Chapter 8

Beyond NP-completeness

At the conclusion of Chapter 6, we stated that proving a problem is NP-complete does not make it go away. The subject of this chapter is to go beyond NP-completeness and to describe the various approaches that can be taken when confronted with an NP-complete problem.

The first approach (see Section 8.1) is the most elegant. When deriving *approximation algorithms*, we search for an approximate solution, but we also guarantee that it is of good quality. Of course, the approximated solution must be found in polynomial time.

The second approach (see Section 8.2) is less ambitious. Given an NP-complete problem, we show how to characterize particular instances that have polynomial complexity.

The third approach (see Section 8.3) often provides useful lower bounds. The idea is to cast the optimization problem under study in terms of a *linear program*. While solving a linear program with integer variables is NP-complete, solving a linear program with rational variables has polynomial complexity (we are restricted to rational variables because of the impossibility of efficiently encoding real numbers). The difficulty is then to reconstruct a solution of the integer linear program from an optimal solution of that program with rational variables. This is not always possible, but this method at least provides a lower bound on any optimal integer solution.

We briefly introduce, in Section 8.4, *randomized algorithms* as a fourth approach that solves "most" instances of an NP-complete problem in polynomial time.

Finally, we provide in Section 8.5 a detailed discussion of *branch-and-bound* and *backtracking* strategies, where one explores the space of all potential solutions in a clever way. While the worst-case exploration may require exponential time, on average, the optimal solution is found in "reasonable" time.

8.1 Approximation results

In this section, we first define polynomial-time approximation algorithms and (fully) polynomial-time approximation schemes (PTAS and FPTAS). Then, we give some examples of approximation and inapproximability results.

8.1.1 Approximation algorithms

In Chapter 6, we have defined the NP-completeness of problems and exhibited several NP-complete decision problems. As discussed in Section 6.3.4, the target problem is often an optimization problem that has been restricted to a decision problem so that we can prove its NP-completeness.

If the optimal solution of an optimization problem cannot be found in polynomial time, one may want to find an approximate solution in polynomial time.

DEFINITION 8.1. A λ-approximation algorithm is an algorithm whose execution time is polynomial in the instance size and that returns an approximate solution guaranteed to be, in the worst case, at a factor λ away from the optimal solution.

For instance, for each instance $\mathcal{I}$ of a minimization problem, the solution of the approximation algorithm for instance $\mathcal{I}$ must be smaller than or equal to λ times the optimal solution for instance $\mathcal{I}$.

The closer λ to 1, the better the approximation algorithm. We categorize some particular approximation algorithms for which λ is close to 1 as polynomial-time approximation schemes.

DEFINITION 8.2. A *Polynomial-Time Approximation Scheme* (PTAS) is such that for any constant $\lambda = 1+\varepsilon > 1$, there exists a λ-approximation algorithm, i.e., an algorithm that is polynomial in the instance size and guaranteed at a factor λ.

Note that the algorithm may not be polynomial in $1/\varepsilon$ and thus have a high complexity when ε gets close to zero. A Fully PTAS is such that the algorithm is polynomial both in the instance size and in $1/\varepsilon$.

DEFINITION 8.3. A *Fully Polynomial-Time Approximation Scheme*, or FPTAS, is such that for any constant $\lambda = 1 + \varepsilon > 1$, there exists a λ-approximation algorithm that is polynomial in the instance size *and* in $1/\varepsilon$.

The difference between PTAS and FPTAS is simply that the $\forall\varepsilon$ quantifier changes sides. For a PTAS, ε is a fixed constant, so that $2^{\frac{1}{\varepsilon}}$ is a constant as well. On the contrary, the complexity of an FPTAS scheme must be polynomial in $\frac{1}{\varepsilon}$. Of course, having an FPTAS is a stronger property than having a PTAS (i.e., FPTAS $\Rightarrow$ PTAS).

Finally, we define asymptotic PTAS and FPTAS, which add a constant to the approximation scheme. We define formally only the APTAS for a minimization problem, and the definition can easily be extended for maximization problems and AFPTAS.

DEFINITION 8.4. An *Asymptotic Polynomial-Time Approximation Scheme*, or APTAS, is such that for any constant $\lambda = 1 + \varepsilon > 1$, there exists an algorithm, polynomial in the instance size, such that $C_{APTAS} \leqslant \lambda C_{opt} + \beta$ (for a minimization problem), where C_{APTAS} is the cost of the solution of the algorithm, C_{opt} is the cost of an optimal solution, and β is a constant that may depend on ε but should be independent of the problem size.

In the following, we discuss several approximation algorithms, and we show how to prove that an algorithm is an approximation algorithm (possibly an (A)PTAS or (A)FPTAS) or how to prove that a problem cannot be approximated in polynomial time up to any fixed constant λ.

8.1.2 Vertex cover

We consider here the classical vertex cover problem, which was shown to be NP-complete in Section 6.4.3. We discuss a weighted version of this problem in Section 8.3.

We first recall the definition of the vertex cover problem in its optimization problem formulation. Given a graph $G = (V, E)$, we want to find a set of vertices of minimum size that is covering all edges (i.e., any edge in E includes at least one of the vertices of the set).

We consider the following greedy algorithm to solve the problem, called **greedy-vc**. Initialize $S = \emptyset$. Then, while some edges are not covered (i.e., neither of their end vertices are in set S), pick one edge $e = (u, v)$, add both vertices u and v to set S, and mark all edges including u or v as covered. It is clear that **greedy-vc** returns a valid vertex cover, and that it is polynomial in the size of the instance. We now prove that it is a 2-approximation algorithm for the vertex cover optimization problem.

THEOREM 8.1. Greedy-vc *is a* 2*-approximation algorithm for vertex cover.*

Proof. Let A be the set of edges selected by the greedy algorithm. Two edges of A cannot have a common vertex, and, therefore, the size of the cover of this algorithm is $C_{\textbf{greedy-vc}} = 2|A|$. However, all edges selected greedily are independent, and each of them must be covered in any solution; hence, an optimal solution has at least $|A|$ vertices: $C_{opt} \geqslant |A|$. We deduce that $C_{\textbf{greedy-vc}} \leqslant 2 \times C_{opt}$, which concludes the proof. $\square$

Note that this approximation factor of 2 is achieved, for instance, if G consists of two vertices joined by an edge. There is a polynomial-time algorithm that is a $2 - \frac{\log(\log(|V|))}{2\log(|V|)}$ approximation [79], but, for instance, we do not know any polynomial-time algorithms that would be a 1.99 approximation (the problem is still open).

8.1.3 Traveling salesman problem (TSP)

Let $G = (V, E)$ be a complete graph and $w : E \to \mathbb{N}$ be a cost function. The TSP problem consists of finding a cycle $\mathcal{C}$ going through each vertex once and only once, with $\sum_{e \in \mathcal{C}} w(e) \leqslant k$. The decision problem, in which k is a fixed integer, is NP-complete, as mentioned in Section 6.4.5. For the optimization problem, the goal is to minimize k.

First, we prove that TSP cannot be approximated unless P = NP. Then, we propose an approximation algorithm in the particular case where the cost function follows the triangle inequality.

Inapproximability of TSP

THEOREM 8.2. *For any constant $\lambda \geqslant 1$, there does not exist any λ-approximation algorithm for TSP unless $P = NP$.*

To prove such a result, the methodology is often as follows. The idea consists of assuming that there is a λ-approximation algorithm for the target problem (by definition, this is a polynomial-time algorithm). Then, one uses this approximation algorithm to solve in polynomial time a problem that is known to be NP-complete. For TSP, we show how any instance of problem Hamiltonian Cycle (HC, see Definition 6.4) can be solved in polynomial time using any approximation algorithm for TSP.

Proof. Let us assume that there is a λ-approximation algorithm for TSP. We consider an instance $\mathcal{I}_{hc}$ of HC, which is a graph $G = (V, E)$, with $n = |V|$. Then, we build an instance $\mathcal{I}_{tsp}$ of TSP as follows. In the complete graph, we build a cost function such that $w(e) = 1$ if $e \in E$, and $w(e) = \lambda n + 1$ otherwise. The size of $\mathcal{I}_{tsp}$ is obviously polynomial in the size of $\mathcal{I}_{hc}$.

We use the λ-approximation algorithm to solve $\mathcal{I}_{tsp}$. Let C_{algo} be its solution. This solution is such that $C_{algo} \leqslant \lambda C_{opt}$, where C_{opt} is the optimal solution.

We consider the two following cases:

- If $C_{algo} \geqslant \lambda n + 1$, then $C_{opt} > n$. This means that instance $\mathcal{I}_{hc}$ has no solution. Indeed, a Hamiltonian Cycle for $\mathcal{I}_{hc}$ would be a solution of cost n for $\mathcal{I}_{tsp}$.

- Otherwise, $C_{algo} < \lambda n + 1$, and therefore the solution of $\mathcal{I}_{tsp}$ is not using any edge not in E (otherwise, the cost would be at least $\lambda n + 1$). This solution is therefore a Hamiltonian Cycle for $\mathcal{I}_{hc}$, which means that instance $\mathcal{I}_{hp}$ has a solution.

Therefore, the result of the algorithm for $\mathcal{I}_{tsp}$ allows us to conclude whether there is a Hamiltonian Cycle in $\mathcal{I}_{hc}$, which concludes the proof. □

Note that we assumed that λ is constant, but we can even have $\lambda = 1 + 2^{-n}$, since the algorithm would still be polynomial in the instance size (λ can be encoded in logarithmic size, hence in $O(n)$). However, Theorem 8.2 does not forbid the existence of a $2^{2^{-n}}$-approximation algorithm.

Approximation algorithm with triangle inequality

We now assume that the cost function w satisfies the triangle inequality, i.e., for all vertices $v_1, v_2, v_3 \in V$, $w(v_1, v_3) \leqslant w(v_1, v_2) + w(v_2, v_3)$.

The approximation algorithm **spanning-tsp** works as follows. First, we build a minimum spanning tree T of the graph G, which can be done in polynomial time with a greedy algorithm (remove edges by nonincreasing costs while keeping a connected graph, see Section 3.4). Then, we perform a tree traversal of T (once a node u is visited, one completely visits the subtree rooted at one of the children of u before starting to visit any subtree rooted at another child). Each edge of T is visited exactly twice. We extract a solution for TSP, i.e., a Hamiltonian Cycle, by recording the order in which vertices are visited for the first time. From this ordered list of vertices, we build a cycle by taking the edges that link consecutive vertices (recall that the graph is complete).

We now prove that this algorithm is a 2-approximation.

THEOREM 8.3. Spanning-tsp *is a* 2*-approximation algorithm for the traveling salesman problem with the triangle inequality.*

Proof. The optimal cost C_{opt} is at least equal to the sum of the costs of the edges in the minimum spanning tree T, denoted by $w(T)$. Indeed, an optimal solution is a cycle. If we remove an edge from an optimal solution, we obtain a spanning tree, and T is a spanning tree of minimum weight. Therefore, $C_{opt} \geqslant w(T)$.

Now, we consider the cost of the solution returned by the algorithm. We denote this solution by S and its cost by $C_{\textbf{spanning-tsp}}$. Let O be the order in which the vertices are visited in the traversal of T. Vertices that are not leaves of T appear several times in O. S is obtained from O by keeping only the first occurrence of each vertex. Because of the triangular inequality, deleting a vertex from O does not increase the cost of the associated path. (Suppose we delete the vertex y in the sequence (x, y, z) of O; this is equivalent to replacing the two edges (x, y) and (y, z) with the single edge (x, z).) Hence, $C_{\textbf{spanning-tsp}}$ is less than or equal to the cost of the path associated with O. Furthermore, the path associated with O contains each edge exactly twice, and its cost is exactly $2 \times w(T)$. Therefore, $C_{\textbf{spanning-tsp}} \leqslant 2C_{opt}$, which proves the approximation result. □

8.1.4 Bin packing

In this section, we introduce a new classical problem that is the bin packing problem.

DEFINITION 8.5 (BP – Bin Packing)**.** Given n rational numbers (also called objects) $a_1, \ldots, a_n$, with $0 < a_i \leqslant 1$, for $1 \leqslant i \leqslant n$, can we partition them in k bins $B_1, \ldots, B_k$ of capacity 1, i.e., for each $1 \leqslant j \leqslant k$, $\sum_{i \in B_j} a_i \leqslant 1$?

First, we prove the NP-completeness of this problem, then we exhibit several approximation results.

NP-completeness of BP

THEOREM 8.4. *BP is NP-complete.*

Proof. It is straightforward to see that BP is in NP: A certificate is the list, for each bin, of the indices of the numbers it contains.

The reduction comes from 2-PARTITION. We consider an instance $\mathcal{I}_1$ of 2-PARTITION, with n integers $b_1, \ldots, b_n$. We build the following instance $\mathcal{I}_2$ of BP: For $1 \leqslant i \leqslant n$, $a_i = \frac{2b_i}{S}$, with $S = \sum_{i=1}^{n} b_i$, and we set $k = 2$.

It is then straightforward to see that the size of the new instance is polynomial and to check the equivalence of solutions. □

Inapproximability of BP

THEOREM 8.5. *For all $\varepsilon > 0$, there does not exist any $(\frac{3}{2} - \varepsilon)$-approximation algorithm for BP unless P = NP.*

Proof. Let us assume that there is a $(\frac{3}{2} - \varepsilon)$-approximation algorithm for BP. We then exhibit a polynomial algorithm to solve 2-PARTITION.

Given an instance of 2-PARTITION, we execute the algorithm for BP with the a_i as defined earlier. If there exists a 2-PARTITION of the b_i, the algorithm returns at most $2 \times (\frac{3}{2} - \varepsilon) = 3 - 2\varepsilon$ bins, so it returns two bins. Otherwise, the algorithm returns a solution with at least three bins. Thanks to the polynomial approximation algorithm, we can solve 2-PARTITION in polynomial time, which implies that P = NP. This concludes the proof. □

Approximation algorithms for BP

We start with a simple greedy algorithm in which we select objects in a random order, and, at each step, we place the object either in the last used bin where it fits (**next-fit** algorithm) or in the first used bin where it fits (**first-fit** algorithm); otherwise (i.e., the object is not fitting in any used bin), we create a new bin and place the object in this new bin. We prove below that **next-fit** (and, hence, **first-fit**) is a 2-approximation algorithm for the BP problem.

THEOREM 8.6. Next-fit *is a 2-approximation algorithm for BP.*

Proof. Let $A = \sum_{i=1}^{n} a_i$. We have a lower bound on the cost of the optimal solution (the number of bins used by the optimal solution): $C_{opt} \geqslant \lceil A \rceil$.

Now we bound the cost of **next-fit** as follows. If we consider two consecutive bins, the sum of the objects that they contain is strictly greater than 1;

otherwise, we would not have created a new bin. Therefore, if $C_{\textbf{next-fit}} = K$, and B_k is the k-th bin of the solution returned by **next-fit**, for $1 \leqslant k \leqslant K$, then by summing the contents of two consecutive bins, we get

$$\sum_{k=1}^{K-1} \left(\sum_{i \in B_k} a_i + \sum_{i \in B_{k+1}} a_i \right) > K - 1 \; .$$

Moreover, by definition of A, we have $\sum_{k=1}^{K-1} \left(\sum_{i \in B_k} a_i + \sum_{i \in B_{k+1}} a_i \right) \leqslant 2A$, and, therefore, $K - 1 < 2A \leqslant 2\lceil A \rceil$. Finally, $C_{\textbf{next-fit}} = K \leqslant 2\lceil A \rceil \leqslant 2C_{opt}$, which concludes the proof. □

Note that the approximation ratio is tight for the **next-fit** algorithm. Consider an instance of BP with $4n$ objects such that $a_{2i-1} = \frac{1}{2}$ and $a_{2i} = \frac{1}{2n}$, for $1 \leqslant i \leqslant 2n$. Then, if **next-fit** chooses the objects in the sequential order, its solution uses $2n$ bins (one object a_{2i-1} and one object a_{2i} in each bin), while the optimal solution uses only $n + 1$ bins (for $1 \leqslant i \leqslant 2n$, the $2n$ objects a_{2i} in one bin and two objects a_{2i-1} in each of the other n bins).

The previous algorithms can be qualified as *online* algorithms because no sorting is done on the objects, and we can pack them in the bins when they arrive, on the fly. If we have the knowledge of all objects before executing the algorithm, we can refine the algorithm by sorting the objects beforehand. Such algorithms are called *offline* algorithms. The **first-fit-dec** algorithm sorts the objects by nonincreasing size (**dec** stands for decreasing), and then it applies the **first-fit** rule: The object is placed in the first used bin in which it fits; otherwise, a new bin is created.

THEOREM 8.7. *$C_{\textbf{first-fit-dec}} \leqslant \frac{3}{2}C_{opt} + 1$, where $C_{\textbf{first-fit-dec}}$ is the cost returned by the* **first-fit-dec** *algorithm, and C_{opt} is the optimal cost.*

Note that this is not an approximation algorithm as defined above because of the "+1" in the expression, which corresponds to one extra bin that the **first-fit-dec** algorithm may use. This is rather an *asymptotic* approximation algorithm, which is similar to an A(F)PTAS scheme. Indeed, the constant 1 is independent of the problem size, and the algorithm is asymptotically a $\frac{3}{2}$-approximation.

Proof. We split the a_i in four categories:

$$A = \left\{ a_i > \frac{2}{3} \right\} \quad B = \left\{ \frac{2}{3} \geq a_i > \frac{1}{2} \right\} \quad C = \left\{ \frac{1}{2} \geq a_i > \frac{1}{3} \right\} \quad D = \left\{ \frac{1}{3} \geq a_i \right\}$$

Case 1: There is at least one bin containing only objects of category D in the solution of **first-fit-dec**. In this case, at most one bin (the last one) has a sum of objects of less than $\frac{2}{3}$, and it contains only objects of category D. Indeed, if the objects of D of the last bin have not fit in the previous bins, it means that each bin (except the last one) has a sum of objects of at least $\frac{2}{3}$.

Therefore, if we ignore the last bin, $C_{opt} \geqslant \sum_{i=1}^{n} a_i \geqslant \frac{2}{3}(C_{\textbf{first-fit-dec}} - 1)$, which concludes the proof for this case.

Case 2: There is no bin with only objects of category D. In this case, we can ignore the objects of category D because they are added into the bins at the end of the algorithm, and they do not lead to the creation of new bins. We now prove that the solution of **first-fit-dec** for the objects of A, B, and C is optimal. Indeed, in any solution, objects of A are alone in a bin, and there are at most two objects of B and C in a bin, with at most one object of B. The **first-fit-dec** algorithm is placing first each object A and B in a separate bin, then it does the best matching of objects C, because they are placed in the bins by decreasing order. In this case, **first-fit-dec** is optimal. □

Note that the reasoning does not hold if the categories are made differently, with, for instance, $\frac{1}{4}$ instead of $\frac{1}{3}$. Indeed, we can then fit three objects of category C in a single bin, and the reasoning does not hold anymore. However, we point out that it is also possible to prove that $C_{\textbf{first-fit-dec}} \leqslant \frac{11}{9} C_{opt} + 1$, and we refer to [112] for further details. The idea of the proof is similar, but more categories of objects are considered, and the algorithm turns out to be much more complex.

Without allowing an extra bin, we can finally prove that **first-fit-dec** is a $\frac{3}{2}$-approximation algorithm.

THEOREM 8.8. First-fit-dec *is a $\frac{3}{2}$-approximation algorithm for the bin packing problem.*

Proof. Let $k = C_{\textbf{first-fit-dec}}$ be the cost returned by the **first-fit-dec** algorithm, and let $j = \lceil \frac{2}{3} k \rceil$. Bins are numbered from 1 to k, and we consider two cases.

Case 1: If bin j contains an object a_i such that $a_i > \frac{1}{2}$, then if $j' < j$, there is an object $a_{i'}$ in bin j' such that $a_{i'} \geqslant a_i > \frac{1}{2}$. This is true for $1 \leqslant j' < j$, and, therefore, there are at least j objects of size greater than $\frac{1}{2}$ that should be placed in distinct bins. This implies that the optimal cost C_{opt} is greater than j.

Case 2: None of the bins $j' \geqslant j$ contains any object of size strictly greater than $\frac{1}{2}$; there are at least two objects per bin, except for bin k that may contain only one object, hence $2(k-j)+1$ objects in bins $j, j+1, \ldots, k$. None of these objects fits into bins $1, 2, \ldots, j-1$, by definition of **first-fit-dec**. We show below that $2(k-j)+1 \geqslant j-1$, and by combining $j-1$ of these objects with each of the first $j-1$ bins, we obtain that the sum of the a_is is strictly greater than $j-1$, i.e., C_{opt} is greater than j. In order to prove the inequality $2(k-j)+1 \geqslant j-1$, we show that $j = \lceil \frac{2}{3} k \rceil \leqslant \frac{2}{3}(k+1)$. Let $y = j - \frac{2}{3}k$. Note that j and k are integers, and $0 \leqslant y < 1$. Moreover, $k = \frac{3}{2} j - \frac{3}{2} y$. If j is even, then $\frac{3}{2} j$ is an integer; therefore, $\frac{3}{2} y$ is an integer strictly smaller than $\frac{3}{2}$, i.e., $\frac{3}{2} y \leqslant 1$ and $y \leqslant \frac{2}{3}$. Otherwise, $\frac{3}{2} y + \frac{1}{2}$ is an integer, and because $y < 1$, we have $\frac{3}{2} y + \frac{1}{2} < 2$, i.e., $\frac{3}{2} y + \frac{1}{2} \leqslant 1$ and $y \leqslant \frac{1}{3}$. Altogether, $\lceil \frac{2}{3} k \rceil = j = \frac{2}{3} k + y \leqslant \frac{2}{3} k + \frac{2}{3}$.

In both cases, we have

$$C_{opt} \geqslant j = \left\lceil \frac{2}{3}k \right\rceil \geqslant \frac{2}{3} C_{\textbf{first-fit-dec}} \ ,$$

which concludes the proof. □

8.1.5 2-PARTITION

We discuss approximation algorithms for the 2-PARTITION problem. The optimization problem associated with 2-PARTITION is the following: Given n integers $a_1, \ldots, a_n$, find a subset I of $\{1, \ldots, n\}$ such that $\max\left(\sum_{i \in I} a_i, \sum_{i \notin I} a_i\right)$ is minimum. Note that the minimum is always at least $\max\left(P_{max}, P_{sum}/2\right)$, where $P_{max} = \max_{1 \leqslant i \leqslant n} a_i$ and $P_{sum} = \sum_{i=1}^{n} a_i$.

This problem is similar to a scheduling problem with two identical processors. There are n independent tasks $T_1, \ldots, T_n$, and task T_i $(1 \leqslant i \leqslant n)$ can be executed on one of the two processors in time a_i. The goal is to minimize the total execution time. The processors are denoted by P_1 and P_2.

We start by analyzing two greedy algorithms for this problem. Then, we show how to derive a PTAS for 2-PARTITION and even an FPTAS.

Greedy algorithms

The two natural greedy algorithms are the following. We choose tasks in a random order (online algorithm, **greedy-online**) or sorted by nonincreasing execution time (offline algorithm, **greedy-offline**), and we assign the chosen task to the processor that has the lowest current load.

The idea of sorting in the offline algorithm is that a task with a large execution time, if considered at the end of the algorithm, may unbalance the entire execution. However, the offline version requires that all execution times are known beforehand. The online algorithm can be applied in a problem where tasks arrive dynamically (for instance, scheduling user jobs on a biprocessor server).

THEOREM 8.9. **Greedy-online** *is a* $\frac{3}{2}$*-approximation algorithm, and* **greedy-offline** *is a* $\frac{7}{6}$*-approximation algorithm for the 2-PARTITION problem. Moreover, these approximation ratios are tight.*

Proof. First, we consider the **greedy-online** algorithm. Let us assume that processor P_1 finishes the execution at time $M_1 \geqslant M_2$ (where M_2 is the time at which P_2 finishes its execution), and that T_j is the last task executed on P_1. We have $M_1 + M_2 = P_{sum}$. Moreover, since the greedy algorithm chose processor P_1 to execute task T_j, it means that $M_1 - a_j \leqslant M_2$; otherwise, T_j would have been scheduled on P_2. Finally, the cost of **greedy-online** is such that:

$$C_{\textbf{online}} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leqslant \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(P_{sum} + a_j),$$

and since $C_{opt} \geqslant P_{sum}/2$ and $C_{opt} \geqslant a_i$ for $1 \leqslant i \leqslant n$, we have $C_{\textbf{online}} \leqslant C_{opt} + \frac{1}{2}C_{opt} = \frac{3}{2}C_{opt}$, which concludes the proof.

For the offline version of the greedy algorithm, we start as before, but we refine the inequality $a_j \leqslant C_{opt}$. If $a_j \leqslant \frac{1}{3}C_{opt}$, we obtain the approximation ratio of the theorem, i.e., $C_{\textbf{offline}} \leqslant \frac{7}{6}C_{opt}$. We focus now on the case where $a_j > \frac{1}{3}C_{opt}$. Then, $j \leqslant 4$. Indeed, if a_j were the fifth task, because the tasks are sorted by nonincreasing execution times, there would be at least five tasks of time at least $\frac{1}{3}C_{opt}$, and any schedule would need to schedule at least three of these tasks on the same processor, leading to an execution time strictly greater than C_{opt}, and hence a contradiction. Then, we note that, in this case, the cost $C_{\textbf{offline}}$ when we restrict to the scheduling of the first four tasks is identical to the cost when scheduling all tasks. Finally, it is easy to check (exhaustively) that **greedy-offline** is optimal when scheduling at most four tasks. We conclude that $C_{\textbf{offline}} = C_{opt}$ in this case, which ends the proof.

Finally, we prove that the ratios are tight. For **greedy-online**, we consider an instance with two tasks of time 1 and one task of time 2. The greedy algorithm schedules the tasks in time 3 (each task of time 1 on a distinct processor, then the task of time 2 after one of those), while the optimal algorithm takes a time 2 (with the two first tasks on the same processor). For **greedy-offline**, we consider an instance with two tasks of time 3 and three tasks of time 2. The greedy algorithm schedules each task of time 3 on a distinct processor, leading to a total execution time of 7, while the optimal solution consists of grouping those two tasks on the same processor, with a total time of 6. □

PTAS: A $(1+\varepsilon)$-approximation algorithm

THEOREM 8.10. *$\forall \varepsilon > 0$, there is a $(1+\varepsilon)$-approximation algorithm for the 2-PARTITION problem. In order words, 2-PARTITION has a PTAS.*

Proof. We consider an instance $\mathcal{I}$ of 2-PARTITION, $a_1, \ldots, a_n$ (recall that the a_is can be interpreted as the execution time of tasks), and $\varepsilon > 0$.

We classify the tasks into two categories. Let $L = \max(P_{max}, P_{sum}/2)$. The *big* tasks are in the set $T_{big} = \{i \mid a_i > \varepsilon L\}$, while the *small* tasks are in the set $T_{small} = \{i \mid a_i \leqslant \varepsilon L\}$. We consider an instance $\mathcal{I}^*$ of the problem with the tasks of T_{big}, and $\left\lfloor \frac{S}{\varepsilon L} \right\rfloor$ tasks of identical size εL, where $S = \sum_{i \in T_{small}} a_i$.

The proof goes as follows. We show that the optimal schedule for instance $\mathcal{I}^*$ has a cost C^*_{opt} close to the cost C_{opt} of the optimal schedule for instance $\mathcal{I}$, i.e., $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$. Moreover, it is possible to compute the optimal schedule for instance $\mathcal{I}^*$ in a polynomial time. Building upon this schedule, we finally construct a solution to the original instance $\mathcal{I}$, with a guaranteed cost.

First, we prove that $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$. Let *opt* be an optimal schedule for instance $\mathcal{I}$, of cost C_{opt}. Then, let S_1 (resp. S_2) be the sum of the small tasks in this optimal schedule on processor P_1 (resp. P_2). We build a new schedule *sched** in which the big tasks of the optimal schedule *opt* remain on the same processors, but small tasks are replaced with $\left\lceil \frac{S_i}{\varepsilon L} \right\rceil$ tasks of size εL on processor P_i, for $i = 1, 2$. Because

$$\left\lceil \frac{S_1}{\varepsilon L} \right\rceil + \left\lceil \frac{S_2}{\varepsilon L} \right\rceil \geqslant \left\lfloor \frac{S_1 + S_2}{\varepsilon L} \right\rfloor = \left\lfloor \frac{S}{\varepsilon L} \right\rfloor ,$$

we have scheduled at least as many tasks of size εL as the total number of small tasks in instance $\mathcal{I}^*$. Moreover, the execution time on processor P_i, for $i = 1, 2$, has been increased of at most

$$\left\lceil \frac{S_i}{\varepsilon L} \right\rceil \times \varepsilon L \;-\; S_i \leqslant \varepsilon L ,$$

which means that the cost of this schedule is such that $C^*_{sched} \leqslant C_{opt} + \varepsilon L$. Moreover, this schedule is a schedule for instance $\mathcal{I}^*$ and, therefore, $C^*_{sched} \geqslant C^*_{opt}$. Finally, $C^*_{opt} \leqslant C_{opt} + \varepsilon L \leqslant C_{opt} + \varepsilon \times C_{opt}$, which concludes the proof that $C^*_{opt} \leqslant (1 + \varepsilon)C_{opt}$.

Next, we discuss how to find an optimal schedule for instance $\mathcal{I}^*$. First, we provide a bound on the number of tasks in $\mathcal{I}^*$. Because we replaced small tasks of $\mathcal{I}$ with tasks of size εL, we have not increased the total execution time, which is at most $P_{sum} \leqslant 2L$. Each task of $\mathcal{I}^*$ has an execution time of at least εL (small tasks), so there are at most $\frac{2L}{\varepsilon L} = \frac{2}{\varepsilon}$ tasks. Note that this is a constant number because ε is a constant. Moreover, we note that the size of $\mathcal{I}^*$ is polynomial in the size of instance $\mathcal{I}$ (because the size of $\mathcal{I}^*$ is a constant). We can optimally schedule $\mathcal{I}^*$ by trying all $2^{\frac{2}{\varepsilon}}$ possible schedules and keeping the best one. Of course, this algorithm is not polynomial in $1/\varepsilon$, but it is polynomial in the size of the instance $\mathcal{I}$ because it is a constant.

Now we have an optimal schedule *opt** for instance $\mathcal{I}^*$, of cost C^*_{opt}, and we aim to build a schedule *sched* for instance $\mathcal{I}$. For $i = 1, 2$, we let $L^*_i = B^*_i + S^*_i$ be the total execution time of processor P_i in the schedule *opt**, where B^*_i (resp. S^*_i) is the time spent on big (resp. small) tasks. Then, we build the schedule *sched* in which the big tasks are kept on the same processor as in *opt**, and we greedily assign small tasks to processors. First, we assign small tasks to processor P_1 until their processing time does not exceed $S^*_1 + 2\varepsilon L$. Then, we schedule the remaining small tasks to processor P_2. Let us prove now that once all small tasks have been scheduled, the execution time has not increased by more than $2\varepsilon L$.

Because small tasks have a size of at most εL, the greedy algorithm assigns at least a total of $S^*_1 + \varepsilon L$ small tasks on processor P_1. Then, there are at most a total of $S - (S^*_1 + \varepsilon L)$ small jobs to assign to processor P_2. However, by construction of $\mathcal{I}^*$, we have $S^*_1 + S^*_2 = \varepsilon L \left\lfloor \frac{S}{\varepsilon L} \right\rfloor > S - \varepsilon L$ and, therefore, $S - (S^*_1 + \varepsilon L) \leqslant S^*_2$, and the execution time of P_2 in the new schedule *sched* is not greater than in the schedule *opt**.

The schedule *sched* is a schedule for instance $\mathcal{I}$, which is built in polynomial time. The cost of this schedule is at most $C_{sched} \leqslant C^*_{opt} + 2\varepsilon L$. We use the previous result that $C^*_{opt} \leqslant (1+\varepsilon)C_{opt}$ and the fact that $L \leqslant C_{opt}$ to conclude that $C_{sched} \leqslant C^*_{opt} + 2\varepsilon L \leqslant (1+3\varepsilon)C_{opt}$. This is true for all ε, so we can apply this algorithm with $\varepsilon/3$ to obtain the desired ratio. $\square$

Note that a simpler proof can be done by using an optimal schedule for the big tasks, of cost C_{big}, and the **greedy-online** algorithm introduced above. Once the small tasks have been scheduled greedily on the two processors, there are two cases. If the total time has not changed, i.e., it is C_{big}, it is optimal. Otherwise, the processor that ends the execution is executing a small task a_j. This means that before the greedy choice of scheduling task a_j onto this processor, the finishing time of the processor was less than $P_{sum}/2$; otherwise, task a_j would have been assigned to the other processor because of the greedy choice. Finally, the cost of the schedule returned by this algorithm is at most $P_{sum}/2 + a_j \leqslant L + \varepsilon L \leqslant (1+\varepsilon)C_{opt}$.

A PTAS provides an approximate solution that is as close to the optimal as one wants. The only downside is that the algorithm running time increases with the quality of the approximate solution. Some readers may thus be puzzled by the idea of having a PTAS or an FPTAS for an NP-complete problem whose objective function takes values in a discrete set, such as 2-PARTITION. Indeed, a PTAS, for such a problem, enables one to obtain an optimal solution whenever one is ready to pay the cost. Let us consider any given instance $\mathcal{I}$ of 2-PARTITION. Let S be the sum of the elements of $\mathcal{I}$. If $\varepsilon < \frac{1}{S}$, then any $1+\varepsilon$ approximation produces an optimal solution. Indeed, $(1+\varepsilon)C_{opt} < (1+\frac{1}{S})C_{opt} \leqslant C_{opt} + 1$ because $C_{opt} \leqslant S$ and because the objective function can take only integral values. This may be surprising at first sight, but it does not contradict anything we have written so far. One should not forget that the running time of an FPTAS is polynomial in the size of $\frac{1}{\varepsilon}$, that is, in our example, in the size of S. The running time of a PTAS can even be exponential in the size of $\frac{1}{\varepsilon}$. Finding the optimal solution for 2-PARTITION in time exponential in the size of S is quite simple. One generates all the subsets of $\mathcal{I}$ and computes the sum of the elements of each subset. If $\mathcal{I}$ includes n elements, there are $2^n = O(2^S)$ subsets of $\mathcal{I}$. The sum of the elements of each of them is computed in time $O(n)$ and thus $O(S)$. Therefore, readers should not be surprised that, for a given value of ε, an algorithm whose running time is polynomial in the size of the instance can find an optimal solution to 2-PARTITION.

FPTAS for 2-PARTITION

We have provided a PTAS for 2-PARTITION, but the algorithm finds an optimal schedule for instance $\mathcal{I}^*$ (i.e., an optimal schedule of the big tasks), and this is not polynomial in $1/\varepsilon$. Below, we provide an FPTAS, i.e., a $(1+\varepsilon)$-approximation algorithm that is polynomial in the size of $\mathcal{I}$ and in $1/\varepsilon$.

THEOREM 8.11. $\forall\varepsilon > 0$, *there is a* $(1+\varepsilon)$*-approximation algorithm for the 2-PARTITION problem that is polynomial in* $1/\varepsilon$*. In order words, 2-PARTITION has an FPTAS.*

Proof. The idea of the proof is to encode the schedules as *vector sets*, in which the first (resp. second) element of a vector represents the running time of the first (resp. second) processor. Formally, for $1 \leqslant k \leqslant n$, VS_k is the set of vectors representing schedules of tasks $a_1, \ldots, a_k$: $VS_1 = \{[a_1, 0], [0, a_1]\}$, and we build VS_k from VS_{k-1} as follows. For all $[x, y] \in VS_{k-1}$, we add $[x + a_k, y]$ and $[x, y + a_k]$ to VS_k. The optimal schedule is represented by a vector $[x, y] \in VS_n$, and it is such that $\max(x, y)$ is minimized.

The approximation algorithm enumerates all possible schedules, but some of them are discarded on the fly so that we keep a polynomial algorithm.

Let $\Delta = 1 + \frac{\varepsilon}{2n}$. We partition the square $P_{sum} \times P_{sum}$ following the power of Δ, from 0 to Δ^M. We have $M = \lceil \log_\Delta(P_{sum}) \rceil = \left\lceil \frac{\ln(P_{sum})}{\ln(\Delta)} \right\rceil \leqslant \left\lceil \left(1 + \frac{2n}{\varepsilon}\right) \ln(P_{sum}) \right\rceil$. Indeed, note that if $z \geqslant 1$, then $\ln(z) \geqslant 1 - \frac{1}{z}$.

The idea of the algorithm consists of building the vector sets but adding a new vector to a set only if there are no other vectors in the same square of the partitioned $P_{sum} \times P_{sum}$ square. Because M is polynomial in $1/\varepsilon$ and in $\ln(P_{sum})$, and the size of instance $\mathcal{I}$ is greater than $\ln(P_{sum})$, the algorithm is polynomial both in the size of $\mathcal{I}$ and in $1/\varepsilon$. We need to prove that this algorithm is a $(1 + \varepsilon)$-approximation to conclude the proof.

First, let us formally describe the algorithm. Initially, $VS_1^\# = VS_1$. Then, for $2 \leqslant k \leqslant n$, we build $VS_k^\#$ from $VS_{k-1}^\#$ as follows. For all $[x, y] \in VS_{k-1}^\#$, we add $[x + a_k, y]$ (resp. $[x, y + a_k]$) to $VS_k^\#$ if and only if there is no vector from $VS_k^\#$ in the same square. Note that two vectors $[x_1, y_1]$ and $[x_2, y_2]$ are in the same square if and only if $\frac{x_1}{\Delta} \leqslant x_2 \leqslant \Delta x_1$ and $\frac{y_1}{\Delta} \leqslant y_2 \leqslant \Delta y_1$.

We keep at most one vector per square at each step, which gives an overall complexity in $n \times M^2$, which is polynomial both in the size of instance $\mathcal{I}$ and in $1/\varepsilon$.

Next, we prove that for all $1 \leqslant k \leqslant n$ and $[x, y] \in VS_k$ there exists $[x^\#, y^\#] \in VS_k^\#$ such that $x^\# \leqslant \Delta^k x$ and $y^\# \leqslant \Delta^k y$. The proof is done recursively. The result is trivial for $k = 1$. If we assume that the result is true for $k - 1$, then let us consider $[x, y] \in VS_k$. Either $x = u + a_k$ and $y = v$ (case 1), or $x = u$ and $y = v + a_k$ (case 2), with $[u, v] \in VS_{k-1}$. By recursion hypothesis, there exists $[u^\#, v^\#] \in VS_{k-1}^\#$ with $u^\# \leqslant \Delta^{k-1} u$ and $v^\# \leqslant \Delta^{k-1} v$. For case 1, note that $[u^\# + a_k, v^\#]$ may not be in $VS_k^\#$, but we know that there is at least one vector in the same square in $VS_k^\#$; there exists $[x^\#, y^\#] \in VS_k^\#$ such that $x^\# \leqslant \Delta\left(u^\# + a_k\right)$ and $y^\# \leqslant \Delta v^\#$. Finally, we have $x^\# \leqslant \Delta^k u + \Delta a_k \leqslant \Delta^k (u + a_k) = \Delta^k x$ and $y^\# \leqslant \Delta v^\# \leqslant \Delta^k y$, and case 2 is symmetrical. This proves the result.

For $k = n$, we can deduce that $\max(x^\#, y^\#) \leqslant \Delta^n \max(x, y)$. There remains to be proven that $\Delta^n \leqslant (1+\varepsilon)$, where $\Delta^n = \left(1 + \frac{\varepsilon}{2n}\right)^n$. We rearrange the last inequality and study the function $f(z) = \left(1 + \frac{z}{n}\right)^n - 1 - 2z$, for $0 \leqslant z \leqslant 1$, $f'(z) = \frac{1}{n} n \left(1 + \frac{z}{n}\right)^{n-1} - 2$. We deduce that f is a convex function, and that its minimum is reached in $\lambda_0 = n\left(\sqrt[n-1]{2} - 1\right)$. Moreover, $f(0) = -1$ and $f(1) = \left(1 + \frac{1}{n}\right)^n - 3 \leqslant 0$. Because f is convex, and $f(z) \leqslant 0$ for $z = 0$ and $z = 1$, we can deduce that $f(z) \leqslant 0$ for $0 \leqslant z \leqslant 1$. This concludes the proof. □

8.2 Polynomial problem instances

When confronted with an NP-complete problem, one algorithmic solution consists of finding good approximation algorithms. While some problems may have good approximation schemes, such as PTAS or FPTAS (see Section 8.1), some problems cannot be approximated. However, with a slight change of the problem parameters (constant value for a parameter, different rule of the game, etc.), it may be possible to find a good approximation algorithm or even to be able to solve the problem in pseudopolynomial or polynomial time.

The analysis of a problem is comprehensive when we are able to identify at which point the problem becomes NP-complete and then at which point the problem cannot be approximated any more. We refine the problem complexity as follows:

- The class P consists of all optimization problems that can be solved in polynomial time.
- The class FPTAS consists of all optimization problems that have an FPTAS, and it contains P.
- The class PTAS consists of all optimization problems that have a PTAS, and it contains FPTAS.
- The class APX consists of all optimization problems that have a polynomial-time approximation algorithm with a constant ratio, and it contains PTAS.
- Finally, the class NP contains APX: Some problems may be in NP but not in APX.

We also consider the class of problems that can be solved in pseudopolynomial time, PPT. This class includes P but none of the other previous classes. Some problems that can be solved in pseudopolynomial time may not have an FPTAS or may not even be in APX. The problems of (i) finding a pseudopolynomial-time algorithm to solve the problem exactly and (ii) finding good polynomial-time approximation algorithms are not correlated.

In the following, we illustrate how the problem can move from one category to another when parameters are modified. In particular, we check whether the problem can be solved in polynomial time or in pseudopolynomial time, and if there is no polynomial-time algorithm to solve the problem, we investigate polynomial-time approximation algorithms.

8.2.1 Partitioning problems

First, we provide both the optimization and decision versions of the partitioning problem that we consider, and then we investigate variants of the problem.

Optimization problem (PART-OPT). Let $a_1, \dots, a_n$ be n positive integers. The goal is to partition these integers into p subsets $A_1, \dots, A_p$, in order to minimize the maximum (over all subsets) of the sum of the integers in a subset:

$$\min \left(\max_{1 \leqslant j \leqslant p} \sum_{i \in A_j} a_i \right) .$$

Decision problem (PART-DEC). The associated decision problem is the following: Let $a_1, \dots, a_n$ be n positive integers. Given a bound K, is it possible to partition these integers into p subsets $A_1, \dots, A_p$, such that the sum of the integers in each subset does not exceed K? In other words,

$$\text{for all } \ 1 \leqslant j \leqslant p, \quad \sum_{i \in A_j} a_i \leqslant K .$$

We can easily prove, from a reduction from 3-PARTITION, that PART-DEC is NP-complete in the strong sense. No pseudopolynomial algorithm is known to solve PART-DEC. However, PART-OPT is a classical scheduling problem. The goal is to schedule n independent tasks onto p processors, where a_i is the execution time of task T_i, for $1 \leqslant i \leqslant n$, and the goal is to minimize the total execution time. There is a PTAS to approximate this problem [49].

One way to simplify the problem is to restrict it to the case $p = 2$. The problem is then equivalent to 2-PARTITION, and it can be solved in pseudopolynomial time using a dynamic-programming algorithm (see Section 6.2.1). Moreover, this problem is in the class FPTAS, as was shown in Section 8.1.5.

In order to identify polynomial instances of this problem, we consider the following variants:

1. We consider the case in which all integers are equal, i.e., $a_1 = a_2 = \cdots = a_n = a$. In this case, we can find the solution to the optimization problem, which is simply $\left\lceil \frac{n}{p} \right\rceil \times a$. Therefore, we also can solve the decision problem in polynomial time, even in constant time.

2. We change the rule of the game. The subsets must contain only continuous elements, for instance, $[a_i, a_{i+1}, \dots, a_{i'}]$. The subsets are then

intervals, and the problem can be solved in polynomial time. It is the classical chains-on-chains partitioning problem (see Chapter 11), which can be solved, for instance, with a dynamic-programming algorithm in time $O(n^2 \times p)$.

If we consider the problem as a scheduling problem where we must schedule n tasks onto p processors, we can conclude that the problem becomes difficult (NP-complete) as soon as the tasks are different (the case of identical tasks is case 1) and as soon as we are allowed any mapping (no fixed ordering to enforce, such as in case 2). Moreover, while the problem is in PPT and has an FPTAS with $p = 2$ processors, it is no longer in PPT for an arbitrary number of processors and has only a PTAS.

For a deeper analysis of partitioning problems, the interested reader can refer to the chains-on-chains partitioning case study (Chapter 11).

8.2.2 Assessing problem complexity

In this section, we mention two classical approaches when facing NP-complete problems and aiming at identifying polynomial instances. We illustrate these approaches with two different problems.

The first problem is a routing problem, which is discussed extensively in Chapter 13. Given a directed graph $G = (V, E)$ and a set of terminal pairs $\mathcal{R} = \{R_i = (s_i, t_i)\}$, the goal is to connect as many pairs as possible using edge-disjoint simple paths. In a solution $\mathcal{A}$, each $R_i \in \mathcal{A}$ must be assigned a simple path π_i from s_i to t_i in G so that no two paths π_i and π_j, where $R_i \in \mathcal{A}$, $R_j \in \mathcal{A}$ and $i \neq j$, have an edge in common.

The goal is to maximize $|\mathcal{A}|$, the cardinality of $\mathcal{A}$, i.e., the number of connected terminal pairs. It turns out that this routing problem is NP-complete, and Chapter 13 presents approximation algorithms. But how can we find polynomial instances? A first idea is to bound the number of terminal pairs with a constant, but this does not work, as it turns out that the problem remains NP-complete with only two terminal pairs [35]. Another idea is to restrict the problem to some special classes of graphs. We show in Chapter 13 that the problem is polynomial for linear chains and stars, regardless of the number of terminal pairs.

The second problem is a geometric problem, which is investigated in Chapter 14. How can we partition the unit square into p rectangles of given area $s_1, s_2, \ldots, s_p$ (such that $\sum_{i=1}^{p} s_i = 1$) so as to minimize the sum of the p half perimeters of the rectangles? In Chapter 14, we explain the relevance of this problem to parallel computing, and we show that it is NP-complete. What can we do here? The problem becomes polynomial if we restrict to same-size rectangles [64], but this is very restrictive. Another approach is to change the rules of the game and ask for some specific partitioning of the unit square. Indeed, we show in Chapter 14 that the problem becomes polynomial when restricting to column-based partitioning, i.e., imposing that the rectangles are arranged along several columns within the unit square. Going further in that direction, we show that the optimal column-based partitioning is indeed a good approximation of the general solution. We hope that this short discussion will urge the reader to read the full case study of Chapter 14.

8.3 Linear programming

Sometimes the solution of an NP-complete problem can be expressed as the solution of an integer linear program. Once we have written an optimization problem as an integer linear program, we can do three things:

1. Solve the integer linear program to obtain optimal solutions for (very) small instances.
2. Relax the integer linear program into a (rational) linear program and solve it to obtain a bound on the optimal solution for the original problem.
3. Relax the integer linear program into a (rational) linear program, solve the latter program to obtain a rational solution, and build an integral solution from the rational one.

We first introduce the necessary notions and definitions (Section 8.3.1). Then we describe several *rounding* approaches to transform a solution of a relaxed linear program into a solution of the original integer linear program (Section 8.3.2).

8.3.1 Formal definition

Linear programming is a mathematical method in which an optimization problem is expressed as the minimization (or maximization) of a linear function whose arguments are constrained by a set of affine equations and inequalities.

DEFINITION 8.6 (Linear program)**.** A linear program is an optimization problem of the form:

$$\textsc{Minimize} \quad c^T \cdot x \quad \textsc{subject to}$$
$$Ax \leqslant b \quad \text{and} \quad x \geqslant 0$$

where x is an (unknown) vector of variables of size n, A is a (known) matrix of coefficients of size $m \times n$, and b and c are the two (known) vectors of coefficients of respective size m and n (and where c^T is the transpose of vector c).

An *integer linear program* is a linear program whose variables can take only integral values. A *mixed linear program* is a linear program in which some variables must take integral values and some can take rational values.

In the above formal definition, linear programs are given under a canonical form. Therefore, the formal definition of linear programs may look more restrictive than the informal definition we gave right before the formal one. In fact, both definitions are equivalent:

- A maximization problem with the objective function $c^T \cdot x$ is equivalent to a minimization problem with the objective function $-c^T \cdot x$.
- An equality $d^T \cdot x = e$ is equivalent to the set of two inequalities:

$$\begin{cases} d^T \cdot x \leqslant e \\ -d^T \cdot x \leqslant -e. \end{cases}$$

- A variable that can take both positive and negative values can be equivalently replaced by the difference of two nonnegative variables.

An example: Weighted vertex cover

In Section 8.1.2, we have seen the classical version of the vertex cover problem. Given a graph $G = (V, E)$, we want to return a set U of vertices ($U \subset V$) of minimum size that is covering all edges, i.e., such that for each edge $e = (i, j) \in E$, $i \in U$ and/or $j \in U$.

Here, we consider the weighted version of this problem. We assign a weight w_i to each vertex $i \in V$. The problem is then to minimize $\sum_{i \in U} w_i$, where U is once again a vertex cover. This problem amounts to the classical one if $w_i = 1$ for all $i \in V$ and is also NP-complete.

We express this minimization problem as an integer linear program. We introduce a set of Boolean variables, one for each vertex, stating whether the corresponding vertex belongs to the cover. Let x_i be the variable associated with vertex $i \in V$. We will have $x_i = 1$ if i belongs to the cover ($i \in U$) and $x_i = 0$ otherwise.

$$\text{MINIMIZE} \sum_{i \in V} x_i w_i \quad \text{SUBJECT TO}$$

$$\begin{cases} \forall (i,j) \in E & -x_i - x_j \leqslant -1 \\ \forall i \in V & x_i \leqslant 1 \\ \forall i \in V & x_i \geqslant 0 \end{cases} \tag{8.1}$$

We now show that solving the Integer Linear Program (8.1), with $x_i \in \{0, 1\}$, is absolutely equivalent to solving the minimum weighted vertex cover problem for the graph G.

One can easily check that, if U is an optimal solution to the weighted vertex cover problem, then, by letting $x_i = 1$ for any vertex i in U and $x_j = 0$ for any vertex j not in U, one builds a solution to the above linear program for which the objective function takes the value of the cost of the cover U.

Reciprocally, consider an optimal solution to the Integer Linear Program (8.1), with $x_i \in \{0, 1\}$. From this solution, we build a subset U of V as follows. For any vertex i of V, i belongs to U if and only if $x_i = 1$. For any edge

$e = (i, j) \in E$ we have $-x_i - x_j \leqslant -1$, which is equivalent to $x_i + x_j \geqslant 1$. In other words, either x_i or x_j or both variables are equal to 1 (remember that here the x_is are integer variables). Therefore, at least one of the two vertices i and j is a member of U, and U is thus a cover. The objective function is obviously the cost of the cover U. Therefore, U is a cover of minimum weight.

Complexity

In the general case, the decision problem associated with the problem of solving integer linear programs is an NP-complete problem [58, 38]. However, (rational) linear programs can be solved in polynomial time [93]. Hence, the motivation, when confronted with an NP-complete problem, is to express it as an integer or mixed linear program and then to solve this program as if it were a rational linear program. This method is called *relaxation*. However, the solution obtained this way may be meaningless. For instance, in the case of the linear program for the weighted vertex cover problem (Linear Program (8.1)), one of the variables x_i can have a value different from 0 and 1, which does not make any sense because a vertex cannot be *partially* included in the solution. The problem then becomes how to build an integral solution from a rational one. We now focus on this problem, which is called *rounding*.

8.3.2 Relaxation and rounding

Rounding to the nearest integer

The simplest rounding method is the rounding of any rational variable to the nearest integer. (Obviously, this method is not fully defined because one will still have to decide how to handle variables whose values are of the form $z+0.5$ where z is an integer.) We illustrate this method with the weighted vertex cover problem.

Algorithm **lp-wvc** is defined as follows. First, solve the Linear Program (8.1) over the rationals rather than on the integers, and let $\{x_i^*\}_{i \in V}$ be the found optimal solution. Then, any vertex i of V belongs to the cover U if and only if $x_i^* \geqslant \frac{1}{2}$. In other words, we build from the x_i^*s the Boolean variables x_is, by: $x_i = 1 \Leftrightarrow x_i^* \geqslant \frac{1}{2}$. Not only is Algorithm **lp-wvc** correct, it is even an approximation algorithm, as we now prove.

THEOREM 8.12. **lp-wvc** *is a* 2*-approximation algorithm for weighted vertex cover.*

Proof. First, we check that **lp-wvc** returns a cover. Let $(i, j) \in E$ be an edge. Then, because the x_i^*s are a rational solution to the linear program, we have $x_i^* + x_j^* \geqslant 1$, and at least one of them is greater than or equal to 1/2. Therefore, in the solution of our problem, we have either $x_i = 1$ or $x_j = 1$ (we also can have $x_i = x_j = 1$). Therefore, the edge (i, j) is covered, $x_i + x_j \geqslant 1$.

To prove that the algorithm is a 2-approximation, we compare the cost of the algorithm $C_{\textbf{lp-wvc}} = \sum_{i \in V} x_i w_i$ with the cost of an optimal solution C_{opt}.

The result comes from two observations: (i) For all i, we have $x_i \leqslant 2x_i^*$ (whether i has been chosen to be part of the cover or not), and (ii) the optimal solution of the linear program over the integers has necessarily a higher cost than the rational solution (the integer solution is a solution to the rational problem). Because $\sum_{i\in V} x_i^* w_i$ is an optimal solution to the rational problem, $C_{opt} \geqslant \sum_{i\in V} x_i^* w_i$. Finally, we have

$$C_{\textbf{lp-wvc}} = \sum_{i\in V} x_i w_i \leqslant \sum_{i\in V} (2x_i^*) w_i \leqslant 2C_{opt},$$

which concludes the proof. □

Threshold rounding

We do not have any a priori guarantee that the rounding to the nearest integer will produce a valid integer solution. We illustrate this potential problem with the set cover problem.

DEFINITION 8.7 (SET-COVER)**.** Let V be a set. Let $\mathcal{S}$ be a collection of k subsets of V: $\mathcal{S} = \{S_1, \ldots, S_k\}$ where, for $1 \leqslant i \leqslant k$, $S_i \subset V$. Let K be an integer, with $K < k$. Is there a subcollection of at most K elements of $\mathcal{S}$ that covers all elements of V?

SET-COVER is an NP-complete problem [58, 38]. It easily can be coded as an integer linear program. Let $\delta_{i,j}$ be a Boolean constant indicating whether the element $v \in V$ belongs to the subset $s \in \mathcal{S}$. As previously, variable x_s indicates whether the set $s \in \mathcal{S}$ belongs to the solution. The following integer linear program then searches for a minimum set cover. The first inequality just states that, whatever the element v of V, at least one of the subsets containing v must be picked in the solution.

$$\textsc{Minimize} \quad \sum_{s\in\mathcal{S}} x_s \quad \text{SUBJECT TO}$$

$$\begin{cases} \forall v \in V & -\sum_{s\in S} \delta_{v,s} x_s \leqslant -1 \\ \forall s \in \mathcal{S} & x_s \leqslant \ \ 1 \\ \forall s \in \mathcal{S} & -x_s \leqslant \ \ 0 \end{cases} \tag{8.2}$$

Now, consider the following particular instance of minimum cover: $V = \{a, b, c, d\}$ and $\mathcal{S} = \{S_1 = \{a, b, c\}, S_2 = \{a, b, d\}, S_3 = \{a, c, d\}, S_4 = \{b, c, d\}\}$. One can easily see that any two elements of $\mathcal{S}$ define an optimal solution. We write explicitly the Linear Program (8.2) for that instance:

$$\text{Minimize} \quad x_{S_1} + x_{S_2} + x_{S_3} + x_{S_4} \quad \text{subject to}$$

$$\left\{\begin{array}{lr} & -x_{S_1} - x_{S_2} - x_{S_3} \leqslant -1 \\ & -x_{S_1} - x_{S_2} - x_{S_4} \leqslant -1 \\ & -x_{S_1} - x_{S_3} - x_{S_4} \leqslant -1 \\ & -x_{S_2} - x_{S_3} - x_{S_4} \leqslant -1 \\ \forall s \in \{S_1, S_2, S_3, S_4\} & x_s \leqslant 1 \\ \forall s \in \{S_1, S_2, S_3, S_4\} & -x_s \leqslant 0. \end{array}\right. \tag{8.3}$$

By summing the first four inequalities, we obtain $x_{S_1} + x_{S_2} + x_{S_3} + x_{S_4} \geqslant \frac{4}{3}$. Hence, the optimal value of the objective function is not smaller than $\frac{4}{3}$. Then, one can check that $x^*_{S_1} = x^*_{S_2} = x^*_{S_3} = x^*_{S_4} = \frac{1}{3}$ defines an optimal solution of the relaxed (rational) version of the Linear Program (8.3). Rounding this optimal rational solution to the nearest integer would lead to $x_{S_1} = x_{S_2} = x_{S_3} = x_{S_4} = 0$, which, obviously, does not define a cover. To circumvent this problem, rather than to round each variable to the nearest integer, one can use a generalization of this technique: threshold rounding. When variables are 0-1 variables, that is, when variables can take only the values 0 or 1, one first sets a threshold and then rounds to 1 exactly those variables whose values are not smaller than the threshold. This technique leads to an approximation algorithm for the minimum set cover problem.

THEOREM 8.13. *Let $\mathcal{P} = (V, \mathcal{S})$ be an instance of the minimum set cover problem in which each element of V belongs to at most p elements of $\mathcal{S}$. Then, solving the Linear Program* (8.3) *over the rationals and rounding the solution with the threshold $\frac{1}{p}$ builds a cover whose size is at most p times the optimal.*

Proof. Let us consider an optimal solution x^* of the relaxed linear program. Let v be any element of V. By definition of p, v belongs to $q \leqslant p$ elements of $\mathcal{S}$: $S_{\sigma(1)}, \ldots, S_{\sigma(q)}$. The Linear Program (8.2) contains the constraint $-x^*_{S_{\sigma(1)}} - x^*_{S_{\sigma(2)}} - \cdots - x^*_{S_{\sigma(q)}} \leqslant -1$. Therefore, there exists at least one $i \in [1, q]$ such that $x_{S_{\sigma(i)}} \geqslant \frac{1}{q} \geqslant \frac{1}{p}$ and the solution contains at least one element of $\mathcal{S}$ that includes v, namely, $S_{\sigma(i)}$. Thus, the solution is a valid cover. Then, for any element s of $\mathcal{S}$, $x_s \leqslant p \times x^*_s$. Indeed, if $x^*_s \geqslant \frac{1}{p}$, then $x_s = 1$ and $x_s = 0$ otherwise. This completes the proof for the approximation ratio. □

Randomized rounding

In the previous two approaches, the value of a variable in a rational solution was considered to be a deterministic indication of what should be the value of this variable in an integer solution. In the randomized rounding approach, the fractional part of such a value is interpreted as a probability.

Let us consider a nonintegral component x_i^* of an optimal rational solution x^*, and let y_i^* be its fractional part: $x_i^* = \lfloor x_i^* \rfloor + y_i^*$, with $0 < y_i^* < 1$. Then, in randomized rounding, y_i^* is considered to be the probability that, in the integral solution, x_i will be equal to $\lceil x_i^* \rceil$ rather than to $\lfloor x_i^* \rfloor$. In practice, using any uniform random generator over the interval $[0, 1]$, one generates a number $r \in [0, 1]$. If $r \geqslant y_i^*$, then we let $x_i = \lceil x_i^* \rceil$, and $x_i = \lfloor x_i^* \rfloor$ otherwise.

Iterative rounding

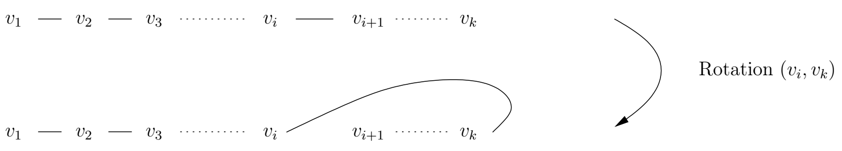
In all the previously described rounding approaches, a single relaxed linear program is solved, and then one tries to build an integral solution from the rational solution. A potential problem of these approaches is that the assignment of a particular value to one of the variables may force the value of some other variables in any valid solution. For instance, let us go back to the example showing that rounding to the nearest integer could lead to nonfeasible solutions to the minimum cover problem. There, setting $x_{S_1} = 0$ and $x_{S_2} = 0$ imposes that $x_{S_3} = x_{S_4} = 1$ (because, respectively, a and b must be covered). Rounding to the nearest integer ignores this implication and leads to an infeasible solution. A way to avoid such a problem is to assign values only to a subset of the variables and then solve the relaxed version of the linear program while taking into account the assignments made so far. This way, we obtain a new rational solution where fewer variables have nonintegral values. The process is then iterated until an integral solution is built (or the transformed linear program has no solution). The smaller the number of variables assigned at each iteration, the higher the probability to end up with a valid solution but also the higher the number of iterations, the complexity, and the execution time.

8.4 Randomized algorithms

In this section, we briefly explore how randomized algorithms can help deal with NP-complete problems. We restrict ourselves to a randomized algorithm to solve the NP-complete HC problem (recall that HC stands for Hamiltonian Cycle, see Definition 6.4, p. 130). Given an undirected graph, the algorithm incrementally builds a cycle, taking random decisions on the next vertex to visit to augment the current path. The algorithm will indeed output a Hamiltonian cycle with high probability as soon as the graph contains enough edges. We will quantify this last statement in what follows.

8.4.1 The algorithm

Consider a graph $G = (V, E)$. How can we build a Hamiltonian cycle in G by taking random decisions? The first idea is to grow a path iteratively by picking any neighbor of the current path head that has not been picked so far. Start by picking a vertex, say v_1, at random, and make it the head of the path. Then, pick any neighbor of v_1, say v_2, and make it the new head of the path. Progress likewise at each step; pick any neighbor v_{k+1} of the current path head v_k, and make it the new head of the path. But what if v_{k+1} is equal to some vertex v_i, $1 \leqslant i \leqslant k-1$, that is already present in the path? Then, the algorithm can perform a *rotation*, as illustrated by Figure 8.1.



v_1 — v_2 — v_3 ⋯ v_i — v_{i+1} ⋯ v_k
Rotation (v_i, v_k)
v_1 — v_2 — v_3 ⋯ v_i v_{i+1} ⋯ v_k

FIGURE 8.1: Rotation (v_k, v_i) of the path. The new head is v_{i+1}.

We obtain the following algorithm, where at each step we pick at random a neighbor u of the current path head v_k among the set of edges originating from v_k that have not been used so far. At the beginning, no edge has been used yet.

8.4.2 Results

What is the probability that Algorithm 8.1 will successfully build a Hamiltonian cycle for G? We would like to express this probability as a function of $n = |V|$, the number of vertices in G. Note that there exist exactly $2^{n(n-1)/2}$ different graphs with n vertices because there are $\binom{n}{2}$ possible edges that can or cannot be added to the graph.

THEOREM 8.14. *There exist constants c and d such that if we pick at random a graph G with n vertices and at least $c \log n$ edges, then with probability at least $1 - \frac{1}{n}$, Algorithm 8.1 will find a Hamiltonian cycle during its first $dn \log n$ steps.*

Proving this theorem is not difficult. This requires, however, some basic knowledge about probability theory (binomial distributions and Markov bound essentially) that is out of the scope of this chapter. We refer the reader to [78] for a proof and many more details about random graphs. We limit ourselves to some comments. First, the randomized algorithm does not give any insight on the P versus NP problem, nor does it help solve all instances of the HC problem. However, on the positive side, we have a fast algorithm that

```
Input: graph $G = (V, E)$ with $n$ vertices
Output: a Hamiltonian cycle in $G$ or failure
foreach $v \in V$ do
    unused($v$) := $\{(v, u) \mid (v, u) \in E\}$
pick a vertex at random and make it the head of the path
while true do
    let $(v_1, \dots, v_k)$ be the current path (with head $v_k$)
    if unused($v_k$) $= \emptyset$ then return failure
    else let $(v_k, u)$ be the first element in unused($v_k$)
    delete edge $(v_k, u)$ from unused($v_k$) and unused($u$)
    if $u \notin \{v_1, \dots, v_{k-1}\}$ then
        add $u$ to the path and let $v_{k+1} = u$ be the new path head
    else
        let $i$ be such that $v_i = u$
        if $k = n$ and $v_i = v_1$ then return $\{v_1, \dots, v_n\}$
        else rotate $(v_k, v_i)$ and let $v_{i+1}$ be the new path head
```

ALGORITHM 8.1: Randomized algorithm for the HC problem.

solves HC in most instances, as soon as the graph has enough edges. This is expected news, as we expect a random graph to be connected and then to have large cliques, or a Hamiltonian cycle, when its number of edges grow. But the beauty of Theorem 8.14 is to quantify this observation.

8.5 Branch-and-bound and backtracking

In this last section, we introduce branch-and-bound and backtracking techniques. The principle is to represent as a tree the search space (i.e., all candidate solutions) and then to explore this tree and remove branches that either lead to no valid solution or lead to solutions that are less good. Such algorithms return exact solutions to an NP-complete problem. For decision problems, the technique is called backtracking, while it is called branch-and-bound for optimization problems. While there is no guarantee on the execution time of such algorithms (the worst case may well be exponential because we may need to explore the entire search space), they are offering practical and often efficient solutions to deal with NP-complete problems.

We first present a small example of a backtracking algorithm with the n-queens problem. Then, we investigate branch-and-bound with the knapsack problem. Finally, we discuss some more complex graph algorithms.

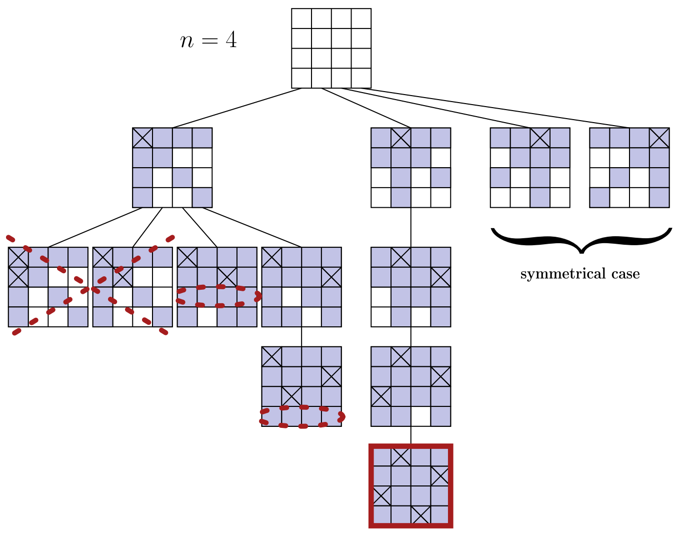


FIGURE 8.2: The n-queens backtracking tree.

8.5.1 Backtracking: The n queens

In a chess game, a queen can move as far as she wants: horizontally, vertically, or diagonally. We consider a chess board with n rows and n columns. The problem is to place n queens on this chess board so that none of them can attack any other in one move.

In any solution, there is exactly one queen per row. Therefore, the search space is of size n^n. However, because of the many constraints, many solutions can be discarded. The idea of the backtracking algorithm is to place a queen on the first row (n possible choices) and then perform a recursive call for the next row. We discard the choices that lead to no solution, and if no solution is found on a branch of the tree, we go up in the tree and try the next possibility (the next branch).

Figure 8.2 illustrates the tree for $n = 4$. Because the problem is symmetrical, we develop only the portion of the tree in which we place the first queen either on the first or on the second column. Once a queen has been placed, the squares on which it is not possible to place another queen have been colored. Therefore, if we place the first queen on the top left corner, the queen on the second row can be placed only on the third or fourth column. If we place it on the third column, there is no further choice for the third queen. If we place it on the fourth column, we can still place the third queen on the second column, but then there is no possibility for the last queen. However, a solution is found by exploring the second branch of the tree.

8.5.2 Branch-and-bound: The knapsack

A branch-and-bound algorithm works in two phases. The *branch* consists of splitting a set of solutions into subsets, while the *bound* consists of evaluating the solutions of a subset by bounding the value of the best solution in this subset.

We consider the knapsack problem, which was introduced in Section 4.2 and that we redefine briefly. Given a set of items $I_1, \ldots, I_n$, where item I_i has a weight w_i and a value c_i $(1 \leqslant i \leqslant n)$, we want to determine the items to include in the collection so that the total weight is less than a given limit W and the total value is as large as possible. We consider the variant of the problem where we have as many units of each item as we want. Let x_i be the number of units of item I_i that we decide to add into the knapsack. The goal is to maximize $\sum_{i=1}^{n} x_i \times c_i$, under the constraint $\sum_{i=1}^{n} x_i \times w_i \leqslant W$.

We consider the running example from [15]. There are four items, and the goal is to find $\max(4x_1 + 5x_2 + 6x_3 + 2x_4)$, under the constraint $33x_1 + 49x_2 + 60x_3 + 32x_4 \leqslant 130$.

The search space is represented as a tree. The leaves of the tree correspond to maximal solutions, i.e., solutions to which we cannot add any item because of the constraint on total weight. At the root of the tree, we have not chosen any item. The root has $\left\lceil \frac{W}{w_1} \right\rceil + 1$ children, which corresponds to picking, respectively, $0, 1, \ldots, \left\lceil \frac{W}{w_1} \right\rceil$ units of I_1. Then, for each of these nodes, we add one child for each possible number of units of the next item that can be chosen. For the last item, we fill the knapsack by adding systematically as many units of this item as we can. A part of the tree corresponding to this example is depicted in Figure 8.3. Its height is equal to the number of different items, n. Each leaf corresponds to a solution, and the number of leaves is exponential in the problem size.

Note that we have ordered the items such that the c_i/w_i are nonincreasing, i.e., the first item has the best value/weight ratio.

Given a search space represented by a tree, the branch-and-bound algorithm works as follows. At the beginning, there is only one active node, the root of the tree. At each step, we choose an active node, and we process its children nodes. If a child has only one child itself, we traverse the branch until we eventually find a leaf or a node with at least two children. Then we evaluate the node as follows: (i) If the node is a leaf, it corresponds to a solution, and we can compute the exact value of this solution. We keep the best solution between case (i) and the previously best known solution; (ii) otherwise, we provide an upper bound on the solutions in the branch by filling the unused weight with the item that has not yet been considered and that has the best value/weight ratio as if it were a liquid, that is, as if we were allowed to use a noninteger number of items. All the nodes from case (ii) become active. Before moving to the next step (i.e., picking up a new active node), we remove the active nodes that will never lead to a better solution than one of the solutions

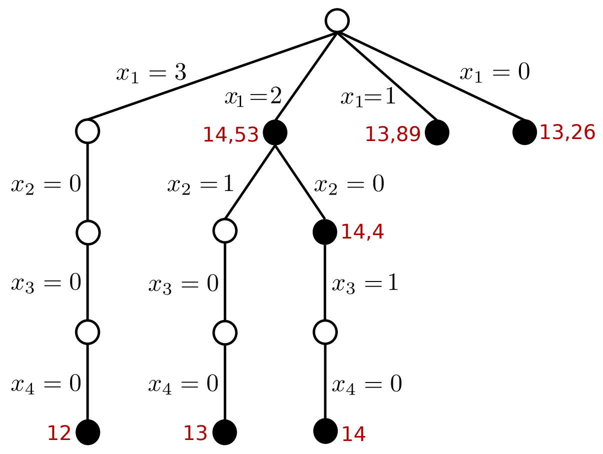


FIGURE 8.3: Branch-and-bound algorithm for the knapsack problem.

already found, i.e., if their upper bound is smaller than the value of the best solution. This corresponds to the pruning of the search space.

In the example (see Figure 8.3), we first process the child node corresponding to $x_1 = 3$. We cannot add any other item in the knapsack, so we reach a leaf of the tree. The value of the solution is $3 \times 4 = 12$. This is the best current solution. Then, we consider the second child node of the root, corresponding to $x_1 = 2$. It has two children, corresponding to $x_2 = 1$ and $x_2 = 0$ (we cannot add more than one unit of item I_2 in the knapsack). Therefore, we evaluate this node. The upper bound is computed with $x_1 = 2$, and all the remaining space $(130 - 66)$ is filled with item I_2, which is the remaining item with the best value/weight ratio. We obtain $2 \times 4 + 5/49 \times (130 - 66) = 14.53$. Because $14.53 \geqslant 12 + 1$, it may be possible to find a better solution than the current one (whose value is 12) in this tree, i.e., a solution whose value is at least 13 (solutions are integers). Therefore, this node becomes active. With $x_2 = 1$, we obtain a solution of value 13. Then, we evaluate the node for $x_2 = 0$ because there may still be a solution of value 14 in this subtree. The evaluation is done by filling the remaining space with item I_3, leading to $2 \times 4 + 6/60 \times (130 - 66) = 14.4$. This branch leads to a solution 14, with $x_3 = 1$. Because the upper bound for this subtree is 14.4, we cannot find a better solution. We evaluate the third child of the root to 13.89 and then the last child to 13.26; therefore, no better solution can be found. There are no more active nodes. The nodes of the tree colored in black are the nodes that have been evaluated.

Note that several strategies can be considered for the choice of the next active node. A depth-first search, as we have done in the example, is very practical because there are few nodes that are simultaneously active. A breadth-first search often leads to poor results. Another strategy consists of picking the active node with the best evaluation. Some hybrid strategies also can be considered. For instance, one can perform a depth-first search until a solution is found and then use a best evaluation strategy to find even better solutions. Such a strategy may allow the pruning of several branches.

Note that other strategies can be used to solve this kind of problem. The branch-and-bound algorithm is often not very efficient in the worst case. However, it often leads to efficient algorithms on average, as we detail in the next section.

8.5.3 Graph algorithms

In this section, we consider two important NP-complete graph problems that we aim to solve with backtracking algorithms. First, we investigate the problem of finding the largest independent set, and then we investigate the graph coloring problem.

8.5.3.1 Independent sets

Let $G = (V, E)$ be a graph with n vertices, numbered from 1 to n. The problem is to find the size of the largest independent set of G, i.e., a subset $S \subseteq V$ such that, for all $i, i' \in S$, $(i, i') \notin E$, and $|S|$ is maximum.

The backtracking algorithm is easy to describe and analyze for this problem. The idea is to explore all possible independent sets and to build a tree with all the solutions to the problem. The root of the tree corresponds to the empty set. The children of the root node correspond to independent sets of size 1, and we add a node only if it is an independent set. The tree is built in a depth-first traversal. First, we search for independent sets containing vertex 1, which correspond to the first child of the root, denoted $\{1\}$ (if $(1, 1) \notin E$). We then try to increase the size of this set by adding vertex 2. The children of $\{1\}$ are the independent sets of size 2 containing vertex 1. If $(1, 2) \in E$, then there is no independent set containing both 1 and 2; therefore, we do not add any node in the solution tree and proceed with vertices $3, \ldots, n$. Otherwise, we add $\{1, 2\}$ as a child node of $\{1\}$ and move to the next level of the tree, trying to add vertices $3, \ldots, n$ to this independent set and building independent sets of size 3. When no vertex can be further added, we *backtrack* up in the tree and develop all remaining branches of the solution tree. The height of the solution tree gives the maximum size of an independent set.

The solution tree has one node per independent set and, therefore, the complexity of the algorithm depends on the number of independent sets, which can be exponential: For a graph with $E = \emptyset$, this number is 2^n. However, for a clique of size n, there are only $n + 1$ independent sets. The analysis aims at determining the average complexity of the algorithm, i.e., the average number of independent sets, denoted I_n.

Let $I(G)$ be the number of independent sets of a graph $G = (V_G, E)$.

$H(G,S)$ equals 1 if S is an independent set of G, and 0 otherwise. Therefore, $I(G) = \sum_{S \subseteq V_G} H(G,S)$, and the sum contains the 2^n possible subsets of V.

The average number of independent sets I_n is then the sum over all possible graphs G with n vertices, divided by the number of such graphs, $2^{n(n-1)/2}$. We obtain

$$I_n = 2^{-n(n-1)/2} \sum_{|V_G|=n} \sum_{S \subseteq V_G} H(G,S) \, .$$

We can invert the two sums, and we examine $\sum_{|V_G|=n} H(G,S)$. Given a set S, this value corresponds to the number of graphs with n nodes that contain S as an independent set. If $|S| = k$, there are $k(k-1)/2$ edges that cannot exist in G, and there are $n(n-1)/2 - k(k-1)/2$ possible edges, which leads to $2^{n(n-1)/2-k(k-1)/2}$ graphs with n vertices such that $H(G,S) = 1$. Finally, since the number of sets S with k vertices is $\binom{n}{k}$, we obtain

$$I_n = \sum_{k=0}^{n} \binom{n}{k} 2^{-k(k-1)/2} \, .$$

On average, the algorithm is much better than in the worst case; for instance, with $n = 40$, $I_n = 3862.9$, while $2^n > 10^{12}$. In fact, for large values of n, $I_n = O(n^{\log(n)})$ and, therefore, the average complexity of the algorithm remains subexponential.

8.5.3.2 Graph coloring

For the graph coloring problem, the backtracking algorithm leads to more efficient results on average than for the independent sets problem because it turns out that the average complexity is, in fact, constant for a fixed number of colors, even when the number of vertices tends to infinity.

Let $G = (V,E)$ be a graph with n vertices, numbered from 1 to n, and K be an integer. The K-coloring problem is to associate a color with each vertex such that two vertices connected by an edge have a different color, where K is the number of colors.

The backtracking algorithm builds all partial colorings of the graph with only a subset of vertices $\{1,\ldots,L\}$, with $1 \leqslant L \leqslant n$. The root of the tree corresponds to the coloring of the empty graph; it is represented by an empty set. It has K children nodes, corresponding to the possible colors for vertex 1. The node is labeled by the set of colors for the vertices that we consider, i.e., the children of the root are labeled $1,\ldots,K$. Similar to the backtracking algorithm for the independent sets problem, we build the tree in a depth-first traversal. We add a node 11 as a child of 1 if and only if $(1,2) \notin E$, then we assign the lowest possible color to the third vertex, and so on. If there is no possible color for one of the vertices, or if we have successfully colored all vertices, we go up in the tree until we can try another color for one of the vertices. (Remember that the backtracking algorithm builds *all* partial colorings of the graph.)

Note that the branch of a tree may stop before a color has been assigned to each vertex, and it may happen that no valid coloring can be found. At level L of the tree, we have all partial colorings of vertices $\{1,\ldots,L\}$, and a valid coloring has been found if the tree has nodes of level n. Graph G restricted to vertices $\{1,\ldots,L\}$ is denoted $H_L(G)$ in the following.

The goal is to determine the average number of nodes $A_{n,K}$ of a backtrack tree generated when coloring a graph of size n with at most K colors. There are $2^{n(n-1)/2}$ different graphs, and we decompose the backtrack trees into levels. If G is a graph with n vertices, we denote by $P(K,H_L(G))$ the number of nodes at level L of the backtrack tree of G. It is equal to the number of correct colorings of graph $H_L(G)$ with K colors. Finally,

$$A_{n,K} = 2^{-n(n-1)/2} \sum_{|V_G|=n} \sum_{L=0}^{n} P(K,H_L(G)) \, .$$

We invert the two sums and examine $\sum_{|V_G|=n} P(K,H_L(G))$, given a level L. Note that there are exactly $2^{n(n-1)/2-L(L-1)/2}$ graphs that share the same graph $H_L(G)$, and, therefore,

$$A_{n,K} = 2^{-n(n-1)/2} \sum_{L=0}^{n} 2^{n(n-1)/2-L(L-1)/2} B_{L,K} = \sum_{L=0}^{n} 2^{-L(L-1)/2} B_{L,K} \, ,$$

where $B_{L,K}$ is the total number of correct colorings with K colors of all graphs with L vertices. Given a coloring, we denote by s_i the number of vertices that are colored with the color i, for $1 \leqslant i \leqslant K$. Because the graphs have L vertices, we have $\sum_{i=1}^{K} s_i = L$. Moreover, an edge can connect only two vertices of different colors, and, thus, the maximum number of edges is $E_{n,K} = s_1s_2 + s_1s_3 + \cdots + s_1s_K + s_2s_3 + \cdots + s_{K-1}s_K = \sum_{1\leqslant i<j\leqslant K} s_is_j$. We compute this value as follows:

$$\begin{aligned} E_{n,K} &= \tfrac{1}{2}\textstyle\sum_{i\neq j} s_is_j = \tfrac{1}{2}\left(\sum_{i,j=1}^{K} s_is_j - \sum_{i=1}^{K} s_i^2\right) \\ &= \tfrac{1}{2}\left(\textstyle\sum_{i=1}^{K} s_i\right)^2 - \tfrac{1}{2}\sum_{i=1}^{K} s_i^2 = \tfrac{1}{2}L^2 - \tfrac{1}{2}\sum_{i=1}^{K} s_i^2 \, . \end{aligned}$$

It is easy to check that $\sum_{i=1}^{K} s_i^2 \geqslant L^2/K$, because $L = \sum_{i=1}^{K} s_i$:

$$\begin{aligned} \textstyle\sum_{i=1}^{K} s_i^2 - L^2/K &= \textstyle\sum_{i=1}^{K} s_i^2 - 2L^2/K + L^2/K \\ &= \textstyle\sum_{i=1}^{K}\left(s_i^2 - 2Ls_i/K + L^2/K^2\right) = \sum_{i=1}^{K}\left(s_i - L/K\right)^2 \geqslant 0 \, . \end{aligned}$$

Therefore, $E_{n,K} \leqslant \frac{1}{2}L^2 - \frac{1}{2}L^2/K = L^2(1-1/K)/2$. The number of graphs $H_L(G)$ with the same coloring is at most $2^{L^2(1-1/K)/2}$. Because there are at most K^L different colorings (counting invalid ones), we obtain $B_{L,K} \leqslant K^L 2^{L^2(1-1/K)/2}$, and, finally,

$$A_{n,K} \leqslant \sum_{L=0}^{n} 2^{-L(L-1)/2} K^L 2^{L^2(1-1/K)/2} \leqslant \sum_{L=0}^{\infty} K^L 2^{L/2} 2^{-L^2/2K} .$$

This infinite series is converging; therefore, $A(n, K)$ is bounded for all n.

8.6 Bibliographical notes

The FPTAS for scheduling independent tasks on two processors (Section 8.1.5) is presented in [95]. Further references for approximation algorithms are the books by Ausiello et al. [5] and by Vazirani [103]. Randomized algorithms (Section 8.4) are dealt with in the books by Mitzenmacher and Upfal [78] and by Motwani and Raghavan [80]. Section 8.5.2 (branch-and-bound) is inspired from [15]. The backtracking graph algorithms (Section 8.5.3) are analyzed in [108].