Chapter 3

Complexity of algorithms

In this chapter, we see how problems may be classified according to their level of difficulty.

Most problems that we consider in these notes are of general character, applying to all members of some family of graphs or digraphs. By an *instance* of a problem, we mean the problem applied to one specific member of the family. For example, an instance of Algorithm 2.6 is the problem of finding all the strongly connected components of a particular digraph.

An *algorithm* for solving a problem is a well-defined procedure which accepts any instance of the problem as *input* and returns a solution to the problem as *output*. Designing computationally efficient algorithms for solving graphs problems is one of the main concern of graph theorists and computer scientists. The two aspects of theoretical interest in this regards are, firstly, to verify that a proposed algorithm does indeed perform correctly and, secondly, to analyse how efficient a procedure is. In the previous chapters, we have already encountered algorithms for solving a number of basic problems. In each case, we have established their validity and estimated their running time.

By the *computational complexity* (or, for short, *complexity*) of an algorithm, we mean the number of basic computational steps (such as arithmetical operations and comparisons) required for its execution. This number clearly depends on the size and nature of the input. In the case of graphs, the complexity is a function of the number of bits required to encode the adjacency list of the input graph $G = (V,E)$, a function of $|V|$ and $|E|$. (The number of bits required to encode an integer k is $\lceil \log_2 k \rceil$.) Naturally, when the input includes additional information, such as weights on the vertices or edges of the graph, this too must be taken into account in calculating the complexity. If the complexity is bounded above by a polynomial in the input size, the algorithm is called a *polynomial-time algorithm*. Such an algorithm is further qualified as *linear-time* if the polynomial is a linear function, *quadratic-time* if it is a quadratic function, and so on.

The classical big-O notation is defined as follows. For functions f and g we write $f = O(g)$ if there are positive numbers n_0 and c such that for every $n \geq n_0$, $f(n) \leq c \cdot g(n)$. We also write $f = \Omega(g)$, which means that $g = O(f)$, and $f = \Theta(g)$, which means that $f = \Omega(g)$ and $f = O(g)$.

3.1 Computational complexity

Polynomial-time solvable algorithms The significance of polynomial-time algorithms is that they are usually found to be computationally feasible, even for large input graphs. By contrast, algorithms whose complexity is exponential in the size of the input have running times which render then unusable even on inputs of moderate size.

For example, Algorithm 1.1 runs in polynomial time: The eulerian tour will be represented with a function *next* such that at the end, $next(e)$ is the edge that immediately follows e in the tour. Each time we insert an edge e, we change the value of *next* for at most two edges: the one of the edge $e_l = v_{l-1}v_l$ if $e = v_l v_{l+1}$ is not in W, or those of $v_{i-1}v_i$ and $e_l = v_{l-1}v_l$ if $e = v_i u$. But an edge is inserted exactly once so the complexity is $O(|E|)$.

The algorithms discussed in Chapter 2 also run in polynomial time. In breadth-first search, each edge is examined for possible inclusion in the tree just twice, (once per endvertex). The same is true for depth-first search. Each time that an edge is considered, a constant number of operations (test, addition or removal in a queue or a stack, mark). Hence, Algorithms 2.2 and 2.3 perform in time $O(|E|)$. The complexity of Algorithms 2.4 and 2.5 is also $O(|E|)$ since they are variation of breadth-first search, as well as the complexity of Algorithm 2.6 because it is a variation of depth-first search.

Although our analysis of these algorithms is admittedly cursory, and leaves out many pertinent details, it should be clear that they do indeed run in polynomial time. A thorough analysis of these and other graph algorithms can be found in the books by Aho et al. [2] and Papadimitriou [6]. On the other hand, there are many basic problems for which polynomial-time algorithms have yet to be found, and indeed might well not exist. Determining which problems are solvable in polynomial time and which are not is evidently a fundamental question. In this connection, a class of probems denoted by $\mathcal{NP}$ (standing for *nondeterministc polynomial-time*) plays an important role. we give here an informal definition of this class: a precise treatment can be found in the book of Garey and Johnson [4], or in Chapter 29 of the *Handbook of Combinatorics* [5].

The classes $\mathcal{P}$, $\mathcal{NP}$ and co-$\mathcal{NP}$ A *decision problem* is a question whose answer is either 'yes' or 'no'. Such a problem belongs to the class $\mathcal{P}$ if there is a polynomial-time algorithm that solves any instance of the problem in polynomial time. It belongs to the class $\mathcal{NP}$ if, given any instance of the problem whose answer is 'yes', there is a certificate validating this fact which can be checked in polynomial time; such a certificate is said to be *succint*. Analogously, a decision problem belongs the the class co-$\mathcal{NP}$ if, given any instance of the problem whose answer is 'no', there is a succint certificate which confirms that this is so. It is immediate from those definitions that $\mathcal{P} \subseteq \mathcal{NP}$. Likewise, $\mathcal{P} \subseteq$ co-$\mathcal{NP}$. Thus

$$\mathcal{P} \subseteq \mathcal{NP} \cap \text{co-}\mathcal{NP}.$$

Consider, for example, the problem of determining whether a graph is bipartite. This decision problem belongs to $\mathcal{NP}$, because a bipartition is a succint certificate: given a bipartition (A,B) of a bipartite graph G, it suffices to check that each edge of G has one endvertex in A and one endvertex in B. The problem also belongs to co-$\mathcal{NP}$ because, by Theorem 2.5, every

nonbipartite graph contains an odd cycle, and any such cycle constitutes a succint certificate of the graph's nonbipartite character. It thus belongs to $\mathcal{NP} \cap$ co-$\mathcal{NP}$. In fact, as shown by Algorithm 2.1 which run in polynomial time, it belongs to $\mathcal{P}$.

Consider now the problem of deciding whether a graph has a *hamiltonian cycle*, that is a spanning cycle or its analog in digraph.

Problem 3.1 (Hamiltonian Cycle)**.**
Instance: A graph G.
Decide: Does G have a hamiltonian cycle?

Problem 3.2 (Directed Hamiltonian Cycle)**.**
Instance: A digraph G.
Decide: Does G have a directed hamiltonian cycle?

If the answer is 'yes', then any hamiltonian cycle would serve as a succint certificate. However, should the answer be 'no', what could consitute a succint certificate confirming this fact? In constrast to the two problems described above, no such certificate is known! In other words, notwithstanding that the Hamiltonian Cycle Problem is clearly member of the class $\mathcal{NP}$, it has not yet been shown to belong to co-$\mathcal{NP}$, and might very well not belong to this class.

We have noted three relations of inclusion among the classes $\mathcal{P}$, $\mathcal{NP}$ and co-$\mathcal{NP}$, and it is natural to ask wether these inclusiuons are proper. Because $\mathcal{P} = \mathcal{NP}$ if and only if $\mathcal{P} =$ co-$\mathcal{NP}$, two basic questions arise, both of which have been posed as conjectures.

Conjecture 3.3.

$$\mathcal{P} \neq \mathcal{NP}$$

Conjecture 3.4 (Edmonds)**.**

$$\mathcal{P} = \mathcal{NP} \cap \text{co-}\mathcal{NP}$$

Conjecture 3.3 is one of the most fundamental open questions in all mathematics. (A prize of one million dollar has been offered for its resolution). It is widely (but not universally) believed that the conjecture is true, that there are problems in $\mathcal{NP}$ for which no polynomial-time algorithm exists. One such problem would be the Directed Hamiltonian Cycle Problem. As we show in Section 3.3, this problem is at least as hard to solve as any problem in the class $\mathcal{NP}$; more precisely, if a polynomial-time algorithm for this problem should be found, it could be adapted to solve any problem in $\mathcal{NP}$ in polynomial time by means of a suitable transformation.

Conjecture 3.4 is strongly supported by empirical evidence. Most decision problems which are known to belong to $\mathcal{NP} \cap$ co-$\mathcal{NP}$ are also known to belong to $\mathcal{P}$. A case in point is the problem of deciding whether a given integer is prime. Although it had been known for some time that this problem belongs to both $\mathcal{NP}$ and co-$\mathcal{NP}$, a polynomial-time algorithm for testing primality was discovered only much more recently, by Agrawal, Kayal and Saxena [1].

3.2 Polynomial reductions

A common approach to problem-solving is to transform the given problem into one whose solution is already known, and then convert that solution into a solution of the original problem. Of course, this approach is feasible only if the transformation can be made rapidly. The concept of polynomial reduction captures this requirement.

A *polynomial reduction* of a problem P to problem Q is a pair of polynomial-time algorithms which transforms each instance I of P to an instance J of Q, and the other which transforms a solution for the instance J to a solution for the instance I. If such a reduction exists, we say that P is *polynomially reducible* to Q, and write $P \preceq Q$. The significance of polynomial reducibility is that if $P \preceq Q$, and if there is a polynomial-time algorithm for solving Q, then this algorithm can be converted into a polynomial-time algorithm for solving P.

In particular, if P and Q are both decision problems, it translates into symbols:

$$P \preceq Q \text{ and } Q \in \mathcal{P} \Rightarrow P \in \mathcal{P}. \tag{3.1}$$

Observe that, since the solution of a decision problem is either 'yes' or 'no', the second algorithm which transforms a solution for the instance J to a solution for the instance I, is trivial: either it is identity (the solution to I is 'yes' if and only if the answer to J is 'yes') or the negation (the answer to I is 'yes' if and only if the answer to J is 'no').

3.3 $\mathcal{NP}$-complete problems

3.3.1 The class $\mathcal{NPC}$

We have just seen how polynomial reductions may be used to produce new polynomial-time algortihms from existing ones. By the same token, polynomial reductions may also be used to link 'hard' problems, ones for which no polynomial-time algorithm exists, as can be seen by writing (3.1) in a different form:

$$P \preceq Q \text{ and } P \notin \mathcal{P} \Rightarrow Q \notin \mathcal{P}.$$

This viewpoint led Cook and Levin to define a special class of seemingly intractable decision problems, the class of $\mathcal{NP}$-complete problems. Informally, there are the problems in the class $\mathcal{NP}$ which are 'at least as hard to solve' as any problem in $\mathcal{NP}$.

Formally, a problem P in $\mathcal{NP}$ is *NP-complete* if $P' \preceq P$ for every problem P' in $\mathcal{NP}$. The class of $\mathcal{NP}$-complete problems is denoted by $\mathcal{NPC}$. It is by no means obvious that $\mathcal{NP}$-complete problems should exist at all. On the other hand, once one such problem has been found, the $\mathcal{NP}$-completeness of other problems may be established by means of polynomial reductions, as follows.

In order to prove that a problem Q in $\mathcal{NP}$ is $\mathcal{NP}$-complete, it suffices to find a polynomial reduction to Q of some known $\mathcal{NP}$-complete problem P. Why is this so? Suppose that P is $\mathcal{NP}$-complete. Then $P' \preceq P$ for all $P' \in \mathcal{NP}$. If $P \preceq Q$, then $P' \preceq Q$ for all $P' \in \mathcal{NP}$, by the

transitivity of the relation $\preceq$. In other words, Q is $\mathcal{NP}$-complete. In symbols:

$$P \preceq Q \text{ and } P \in \mathcal{NPC} \Rightarrow Q \in \mathcal{NPC}.$$

Cook and Levin made a fundamental breakthrough by showing that there do indeed exist $\mathcal{NP}$-complete problems. More precisely, they proved that the satisfiability problem for boolean formulae is $\mathcal{NP}$-complete. We now describe this problem, and examine the theoretical and practical implications of their discovery.

3.3.2 Boolean formulae and satisfiability

A *boolean variable* is a variable which takes on one of two values, $false$ or $true$. Boolean variables may be combined into *boolean formulae*, which may be defined recursively as follows.

- Every boolean variable is a boolean formula.
- If f is a boolean formula, then so too is $(\neg f)$, the *negation* of f.
- If f_1 and f_2 are boolean formulae, then so too are:
  - $(f_1 \vee f_2)$, the *disjunction* of f_1 and f_2,
  - $(f_1 \wedge f_2)$, the *conjunction* of f_1 and f_2.

These three operations may be thought of informally as 'not f', 'f_1 or f_2', and 'f_1 and f_2', respectively.

An assignment of values to the variables is called a *truth assignment*. Given a truth assignment, the value of the formula may be computed according to the following rules:

- if $f = false$, then $(\neg f) = true$, else $(\neg f) = false$;
- if $f_1 = true$ or $f_2 = true$, then $(f_1 \vee f_2) = true$, else $(f_1 \vee f_2) = false$;
- if $f_1 = true$ and $f_2 = true$, then $(f_1 \wedge f_2) = true$, else $(f_1 \vee f_2) = false$.

Two boolean formulae are *equivalent* (written $\equiv$) if they take the same value for each assignment of the variable involved. It follows easily from the above rules that disjunction and conjunction are *commutative* and *associative*. Hence, all the formulae obtained from k subformulae $f_1, f_2, \ldots, f_k$ by means of disjunction are all equivalent. Any of these is denoted by $(f_1 \vee f_2 \vee \cdots \vee f_k)$.

A boolean formula is *satisfiable* if there is a truth assignment of its variables for which the value of the formula is $true$. Clearly, some boolean formulae are satisfiable and some are not. This poses the general problem:

Problem 3.5 (SAT ; Boolean Satisfiability)**.**
Instance: a boolean formula f.
Decide: Is f satisfiable?

Observe that SAT belongs to $\mathcal{NP}$. Indeed given an appropriate truth assignment, it can be checked in polynomial time that the value of the formula is indeed $true$.

Theorem 3.6 (Cook – Levin)**.** *The problem* SAT *is $\mathcal{NP}$-complete.*

The proof of the Cook-Levin Theorem involves the notion of a Turing machine, and is beyond the scope of these notes. A proof may be found in the books of Garey and Johnson [4] or Sipser [7].

Many combinatorial problems are shown to be $\mathcal{NP}$-complete. See for example [4] or [3]. One of the most celebrated and to which many reductions are proved, is 3-SAT. To define it, we need a few more definitions.

A variable x, or its negation $\bar{x}$, is a *literal*, and a disjunction of literals is a *clause*. Any conjuction of disjunctive clauses is referred to as a formula in *conjunctive normal form*. It can be shown that every boolean formula is equivalent, via a polynomial reduction, to one in conjunctive normal form (Exercise 3.1). Furthermore, every boolean formula is equivalent, again via a polynomial reduction, to one in conjunctive normal form with exactly three literals per clause. The decision problem for such boolean formulae is known as 3-SAT.

Problem 3.7 (3-SAT)**.**
Instance: a boolean formula f in conjunctive normal form with exactly three literals per clause.
Decide: Is f satisfiable?

Theorem 3.8. *The problem* 3-SAT *is $\mathcal{NP}$-complete.*

Proof. By Theorem 3.6, it suffices to prove that SAT $\preceq$ 3-SAT. Let f be a boolean formula in conjunctive normal form. We show how to construct, in polynomial time, a boolean formula f' in conjunctive normal form such that:

(i) each clause in f' has three literals;

(ii) f is satisfiable if and only if f' is satisfiable.

Such a formula f' may be obtained by the addition of new variables and clauses, as follows.

Suppose that some clause of f has just two literals, for instance the clause $(x_1 \vee x_2)$. In this case, we simply we simply replace this clause by two clauses with three literals, $(x_1 \vee x_2 \vee x)$ and $(x_1 \vee x_2 \vee \bar{x})$, where x is a new variable. Clearly $(x_1 \vee x_2)$ is equivalent to the conjunction of these two clauses.

Clauses with single literals may be dealt with in a similar manner. If x_1 is a clause, then we first replace this clause by the two clauses with two literals $(x_1 \vee x)$ and $(x_1 \vee \bar{x})$, where x is a new variable. We then replace each of these two clauses by two clauses with three literals as above.

Now suppose that some clause $(x_1 \vee x_2 \vee \cdots \vee x_k)$ of f has k literals, where $k \geq 4$. In this case, we add $k-3$ new variables $y_1, y_2, \ldots, y_{k-3}$ and form the following $k-2$ clauses, each with three literals.

$$(x_1 \vee x_2 \vee y_1), (\bar{y}_1 \vee x_3 \vee y_2), (\bar{y}_2 \vee x_4 \vee y_3), \ldots, (\bar{y}_{k-4} \vee x_{k-2} \vee y_{k-3}), (\bar{y}_{k-3} \vee x_{k-1} \vee x_k).$$

One may verify that $(x_1 \vee x_2 \vee \cdots \vee x_k)$ is equivalent to the conjunction of these $k-2$ clauses. □

3.3.3 Some $\mathcal{NP}$-completeness proofs

As we have observed, in order to show that a decision problem Q in $\mathcal{NP}$ is $\mathcal{NP}$-complete, it suffices to find a polynomial reduction to Q of a known $\mathcal{NP}$-complete problem. This is generally easier said than done. What is needed is to first decide on an appropriate $\mathcal{NP}$-complete problem P and then come up with a suitable polynomial reduction. In the case of graphs, the latter step is often achieved by means of a construction whereby certain special subgraphs, referred to as 'gadgets', are inserted into the instance of P so as to obtain an instance of Q with the required properties. We now describe an illustration of this technique by showing how 3-SAT may be reduced to the Directed Hamiltonian Cycle Problem via an intermediate problem, the Exact Cover Problem.

Let $\mathcal{A}$ be a family of subsets of a finite set X. An *exact cover* of X by $\mathcal{A}$ is a partition of X, each member of which belongs to $\mathcal{A}$. For instance, if $X = \{x_1, x_2, x_3\}$ and $\mathcal{A} = \{\{x_1\}, \{x_1, x_2\}, \{x_2, x_3\}\}$, then $(\{x_1\}, \{x_2, x_3\})$ is an exact cover of X by $\mathcal{A}$. This notion give rise to the following decision problem.

Problem 3.9 (Exact Cover)**.**
Instance: a set X and a family $\mathcal{A}$ of subsets of X.
Decide: Is there an exact cover of X by $\mathcal{A}$?

We first describe a polynomial reduction of 3-SAT to the Exact Cover Problem, and then a polynomial reduction of the Exact Cover Problem to the Directed Hamiltonian Cycle Problem. Since 3-SAT is $\mathcal{NP}$-complete by Theorem 3.8, this implies that the Exact Cover Problem and the Directed Hamiltonian Cycle Problem are $\mathcal{NP}$-complete.

Theorem 3.10. 3-SAT *is polynomially reducible to the Exact Cover Problem.*

Proof. Let f be an instance of 3-SAT, with variables $x_1, \ldots, x_n$ and clauses $f_1, \ldots, f_m$. We first construct a graph G from f by setting:

$$\begin{aligned} V(G) &= \{x_i \mid 1 \le i \le n\} \cup \{\overline{x}_i \mid 1 \le i \le n\} \cup \{f_j \mid 1 \le j \le m\}, \\ E(G) &= \{x_i \overline{x}_i \mid 1 \le i \le n\} \cup \{x_i f_j \mid x_i \in f_j\} \cup \{\overline{x}_i f_j \mid \overline{x_i} \in f_j\}, \end{aligned}$$

where the notation $x_i \in f_j$ (resp. $\overline{x_i} \in f_j$) signifies that x_i (resp. $\overline{x_i}$) is a literal of the clause f_j. We then obtain an instance $(X, \mathcal{A})$ of the Exact Cover Problem from this graph G by setting:

$$\begin{aligned} X &= \{f_j \mid 1 \le j \le m\} \cup E(G), \quad \text{and} \\ \mathcal{A} &= \{E(x_i) \mid 1 \le i \le n\} \cup \{E(\overline{x_i}) \mid 1 \le i \le n\} \cup \{\{f_j\} \cup F_j \mid F_j \subset E(f_j), 1 \le j \le m\}, \end{aligned}$$

where $E(x)$ denotes the set of edges incident to vertex x in the graph G.

It can be verified that the formula f is satisfiable if and only if the set X has an exact cover by the family $\mathcal{A}$. □

Corollary 3.11. *The Exact Cover Problem is $\mathcal{NP}$-complete.*

Theorem 3.12. *The Exact Cover Problem is polynomially reducible to the Directed Hamiltonian Cycle Problem.*

Proof. Let $(X, \mathcal{A})$ be an instance of the Exact Cover Problem, where $X = \{x_i, 1 \leq i \leq n\}$ and $\mathcal{A} = \{A_j \mid 1 \leq j \leq m\}$. We construct a digraph D as follows. Let P be a directed path whose arcs are labelled by the elements of X, Q a directed path whose arcs are labelled by the elements of $\mathcal{A}$, and for $1 \leq j \leq m$, R_j a directed path whose vertices are labelled by the elements of A_j. The paths P, Q, and R_j, $1 \leq j \leq m$, are assumed to be pairwise disjoint. We add an arc from the start of P to the start of Q, and from the terminus of Q to the terminus of P. For $1 \leq j \leq m$, we also add an arc from the tail of the arc labelled A_j of Q to the start of R_j, and from the terminus of R_j to the head of the arc labelled A_j of Q.

For $1 \leq j \leq m$, we now transform the directed path R_j into a digraph D_j by replacing each vertex labelled x_i of R_j by a *symmetric path* P_{ij} of length two, that is the symmetric digraph obtained from a path of length two by replacing each edge by two arcs in each direction. Moreover, for every such symmetric path, we add an arc from the start of $P_{i,j}$ to the tail of the arc labelled x_i of P, and one from the head of x_i to the terminus of $P_{i,j}$. We denote the resulting digraph by D. See Figure 3.1.

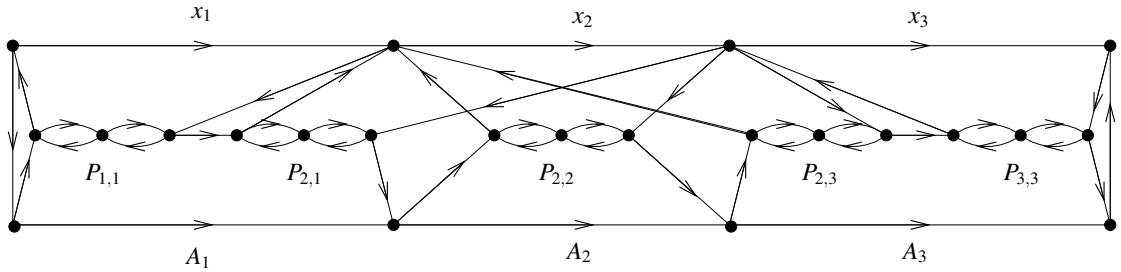

Figure 3.1: The digraph D when $X = \{x_1, x_2, x_3\}$ and $\mathcal{A} = \{\{x_1, x_2\}, \{x_2\}, \{x_2, x_3\}\}$.

Observe now, that the digraph D has a directed hamiltonian cycle C if and only if the set X has an exact cover by $\mathcal{A}$.

Suppose first that D has a directed hamiltonian cycle C. If C does not use the arc labelled A_j in Q, it is obliged to traverse D_j from its start to its terminus. Conversely, if C uses the arc labelled A_j in Q, it is obliged to include each one of the paths $P_{i,j}$ in D_j in its route from the start of P to the terminus of P. Moreover, C traces exactly one of the paths $P_{i,j}$ (for $x_i \in A_j$) in travelling from the head of the arc of P labelled x_i to its tail. Hence A_j that label the arcs of $Q \cap C$ form a partition of X.

Reciprocally, if we have an exact cover of X by $\mathcal{A}$, a directed hamiltonian cycle C of D may be constructed by taking the arcs labelled A_j when A_j is not a member of the exact cover and all the paths P_{ij} for X_j in the cover and $x_i \in X_j$ and completing by adding arcs in an obvious way.

Finally, the number of vertices of D is

$$|E(D)| = |X| + |\mathcal{A}| + 3\sum_{j=1}^{m} |A_J| + 2.$$

This function is bounded above by a linear function of the size of the instance $(X, \mathcal{A})$, so the above reduction is indeed polynomial. □

Corollary 3.13. *The Directed Hamiltonian Cycle Probem is $\mathcal{NP}$-complete.*

A huge number of decision problems have been shown to be $\mathcal{NP}$-complete. See for example the book of Garey and Johnson [4].

3.4 $\mathcal{NP}$-hard problems

We now turn to the computational complexity of optimization problems such as the Travelling Salesman Problem. An *edge-weighted graph* is a pair (G,w) where $G=(V,E)$ is a graph and $w : E \to \mathbb{R}$ is a *weight function*.

Problem 3.14 (Travelling Salesman)**.**
Instance: an edge-weighted complete graph (G,w).
Find: a hamiltonian cycle C of G of minimum weight, i.e. such that $\sum_{e\in E(C)} w(e)$ is minimum.

This problem contains the Hamiltonian Cycle Problem as a special case. To see this, associate with a given graph G the edge-weighted complete graph on $V(G)$ in which the weight function w is defined by $w(uv)=0$ if $uv \in E(G)$, and $w(uv)=1$ otherwise. The resulting edge-weighted complete graph has a hamiltonian cycle of weight zero if and only if G has a hamiltonian cycle. Thus, any algorithm for solving the Travelling Salesman Problem will also solve the Hamiltonian Cycle Problem, and we may conclude that the former problem is at least as hard as the latter. Because the Hamiltonian Cycle Problem is $\mathcal{NP}$-complete, (See Exercise 3.2), the Travelling Salesman Problem is at least as hard as any problem in $\mathcal{NP}$. Such problems are called *$\mathcal{NP}$-hard*.

Observe that every optimization problem implicitely includes an infinitude of decision problems. For example, the Travelling Salesman Problem includes, for each real number r, the following decision problem. Given an edge-weighted graph (G,w), is there a hamiltonian cycle of weight at most r? If one of these problems is $\mathcal{NP}$-complete, then the optimization problem is $\mathcal{NP}$-hard. However, the problem may still be $\mathcal{NP}$-hard even if all these problems are polynomial-time solvable. For example, it is the case for the following basic problem.

Problem 3.15 (Maximum Stable Set)**.**
Instance: a graph G.
Find: a stable set of maximum size in G.

If k is a fixed integer not depending on $|V(G)|$, the existence of a stable set of size k can be decided in polynomial time, simply by means of an exhaustive search, because the number of k-subsets of $V(G)$ is bounded above by $|V(G)|^k$. However, if k depends on $|V(G)|$, this is no longer true. Indeed, the problem of deciding whether a graph G has a k-clique, where k depends on $|V(G)|$ is $\mathcal{NP}$-complete.

Theorem 3.16. *The following problem is $\mathcal{NP}$-complete.*
Instance: a graph G
Decide: $\alpha(G) \geq |V(G)|/3$?

Proof. Let Φ be a 3-SAT formula with n variables $x_1,\ldots,x_n$ and m clauses $C_1,\ldots,C_m$. Let us create a graph G_Φ as follows:

- For each clause $C_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$, we create a 3-cycle (v_1^j, v_2^j, v_3^j). These cycles are called the *clause gadgets*.
- For any two vertices v_k^j and $v_{k'}^{j'}$ in different clause gadgets, we add an edge between v_k^j and $v_{k'}^{j'}$ if and only if there is a variable x_i such that $\{\ell_k^j, \ell_{k'}^{j'}\} = \{x_i, \bar{x}_i\}$.

By construction $|V(G_\phi)| = 3m$. Let us prove that G_Φ has a stable set of size m if and only if Φ is satisfiable.

Suppose Φ has a satisfying truth assignment f. Then at least one variable in each clause is satisfied by f. Define S to be a set of vertices in G_Φ found by selecting one of vertices corresponding to the satisfied variable in each clause gadget. Since we picked one vertex for each clause, there are clearly m vertices in S. Now consider two distinct vertices v_k^j and $v_{k'}^{j'}$ of S. They are not in the same clause gadget, since we only select a single vertex for each gadget, and $\{\ell_k^j, \ell_{k'}^{j'}\} \neq \{x_i, \bar{x}_i\}$, because f can not have satisfied both of x_i and $\bar{x}_i$. Therefore x and y are not adjacent. Hence S is a stable set of size m.

Suppose now that G_Φ has a stable set S of size m. Since the clause gadget are complete subgraphs, there is at most vertex vertex of S per clause gadget. But in fact, since there are exactly m clause gadgets, S must contain exactly one node from each clause gadget. Let f be the truth assignment defined by $f(x_i) = true$ if there exists a literal $\ell_k^j = x_i$ whose associated vertex v_k^j is in S, and $f(x_i) = false$ otherwise. Clearly, f satisfies Φ. □

Since a subset S of $V(G)$ is a stable set in G if and only if S is a clique in $\overline{G}$, the following problem is polynomially equivalent to the Maximum Stable Set Problem, and thus is $\mathcal{NP}$-hard also.

Problem 3.17 (Maximum Clique)**.**
Instance: a graph G.
Find: a clique of maximum size in G.

A huge collection of optimization problems have been shown to be $\mathcal{NP}$-hard, see [3].

3.5 Approximation algorithms

For $\mathcal{NP}$-hard optimization problems of practical interest, such as the Travelling Salesman Problem, the best that one can reasonably expect of a polynomial-time algorithm is that it should always return a feasible solution which is not too far from optimality.

Given a real number $r \geq 1$, an *r-approximation algorithm* for a minimization problem is an algorithm that returns a feasible solution whose value is no more than r times the optimal value; similarly, an *r-approximation algorithm* for a maximization problem is an algorithm that returns a feasible solution whose value is no less than r times the optimal value; the smaller the value of r, the better the approximation. Naturally, the running time of the algorithm is an equally important factor. We give an example.

Problem 3.18 (Maximum Cut)**.**
Instance: a graph G.
Find: a spanning bipartite subgraph F of G with the maximum number of edges.

It can be shown that the Maximum Cut Problem is $\mathcal{NP}$-hard (Exercise 3.4).

Theorem 3.19. *The Maximum Cut Problem admits a polynomial-time* 2*-approximation algorithm.*

Proof. We now describe an algorithm that find a bipartite subgraph F such that $|E(F)| \geq |E(G)|/2$. Since an subgraph of G cannot have more than $|E(G)|$ edges, this is a 2-approximation algorithm.

Algorithm 3.1 (Maximum-Cut Approximation)**.**

2. Take any ordering $v_1, v_2, \ldots, v_n$ of the vertices; $A := \emptyset$; $B := \emptyset$; $E := \emptyset$.
2. For $i = 1$ to n, if v_i has more neighbours in A than in B, then add v_i to B and all edges joining v_i to elements of A to E. Else add v_i to A and all edges joining v_i to elements of B to E.
3. Return $((A,B),E)$.

This algorithm examined every vertex exactly once and each time its examines a vertex it must counts the numbers of vertices in A and B and compare them. Then its complexity is at most $O(|V|^2)$.

Let us now show that the returned bipartite graph F satisfies $|E(F)| \geq |E(G)|/2$. Therefore let us denote by $F_i = ((A_i, B_i), E_i)$ the bipartite graph constructed after step i that is after having examing v_i at Step 2 in the above algorithm. and $G_i = G\langle\{v_1, \ldots, v_i\}\rangle$. We prove by induction that $|E(F_i)| \geq |E(G_i)|/2$, the result holding vacuously when $i = 0$. Suppose now that $i > 0$ and that the result holds for $i-1$. We obtained F_i from F_{i-1} by adding V_i to the part in which it has the smaller number of neighbours. Hence, the number of edges incident to v_i that we add to F_{i-1} is at least has large has half the number of edges joining v_i to a vertex in $V(F_{i-1}) = \{v_1, \ldots, v_{i-1}\}$. Since $|E(F_{i-1})| \geq |E(G_{i-1})|/2$ by the induction hypothesis, we have $|E(F_i)| \geq |E(G_i)|/2$. □

The analog of the Maximum Cut Problem in edge-weighted graph, called the Weighted Maximum Cut Problem also admits a polynomial-time 2-approximation algorithm. See Exercise 3.7.

If some algorithms admits polynomial-time approximation algorithms, some others do not. For example, this is the case for the Travelling Salesman Problem: for any $t \geq 2$, there cannot exists a polynomial-time t-approximation algorithm for solving the Travelling Salesman Problem, unless $\mathcal{P} = \mathcal{NP}$. (Exercise 3.8). However some special cases the Travelling Salesman Problem admit polynomial-time approximation algorithm. For example, such an algorithm, when the weights satisfies the triangle inequality, is discussed in Section 4.2.3.

For more on approximation algorithms, we refer the interested reader to the book of Vazirani [8].

3.6 Exercises

Exercise 3.1. Let f_1 and f_2 be two boolean formulae in conjuctive normal form.
1) Show that:

a) $f_1 \wedge f_2$ is in conjunctive normal form;

b) $f_1 \vee f_2$ is equivalent to a boolean formula in conjunctive normal form;

c) $\neg f_1$ is equivalent to a boolean formula in conjunctive normal form;

2) Deduce that every boolean formula is equivalent to a boolean formula in conjunctive normal form.

Exercise 3.2.
1) Describe a polynomial-time reduction of the Directed Hamiltonian Cycle Problem to the Hamiltonian Cycle Problem.
2) Deduce that the Hamiltonian Cycle Problem is $\mathcal{NP}$-complete.

Exercise 3.3.

A path is *hamiltonian* in a graph G if it goes through all vertices of G. The Hamiltonian Path Problem consists in deciding if a given graph contains a hamiltonian path. Show that the Hamiltonian Path Problem is $\mathcal{NP}$-complete.

Exercise 3.4. Show that the Maximum Cut Problem is $\mathcal{NP}$-hard.

Exercise 3.5. A *vertex cover* in graph is a set S of vertices such that every edge of G has an endvertex in S. The Vertex Cover Problem consists in finding a minimum vertex cover of the input graph, that is a vertex cover with the minimum number of vertices.

1) Show that the Vertex Cover Problem is $\mathcal{NP}$-hard.

2) A *matching* in a graph G is a set of pairwise disjoint edges. A matching M is *maximal* in G if for any edge $e \in E(G) \setminus M$, the set $M \cup \{e\}$ is not a matching.

a) Show that if M is a maximal matching in G, then $G - V(M)$ is a stable set.

b) Deduce a 2-approximation algorithm for the Vertex Cover Problem.

Exercise 3.6. A *feedback arc set* in a digraph D is a set of arcs F such that $D \setminus F$ is acyclic. Show that finding a feedback arc set of minimum cardinality in a digraph is an $\mathcal{NP}$-hard problem.

Hint: Use reduction from the Vertex Cover Problem. From an undirected graph G, you may construct a digraph with vertex set $\bigcup_{v \in V(G)} \{v^-, v^+\}$.

Exercise 3.7. Describe a polynomial-time 2-approximation algorithm for the following problem, called the Weighted Maximum Cut Problem.
Instance: an edge-weighted graph (G,w).
Find: a spanning bipartite subgraph F of G with the maximum weight.

Exercise 3.8.
1) Let G be a graph on n vertices, $n \geq 3$ vertices, and let t be a positive integer. Consider the edge-weighted complete graph (K,w), where $V(K) = V(G)$, in which $w(e) = 1$ if $e \in E(G)$ and $w(e) = (t-1)n+2$ if $e \in E(K) \setminus E(G)$. Show that:

a) (K,w) has a hamiltonian cycle of weight n if and only if G has a hamiltonian cycle;

b) any hamiltonian cycle of (K,w) of weight greater than n has weight at least $tn+1$.

2) Deduce that, unless $\mathcal{P} \neq \mathcal{NP}$, there cannot exist a polynomial-time t-approximation algorithm for solving the Travelling Salesman Problem.

Bibliography

[1] M. Agrawal, N. Kayal and N. Saxena. PRIMES is in P. *Ann. of Math. (2)* 160:781–793, 2004.

[2] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, Reading, MA, 1975. Second printing.

[3] P. Crescenzi and V. Kann, eds. *A $\mathcal{NP}$-compendium of $\mathcal{NP}$ optimization problems*. `http://www.csc.kth.se/ viggo/problemlist/`.

[4] M. R. Garey and D. S. Johnson, *Computers and intractability. A guide to the theory of NP-completeness.* A Series of Books in the Mathematical Sciences. W. H. Freeman and Co., San Francisco, Calif., 1979.

[5] R. L. Graham, M. Grötschel and L. Lovász, eds. *Handbook of Combinatorics*. Vol. 1,2. Elsevier, Amsterdam, 1995.

[6] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[7] M. Sipser. *Introduction to the Theory of Computation.* Second edition. Course Technology, Boston, MA, 2005.

[8] V. V. Vazirani. *Approximation Algorithms*. Springer, Berlin, 2001.