the assignments in the exercise section provide a crash course.

We emphasize that, throughout the book, we express all space requirements in *computer words*, except when we aim at optimizing the space needed by some large data structure, where we always add "bits of space" to make it clear. We use the *random access model* (RAM) to describe the content, with a model in which the computer word size w is considered to be $\Omega(\log U + \log n)$ bits, where U is the maximum input value and n the problem size. Basic arithmetic, logical, and bit-manipulation operations on words of length w bits are assumed to take constant time. For example, an array $A[1..U]$ with each $A[i] \in [1..U]$, can be represented in $U(\lfloor \log_2 U \rfloor + 1)$ bits (which we often simplify to $U \log U$), in the fixed-length bit-fields representation (see Exercise 2.7). A static tree of n nodes can be stored as a constant number of arrays of length n (Exercise 2.8), leading to $O(n \log n)$-bit representation. Even a dynamic tree can be maintained using $O(n \log n)$ bits, with tailor-made memory-allocation routines exploiting *dynamic tables*. We do not need to touch on this area in this book, since we have chosen to modify the algorithms to make use instead of *semi-dynamic structures*, in which elements are inserted at the beginning and updated later.

We often talk about succinct data structures. Such structures occupy space equal to the storage required for representing the input, plus a lower-order term. More formally, suppose that the input to the data structure is an element X taken from a set of elements $\mathcal{U}$. Then, a *succinct* data structure on X is required to occupy $\log |\mathcal{U}|(1 + o(1))$ bits of space. For example, a text T of length n from an alphabet Σ of size σ is an element of the set $U = \Sigma^n$ of all strings of length n over alphabet Σ. A data structure that takes as input T is succinct if it uses $n \log \sigma (1 + o(1))$ bits of space.

If a data structure built on T can be used to restore T after T itself has been deleted, we say that the data structure is a *representation* of T. If the same data structure occupies $n \log \sigma (1 + o(1))$ bits of space, we call it a *succinct representation* of T.

Observe that succinct space is the information-theoretic minimum, since otherwise two inputs would necessarily need to have the same representation. However, there could be *compressed* representations that occupy less than succinct space on *some inputs*. Many of the succinct data structures and representations covered in the book can be made compressed, but, for clarity, we do not explore this wide area of research systematically; this topic is touched on only in Section 10.7.1 and Chapter 12.

2.3 Reductions

One important algorithm design technique is to exploit connections between problems. There are some fundamental problems that have been studied for decades and sophisticated algorithms have already been developed for them. When formulating a new problem, for example, motivated by an analysis task in high-throughput sequencing, one should first look to see whether a solution to some widely studied problem could be used to solve the newly formulated problem. Sometimes one can exploit such a connection by modifying an algorithm for the old problem to solve the new problem, but, if the

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

connection is close enough, one may even be able to *reduce* the new problem to the old problem.

This reduction means that any input to the new problem can be recast as an input of the old problem such that, on solving the old problem with some algorithm, one solves also the new problem. The running time of such an algorithm for the new problem is then the combined running time of recasting the input as an input to the old problem, the running time of the algorithm for the old problem, and then the time needed for recasting this output as an output for the original problem. Since we frequently exploit this kind of reduction in this book, we illustrate one such reduction in Example 2.2.

Example 2.2 Reduction to a tractable problem – longest increasing subsequence

Let the old problem be the one of finding the *longest common subsequence (LCS)* of two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$. For example, with

- $A = \texttt{ACAGTGA}$
- $B = \texttt{CCTGTAG}$

the answer is $C = \texttt{CGTA}$, since C is a subsequence of both strings, meaning that C can be obtained by deleting zero or more characters from A and from B, and there is no common subsequence of length 5 or more. We study this problem in Section 6.2, where we give an algorithm running in time $O(|M| \log \sigma)$, where M is the set of position pairs (i, j) of A and B with $a_i = b_j$, and σ is the alphabet size.

Let the new problem be the one of finding the *longest increasing subsequence (LIS)* in a sequence of numbers $S = s_1, s_2, \ldots, s_k$. For example, on $S = 1, 8, 4, 1, 10, 6, 4, 10, 8$, possible answers are $1, 4, 6, 8$ or $1, 4, 6, 10$. We can reduce LIS to LCS as follows. Sort the set of distinct numbers of S in $O(k \log k)$ time and replace each occurrence of a number in S with its *rank* in this sorted order: the smallest number gets rank 1, the second smallest gets rank 2, and so on. In our example, S is replaced by a sequence of ranks $R = 1, 4, 2, 1, 5, 3, 2, 5, 4$. Clearly, any longest increasing subsequence of ranks in R can be mapped to a longest increasing subsequence of S. Now $A = R$ (taken as a string) and $B = \min(R)(\min(R) + 1) \cdots \max(R) = 123 \cdots \max(R)$ form the input to the LCS. Observe that $\max(R) \leq k$. In our example,

- $A = 142153254$
- $B = 12345$.

Any longest common subsequence of A and B defined as above can be mapped to a longest subsequence of R, and hence of S. In our example, $C = 1234$ is one such longest common subsequence, and also the longest increasing subsequence of R, and hence of S.

The $O(|M| \log \sigma)$ algorithm for LCS simplifies to $O(k \log k)$, since $\sigma \leq k$, and each index i in A $(= R)$ is paired with exactly one index j in B. The reduction itself took $O(k \log k)$ time to create, the solution to the reduced instance took the same

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

$O(k \log k)$ time, and casting the result of the LCS problem as a longest increasing subsequence in S can be done in time $O(k)$. Thus, the overall running time to solve the LIS problem is $O(k \log k)$.

Another kind of reduction will concern the *hardness* of some problems. There is a class of fundamental problems for which no efficient solutions are known. (By efficient, we mean an algorithm with *polynomial* running time with respect to the input size.) These hard problems are known only to have *exponential* time algorithms; such a running time can be achieved, for example, by enumerating through all possible answers, and checking whether one of the answers is a proper solution. A well-known example of a hard problem is the *Hamiltonian path* problem, in which one asks whether there is a route in a graph (for example, a railway network) that visits all the vertices (cities) exactly once. With n vertices, a trivial solution is to consider all $n! = \Omega(2^n)$ permutations of vertices, and to check for each one of them whether all consecutive vertices (cities) are connected by an edge (a railway line).

Assuming now that an old problem is hard, if we show that we can solve the old problem by recasting it as a new problem, by appropriately converting the input and output, then we can conclude that the new problem is at least as hard as the old problem. In this context, the running time of converting the input and output is just required to be polynomial in their sizes. Let us now describe these notions in slightly more formal terms.

Fix an alphabet Σ of size at least two. A *decision problem* is a subset of words over Σ^*. For example, if Σ is the ASCII alphabet, then under the standard convention for encoding graphs, an input to the Hamiltonian path problem is

$$w_G = (\{1,2,3,4\}, \{(1,2),(2,3),(1,3),(3,4)\}) .$$

If Π_H is the Hamiltonian path problem, $w_G \in \Pi_H$ holds. Having an algorithm for solving Π_H means that, for any input word $w \in \Sigma^*$, we can decide whether $w \in \Pi_H$.

Among all problems, one class of problems is of particular interest, called NP (nondeterministic polynomial time). A problem Π is in NP if each $w \in \Pi$ admits a "certificate" that can be checked in time polynomial in the length of w. For example, a certificate for Π_H is a permutation of the vertex set (the Hamiltonian path), which can be checked in polynomial time in the size of the vertex set to be a Hamiltonian path (see Exercises 2.4 and 2.5). A certificate for the particular instance w_G can be the sequence of vertices $w'_g = (1,2,3,4)$ or the sequence $w''_g = (2,1,3,4)$.

It is an open question whether every problem in NP can be also decided in time polynomial in the length of the input. To address this question, the following notion has been introduced: say that a problem Π is *NP-hard* if, for every problem $\Pi' \in$ NP, there is a polynomial-time algorithm that, given any $w \in \Sigma^*$, returns $f(w) \in \Sigma^*$ such that

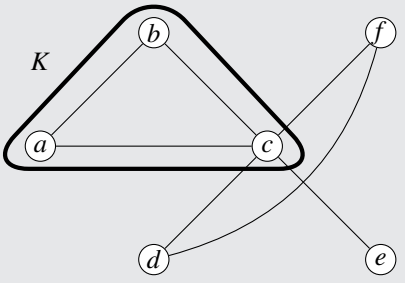
$$w \in \Pi' \Leftrightarrow f(w) \in \Pi.$$

A problem is said to be *NP-complete* if it is NP-hard and it belongs to NP. Having a polynomial-time algorithm for an NP-complete problem would imply that all problems in NP can be solved in polynomial time. Many problems are NP-complete, including also the Hamiltonian path problem mentioned above. To show that some problem Π

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

is NP-hard, it suffices to exhibit a polynomial-time reduction to Π from *any* NP-hard problem. Example 2.3 provides one illustrative NP-hardness reduction.

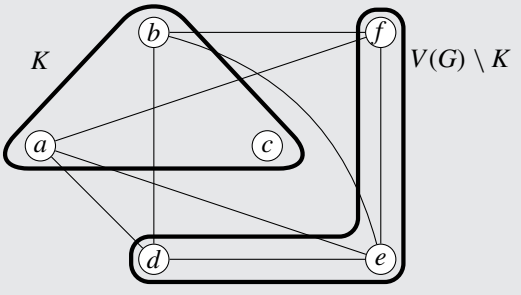
Example 2.3 Reduction from an intractable problem – Vertex Cover Let the old NP-hard problem be the *Clique* problem: given an undirected graph G and an integer k, decide whether G has a clique of size k, namely a subset K of vertices such that every two vertices in K are connected by an edge. For example, in the graph G below, $K = \{a, b, c\}$ is a clique of size 3.



G

Let the new problem be the *Vertex Cover* problem: given an undirected graph G and an integer k, decide whether G has a vertex cover of size t, namely a subset T of vertices such that every edge of G has at least one extremity in T. For example, in the graph G above, $T = \{a, c, f\}$ is a vertex cover.

We now reduce Clique to Vertex Cover. Given a graph G, whose set of vertices is V, and an integer k, as input for the Clique problem, we construct the complement graph of G, denoted $\overline{G}$, which has the same set V of vertices, and in which an edge between two vertices is present if and only if it is absent in G. Observe that, if K is a clique in G, then $V \setminus K$ is a vertex cover in $\overline{G}$. Indeed, if there were an edge in $\overline{G}$ between u and v with both u and v outside $V \setminus K$ then in G there would be no edge between u and v, contradicting the fact that K is a clique. See the figure below for an example. Conversely, if T is a vertex cover in $\overline{G}$, then $V \setminus T$ is a clique in G.



$\overline{G}$

Therefore, we can solve the Clique problem on G and k by solving the Vertex Cover problem on $\overline{G}$ and $t = |V| - k$. If the answer for the Vertex Cover problem

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

is "yes", then G has a clique of size k; if the answer is "no", then G does not have a clique of size k.

We have shown that the Vertex Cover problem is at least as hard as the Clique problem (up to a polynomial-time reduction). Since the clique problem is NP-hard, we can conclude that also the Vertex Cover problem is NP-hard. Observe also that Vertex Cover too belongs NP, since given a subset T of vertices (the "certificate") we can check in polynomial time whether it is a vertex cover. Thus, Vertex Cover is NP-complete.

2.4 Literature

The Morris–Pratt algorithm appeared in Morris & Pratt (1970), and the Knuth–Morris–Pratt algorithm in Knuth *et al.* (1977). An exercise below hints how to extend this algorithm for multiple patterns, that is, to rediscover the algorithm in Aho & Corasick (1975).

The longest increasing subsequence (LIS) problem has a more direct $O(k \log k)$ solution (Fredman 1975) than the one we described here through the reduction to the longest common subsequence (LCS) problem. As we learn in Section 6.7, the LCS problem can also be solved in $O(|M| \log \log k)$ time, where k is the length of the shorter input string. That is, if a permutation is given as input to the LIS problem such that sorting can be avoided (the rank vector is the input), the reduction gives an $O(k \log \log k)$ time algorithm for this special case. For an extensive list of NP-complete problems and further notions, see Garey & Johnson (1979).

Exercises

2.1 Consider the `fail`$(\cdot)$ function of the Morris–Pratt (MP) algorithm. We should devise a linear-time algorithm to compute it on the pattern to conclude the linear-time exact pattern-matching algorithm. Show that one can modify the same MP algorithm so that on inputs $P = p_1 p_2 \cdots p_m$ and $T = p_2 p_3 \cdots p_m \#^m$, where $\#^m$ denotes a string of m concatenated endmarkers #, the values `fail`(2), `fail`(3), …, `fail`(m) can be stored on the fly before they need to be accessed.

2.2 The Knuth–Morris–Pratt (KMP) algorithm is a variant of the MP algorithm with optimized `fail`$(\cdot)$ function: `fail`$(i) = i'$, where i' is largest such that $p_1 p_2 \cdots p_{i'} = p_{i-i'+1} p_{i-i'+2} \cdots p_i$, $i' < i$, and $p_{i'+1} \neq p_{i+1}$. This last condition makes the difference from the original definition. Assume you have the `fail`$(\cdot)$ function values computed with the original definition. Show how to update these values in linear time to satisfy the KMP optimization.

2.3 Generalize KMP for solving the *multiple pattern-matching* problem, where one is given a set of patterns rather than only one as in the exact string matching problem. The goal is to scan T in linear time so as to find exact occurrences of any pattern in the given set. *Hint.* Store the patterns in a tree structure, so that common prefixes of patterns share the same subpath. Extend `fail`$(\cdot)$ to the positions of the paths in the tree. Observe that,

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

unlike in KMP, the running time of the approach depends on the alphabet size σ. Can you obtain scanning time $O(n \log \sigma)$? Can you build the required tree data structure in $O(M \log \sigma)$ time, where M is the total length of the patterns? On top of the $O(n \log \sigma)$ time for scanning T, can you output all the occurrences of all patterns in linear time in the output size?

2.4 Show that a certificate for the Hamiltonian path problem can be checked in time $O(n)$ (where n is the number of vertices) assuming an adjacency representation of the graph that uses $O(n^2)$ bits. *Hint.* Use a table of n integers that counts the number of occurrences of the vertices in the given certificate.

2.5 Suppose that we can afford to use no more than $O(m)$ space to represent the adjacency list. Show that a certificate for the Hamiltonian path can now be checked in time $O(n \log n)$.

2.6 Find out how bit-manipulation routines are implemented in your favorite programming language. We visualize below binary representations of integers with the most-significant bit first. You might find useful the following examples of these operations:

- *left-shift*: $0000000000101001 << 2 = 0000000010100100$,
- *right-shift*: $0000000010100100 >> 5 = 0000000000000101$,
- *logical or*: $0000000000101001 \mid 1000001000001001 = 1000001000101001$,
- *logical and*: $0000000000101001 \,\&\, 1000001000001001 = 0000000000001001$,
- *exclusive or*: $0000000000101001 \oplus 1000001000001001 = 1000001000100000$,
- *complement*: $\sim 0000000000101001 = 1111111111010110$,
- *addition*: $0000000000101001 + 0000000000100001 = 0000000001001010$, and
- *subtraction*: $0000000000101001 - 0000000000100001 = 0000000000001000$, $0000000000001000 - 0000000000000001 = 0000000000000111$.

These examples use 16-bit variables (note the overflow). Show two different ways to implement a function `mask`(B, d) that converts the d most significant bits of a variable to zero. For example, `mask`$(1000001000001001, 7) =$ `0000000000001001`.

2.7 Implement with your favorite programming language a fixed-length bit-field array. For example, using C++ you can allocate with

```
A=new unsigned[(n*k)/w+1]
```

an array occupying roughly $n \cdot k$ bits, where w is the size of the computer word (unsigned variable) in bits and $k < w$. You should provide operations `setField(A,i,x)` and `x=getField(A,i)` to store and retrieve integer x from A, for x whose binary representation occupies at most k bits.

2.8 Implement using the above fixed-length bit-field array an $O(n \log n)$-bit representation of a node-labeled static tree, supporting navigation from the root to the children.

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press

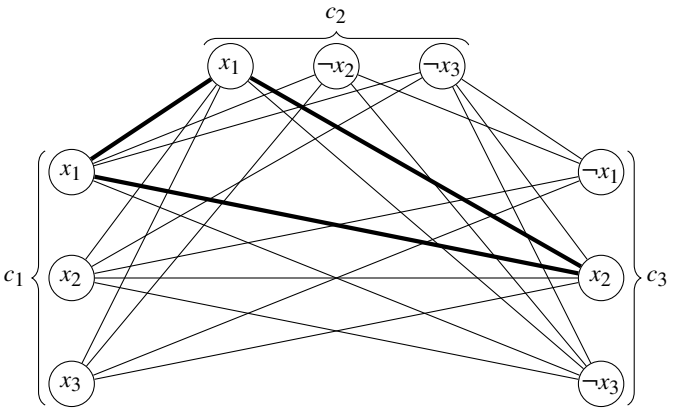


Figure 2.1 A reduction of the 3-SAT problem to the Clique problem. A clique in G_f is highlighted; this induces either one of the truth assignments $(x_1, x_2, x_3) = (1, 1, 0)$ or $(x_1, x_2, x_3) = (1, 1, 1)$.

2.9 Recall how stack, queue, and *deque* work. Implement them using your favorite programming language using doubly-linked lists.

2.10 Given an undirected graph G, a subset $S \subseteq V(G)$ is called an *independent set* if no edge exists between the vertices of S. In the independent-set problem we are given an undirected graph G and an integer k and are asked whether G contains an independent set of size k. Show that the independent-set problem is NP-complete.

2.11 A Boolean formula $f(x_1, \ldots, x_n)$ is in *3-CNF form* if it can be written as

$$f(x_1, \ldots, x_n) = c_1 \wedge \cdots \wedge c_m,$$

where each c_i is $y_{i,1} \vee y_{i,2} \vee y_{i,3}$, and each $y_{i,j}$ equals x_k or $\neg x_k$, for some $k \in \{1, \ldots, n\}$ (with $y_{i,1}$, $y_{i,2}$, $y_{i,3}$ all distinct). The subformulas c_i are called *clauses*, and the subformulas $y_{i,j}$ are called *literals*. The following problem, called 3-SAT, is known to be NP-complete. Given a Boolean formula $f(x_1, \ldots, x_n)$ in 3-CNF form, decide whether there exist $\alpha_1, \ldots, \alpha_n \in \{0, 1\}$ such that $f(\alpha_1, \ldots, \alpha_n)$ is true (such values α_i are called a *satisfying truth assignment*).

Consider as a "new" problem the clique problem from Example 2.3. Show that clique is NP-complete by constructing a reduction from 3-SAT. *Hint.* Given a 3-CNF Boolean formula

$$f(x_1, \ldots, x_n) = (y_{1,1} \vee y_{1,2} \vee y_{1,3}) \wedge \cdots \wedge (y_{m,1} \vee y_{m,2} \vee y_{m,3}),$$

construct the graph G_f as follows (see Figure 2.1 for an example):

- for every $y_{i,j}$, $i \in \{1, \ldots, m\}$, $j \in \{1, 2, 3\}$, add a vertex $y_{i,j}$ to G_f; and
- for every $y_{i_1,j}$ and $y_{i_2,k}$ with $i_1 \neq i_2$ and $y_{i_1,j} \neq \neg y_{i_1,k}$, add the edge $(y_{i_1,j}, y_{i_2,k})$.

Show that f has a satisfying assignment if and only if G_f has a clique of size m.

https://doi.org/10.1017/CBO9781139940023.004 Published online by Cambridge University Press