IFT800 - Algorithmique

Manuel Lafond
Université de Sherbrooke

Contents

Chapter 1

Introduction

As a disclaimer, I must first acknowledge that this English version was 99% translated from French using automated software. Although current tools offer unprecedented accuracy, they are still prone to mistakes. I do read the translation and apply corrections where needed, but please let me know if I have missed complete nonsense in this document.

In this course, we will discuss techniques to solve difficult algorithmic problems efficiently. From our point of view, a problem is *difficult* if there is no known polynomial time algorithm to solve it. Our goal is to develop algorithms for these problems that offer *theoretical guarantees* on the quality of the solution obtained or on their speed. We will study two approaches:

1. approximation algorithms, which guarantee to always return a solution within a factor close to optimal;

2. algorithms with parameterized complexity, which guarantee an exponential time, but only with respect to a parameter k which is small in practice.

For example, the travel salesperson problem is NP-complete, but there is a polynomial time algorithm that returns a route that is, at worst, twice the optimal (under the triangle inequality condition). Or, the VERTEX-COVER problem is NP-complete, but if k is the number of vertices in a cover, there is an algorithm in $O(2^k \cdot n)$ time (VERTEX-COVER is defined in chapter 2).

In other words, no one knows an efficent solution to solve NP-complete problems. This does not mean that any heuristic can be implemented, as these can return completely incorrect solutions on some instances. We want

to implement algorithms whose performance is *demonstrable*, whether in terms of approximation or complexity.

1.1 What is an NP-complete problem?

If a problem is NP-complete, there is no known polynomial time algorithm that solves it. The formal definition of an NP-complete problem is beyond the scope of this course. The intuition that will suffice for this course is that a first NP-complete problem was discovered in the early 1970s. This problem is called SAT and consists of determining whether a boolean expression can be satisfied. For example, consider the boolean expression

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3})$$

In the SAT problem, we want to know if we can assign the value True or False to each of the x_i so that ϕ evaluates to True. A solution here would be $x_1 = True, x_2 = True, x_3 = False$. This example is easy and has several solutions, but on larger instances, no one knows a polynomial time algorithm to decide if it is possible that $\phi = True$. In fact, the best known algorithm is to test every combination of the values of x_i, which takes a time $\Omega(2^n)$.

In short, no one knows an efficient algorithm for SAT, despite years of research. It turns out that SAT can be *reduced* to other problems. For example, it has been shown that finding a maximum size clique in a graph is at least as hard as the SAT problem. That is, we can transform a ϕ instance of SAT into a G graph such that if we had a $O(n^c)$ algorithm that finds the largest G click, this algorithm could

That is, we can transform a ϕ instance of SAT into a graph G such that if we had a $O(n^c)$ algorithm that finds the largest clique in G, this algorithm could determine whether SAT is satisfiable or not. Everything happens in the transformation of ϕ into G in order to preserve equivalence.

We invite you to take the course IFT503/IFT711 for more details on this subject. For our purposes, it is sufficient to know that if a problem is NP-complete, it probably does not admit a polynomial time algorithm because such an algorithm would solve SAT, something that the greatest geniuses of the last decade have not been able to do.

Unless otherwise stated, all problems studied in this course are NP-complete.

1.2 What are the prerequisites for this course

We assume that the common notions in algorithms are known. These include

- knowledge of the main concepts of discrete mathematics, including sets, sequences, permutations, and associated notations and operators ($\cap, \cup, |X|$, etc.).
- a knowledge of common proof techniques: direct proof, proof by contradiction, proof by induction, proof by counter-example.
- familiarity with the O notation. In particular, it will be useful to be able to quickly evaluate the complexity of an algorithm by only going through the code structure (loops, recursive calls, ...).
- familiarity with graphs. A graph $G = (V, E)$ is a structure where V is the set of vertices and E the edges, where E contains pairs of vertices.
- a knowledge of divide-and-conquer algorithms and of the master theorem for recurrence analysis. A knowledge of linear homogeneous recurrences may also help (but we will discuss this further).
- knowledge of dynamic programming. In particular, how to establish a recurrence that expresses optimality, and how to transpose this recurrence into code.

It is important to note that this course is *theoretical* in nature. Approximation and parameterized algorithms are now very useful in practice, and many of the approaches presented are implemented in libraries. In this course, we will not dwell on practical considerations, or even try to convince ourselves of the applications of the notions presented. The methods presented are chosen to illustrate the most common techniques, and not to illustrate the most applicable algorithms.

The objective of this course is to train you to apply these techniques to new problems — the practical interest of this course does not therefore lie in the specific algorithms chosen, but rather in the *ideas* behind these algorithms that you will be able to apply to future problems that you will encounter.

Part I

Approximation algorithms

Chapter 2

Approximation algorithms

In approximation algorithms, we try to create a polynomial time algorithm that returns a solution that is possibly sub-optimal, but as close to optimal as possible, in a **demonstrable** way. Most of the time, we are not going to worry about the exact complexity of our algorithm — our only interest is polynomial time, even if it is $O(n^{100})$.

2.1 Approximating a minimization problem

Let P be a minimization problem, i.e. a problem in which we want to minimize a certain objective function among a set of feasible solutions. For an instance X of P, let $OPT(X)$ be the value of an optimal solution for X.

Let A be an algorithm that produces a feasible solution for any instance of P, and let $APP(X)$ be the value of the solution output by A when given the input X.

We say that A is a c-approximation if, for any X instance of P,

$$APP(X) \leq c \cdot OPT(X)$$

So when $c > 1$, the algorithm can give a sub-optimal solution, but not more than c times too big.

2.2 Approximating a maximization problem

If P is a maximization problem, we say that A is a c-approximation if, for any instance X of P,

$$APP(X) \geq c \cdot OPT(X)$$

So when $c < 1$, the algorithm can give a sub-optimal solution, but not more than c times too small. Note that in some texts, $c \geq 1$ and $APP(X) \geq \frac{1}{c} \cdot OPT(X)$ are required. This is so that $c > 1$ in both types of problems. In this course, we prefer to use c directly and not its inverse.

In the remainder of this document, we will often write OPT instead of $OPT(X)$ and APP instead of $APP(X)$.

2.3 A first example with 3-SET-COVER

Our first example of approximation will be trivial, but instructive, as it will allow us to introduce a fundamental technique in approximation. We study the problem 3-SET-COVER. We are given sets $S_1, \ldots, S_m$ of size 3, and we are asked to cover a universe U of size n with a minimum number of sets.

3-SET-COVER
Input: a universe $U = \{u_1, \ldots, u_n\}$ and a collection of sets $S = \{S_1, \ldots, S_m\}$ such that $|S_i| = 3$ for all $i \in [m]$
Output: a subset $S^* \subseteq S$ of minimum size such that $\bigcup_{S_i \in S^*} S_i = U$.

It is assumed that each $u_i \in U$ has a set of S that contains it. Here is an example instance:

$U = \{1, 2, 3, 4, 5\}$ $S_1 = \{1, 3, 4\}, S_2 = \{1, 4, 5\}, S_3 = \{2, 3, 5\}$
An optimal solution: choose $S^* = \{S_1, S_3\}$

Here, a feasible solution is any $S^* \subseteq S$ such that the union of the elements of S^* gives U. The value we want to minimize is $|S^*|$. Here is a 3-approximation.

```
fonction setcover(U, S)
    S* = {}
    for i = 1..n do
        Let u_i be the i-th element of U
        if u_i is not covered by S* then
            Add to S* any set that contains u_i
    end
    return S*
```

It is clear that this algorithm runs in polynomial time and always returns a valid solution, i.e. an S^* subset that covers all the u_i. But how do we know that it is a 3-approximation? Let's start by asking ourselves what OPT is worth. Since there are n items to cover and each S_i can cover at most 3 elements, we know that

$$OPT \geq \left\lceil \frac{n}{3} \right\rceil \geq \frac{n}{3}$$

But what is APP worth? In the worst case, we add a set to S^* for each element of U. So

$$APP = |S^*| \leq n$$

We get

$$APP \leq n = 3 \cdot \frac{n}{3} \leq 3 \cdot OPT$$

So, $APP \leq 1/3 \cdot OPT$ and that's a 3-approximation.

2.4 Fundamental technique: finding a bound on OPT

The above example illustrates a very important idea in approximation. To show that we are not too far from OPT, we find a bound on OPT, a bound on APP, and we compare them. It's very simple: we just have to perform the following two steps

1. find a bound on OPT;
2. show that APP is not too far from this bound.

If we have a minimization problem, it takes the following form: we show that $OPT \geq k$ for a certain k. Then, we show that $APP \leq c \cdot k$. These two

facts give the chain of inequalities

$$APP \leq c \cdot k \leq c \cdot OPT$$

and you get a c-aprpoximation.

If we have a maximization problem, we show that $OPT \leq k$ for a certain k. Next, we show that $APP \geq c \cdot k$, which yields

$$APP \geq c \cdot k \geq c \cdot OPT$$

The difficulty with this technique is to find that "middle point" where OPT and APP meet. We will see several examples during this course.

2.5 A 2-approximation for VERTEX-COVER

Let $G = (V, E)$ a graph. A set $X \subseteq V$ is a *vertex cover* if, $\forall uv \in E$, we have $u \in X$ or $v \in X$ (or both). Put another way, X is "touching" all the edges.

VERTEX-COVER
Input: a graph $G = (V, E)$
Output: vertex cover $X \subseteq V$ of minimum size

To approximate this problem, we observe that for each edge $uv \in E$, we must include either u or v in the X solution. Not knowing which one to include, our approximation algorithm includes both. This will have the effect of covering other edges, all those that are incident to u and v. Our algorithm iteratively finds uncovered edges and adds its two ends until it covers the entire E.

fonction *vc-matching*($G = (V, E)$)
  $X = \{\}$
  while X *does not cover every edge* **do**
    Let $uv \in E$ be an edge not covered by X
    $X \leftarrow X \cup \{u, v\}$
  end
  return X

This algorithm is called *vc-matching* because the set of edges for which

the two ends have been added form a *matching*, i.e. a set of edges that do not share any vertex. It is possible to show that this algorithm is a 2-approximation. In fact, any algorithm that returns a matching is a 2-approximation.

Theorem 1. *The* vc-matching *algorithm is a 2-approximation to the VERTEX-COVER problem.*

Proof. Let X^* be a minimum vertex cover of G and let $OPT = |X|$. Let $u_1v_1, u_2v_2, \ldots, u_kv_k$ be the edges of G for which the algorithm added the two vertices, and let X be the cover returned by the algorithm. We observe that for any $i \neq j$, $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$ (this is because once we add u_i and v_i, all edges touching u_i and v_i are covered — if you don't see it, take the time to convince yourself). This means that every u_i and v_i is a different vertex.

Now, for any $i \in [k]$, X^* must contain either u_i or v_i (or both). Since all u_i and v_i are distinct, this implies that $OPT = |X^*| \geq k$. The cover returned by *vc-cover* is $X = \{u_1, v_1, u_2, v_2, \ldots, u_k, v_k\}$, and thus $|X| = 2k$. We get

$$APP = |X| = 2k \leq 2|X^*| = 2OPT$$

and so we have a 2-approximation. □

Note again the use of the technique. We first found a bound on OPT, this time directly linked to the solution returned by the algorithm. We compared OPT to the size of our solution and obtained a 2-approximation. Despite its simplicity, we do not know of an algorithm that offers a better approximation ratio. A well-known conjecture states that if NP-complete problems do not have an algorithm in polynomial time, then it is impossible to do better than this algorithm in terms of approximation.

Note that we passed a bit quickly on the statement $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$. This would not have been tolerated in a 1st or 2nd year course, but we allow ourselves a few such jumps, given the advanced nature of the course. When writing your own proofs, it is important to ask yourself if such jumps are appropriate — they often lead to erroneous proofs (even reputable researchers fall into these traps). When in doubt, detail!

2.6 One last simple example with MAX-SAT

In the world of Boolean expressions, a *clause* is a set of boolean variables linked by logical "or". A boolean variable x_i can be *positive* (x_i) or *negative* ($\overline{x}_i$).

For example, here is a clause C involving three variables:

$$C = x_1 \vee \overline{x}_2 \vee x_3$$

An *assignment* assigns the value $True$ or $False$ to each variable. If $x_i = True$, all positive occurrences of x_i evaluate to $True$ and negative occurrences to $False$. If $x_i = False$, all negative occurrences of x_i evaluate to $True$ and positive occurrences to $False$. For example, if we take the assignment $x_1 = True, x_2 = False, x_3 = False$, the above clause results in

$$C = True \vee True \vee False$$

which evaluates to $True$.

A clause is *satisfied* by an assignment A if the clause evaluates to $True$ with this assignment. For this to occur, at least one variable in the clause must value $True$. The only way not to satisfy the above clause is $x_1 = False, x_2 = True, x_3 = False$.

In the MAX-SAT problem, we receive a set of clauses (with an arbitrary number of variables), and we want to find an assignment that maximizes the number of satisfied clauses.

MAX-SAT
Input: a set of clauses $C_1, \ldots, C_m$ on boolean variables $x_1, \ldots, x_n$
Output: an assignment A that satisfies the maximum number of clauses among $C_1, \ldots, C_m$.

Here is a $\frac{1}{2}$-approximation.

fonction *msat($C_1, \ldots, C_m$ on variables $x_1, \ldots, x_n$)*
  Let A be the assignment with
  $x_1 = True, x_2 = True, \ldots, x_n = True$
  if *A satisfies at least $m/2$ clauses* **then**
    return A
  Let $\overline{A}$ be the assignment with
  $x_1 = False, x_2 = False, \ldots, x_n = False$
  return $\overline{A}$

Theorem 2. *The msat algorithm is a $\frac{1}{2}$-approximation.*

Proof. There are m clauses, so trivially, $OPT \leq m$ (our bound on OPT). If the algorithm returns A, then $APP \geq m/2$ and we have

$$APP \geq m/2 \geq OPT/2$$

Otherwise, assume that A satisfies $k < m/2$ clauses. Note that any clause not satisfied by A has the form $C_i = (\overline{x}_a \vee \overline{x}_b \vee \overline{x}_c)$. Such a clause will be satisfied by $\overline{A}$. Therefore, $\overline{A}$ satisfies at least $m - k > m - m/2 = m/2$ clauses. We deduce that if the algorithm returns $\overline{A}$, we still have $APP \geq m/2 \geq OPT/2$.

So we have a $\frac{1}{2}$-approximation. □

2.7 Can our analyses be refined?

The approximation ratios obtained depend on our ability to demonstrate them. We can then ask ourselves whether, with more refined analyses, we could show that our algorithms are better than we think. For example, perhaps *msat* above is in fact a $\frac{5}{6}$ approximation, but we are unable to demonstrate it.

When we produce an algorithm and a proof of approximation, we like to argue that our analysis is *tight*, i.e. that we could not improve the demonstrated approximation ratio. To do so, we just have to give an example instance in which our algorithm reaches exactly the demonstrated ratio. One instance is enough, but we usually need to provide a family of instances.

Let's take the algorithm again for 3-SET-COVER.

```
fonction setcover(U, S)
    S* = {}
    for i = 1..n do
        Let u_i be the i-th element of U
        if u_i is not covered by S* then
            Add to S* any set that contains u_i
    end
    return S*
```

Is it possible that this algorithm does better than a 3-approximation? No, because there are some instances where this algorithm gives $APP =$

$3 \cdot OPT$. Or more precisely, we will build a family of instances such that $APP = (3-\epsilon)OPT$, with ϵ tending towards 0 when n tends towards infinity. The following theorem formulates this in an alternative way.

Theorem 3. *For any $\epsilon > 0$, there is an instance of 3-SET-COVER such as $APP \geq (3-\epsilon)OPT$.*

Proof. Let $U = \{u_1, \ldots, u_n\}$, where n is a multiple of 3. Let the sets S contain

$$
\begin{aligned}
&S_1 = \{u_1, u_2, u_3\} \\
&S_2 = \{u_1, u_2, u_4\} \\
&\ldots \\
&S_{n-2} = \{u_1, u_2, u_n\} \\
&S'_2 = \{u_4, u_5, u_6\} \\
&S'_3 = \{u_7, u_8, u_9\} \\
&\ldots \\
&S'_{n/3} = \{u_{n-2}, u_{n-1}, u_n\}
\end{aligned}
$$

Since the algorithm does not specify how to choose the set to cover a new element, we can assume that it makes the worst possible choice. We can be convinced that the algorithm could return $S^* = \{S_1, S_2, \ldots, S_{n-2}\}$ of size $n-2$. However, the optimal solution is $\{S_1, S'_2, S'_3, \ldots, S'_{n/3}\}$ of size $n/3$.

If we want to know the approximation ratio, we must find c such that $APP = c \cdot OPT$, i.e. the c such that $n - 2 = c \cdot (n/3)$. We find $c = 3 - 6/n$.

$$APP = n - 2 = (3 - 6/n)OPT$$

By making n tend towards infinity, we obtain the result. □

Chapter 3

Approximation using the fundamental approach

In this chapter, we will see various examples of applications of the fundamental technique. As we will see, we sometimes have to be creative in our choice of bounds on OPT and APP.

3.1 The Traveler Salesperson Problem, Metric Version

Let $G = (V, E)$ a complete graph, i.e. every possible edge is present (and thus $|E| = \binom{n}{2} = n(n-1)/2$). Let $f : E \to \mathbb{R}^{>0}$ a function that assigns a positive weight to each edge. We say that f satisfies *the triangle inequality* if, for any triplet u, v, w of distinct vertices, we have

$$f(uw) \leq f(uv) + f(vw)$$

We will also say that f is a *metric*. The triangle inequality is a condition often applicable in practice, and greatly facilitates the development of approximation algorithms.

A cycle $C = (v_1, v_2, \ldots, v_n, v_1)$ of G is Hamiltonian if C traverses each vertex of V exactly once and then returns to v_1. The weight of C, denoted $f(C)$, is equal to the sum of the edge weights of C, i.e.

$$f(C) = f(v_n v_1) + \sum_{i=1}^{n-1} f(v_i v_{i+1})$$

In the problem of the traveling salesperson, one seeks to visit each city

and return home in a way that minimizes the sum of the distances traveled.

TRAVELING-SALESPERSON
Input: A complete graph $G = (V, E)$ and a metric $f : E \to \mathbb{R}^{>0}$
Output: A Hamiltonian cycle of minimum weight.

At first glance, it is not easy to find a *useful* bound on OPT. Reflecting on this, we can see that a Hamiltonian cycle is a sub-graph that must cover each vertex. We then remember that we know a notion on graphs that aims to cover each vertex: minimum spanning trees (MST). Remember that a spanning tree is a G sub-graph which is a tree (therefore connected and acyclic), which contains all the vertices of G. This spanning tree is an MST if the sum of the edge weights is minimum among all the possibilities. Prim's algorithm or Kruskal's algorithm can find an MST in polynomial time.

Lemma 1. *Let G be an instance of TRAVELLING-SALESPERSON and let $MST(G)$ the weight of a minimum spanning tree of G. Then $OPT(G) \geq MST(G)$.*

Proof. Let C be a Hamiltonian cycle of G minimum weight. The edge weight of C is $OPT(G)$. If we remove an edge of C, we get a path that passes through each vertex once. It is therefore a spanning tree, and therefore the weight of its edges is at least as large as $MST(G)$ (since $MST(G)$ is the minimum possible). Let k be the weight of the edges of this path. We have $OPT(G) \geq k \geq MST(G)$. □

We can therefore try to use an MST as a "middle point" for our approximation, and thus we want to find a cycle that is not too far from the weight of an MST. The idea is to find an MST, to take the pre-order traversal order of this tree. This order will become the visiting order of our cities.

As a reminder, if we have a rooted tree T, the pre-order traversal first goes to the root, then visits its children recursively.

```
fonction preOrder(T = (V, E), v, list)
    //v is the current node
    list.append(v)
    for w child of v do
        preOrder(T, w, list)
    end
```

Our algorithm for TRAVELLING-SALESPERSON is as follows.

```
fonction approxTSP(G = (V, E), f)
    T = getMST(G)              // Get a minimum spanning tree
    Root T at an arbitrary node r
    C = ()                                          // empty list
    preOrder(T, r, C)
    C.append(C.first)                  // To complete the cycle
    return C
```

We can show that this is a 2-approximation. The idea is that the resulting cycle does a bit like going through each T edge twice, thanks to the triangular inequality.

Theorem 4. *The approxTSP algorithm is a 2-approximation.*

Proof. Let k be the weight of an MST T of G, and let $C = (v_1, \ldots, v_n, v_1)$ be the cycle returned by the algorithm. Let $f(C)$ be the weight of the edges of C. We want to show that $f(C) \leq 2k$, because we know that $OPT \geq k$.

Let v_i, v_j be two consecutive vertices in C (so $j = i+1$, or $i = n, j = 1$). Let $P_{i,j}$ be the path from v_i to v_j in T and let $f(P_{i,j})$ be the weight of the edges of $P_{i,j}$. Thanks to the triangular inequality, it is possible show that

$$f(v_i v_j) \leq f(P_{i,j})$$

So,

$$f(C) = \sum_{i=1}^{n-1} f(v_i v_{i+1}) + f(v_n v_1) \leq \sum_{i=1}^{n-1} f(P_{i,i+1}) + f(P_{n,1})$$

It can be noticed that in a pre-order traversal, the paths go through each edge twice: once going down, and once going up. So, $\sum_{i=1}^{n-1} f(P_{i,i+1}) +$

$f(P_{n,1}) \leq 2k$. We get

$$APP = f(C) \leq \sum_{i=1}^{n-1} f(P_{i,i+1}) + f(P_{n,1}) \leq 2k \leq 2OPT$$

which concludes the proof. □

One wonders how the inventor of this algorithm could have thought of it in the first place. Again, it's all in the fundamental approach. We start by asking what could give a bound on OPT. Once the link with MST is made, we wonder how an MST could give us a cycle. Once this idea is established, the calculations come naturally.

Note that if we do not have the triangular inequality, there is *no* c-approximation, even if $c = n^k$ for a constant k, unless P = NP. The conjecture P $\neq$ NP stipulates that NP-complete problems do not admit a polynomial time algorithm, which most researchers believe. Put another way, don't look for an approximation algorithm for non-metric TRAVELLING SALESPERSON.

3.2 Improvement to a $3/2$-approximation

We're going to sketch a 3/2-approximation. This one is not extremely difficult, but involves Eulerian cycles and perfect matchings, which requires some additional knowledge.

In a graph, an Eulerian cycle is a cycle C in which the same vertex can be visited several times, and in which each edge is traversed exactly once. A classical result of graph theory stipulates the following theorem. The proof is left as exercise and is easily found online. It is in fact one of the first results of graph theory, discovered by the famous Euler.

Theorem 5. *A G graph has an Eulerian cycle if and only if each vertex has an even degree.*

Remember that the degree of a vertex is its number of neighbors. The link with the Hamiltonian cycles is the following, also left in exercise.

Lemma 2. *Let $G = (V, E)$ and f a metric on the edges. Let C be an Eulerian cycle of G. Then there is a Hamiltonian cycle C' with a weight equal to or less than that of C.*

The idea of the proof is to take the order of the C vertices according to their first appearance. If C contains the sequence $v_i, v_{i+1}, \ldots, v_{i+k}$ and that

in C', it becomes v_i, v_{i+k} because $v_{i+1}, \ldots, v_{j-1}$ have already been visited, we use the triangular inequality to argue that $f(v_i v_{i+k}) \leq \sum_{j=i}^{i+k-1} f(v_j v_{j+1})$. In the end, we end up with the same weight as C.

So, if our MST T had each vertex with an even number of neighbors, it would be Eulerian and we could transform it into a Hamiltonian cycle with the same weight, giving a 1 approximation! Of course, a tree has leaves, and those leaves have 1 neighbor, an odd number. The idea is to add to T a few edges at odd degree vertices so that it becomes Eulerian.

Let X be the odd degree vertices of T. It is possible to demonstrate that $|X|$ is even (also an exercise: the number of odd vertices of a graph is always even). We will add a set of edges M to T in order to increase the degree of each vertex of X by 1. This way, T with M will be Eulerian. To do this, we will take a perfect matching in X. That is, we partition X into exactly $|X|/2$ pairs so as to minimize the weight of the edges between the chosen pairs. Remarkably, this can be done in polynomial time. The chosen pairs correspond to a M set of $|X|/2$ edges. The interest of this perfect matching is the following.

Lemma 3. *Let M be a perfect matching between the elements of $X \subseteq V(G)$ in G, with $|X|$ even. Then $OPT(G) \geq 2f(M)$.*

The idea of the proof is that since f is a metric, $OPT(G[X]) \leq OPT(G)$, where $G[X]$ is the graph induced by X. Moreover, $OPT(G[X])$ is the weight of a cycle consisting of the union of two perfect matchings.

Our algorithm constructs an MST T, finds a perfect matching M between odd vertices, then adds the M edges to T to get an Eulerian graph. From this, an Hamiltonian cycle can be obtained.

fonction *approxTSP2($G = (V, E), f$)*
  $T = getMST(G)$
  Let X be the odd degree vertices of T
  Let M be a minimum perfect matching of the elements of X
  Let $T' = (V(T), E(T) \cup M)$
  Let C' be an Eulerian cycle of T'
  Let C be a Hamiltonian cycle of G obtained from C'
  return C

Theorem 6. *approxTSP2 is a 3/2-approximation.*

Proof. The weight of C returned by the algorithm is $f(T) + f(M)$. We obtain $APP = f(T) + f(M) \leq OPT + OPT/2 = 3/2 \cdot OPT$. □

3.3 k-center problem

In the k-centers problem, one receives a set of points $P = (p_1, \ldots, p_n)$ in d-dimensions (so $p_i = (u_1, u_2, \ldots, u_d)$). Our goal is to find the k locations among the P points that are the most "centralized". To determine this, we have a metric $dist$ between each pair of points, i.e. $dist(p_i, p_j)$ is a numerical value such that for any p_i, p_j, p_k, we have

$$dist(p_i, p_k) \leq dist(p_i, p_j) + dist(p_j, p_k)$$

A classic example of a metric is the Euclidean distance, where the distance between two points $p = (u_1, \ldots, u_d)$ and $q = (v_1, \ldots, v_d)$ is $dist(p, q) = \sqrt{\sum_{h=1}^{d}(u_i - v_i)^2}$. However, our algorithm works on any metric.

Our goal is to choose k centers in order to minimize the longest distance to travel to one of these centers.

To formalize the problem, let's say $P' \subseteq P$ and $p \in P$. We define

$$dcentre(p, P') = \min_{p' \in P'}(dist(p, p'))$$

If we interpret P' as a list of centers, this represents the distance to the nearest center.

Our problem is as follows.

k-CENTERS
Input: points P, integer k, metric $dist$
Output: $P' \subseteq P$ such that $|P'| = k$ which minimizes $\max_{p \in P}(dcentre(p, P'))$

First, we observe that our optimization criterion concerns a distance between two points (a point p and a center). Therefore, there are $p, q \in P$ such that $OPT = dist(p, q)$. There are $\binom{|P|}{2} \in O(|P|^2)$ possible distances, one for each pair of points. Let

$$d_1, d_2, \ldots, d_m$$

the list of possible distances sorted in ascending order, where $m = \binom{|P|}{2}$.

First we ask ourselves whether $OPT = d_1$. If yes, so much the better, otherwise, we wonder if $OPT = d_2$, then if $OPT = d_3$, and so on. To answer these questions, we will use auxiliary graphs. For $d_i \in \{d_1, d_2, \ldots, d_m\}$, we define

$$G[d_i] = (P, \{p_1p_2 : dist(p_1, p_2) \leq d_i\})$$

as the graph whose vertices are the points P, and we add an edge between two points if the distance does not exceed d_i.

It turns out that determining whether P admits k centers with a distance not exceeding d_i is equivalent to finding an *dominating set* with k vertices in $G[d_i]$. Recall that in a graph $G = (V, E)$, a set $X \subseteq V$ is *dominating* if $\forall v \in V \setminus X$, v has at least one neighbor in X. Put another way, X dominates everything outside X.

The equivalence between centers and dominating sets can be demonstrated as follows.

Lemma 4. *Let $P' \subseteq P$ be a set of k centers. Then $\max_{p \in P}(dcentre(p, P')) \leq d_i$ if and only if P' is a dominating set in $G[d_i]$.*

Proof. Recall that to demonstrate an "if and only if", two directions must be proven, one for each side of the involvement.
($\Rightarrow$): we prove that if $\max_{p \in P}(dcentre(p, P')) \leq d_i$, then P' is a dominating set of $G[d_i]$.

Let $p \in P \setminus P'$. We know that p is at most distance d_i away from its center $p' \in P'$. By the definition of $G[d_i]$, this means that there is an edge between p and p'. So p is dominated by an element of P'. Since this is true for any $p \in P \setminus P'$, P' is a dominating set of $G[d_i]$.

($\Leftarrow$): we prove that if P' is a dominating set of $G[d_i]$, then any p point is at most d_i away from a P' point.

If $p \in P'$, then p is a center and we don't have to consider it. Otherwise, we know that in $G[d_i]$, p has a neighbor p' in P' that dominates it. By the definition of $G[d_i]$, $dist(p, p') \leq d_i$, which implies that $dcentre(p, P') \leq d_i$. So any point is at most d_i away from a center. □

A consequence of the previous lemma is that OPT is equal to the smallest d_i such that $G[d_i]$ contains a dominating set of size k. A strategy would therefore be to iterate through the graphs $G[d_1], G[d_2], \ldots, G[d_m]$ and, for each of them, check whether it admits a dominating set of size k. Unfortunately, there is no known polynomial time algorithm to answer this question

(otherwise, we might have an optimal algorithm). We will therefore consider each $G[d_i]$ from the smallest to the largest, and try to find a possibly sub-optimal dominating set in a greedy way.

Let's first consider a simple algorithm to find a dominating set.

```
fonction getDomSet(G = (V, E))
    X = {}
    while |V| > 0 do
        Choose u ∈ V arbitrarily; Add u to X; Remove from G the
         top v and all its neighbors;
    end
    return X
```

This returns a dominating set because every time we add a vertex, its neighbors become dominated. On the other hand, it is not necessarily an optimal dominating set. We can still use this subroutine for our approximation.

```
fonction kCentres(P, k, dist)
    D = {}                                          // Distances
    for each pair of vertices p, q ∈ P do
        D.append(dist(p, q))
    end
    Sort D = {d_1, d_2, ..., d_m} in ascending order
    for i = 1 to m do
        Build G[d_i]
        P' = getDomSet(G[d_i])
        if |P'| ≤ k then
            return P'
        end
    end
```

Note that we stop at the first $G[d_i]$ that admits a dominant set P' such that $|P'| \leq k$, and not such that $|P'| = k$. We leave it up to you to think about this technicality.

Theorem 7. *kCentres is a 2-approximation.*

Proof. Let $d_i = OPT$. To prove the theorem, we hope that the algorithm

finds a dominating set in a $G[d_j]$ such that $d_j \leq 2\dot{d}_i$. To simplify, we will assume that $G[2d_i]$ is constructed (in reality, we should take the highest d_j which is less than $2d_i$).

We know that $G[d_i]$ admits a dominating set $P' = \{p'_1, p'_2, \ldots, p'_k\}$ of size k. The centers of P' determine groups, called clusters. That is, for any $p \in P$, we say that p chooses the center $p' \in P'$ if p' is the closest choice to p. For each $j \in \{1, 2, \ldots, k\}$, we define the cluster

$$C_j = \{p \in P : p \text{ chooses the center } p'_j \}$$

Therefore, all the points in the same C_j are at a distance of at most d_i from p'_j. By the triangular inequality, if $p, q \in C_h$, then

$$dist(p, q) \leq dist(p, p'_j) + dist(p'_j, q) \leq 2d_i$$

A consequence of this is that in $G[2d_i]$, all the points of a single cluster are neighbors to each other. This means that the algorithm $getDomSet$, when receiving $G[2d_i]$ will first choose a $u \in V$ belonging to a certain cluster C_j, then remove all points of C_j. At the 2nd iteration, it will remove all the points from another cluster, and so on. Since there are k clusters, $getDomSet$ will return a set with at most k vertices.

We deduce that the algorithm returns a dominant set of $G[2d_i]$, which implies that $APP \leq 2d_i = 2OPT$. □

Chapter 4

Greedy approach and local search

In the previous chapter, we studied an approach that first finds a bound on OPT, and then develops an algorithm with respect to the bound found. This works well when possible, but unfortunately it is not always obvious how to apply it. Another way is to first create an algorithm, and then find the bound on OPT relative to that algorithm. This approach often fails to find a good approximation factor. On the other hand, when it works, the resulting algorithm is often simple to implement, even if the analysis is sometimes difficult.

In an optimization problem, two types of algorithms often come naturally:

- **the greedy approach**. We start with an empty solution and find the element to be added that is "the most promising". We add it, update our instance and repeat.

  It's greedy because at each step, we take the choice that seems better *now* without worrying about the future.

- **the local improvement approach**. We start with a solution of some kind, then we try to see if we can make a small local modification to improve it. If so, we do it and repeat. If not, we return this solution.

  Of course, this tends to fall into local min/max, but sometimes gives approximations with good theoretical guarantees.

In this chapter we will see an example of each approach with SET-COVER and MAX-CUT.

4.1 Local search and MAX-CUT

Let $G = (V, E)$ be a graph. A *bipartition* of G is a separation of the vertices into two non-empty parts. More specifically, a bipartition is a pair of sets (V_1, V_2) such that $V_1 \cup V_2 = V$ with $V_1 \neq \emptyset, V_2 \neq \emptyset$ and $V_1 \cap V_2 = \emptyset$. A bipartition is sometimes called a *cut*.

The edges of a bipartition (V_1, V_2) are denoted $E(V_1, V_2)$ and correspond to the edges that cross on both sides of the bipartition. In other words,

$$E(V_1, V_2) = \{uv \in E : u \in V_1, v \in V_2\}$$

In the MAX-CUT problem, we are looking for a bipartition that maximizes crossing edges.

MAX-CUT
Input: a graph $G = (V, E)$.
Output: a bipartition (V_1, V_2) that maximizes $|E(V_1, V_2)|$.

In the spirit of local search, we will start with an arbitrary bipartition and move a vertex if it allows to increase the cut.

```
fonction maxCutLocal(G = (V, E))
    Let u be any vertex of V
    V_1 = {u}
    V_2 = V \ {u}
    finished = False
    while not finished do
        finished = True
        for v ∈ V_1 do
            if |E(V_1 \ {v}, V_2 ∪ {v})| > |E(V_1, V_2)| then
                V_1.remove(v)
                V_2.insert(v)
                finished = False
            end
        end
        for v ∈ V_2 do
            if |E(V_1 ∪ {v}, V_2 \ {v})| > |E(V_1, V_2)| then
                V_1.insert(v)
                V_2.remove(v)
                finished = False
            end
        end
    end
    return (V_1, V_2)
```

A first question concerns the time required by this algorithm. Is it possible that it loops infinitely? It cannot be excluded that the algorithm cycles through a series of movements without ever ending. A first element to demonstrate with this type of algorithm is not only that it finishes, but also that it finishes in polynomial time. This is often complex, but in our example, a simple proof is enough.

Lemma 5. *The maxCutLocal algorithm ends after $O(n^2)$ iterations of the main loop.*

Proof. Let (V_1, V_2) be the bipartition in memory at the beginning of an iteration of the loop, and let (V_1', V_2') the bipartition at the end of the same iteration. If $finished = False$, at least one movement has been made that increases the number of edges that cross. This implies that $E(V_1', V_2') > E(V_1, V_2)$. So the value of the cut increases by at least 1 with each loop

pass. Since the number of edges of a bipartition is at most $\binom{n}{2} \in O(n^2)$, the maximum number of times we can increase the value of the cut is $O(n^2)$. □

This implies that the algorithm takes a polynomial time since an iteration of the loop can be done in time $O(n^2)$. Let us now focus on the performance of *maxCutLocal* in terms of approximation. Note that this algorithm was developed independently at any bound on OPT. This complicates the analysis, as we are now constrained to bound OPT according to the algorithm (whereas in the previous chapter we were free to set OPT without constraints). It is therefore difficult to describe a general analysis technique for this type of algorithm — it seems that every gluttonous algorithm requires an ad hoc analysis.

For a vertex $v \in V$, we denote by $N(v)$ the set of neighbors of v in G.

Lemma 6. *Let (V_1, V_2) be the solution returned by maxCutLocal. For any $v \in V_1$, v has at least half of its neighbors in V_2, i.e.*

$$|N(v) \cap V_2| \geq |N(v)|/2$$

In addition, for any $v \in V_2, |N(v) \cap V_1| \geq |N(v)|/2$.

Proof. We prove for $v \in V_1$ only. The case $v \in V_2$ is identical.

Suppose for the sake of contradiction that there exists $v \in V_1$ such that v has strictly less than $|N(v)|/2$ neighbors in V_2. Then v has strictly more than $|N(v)|/2$ neighbors in V_1. The contribution of v to $E(V_1, V_2)$ is smaller than $|N(v)|/2$. If we transfer v into V_2, the number of edges of the bipartition is changed by an amoutn of

$$|N(v) \cap V_1| - |N(v) \cap V_2|$$

because we gain the edges of v towards V_1, but we lose the edges of v towards V_2. This difference is strictly greater than 0. So the transfer of v increases the cut. This is a contradiction, because the algorithm would then have made this transfer. □

So the intuition is that each vertex has half of the neighbors on the other side. In the best of worlds, each vertex would have *all* its neighbors on the other side, which forms our bound on OPT and gives our $1/2$ approximation.

Theorem 8. *maxCutLocal is a $1/2$ approximation.*

Proof. Note that the number of edges in a bipartition is equal to the sum, over all vertices, of the number of neighbors crossing, divided by 2 because

each edge is counted twice. That is to say, for any bipartition (V_1', V_2'), we have

$$|E(V_1', V_2')| = \frac{1}{2}\left(\sum_{v \in V_1'} |N(v) \cap V_2'| + \sum_{v \in V_2'} |N(v) \cap V_1'|\right)$$

In the best possible case, each intersection contain all neighbors. Therefore,

$$OPT \leq \frac{1}{2} \cdot \sum_{v \in V} |N(v)|$$

In the case of the *maxCutLocal* algorithm, we know that each intersection contains at least half of the neighborhoods.

$$\begin{aligned} APP \geq \frac{1}{2} \cdot \sum_{v \in V} \frac{|N(v)|}{2} &= \frac{1}{2} \cdot \frac{1}{2} \sum_{v \in V} |N(v)| \\ &\geq \frac{1}{2} \cdot OPT \end{aligned}$$

□

4.2 Greedy algorithm for SET-COVER

It turns out that the greedy strategy works well for SET-COVER, in which each set has an arbitrary number of elements. The goal is to cover a universe U with a minimum number of sets.

SET-COVER
Input: universe $U = \{u_1, \ldots, u_n\}$, sets $S = \{S_1, \ldots, S_m\}$
Output: sets $S^* \subseteq S$ that cover U and minimize $|S^*|$.

For precision, we say that $S^* \subseteq S$ covers U if $\bigcup_{S' \in S^*} S' = U$. As an exercise, we will also ask to study the version in which each $S_i \in S$ has a different cost, and we try to minimize the total cost of S^* instead of its number of elements.

A very intuitive idea is to start with $S^* = \{\}$ and then add the set $S_i \in S$ that covers a maximum number of elements. Then add the next set that covers a maximum of remaining elements, and so on. This gives the following algorithm.

```
fonction setCoverGreedy(U = {u_1, ..., u_n}, S = {S_1, ..., S_m})
    R = U //Elements remaining to be covered
    while R ≠ ∅ do
        Let S_i ∈ S that maximizes |S_i ∩ R|        // greedy choice
        S*.append(S_i)
        R = R \ S_i
    end
    return S*
```

As in MAX-CUT, this algorithm was not developed with a bound on OPT in mind, again requiring an ad hoc analysis.

For the setCoverGreedy algorithm, we can look at the "cost" of covering each $u_k \in U$. We will rely on the following intuition. If it took a whole set to cover only one u_k and no other items, then u_k was expensive (it took a set just for itself). But if u_k was covered by adding a S_j, and S_j also covered 99 of other new U items, then the cost of u_k is much less because the cost of covering u_k was spread over 100 elements.

More rigorously, we say that each $S_i \in S$ has a cost of 1. Hence the cost of a solution S^* is equal to $|S^*|$. The moment a S_i set is added to S^* by the algorithm, it distributes its 1 cost to all new covered elements. That is, when S_i is added to S^*, each element $u_k \in S_i \cap R$ is assigned the cost

$$cost(u_k) = \frac{1}{|S_i \cap R|}$$

where, again, R is the set of elements remaining to be covered **at the moment when S_i is added to S^***. The cost of each u_k is therefore dependent on how many other elements were covered at the same time as it.

The interest of this distribution of costs is that it gives us another way of measuring $APP = |S^*|$.

Lemma 7. *Let S^* be the set returned by setCoverGreedy. Then*

$$\sum_{k=1}^{n} cost(u_k) = |S^*|$$

Proof. Each time we add a S_i to S^*, we distribute a total of 1 across the elements of $S_i \cap R$. Since each element u_k receives a cost only once, each $S_i \in S^*$ contributes exactly 1 to the sum of the costs, which explains

$\sum_{k=1}^{n} cost(u_k) = |S^*|$. □

Now, the $cost(u_k)$ will serve as a "middle point" between OPT and APP. We still have to establish a link between OPT and these costs. In the algorithm, more and more items are covered. We can therefore order U so that u_1 is the element covered first, u_2 is covered second, and so on. (several elements could be covered in the same iteration, in which case an arbitrary order is determined).

So we assume that $u_1, u_2, \ldots, u_n$ gives us the order in which the elements of U are covered. Let's take any u_k.

Lemma 8. *For every $k \in \{1, 2, \ldots, n\}$, we have*

$$OPT/(n - k + 1) \geq cost(u_k)$$

Proof. Just before u_k becomes covered, $|R| = n - k + 1$ items remain to be covered. We know that the optimal solution is able to cover these $n - k + 1$ elements of R with OPT sets (because OPT sets can cover all the elements). So, necessarily, there is a set $S_i \in S^*$ such that S_i contains $(n-k+1)/OPT$ elements of R (because otherwise it is impossible to cover R with OPT sets).

Since the algorithm maximizes $|S_i \cap R|$, it will therefore choose to add to S^* a S_i set such that $|S_i \cap R| \geq (n - k + 1)/OPT$. This gives

$$OPT/(n - k + 1) \geq 1/|S_i \cap R|$$

Remember that this S_i contains u_k because we are at the point just before u_k is covered. This concludes the proof, because $1/|S_i \cap R| = cost(u_k)$. □

We now have everything we need to compare $APP = |S^*| = \sum_{k=1}^{n} cost(u_k)$ and OPT. We will find ourselves with what is called the *harmonic series* denoted H_n. More precisely, we define

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$

It turns out that our algorithm is a H_n-approximation.

Theorem 9. *setCoverGreedy is a H_n-approximation.*

Proof. We know that

$$\begin{aligned} APP = \sum_{k=1}^{n} cost(u_k) &\leq \sum_{k=1}^{n} OPT/(n-k+1) \\ &= OPT \cdot \sum_{k=1}^{n} 1/(n-k+1) \\ &= OPT \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right) \\ &= H_n \cdot OPT \end{aligned}$$

which concludes the proof. □

It is possible to demonstrate that $H_n \in O(\log n)$. It is therefore often said that *setCoverGreedy* is a $O(\log n)$-approximation. It turns out that if P $\neq$ NP, there is no better algorithm, i.e. there is no $f(n)$-approximation for any $f(n) \in o(\log n)$ (where the notation o refers to functions that grow strictly slower than $\log n$).

Chapter 5

Probabilistic algorithms

We will see in this chapter some examples of algorithms that can make random choices. These algorithms do not always return the same solution and its quality may vary according to chance. However, in many cases, these algorithms can give a reasonable approximation factor *in expectation*. We will then see that it is sometimes possible to *derandomize* some probabilistic algorithms and make them completely deterministic with the same theoretical guarantees.

5.1 Basics

For our purposes, a probabilistic algorithm runs in polynomial time if it *always* returns a solution in polynomial time, regardless of its random choices. This differs from some analyses that require polynomial time in expectation only. Let A be an approximation algorithm. To define the expectation of the value, we note that the value $APP(A)$ of the solution returned by A is a random variable. For a numerical value K, we write

$$Pr[APP(A) = K]$$

to denote the probability that A returns a solution of value K. It is assumed that the possible values of K form a countable set.

The expectation of the APP value of A is denoted by

$$\mathbb{E}[APP(A)] = \sum_K Pr[APP(A) = K] \cdot K$$

where the sum applies to all possible values of APP. The notion of proba-

bilistic c-approximation is defined as follows:

- if A solves a minimization problem, A is a c-approximation if $\mathbb{E}[APP(A)] \leq c \cdot OPT$;
- if A solves a maximization problem, A is a c-approximation if $\mathbb{E}[APP(A)] \geq c \cdot OPT$.

Note that most of the time, we will not calculate the expectation directly in the above way and will use a few tricks to simplify our calculations. One of the most important is the *linearity of expectation*, well known in probability and statistics.

Theorem 10. *Let $X_1, X_2, \ldots, X_n$ be random variables. Then*

$$\mathbb{E}[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

5.2 A probabilistic $1/2$-approximation for MAX-CUT

One of the simplest examples of probabilistic approximation is for MAX-CUT. Tto construct a bipartition (V_1, V_2), we will simply put each vertex in one of the two sides randomly, each with probability $1/2$.

```
fonction maxCutProba(G = (V, E))
    V1 = ∅
    V2 = ∅
    for v ∈ V do
        res = flipCoin()
        if res = tails then
            V1.append(v)
        else
            V2.append(v)
        end
    end
    return (V1, V2)
```

Despite the simplicity of this algorithm, half of the edges will end up in the cut on average, because each edge has a 50/50 chance of having its ends separated. We formalize this idea below.

Theorem 11. *The maxCutProba algorithm is a* $1/2$ *probabilistic approximation.*

Proof. In this problem, APP is the number of separated uv edges in (V_1, V_2). For $uv \in E$, we define the random variable

$$\mathbb{1}_{uv} = \begin{cases} 1 & \text{if } u \in V_1 \text{ and } v \in V_2, \text{ or if } u \in V_2 \text{ and } v \in V_1 \\ 0 & \text{otherwise} \end{cases}$$

We have $APP = \sum_{uv \in E} \mathbb{1}_{uv}$. The expectation of APP is therefore

$$\mathbb{E}[APP] = \mathbb{E}[\sum_{uv \in E} \mathbb{1}_{uv}] = \sum_{uv \in E} \mathbb{E}[\mathbb{1}_{uv}]$$

Here, $\mathbb{1}_{uv}$ is the probability that u and v will be separated in a solution. It can be argued that $\mathbb{1}_{uv} = 1/2$, because there are 4 equiprobable combinations of possible locations for u and v, and 2 of these cases separate u and v. Therefore,

$$\mathbb{E}[APP] = \sum_{uv \in E} \mathbb{E}[\mathbb{1}_{uv}] = \sum_{uv \in E} 1/2 = |E|/2$$

Since $OPT \leq |E|$, we get $\mathbb{E}[APP] = |E|/2 \geq OPT/2$. □

Note that this algorithm also works for the MAX-CUT version with weights, because each weight has a 50/50 chance of ending up in the cut. The advantage of this algorithm is that it is easy to implement and is very fast. In practice, a common technique with this type of algorithm is to run it several times and keep the best solution.

5.3 A probabilistic $7/8$-approximation for MAX-3-SAT

MAX-3-SAT is the variant of MAX-SAT in which each clause has exactly 3 variables.

MAX-3-SAT
Input: a set of clauses $C_1, \ldots, C_m$ with 3 variables each, on variables $x_1, \ldots, x_n$;
Output: an assignment of the x_i variables that maximizes the number of satisfied clauses.

There is a very simple algorithm for this problem: choose an assignment at random.

fonction *max3satProba*($C_1, \ldots, C_m$ *on variables* $x_1, \ldots, x_n$)
  A = empty assignment
  for $i = 1..n$ **do**
    $res = flipCoin()$
    if $res = tails$ **then**
      Assign $x_i = True$ in A
    else
      Assign $x_i = False$ in A
  end
  return A

Consider a C_i clause, for example $C_i = (x_i \vee \overline{x}_j \vee x_k)$. The only way to *not* satisfy C_i is that all of its variables evaluate to $false$. There are 8 combinations of assignments for the 3 variables of C_i, and 7 of these combinations satisfy C_i. Since each combination is chosen with the same probability, there is therefore a 7/8 probability of satisfying C_i. Since this is true for each clause, by the linearity of expectation, we expect that a 7/8 proportion of the clauses will be satisfied.

The following theorem follows. The proof only formalizes the above paragraph.

Theorem 12. *The max3satProba algorithm is a probabilistic* 7/8*-approximation.*

Proof. Let A be the assignment returned by the algorithm. For a C_i clause, let us define

$$\mathbb{1}_{C_i} = \begin{cases} 1 & \text{if } A \text{ satisfies } C_i \\ 0 & \text{otherwise} \end{cases}$$

We get

$$\mathbb{E}[APP] = \mathbb{E}[\sum_{i=1}^{m} \mathbb{1}_{C_i}] = \sum_{i=1}^{m} \mathbb{E}[\mathbb{1}_{C_i}] = \sum_{i=1}^{m} 7/8 = 7m/8$$

We know that $OPT \leq m$, and therefore $\mathbb{APPP}[E] = 7m/8 \geq 7/8 \cdot OPT$. □

5.4 Derandomization

Purists will say that a probabilistic c-approximation is not appropriate, because it could be that, by some misfortune, APP is very far from OPT. What is the purpose of providing theoretical guarantees if they are only guaranteed with a certain probability? A first answer is that it is possible to show that if we execute the same algorithm several times and keep the best solution, the probability of moving away from the expectation decreases exponentially. If this interests you, search for "Chernoff bound".

Another answer is that sometimes the effect of chance can be eliminated by transforming the probabilistic algorithm into a deterministic algorithm while maintaining the same guarantees of approximation. This is called *derandomization*.

The idea is first to compute the expectation of a given instance, a bit like in MAX-CUT or MAX-3-SAT analysis. We try to add an element to our solution and calculate the effect of setting this choice on the expectation. If it has not worsened, we fix the choice. If it has worsened, we do not make this choice and move on to the next one. If each of the choices maintains the hope that we originally had, we will have fixed a set of deterministic choices that gives a solution that reaches the original hope.

Derandomization of MAX-3-SAT

As an example, let's take a look at the "derandomized" version of MAX-3-SAT. Let A_j be an assignment of the variables $x_1, x_2, \ldots, x_j$. This is a partial assignment, because A_j assigns the value of some variables, but not all (except if $j = n$). The idea is that some clauses are already satisfied by A_j, some will never be, and some may be.

We assume that we set A_j and assign $x_{j+1}, \ldots, x_n$ randomly. The probability of satisfying a clause C_i *given* A_j can be expressed as follows:

$$Pr[C_i \text{ is satisfied } |A_j] = \begin{cases} 1 & \text{if } A_j \text{ already satisfies } C_i \\ 1 - \frac{1}{2^k} & \text{otherwise, where } k \text{ is the number of variables} \\ & \text{from } C_i \text{ not assigned by } A_j \end{cases}$$

For example, suppose that according to A_2, we have $x_1 = true$ and $x_2 = false$. Let $C_i = (\overline{x}_1 \vee x_3 \vee \overline{x}_4)$. So knowing that A_2 affects $x_1 = true$, we know that we are not going to satisfy C_1 with x_1. This leaves 2 variables capable of satisfying C_i with probability $1 - 1/4 = 3/4$. This corresponds to 3 chances out of 4 to satisfy C_i with x_3 or x_4.

We can argue that the expectation of APP given A_j is

$$\mathbb{E}[APP|A_j] = \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_j]$$

The idea of the derandomization is to start with an empty A_0 assignment. As seen above, $\mathbb{E}[APP] = \mathbb{E}[APP|A_0] = 7m/8$. We will try to assign $x_1 = true$ in a temporary assignment A_1^+ and calculate $\mathbb{E}[APP|A_1^+]$. We then try to assign $x_1 = false$ in an assignation A_1^- and calculate $\mathbb{E}[APP|A_1^-]$. We choose A_j to be the choice with maximum expected value.

fonction *max3satDerand($C_1, \ldots, C_m$ on variables $x_1, \ldots, x_n$)*
  A_0 = empty assignment
  for $j = 1..n$ **do**
    A_j^+ = copy of A_{j-1} with $x_j = true$
    $E_j^+ = \mathbb{E}[APP|A_j^+] = \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_j^+]$
    A_j^- = copy of A_{j-1} with $x_j = false$
    $E_j^- = \mathbb{E}[APP|A_j^-] = \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_j^-]$
    if $E_j^+ \geq E_j^-$ **then**
      $A_j = A_j^+$;
    else
      $A_j = A_j^-$
    end
  end
  return A_n

Note that even though it does probability calculations, this algorithm is entirely deterministic. The key idea is that after each loop, we set x_j in A_j such that $\mathbb{E}[APP|A_j] \geq 7m/8$.

Lemma 9. *After the j-th loop of the algorithm, $j \in \{0, 1, \ldots, n\}$, we have $\mathbb{E}[APP|A_j] \geq 7m/8$.*

Proof. Let $j \in \{0, 1, dots, n\}$. We show by induction on j that $\mathbb{E}[APP|A_j] \geq 7m/8$.

As a base case, we take $j = 0$. We have already argued that $\mathbb{E}[APP|A_0] \geq 7m/8$ because a priori, each clause has $7/8$ chances to be satisfied.

Let us consider $j > 0$, assuming by induction that $\mathbb{E}[APP|A_{j-1}] \geq 7m/8$. The expectation of A_{j-1} had been calculated using the fact that there was

a one in two chance of assigning $x_j = true$, and a one in two chance of assigning $x_j = false$. So we end up with

$$\begin{aligned}
\mathbb{E}[APP|A_{j-1}] &= \sum_{i=1}^{m} Pr[C_i \text{ is satisfied } |A_{j-1}] \\
&= \sum_{i=1}^{m} \left(\frac{1}{2} Pr[C_i \text{ is satisfied } |A_j^+] + \frac{1}{2} Pr[C_i \text{ is satisfied } |A_j^-] \right) \\
&= \frac{1}{2}\mathbb{E}[APP|A_j^+] + \frac{1}{2}\mathbb{E}[APP|A_j^-] \\
&= \frac{1}{2}E_j^+ + \frac{1}{2}E_j^- \\
&\geq 7m/8
\end{aligned}$$

The last inequality is simply because $\mathbb{E}[APP|A_{j-1}] \geq 7m/8$ by induction. This inequality also implies that one of E_j^+ or E_j^- is at least $7m/8$. Since A_j is chosen with respect to $max(E_j^+, E_j^-)$, $\mathbb{E}[APP|A_j] \geq 7m/8$. □

Theorem 13. *The max3satDerand algorithm is a* $7/8$ *approximation (deterministic).*

Proof. Let A_n be the assignment after n loops. So A_n is returned by the algorithm and assigns each variable. By the previous lemma, $\mathbb{E}[APP|A_n] \geq 7m/8$. Note that with A_n, each clause is satisfied or not, and there is no more probabilistic choice possible. So $\mathbb{E}[APP|A_n] \geq 7m/8$ implies that A_n satisfies at least $7m/8$ clauses. Since $OPT \leq m$, we deduce that we have a $7/8$-approximation. □

Chapter 6

Polynomial time approximation schemes (and KNAPSACK)

What is the ultimate approximation ratio? If we solve a minimization problem, the best we can hope for an NP-complete problem is

$$APP \leq (1 + \varepsilon) \cdot OPT$$

with very small ε. In the same way, the ideal for a maximization problem is

$$APP \geq (1 - \varepsilon) \cdot OPT$$

It turns out that for some problems, it is possible to reach an accuracy arbitrarily close to 1 (and thus an ε arbitrarily close to 0). To do this, we give the desired ε to the algorithm, and it manages to guarantee our approximation factor. This is called an *polynomial time approximation scheme*.

6.1 Polynomial Time Approximation Scheme (PTAS)

.

A PTAS is an approximation algorithm A that receives as input an instance *and* a ε parameter, and such that:

- A runs in polynomial time relative to the n size of the instance, where ε is treated as a constant;
- if A solves a minimization problem, $APP \leq (1 + \varepsilon) \cdot OPT$

- if A solves a maximization problem, $APP \geq (1-\varepsilon)\dot{O}PT$.

The first point specifies that the complexity is independent of ε, it is common for $1/\varepsilon$ to contribute to the complexity. Some PTAS's, for example, are executed in time $O(\frac{1}{\varepsilon}n^2)$, or in time $O(2^{2^{1/\varepsilon}}n)$. This is acceptable if $\varepsilon = 1/2$, but if $\varepsilon = 1/n$, this has a serious impact on complexity. Other definitions that consider ε in complexity exist. For example, an FPTAS requires that complexity is polynomial in n *and* $1/\varepsilon$.

Of course, it is not always possible to find a PTAS, or some PTAS have an absurd complexity and have no practical application. We will give an example of a PTAS that turns out to be applicable in practice.

6.2 KNAPSACK Problem

In the KNAPSACK problem, we receive the capacity W of a bag and objects, each having a weight w_i and a value v_i. The objective is to fill the bag with a maximum total value without exceeding the capacity. This problem is called the KNAPSACK.

Let $R = \{(w_1, v_1), \ldots, (w_n, v_n)\}$ be a set of objects, where the object i has weight w_i and value v_i. For $R' \subseteq R$, we will denote $w(R') = \sum_{(w_i,v_i)\in R'} w_i$ and $v(R') = \sum_{(w_i,v_i)\in R'} v_i$.

KNAPSACK
Input: integer W, objects $R = \{(w_1, v_1), \ldots, (w_n, v_n)\}$.
Output: a subset $R' \subseteq R$ such that $w(R') \leq W$ that maximizes $v(R')$.

There is a classic dynamic programming algorithm for KNAPSACK. Let's first denote

$$v_{max} = \max\{v_1, \ldots, v_n\}$$

Note that $OPT \leq n \cdot v_{max}$. We will iterate through each possible profit from 1 to nv_{max} and ask ourselves if it is possible to reach this profit.

For $i \in \{1, 2, \ldots, n\}$ and for $p \in \{1, 2, \ldots, n \cdot v_{max}\}$, we define a $B(i, p)$ function as the minimum possible weight allowing to reach a profit p using only the $(w_1, v_1), \ldots, (w_i, v_i)$ objects, i.e. the first i objects. Put another way,

$$B(i,p) = \min\{w : \exists R' \subseteq \{(w_1, v_1), \ldots, (w_i, v_i)\} \text{ such that } w(R') = w \text{ and } v(R') = p$$

We look for the maximum value of p such that $B(n,p) \leq W$, because in this case, $B(n,p)$ corresponds to a way of choosing objects among all those in the input that has a p profit and a weight below the capacity.

We have that $B(i,0) = 0$ for all i, because we can always have a profit 0 with a weight 0. For the other values, it is possible to argue that

$$B(i,p) = min \begin{cases} B(i-1,p) \\ B(i-1,p-v_i) + w_i \end{cases}$$

This is because there are two ways to achieve a profit p. Maybe we don't use (w_i, v_i) and we reach a p profit with the first $i-1$ items. This can be done with a weight $B(i-1,p)$. Otherwise, we do use (w_i, v_i). In this case, we had a profit $p - v_i$ with the first $i-1$ elements and we added v_i. We also added a weight of w_i to the bag. The weight in this case is $B(i-1, p-v_i) + w_i$. We just take the minimum of the two.

```
DontPrintSemicolon fonction
 sacados(W, (w_1, v_1), (w_2, v_2), ..., (w_n, v_n))
   B = table of dimension n × nv_max;
   Initialize B(i,p) = ∞ for all i,p;
   B(i,0) = 0 for all i;
   pmax = 0;
   for i = 1..n do
      for p = 1..nv_max do
         B(i,p) = min(B(i-1,p), B(i-1,p-v_i) + w_i);
         if B(i,p) ≤ W and p > pmax then
            pmax = p;
         end
      end
   end
   return pmax;
```

This algorithm returns only the maximum profit instead of a set of concrete objects. If we want, we can reconstruct a concrete solution reaching a *pmax* profit from the B table, which requires applying dynamic programming *backtracking*.

```
fonction
 sacadosBacktrack(W, (w_1, v_1), (w_2, v_2), ..., (w_n, v_n), B, pmax)
   X = ∅
   i = n
   while i > 0 do
       if B(i, pmax) = B(i − 1, pmax − v_i) + w_i then
           X.append((w_i, v_i))
       i − −
   end
   return X
```

The complexity of the algorithm is $O(n^2 v_{max})$, whether we compute $pmax$ only or a concrete solution X. This is not necessarily polynomial. For example, it is possible that $v_{max} = 2^n$, in which case the time is actually exponential (in reality, the complexity of an algorithm is measured with respect to the number of bits in the input. The fact is that even if $v_{max} = 2^n$, it adds $\log(2^n) = n$ bits to the input for its representation and a time of 2^n is always exponential). In fact, the KNAPSACK problem is NP-complete, and there is probably no such thing as a purely polynomial time algorithm.

The above algorithm is called *pseudo-polynomial*, because it is polynomial in the numerical values of the input, but not polynomial in the number of bits required to represent these values.

6.3 A PTAS for KNAPSACK

Since v_{max} is a problem, why not reduce the values of the objects so that v_{max} is polynomial? If we divide all the values by the same K factor, the problem remains the same. On the other hand, since all v_i must be integers, we will take the floor value of the division.

We don't know by which factor K to divide our v_i, so we'll keep it as a variable and hope to determine it during our analysis. To simplify our life, we will sometimes write

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor$$

to denote the modified weight. So we will create an instance with the same W capacity, but the objects become

$$R' = \{(w_1, v'_1), \ldots, (w_n, v'_n)\}$$

If for example $K = v_{max}/n$, the values v'_i will all be polynomial. But remember that K is unknown yet (we will discover later that $K = v_{max}/n{\cdot}\varepsilon$). Our algorithmic strategy is simple: modify the values and use dynamic programming.

fonction *sacadosPoly*$(W, (w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n), K)$
  For each $i \in \{1, \ldots, n\}$, define $v'_i = \lfloor v_i/K \rfloor$
  $X = sacados(W, (w_1, v'_1), \ldots, (w_n, v'_n))$
  return X

This algorithm does not always give the optimal solution since accuracy is lost in the floor values. In other words, the X solution returned by the algorithm is optimal **relative to the** v'_i values, but not necessarily relative to the v_i values. We now need to quantify this loss of precision with respect to K. To quantify the loss of the floor values, we will use the following, left in exercise.

Lemma 10. *For all v_i, since $v'_i = \lfloor v_i/K \rfloor$, we have*

$$\frac{v_i}{K} - 1 \leq v'_i \leq \frac{v_i}{K}$$

If we isolate v_i, we get $Kv'_i \leq v_i \leq K(v'_i + 1)$.

Let X be the solution returned by *sacadosPoly*. So $APP = \sum_{(w_i,v_i)\in X} v_i$ is the sum of the values **compared to the original values** v_i.

As usual, our goal is to develop an inequality chain of the style $APP \geq \ldots \geq c \cdot OPT$. We just need to fill in the three little dots. Let's start with what we know.

$$APP = \sum_{(w_i,v_i)\in X} v_i \geq K \sum_{(w_i,v_i)\in X} v'_i$$

We're a bit blocked. Let's try to connect this last expression with OPT. Let X^* be an optimal solution for the v_i values. If we divide the element values of X^* by K, we can interpret X^* as a solution for the values v'_i. We know that X is optimal with respect to these values. So $\sum_{(v_i,w_i)\in X^*} v'_i \leq \sum_{(v_i,w_i)\in X} v'_i$.

Let's see what we can say about OPT.

$$\begin{aligned} OPT = \sum_{(w_i,v_i)\in X^*} v_i &\leq \sum_{(w_i,v_i)\in X^*} K(v_i'+1) \\ &= K|X^*| + K\sum_{(w_i,v_i)\in X^*} v_i' \\ &\leq K|X^*| + K\sum_{(w_i,v_i)\in X} v_i' \end{aligned}$$

So, $K\sum_{(w_i,v_i)\in X} v_i' \geq OPT - K|X^*|$. By reconnecting with what we know about APP, we have

$$APP \geq K\sum_{(w_i,v_i)\in X} v_i' \geq OPT - K|X^*|$$

Now it's finally time to choose K. K must contain v_{max} somewhere for the values to become polynomial. In addition, K should help us get rid of $|X^*|$. With a little trial and error, we can see that we can set $K = \varepsilon \cdot v_{max}/n$ for all ε. Since $n \leq v_{max}$, and since $OPT \geq v_{max}$, we obtain

$$APP \geq OPT - \frac{\varepsilon \cdot v_{max}}{n}|X^*| \geq OPT - \varepsilon \cdot v_{max} \geq OPT - \varepsilon \cdot OPT = (1-\varepsilon)\cdot OPT$$

If we pass the value $K = \varepsilon \cdot v_{max}/n$ to *sacadosPoly*, the complexity of the dynamic programming becomes

$$O(n \cdot nv_{max}/K) = O(n^2 \cdot v_{max} \cdot n/(\varepsilon \cdot v_{max}))) = O(\frac{1}{\varepsilon}n^3)$$

The better the desired precision, the closer ε is to 0, and the more $1/\varepsilon$ increases complexity. The advantage is that we can adjust ε. We conclude with the following theorem.

Theorem 14. *For any $\varepsilon > 0$, there is a $(1-\epsilon)$-approximation to the KNAPSACK problem. which runs in time $O(\frac{1}{\varepsilon}n^3)$.*

Chapter 7

Approximation and Linear Programming

A linear program is an ultra-generic way of formulating an optimization problem. We assume that we have a set of numerical variables $x_1, \ldots, x_n$ (possibly others). One must specify the objective function to be minimized or maximized, and this function must be linear with respect to the variables. We also specify a set of *linear constraints*. A linear constraint is an inequality in which each term contains only one variable, possibly multiplied by a constant. Here is an example:

$$
\begin{aligned}
&\text{Minimize} \\
&\qquad 10x_1 + 12x_2 + 4x_3 \\
&\text{Subject to} \\
&\qquad x_1 + 2x_2 \geq 5 \\
&\qquad 3x_2 - x_3 \geq 3 \\
&\qquad x_1 + x_2 \geq 2 \\
&\qquad 0 \leq x_i \leq 1 \qquad\qquad \text{for each } x_i \in \{x_1, x_2, x_3\}
\end{aligned}
$$

The first two lines indicate the objective, and the other lines the constraints (the last line actually represents 6 different constraints).

Generally speaking, a linear program is often expressed in matrix form.

$$\begin{aligned} &\text{Minimize} \\ &\qquad \boldsymbol{c}^T\boldsymbol{x} \\ &\text{Subject to} \\ &\qquad \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ &\qquad \boldsymbol{x} \geq 0 \end{aligned}$$

where $\boldsymbol{x} = (x_1, \ldots, x_n)$ is the vector of our variables, $\boldsymbol{c}$ represents the coefficients of the variables to be optimized, $\boldsymbol{A}$ is a matrix with n columns, and $\boldsymbol{b}$ is a vector of constants. Here, only the vector $\boldsymbol{x}$ is unknown, while the values of $\boldsymbol{c}, \boldsymbol{A}$ and $\boldsymbol{b}$ contain only known constant values. We will often denote by $\boldsymbol{x}^*$ a solution vector, i.e. a vector containing concrete values for the x_i. The values of $\boldsymbol{x}^*$ are denoted by $x_1^*, \ldots, x_n^*$.

Assuming that the domain of x_i is real, a remarkable result in computer science is the following.

Theorem 15. *There is a polynomial time algorithm which, given a linear program, finds an optimal solution to the values of x_i such that each constraint is satisfied.*

One of the best known algorithms is the ellipsoid method. We will only use this theorem as a black box. The resolution of linear programs is the subject of an entire semester course. We are rather interested in using this theorem for approximation purposes.

A linear program LP for short

7.1 LP for approximation

As we have seen, it is not always easy to find a relevant bound on OPT. The usefulness of linear programs lies in their ability to provide such bounds. The idea is to express an LP so that the optimal of the objective function gives a bound on OPT. Note that the linear program does not have to be exactly our problem — it only has to give a bound on OPT somehow. The most common way is to use an *relaxation* to our problem.

Definition 1. *Let P be a problem on variables $\boldsymbol{x}$ and L a linear program on variables $\boldsymbol{x}$. It is assumed that P and L are both of the same type (minimization or maximization). We say that L is a* relaxation *of P if, for any feasible solution $\boldsymbol{x}'$ of P, $\boldsymbol{x}'$ satisfies all the constraints of L.*

Note that $\boldsymbol{x}'$ is not necessarily optimal for L, only for P. Note also that feasibility only goes one way: a feasible solution for L is not necessarily feasible for P. The advantage of expressing a relaxation for P is that the optimal for L automatically gives a bound on the optimal for P.

Theorem 16. *Let L be a linear program that is a relaxation for a problem P. Let OPT_L be the optimal for L and OPT_P the optimal for P. Then:*

- *if P is a minimization problem, $OPT_P \geq OPT_L$;*
- *if P is a maximization problem, $OPT_P \leq OPT_L$.*

Proof. Let's assume that P is a minimization problem and be $\boldsymbol{x}'$ an optimal solution for P. The value of $\boldsymbol{x}'$ for P is OPT_P. Also, $\boldsymbol{x}'$ satisfies all the constraints of L (since it is a relaxation). Thus, $\boldsymbol{x}'$ is one of the possible solutions for L, but not necessarily the optimal for L (L might find a better solution $\boldsymbol{x}^*$). We deduce that $OPT_L \leq OPT_P$. The proof in the maximization case is identical. □

The high-level approach of the LP approximation for a P minimization problem is as follows.

1. express relaxation L for P;
2. solve L in polynomial time, which results in a vector $\boldsymbol{x}^*$ of value OPT_L compared to L;
3. transform $\boldsymbol{x}^*$ into a feasible solution $\boldsymbol{x}'$ for P of value APP;
4. argue that $APP \leq c \cdot OPT_L$ ($\leq OPT$).

Points 3 and 4 are often the most difficult to handle and sometimes require creativiy.

7.2 Relaxation by Integer Linear Programs (ILP)

Most of our algorithmic problems can be expressed with an *Integer Linear Programs* (ILP), which are different from an LP. An ILP is like an LP, but with the ability to add constraints such as

$$x_i \in \{0, 1\}$$

More generally, one can require that $x_i \in \{i_1, \ldots, i_k\}$, where i_j are integers. This is a fundamental difference. In a normal LP, one usually requires $x_i \geq 0$,

or $0 \leq x_i \leq 1$, which allows the x_i to be real. The restriction to integers makes the problem NP-complete.

However, ILPs suggest a completely natural relaxation. Just transform the $x_i \in \{0,1\}$ into $0 \leq x_i \leq 1$ constraints. Any solution that is feasible for the ILP is feasible for LP, so it is a relaxation. The problem is that in a relaxation, a solution $\boldsymbol{x}^*$ can be *fractional*. We then need some ways to handle these fractions.

Before using LP and ILP for approximation, here are some common tricks when developing an ILP, assuming variables $x_i \in \{0,1\}$:

- express the logical OR $x_i \vee x_j$: constraint $x_i + x_j \geq 1$;
- express the negation $\overline{x}_i$: write $(1 - x_i)$;
- express the implication $x_i \Rightarrow x_j$: knowing that $x_i \Rightarrow x_j$ is equivalent to $\overline{x}_i \vee x_j$, we can add the constraint $1 - x_i + x_j \geq 1$. We can simplify to $x_j - x_i \geq 0$, but we often prefer the former.

Finally, let's focus on ideas that are *not* allowed:

- you should not multiply two variables in the same constraint (or in the objective function). For example, $10x_1 + 3x_2 \cdot x_3 \leq 1$ is forbidden, because it is not linear;
- you should not use non-linear functions, even if seemingly simple, such as $min(x_1, x_2), max(x_1, x_2), |x_i|, \lfloor x_i \rfloor, \lceil x_i \rceil$;

7.2.1 Application to VERTEX-COVER

Let $G = (V, E)$ be a graph. Recall that in VERTEX-COVER, we are looking for an $X \subseteq V$ of minimum size that touches each edge. This can be expressed with an ILP. We will assume that $V = \{v_1, \ldots, v_n\}$. For each v_i, we will create a variable $x_i \in \{0,1\}$. The idea is that if in a solution, we have $x_i = 1$, this represents $v_i \in X$, and if $x_i = 0$, $v_i \notin X$.

Minimize

$$\sum_{i=1}^{n} x_i$$

Subject to

$$\begin{aligned} x_i + x_j &\geq 1 && \text{for all } v_i v_j \in E \\ x_i &\in \{0,1\} && \text{for all } v_i \in V \end{aligned}$$

This expresses VERTEX-COVER because we want to minimize the number of variables with $x_i = 1$ corresponding to what we will put in X. Constraints $x_i + x_j = 1$ require that for each edge $v_i v_j \in E$, we add at least one of v_i or v_j to X.

This ILP is NP-complete to solve. We will therefore use the following relaxation.

Minimize

$$\sum_{i=1}^{n} x_i$$

Subject to

$$\begin{aligned} x_i + x_j &\geq 1 && \text{for all } v_i v_j \in E \\ 0 \leq x_i &\leq 1 && \text{for all } v_i \in V \end{aligned}$$

This LP can be solved in polynomial time. Let $\boldsymbol{x}^*$ be an optimal solution to this LP. The problem is that $\boldsymbol{x}^*$ can express the presence of a v_i in X in a *fractional* way. That is, how to interpret $x_i = 1/4$? Or $x_i = 0.6180339\ldots$? What does this correspond to in a concrete solution to VERTEX-COVER? This is where we often have to be creative. We have to find a way to transform the fractional solution $\boldsymbol{x}^*$ into a concrete solution.

In the case of the VERTEX-COVER, it turns out that we just have to round the x_i to the nearest integer. This amounts to including in our X solution every v_i such as $x_i^* \geq 1/2$. We then have to argue that we don't go too far from the optimal $\sum_{i=1}^{n} x_i^*$. The algorithm can be summarized as follows.

```
fonction vcLP(G = (V, E))
    Build the relaxation LP above for G
    Obtain an optimal solution x* for the LP
    X = ∅
    for i = 1..n do
        if x*_i ≥ 1/2 then
            X.append(v_i)
    end
    return X
```

It must now be argued that this gives a good approximation. There are two elements here. We must make sure that X is indeed a feasible solution (i.e. X touches each edge), and then we must analyze the approximation ratio.

Theorem 17. *The vcLP algorithm is a* 2 *approximation.*

Proof. First we will argue that for any $v_iv_j \in E$, we have $v_i \in X$ or $v_j \in X$ (or both). Suppose that this is not the case and that there is $v_iv_j \in E$ such that $v_i \notin X$ and $v_j \notin X$. Since neither was added to X, we had $x_i^* < 1/2$ and $x_j^* < 1/2$. This is a contradiction because $\boldsymbol{x}^*$ is supposed to satisfy every constraint in our LP, whereas here we would have $x_i^* + x_j^* < 1$. We deduce that X covers each edge.

Let's analyze the approximation ratio. Since the LP L is a VERTEX-COVER relaxation, we have $OPT \geq OPT_L = \sum_{i=1}^{n} x_i^*$. For each v_i that is in X, we had $x_i^* \geq 1/2$. This means that $1 \leq 2x_i^*$. So,

$$APP = |X| = \sum_{v_i \in X} 1 \leq \sum_{v_i \in X} 2x_i^* = 2 \sum_{v_i \in X} x_i^* \leq 2 \sum_{i=1}^{n} x_i^* \leq 2 \cdot OPT$$

□

This approach is called the *rounding technique*. We take the values of $\boldsymbol{x}^*$ and round them to obtain a vector in the $\{0, 1\}$. Of course, this does not always (in fact, rarely) work. There are many ways to round, and sometimes we have to be creative.

7.3 Packet delivery over a ring network

Let $G = (V, E)$ a cycle of n vertices. Let $V = \{1, 2, \ldots, n\}$ and $E = \{i, i{+}1\} : i \in \{1, \ldots, n\}\}$. We define $n + 1 = 1$, which simplifies the notation.

Here, G represents a circular network. We have a set of packets to deliver, where each packet has a source and a destination. For each packet (i, j), we must choose to circulate clockwise or counterclockwise. The load of an edge $e \in E$ is the number of packets that pass through e. The goal is to minimize the maximum load.

For each (i, j), we denote by $H_{i,j}$ the clockwise path from i to j, and by $A_{i,j}$ the counterclockwise path.

RING-DELIVERY
Input: cycle $G = (V, E)$ of n vertices, packets $P = \{(i_1, j_1), \ldots, (i_m, j_m)\}$
Output: a choice of paths W which contains $H_{i,j}$ or $A_{i,j}$ for each $(i, j) \in P$, which minimizes the maximum number of packets on an edge, i.e. minimizes

$$\max_{e \in E} |\{W' \in W : e \text{ is on } W\}|$$

We will reformulate this problem into an ILP. We mentioned above that we could not use min and max in an ILP, whereas our criterion to be minimized contains max. We will use a trick to "simulate" this max using additional variables. This is possible when we minimize a maximum. Our variables will be:

- h_{ij} which indicates if we take the path from i to j clockwise;
- a_{ij} which indicates if we take the path from i to j counterclockwise;
- c_e which counts the number of paths that pass through the e edge;
- c_{max} which is the maximum c_e (what we want to minimize).

The variables h_{ij} and a_{ij} should be 0 or 1, but we go directly to relaxation.

Minimize

$$c_{max}$$

Subject to

$$h_{ij} + a_{ij} \geq 1 \qquad \text{for all } (i, j) \in P$$

$$c_e = \sum_{h_{ij}: e \in H_{ij}} h_{ij} + \sum_{a_{ij}: e \in A_{ij}} a_{ij} \qquad \text{for all } e \in E$$

$$c_{max} \geq c_e \qquad \text{for all } e \in E$$

$$0 \leq a_{ij}, h_{ij} \leq 1 \qquad \text{for all } (i, j) \in P$$

Let's imagine that $h_{ij} \in \{0,1\}$ and $a_{ij} \in \{0,1\}$. The first constraint ensures that a path is chosen for each $(i,j) \in P$. The second ensures that c_e is the number of chosen paths that pass through e. The third constraint makes sure that c_{max} is greater than all c_e. Since we are trying to minimize c_{max}, this variable has no interest in exceeding the maximum load of an edge and will therefore be equal.

Since the ILP version models the problem, our LP is a relaxation and therefore $OPT \geq OPT_{LP}$.

We will use the same rounding trick as in VERTEX-COVER. For each $(i,j) \in P$, h^*_{ij} or a^*_{ij} must be at least $1/2$. We take the larger of the two, which ensures a 2-approximation.

```
fonction ringDeliveryLP(G = (V, E), P)
    Build the relaxation LP above for G
    Get an optimal solution for the LP, with values h*_ij and a*_ij
    W = ∅
    for (i, j) ∈ P do
        if h_ij ≥ 1/2 then
            Add H_ij to W
        else
            Add A_ij to W
        end
        return W
    end
```

A bit like VERTEX-COVER, this is a 2-approximation.

Theorem 18. *ringDeliveryLP is a 2-approximation.*

Proof. We denote by $h^*_{ij}, a^*_{ij}, c^*_e$ and c^*_{max} the values of an optimal solution to the LP. Let W be the set of paths returned by ringDeliveryLP. Let $e \in E$ and let W_e be the paths of W that contain e. Let $W' \in W_e$, and say that W' goes from i to j. If $W' = H_{ij}$, we added W' to W because $h_{ij} \geq 1/2$. If $W' = A_{ij}$, it's because $a_{ij} \geq 1/2$. In both cases, h_{ij} or a_{ij} contributes at least $1/2$ to c_e (because e is on the path W'). This means that $c_e \geq 1/2 \cdot |W_e|$. The maximum load of W is APP, which means that $c_{max} \geq 1/2 \cdot APP$. That is, $APP \leq 2 \cdot c_{max} = 2 \cdot OPT_{LP} \leq 2 \cdot OPT$. □

7.4 Randomized Rounding

We will now see a technique in which rounding is done probabilistically. We use the MAX-COVERAGE problem as an example. In this problem, we want to cover a maximum number of elements with k sets.

MAX-COVERAGE
Input: universe $U = \{u_1, \ldots, u_n\}$, sets $S = \{S_1, \ldots, S_m\}$ and integer k.
Output: sets $S' \subseteq S$ such that $|S'| = k$ and which maximize $|\bigcup_{S_i \in S'} S_i|$, i.e. the total number of items covered by S'.

We can express this problem with an ILP — we give the LP relaxation directly here. We will have two types of variables: x_i corresponds to the choices of a set S_i, and y_j corresponds to covering the element u_j.

Maximize

$$\sum_{j=1}^{n} y_j \qquad \text{(maximize the number of items covered)}$$

Subject to

$$\sum_{i=1}^{m} x_i = k \qquad \text{(choose } k \text{ sets)}$$

$$y_j \leq \sum_{S_i : u_j \in S_i} x_i \qquad \text{for all } u_j \in U$$

$$0 \leq x_i \leq 1 \qquad \text{for } i = 1..m$$

$$0 \leq y_j \leq 1 \qquad \text{for } j = 1...n$$

In exercise, you can demonstrate that this is indeed a relaxation of MAX-COVERAGE and so that $OPT \leq OPT_{LP}$. Again, this is because any feasible solution to MAX-COVERAGE gives the LP a feasible solution, so the LP can only do better.

Let $\boldsymbol{x}^* = (x_1^*, x_2^*, \ldots, x_m^*)$ the values of $\boldsymbol{x}$ of an optimal solution to this LP, and let $\boldsymbol{y} = (y_1^*, \ldots, y_n^*)$ the values of $\boldsymbol{y}$. The sum of these x_i^* must be k and represent fractions of presence of sets of S. We will interpret these x_i^*

as probabilities. In fact, the sum of the x_i^* should be 1 to do so, whereas it is currently k. We will therefore interpret

$$(x_1^*/k, x_2^*/k, \ldots, x_m^*/k)$$

as a probability distribution. We will draw k sets at random according to this probability, with replacement. So we may draw the same set several times — in that case, too bad, we return less than k different sets and cover less than we could have.

fonction *maxcoverLP(U, S, k)*
  Build the relaxation LP L above
  Obtain an optimal solution $\boldsymbol{x}^*$ for L
  $S' = \emptyset$
  while $|S'| < k$ **do**
    Choose an index i at random, where each i is chosen with probability x_i^*/k
    Add S_i to S'
  end
  return S'

In practice, one would of course want to avoid choosing the same set twice. We could add another set when this happens, but our calculations would be much more complex.

To analyze the precision of this algorithm, we must look at the probability of covering an element $u_j \in U$. The LP already gives us a "coverage fraction" of u_j with the value of y_j^*. The probability of covering u_j will therefore be put into perspective with y_j^*. Since the latter value contributes to OPT_{LP}, we will be able to compare ourselves to the optimal.

Lemma 11. *For any $u_j \in U$, the probability that S' covers u_j is at least $y_j^* \cdot (1 - 1/e)$, where $e \simeq 2.71828$ is Euler's constant and S' is the set returned by maxCoverageLP.*

Proof. If only one set is chosen, the probability of covering u_j is the sum of the probabilities of the sets containing u_j, which is $\sum_{S_i : u_j \in S_i} x_i^*/k = \frac{1}{k} \cdot \sum_{S_i : u_j \in S_i} x_i^*$. It turns out that the LP specifies a constraint with this sum. That is, we know that $\sum_{S_i : u_j \in S_i} x_i^* \geq y_j^*$, and thus the probability of

covering u_j by choosing one set is at least

$$\frac{1}{k} \cdot \sum_{S_i : u_j \in S_i} x_i^* \geq \frac{y_j^*}{k}$$

So we have a probability $1 - \frac{y_j^*}{k}$ of *not* covering u_j with a set.

Since each set is chosen independently of the previous choices, the probability of never covering u_j by choosing k sets is therefore $\left(1 - \frac{y_j^*}{k}\right)^k$. And the probability of covering u_j is $1 - \left(1 - \frac{y_j^*}{k}\right)^k$. It is possible to show that

$$1 - \left(1 - \frac{y_j^*}{k}\right)^k \geq (1 - 1/e) y_j^*$$

but we leave it in exercise. One way to see it is that the function on the left, with respect to y_j^*, is convex and greater than or equal to the function on the right at the points $y_j^* \in \{0, 1\}$, while the function on the right is linear. This demonstrates the lemma. □

Theorem 19. *maxcoverLP is a $(1 - 1/e)$-probabilistic approximation.*

Proof. Let I_j be an indicator variable such that I_j =1 if the algorithm covers u_j, and $I_j = 0$ otherwise. We have

$$\begin{aligned} \mathbb{E}[APP] = \mathbb{E}[\sum_{j=1}^{n} I_j] = \sum_{j=1}^{n} \mathbb{E}[I_j] &\geq \sum_{j=1}^{n} (1 - 1/e) y_j^* \\ &= (1 - 1/e) \sum_{j=1}^{n} y_j^* \\ &= (1 - 1/e) \cdot OPT_{LP} \end{aligned}$$

□

Part II

Algorithms with parameterized complexity

Chapter 8

Parameterized complexity

In this second part, we are going to study *exact* algorithms, that is to say which guarantee to return us an optimal solution. Since we are studying NP-complete problems, we cannot expect these algorithms to run in polynomial time. Instead, we must move towards exponential algorithms.

This is a major paradigm shift from approximation algorithms. It is no longer necessary to compare the solution of the algorithm with OPT. Also, our algorithms are no longer required to be in polynomial time.

However, we wish to *limit* the magnitude of the exponential complexity. We will assume that we are dealing with data on which a certain k parameter is small, independently of n. In this case, an algorithm in time $O(2^k \cdot n)$ will be much more efficient than an algorithm in time $O(2^n)$. If k remains small, $O(2^k \cdot n)$ remains practical for n going into the tens of thousands, while $O(2^n)$ will never really go further than $n = 100$. For example, one may wonder if there is a VERTEX-COVER of size at most k, and we can assume that no matter how many n vertices there are, we are only interested in whether we could block entry at k people or less.

The spirit of parameterized complexity is therefore to isolate the exponential part of an algorithm on a parameter k, while n must remain polynomial. In fact, we will allow a complexity $f(k)$ for *any* function f with respect to the part affecting k. The important thing is that k is not treated as a constant, so the polynomial that affects n must remain independent of k. Therefore, a complexity $O(2^k \cdot n^{100})$ is acceptable, but *not* a complexity $O(n^k)$.

Brute-force algorithms try all possible solutions and often result in an $O(2^n)$ algorithm. Parameterized compelxity can be seen as "intelligent brute-force", in which an attempt is made to limit the enumeration of pos-

sibilities according to k and not according to n.

8.1 Defining an FPT algorithm

Let P be an algorithmic problem. Unlike approximation algorithms, there is no fundamental distinction between a minimization or maximization problem (with a few exceptions). Let I be an instance of P and let k be a numerical value. We will say that (I, k) is an instance of P parameterized by k.

Definition 2. *A P problem is fixed parameter tractable (FPT) if there is an algorithm that, for any (I, k) parameterized instance of P, returns an optimal solution in time $O(f(k) \cdot n^c)$, where f is an arbitrary function, n is the size of the I instance, and c is a constant independent of k.*

We will say that P is FPT in parameter k.

Note that the interpretation of k is not given. It is up to us to choose what this k represents. Let's emphasize again that the exponent that affects n must be independent of k. The f function, on the other hand, is arbitrary, which sometimes allows complexities disconnected from practice. For example, an algorithm in time

$$O(2^{2^{2^{2^k}}} \cdot n^{1000})$$

is eligible. We will avoid these cases most of the time, and our algorithms will instead have a practical form such as $O(2^k \dot{n})$.

8.2 The canonical example: VERTEX-COVER

There is a very simple $O(2^k \cdot n)$ algorithm for VERTEX-COVER, where k is the number of vertices of an optimal solution.

Let's first analyze the naive brute-force algorithm.

```
fonction vertexCoverBrute(G = (V, E))
    X = V
    for each subset S ⊆ V do
        if S covers all the edges and |S| < |X| then
            X = S
        end
    end
    return X
```

There are 2^n possible subsets, and checking if a subset covers the edges takes a time $O(n+m)$, where $m = |E|$. The complexity of this algorithm is $O(2^n \cdot (n+m))$.

Let's study the parameterized version of this problem.

VERTEX-COVER
Input : graph $G = (V, E)$
Parameter : k, the maximum desired size of a solution
Output : a set $X \subseteq V$ covering the edges such that $|X| \leq k$, or *null* if such a set does not exist

Note that in reality, we are not looking for the optimal solution. We only want to know if a VERTEX-COVER of size k or less exists. This contrasts with the notion of exact algorithm stated earlier. The fact is that if a parameterized VERTEX-COVER algorithm exists, one can try all k in ascending order to find the optimal one.

Here is a **bad** attempt to get an FPT algorithm for this problem. We try to be more brilliant than the naive algorithm and we think that we can use the k parameter to limit our brute force search. This doesn't work.

```
fonction vertexCoverSemiBrute(G = (V, E), k)
    for i = 1...k do
        for each subset S ⊆ V size i do
            if S covers all the edges and |S| < |X| then
                return X
            end
        end
    end
    return null
```

This algorithm is correct, but what is its complexity? There are $\binom{n}{i}$ possible subsets of i vertices, and $\binom{n}{i} \in \Omega(n^i)$ (reminder: $\Omega(f(n))$ can be seen as the inverse of O and means "grows at least as fast as $f(n)$"). In the worst case, the algorithm will try everything and the number of sets tested will be at least

$$\sum_{i=1}^{k} \binom{n}{i} \geq \binom{n}{k} \in \Omega(n^k)$$

The complexity of this algorithm is therefore $\Omega(n^k)$. Since the exponent of n depends on k, it is not parameterized complexity.

Here is an algorithm that works. We observe that for each uv edge, we must include u in our coverage, or v. So we will choose an arbitrary uv edge and connect to both possibilities. If we decide to add u, its incident edges are covered and we can remove them from the graph (same thing with v). After choosing to add u or v, we get one less choice of our vertex cover and we can decrement k. We stop when we have found a cover, or when k has become 0 and we are no longer allowed to add anything. We will pass as a parameter the X set we have built so far (at the initial call, $X = \emptyset$).

```
fonction vertexCoverFPT(G = (V, E), k, X)
    if X covers all edges then
        return X
    end
    if k = 0 then
        return null
    end

    Let uv ∈ E be chosen arbitrarily; Or G_u obtained from G by
     removing u and its edges;
     X_u = vertexCoverFPT(G_u, k − 1, X ∪ {u})
    Let G_v be obtained from G by removing v and its edges
    X_v = vertexCoverFPT(G_v, k − 1, X ∪ {v})

    if X_u ≠ null then
        return X_u
    else if X_v ≠ null then
        return X_v
    else
        return null
    end
```

This algorithm is correct because it tests all the possibilities for the uv edge, which must be covered no matter what we do. The algorithm performs what is called a bounded branching, a very common technique in parameterized complexity. To evaluate the complexity of this algorithm, just note that it builds a recursion tree where each vertex represents a call, with the root representing the initial call. Each vertex has 2 children (2 recursive calls) and the height of the tree is bounded by k. So we have a binary tree of height k, and we know (don't we?) that such a tree has $O(2^k)$ vertices. Since the time to process a call is $O(n)$, the complexity is $O(2^k \cdot n)$.

8.3 Another example: MAX-CLIQUE

Let $G = (V, E)$ a graph. Recall that a clique is a set of vertices X such that for any distinct $u, v \in X$, $uv \in E$. We can try to parameterize the maximum clique problem as follows.

MAX-CLIQUE (parameter "clique size")
Input : graph $G = (V, E)$
Parameter : k, the desired size of a clique
Output : a $X \subseteq V$ clique of size k, if it exists, or *null* otherwise

The naive algorithm would test all subsets of size k and would take at least $O(n^k)$ time. It is therefore not a viable option.

It turns out that despite years of research, no one has found an FPT algorithm for MAX-CLIQUE parameterized by the size of the click. In fact, it is believed that it is impossible that there is a $O(f(k) \cdot n^c)$ algorithm for this problem. MAX-CLIQUE is known as $W[1]$-complete. There is a formal definition of what this means, but it is beyond the scope of these notes. Let's say that $W[1]$-complete is the NP-complete analog for FPT problems, and that a $W[1]$-complete problem means that the problem is probably not FPT.

However, another parameterization can be attempted. Let's say that the G degree is not too large (the degree is the maximum number of neighbors of a node). Therefore, each vertex has at most k neighbors.

MAX-CLIQUE (parameter "maximum degree")
Input : graph $G = (V, E)$
Parameter : k, the maximum degree of a G vertex
Output : a $X \subseteq V$ clique of maximum size

Note some subtleties here. Since the k parameter has no (direct) link with the maximum clique, we require that the return value be the maximum clique. One must therefore manage to isolate the complexity from the degree of the graph. It becomes possible to make an "intelligent brute force". Just consider each vertex and see if it and its neighborhood form a fairly large clique.

```
fonction maxCliqueDegreFPT(G = (V, E) of degree at most k)
    X = ∅
    for v ∈ V do
        for each subset S ⊆ N(v) do
            if S ∪ {v} is a clique and |S| + 1 > |X| then
                X = S ∪ {v}
            end
        end
    end
    return X
```

When developing such an algorithm, one must of course demonstrate that it is correct, and that its complexity is FPT.

Theorem 20. *The algorithm maxCliqueDegreFPT returns a maximum size clique and runs in time $O(k^2 2^k \cdot n)$. The MAX-CLIQUE problem is therefore FPT in the parameter "degree of the graph".*

Proof. It is clear that the algorithm is correct. The maximum clique is formed by a vertex $v \in V$ and a subset of its neighborhood and the algorithm tests all possibilities.

For complexity, the main loop iterates on n vertices. For each $v \in V$, the subsets of $N(v)$ are listed. Since $|N(v)| \leq k$, there are at most 2^k such subsets. Moreover, each subset is of size at most k, and checking that each pair of $N(v)$ vertices has an edge can be made in time $O(\binom{k}{2}) = O(k^2)$. So, the processing of a $v \in V$ is done in time $O(k^2 2^k)$ and there are n vertices to process. The complexity is therefore $O(k^2 2^k \cdot n)$. □

Can VERTEX-COVER be parameterized by the degree of G?

The same ideas of parameterization by the G degree fail on VERTEX-COVER. In fact, this is probably not possible. VERTEX-COVER is therefore in the opposite situation to MAX-CLIQUE.

The reason why it is not possible is that VERTEX-COVER is NP-complete even when the input graph has degree 3. If VERTEX-COVER was FPT in the degree of G, we would have an algorithm in time $O(f(k) \cdot n^c)$ for the problem, where k is the maximum degree. This would imply that if $k = 3$, we would have an algorithm $O(f(3) \cdot n^c)$. But here, $f(3)$ is a constant regardless of f (for example, if $f(k) = 2^k$, then $2^3 = 8$ is a constant). So

with $k = 3$, we would have an algorithm in time $O(n^c)$, which is in polynomial time, for VERTEX-COVER with degree 3. We would have solved a complete NP-complete problem in polynomial time! This means that if VERTEX-COVER was FPT in the degree parameter, we would have P = NP. We would need another course to appreciate the depth of this statement. Let's just say that no one has proven that this is not possible, but it is very unlikely to happen.

The usual way to formulate this kind of result is to use the contraposition, as follows.

Theorem 21. *If $P \neq NP$, then VERTEX-COVER parameterized by the degree of the graph is not FPT.*

Chapter 9

Branching algorithms

The idea of a branching algorithm is to identify a limited number of possible cases that could form an optimal solution, and then to recursively branch into each possibility. In order to obtain an FPT algorithm, two conditions must generally be satisfied:

- the number of cases must be bounded by a function $f(k)$;
- each case must reduce the value of k.

If these two conditions are met, the recursion tree will be such that each node has $f(k)$ children and such that the depth is bounded by $p(k)$, where f and p are any function. Consequently, the tree will have at most $f(k)^{p(k)}$ vertices, giving an FPT algorithm (assuming that each recursion is in polynomial time).

We have already seen an example of a branching algorithm with VERTEX-COVER. Each node of the recursion tree had $f(k) = 2$ children and the depth was bounded by $p(k) = k$, giving an algorithm in time $O(2^k n^c)$.

Many problems can be solved with branching algorithms. The analysis of the recursion tree sometimes becomes complex, and we will develop tools to do so. But first, let's see some basic examples.

9.1 3-HITTING SET

In the 3-HITTING-SET problem, we receive sets $S_1, S_2, \ldots, S_m$ each having 3 elements, on universe $U = \{u_1, \ldots, u_n\}$. One must choose a minimum number of elements of U in order to have an intersection with each S_i. We can imagine a situation in which students each offer 3 days of availability for

an appointment with a professor, and the professor must choose a minimum number of days so that each student has an availability among these days. We will parameterize by the number of items chosen.

3-HITTING-SET
Input : sets $S = \{S_1, \ldots, S_m\}$ each size 3 on universe $U = \{u_1, \ldots, u_n\}$
Parameter : k, the size of a solution
Output : subset $X \subseteq U$ such that $X \cap S_i \neq \emptyset$ for all $i = 1...m$ and such that $|X| \leq k$, or null if non-existent

The idea is about the same as VERTEX-COVER (in fact, this problem is a generalization of VERTEX-COVER). For each $S_i = \{a, b, c\}$, we have to include a, b or c. We branch into the three possibilities and stop when we have reached k elements. When we branch into one of the cases, we remove all the S_i that become covered.

fonction *3hitset*(S, U, k, X)
  if $k < 0$ **then**
    return *null*
  end
  if $S = \emptyset$ **then**
    return X
  end
  Choose $S_i \in S$ arbitrarily
  Let a, b, c be the elements of S_i
  $X_a = 3hitsandFPT(S \setminus \{S_j : a \in S_j\}, U, k-1, X \cup \{a\})$
  $X_b = 3hitsetFPT(S \setminus \{S_j : b \in S_j\}, U, k-1, X \cup \{b\})$
  $X_c = 3hitsetFPT(S \setminus \{S_j : c \in S_j\}, U, k-1, X \cup \{c\})$
  if *one of* X_a, X_b *or* X_c *is not null* **then**
    Return the one from X_a, X_b, X_c that is not *null*
  else
    return *null*
  end

The algorithm is correct because it tests every way to have an intersection with every S_i.

The complexity of this algorithm can be quickly evaluated. It takes time $O(m)$ for each call to find the S_j to remove. The algorithm creates a recursion tree where each node has 3 children, and the depth is bounded by k. So there are $O(3^k)$ nodes. The complexity is $O(3^k \cdot m)$.

9.2 CLUSTER-EDITING

In CLUSTER-EDITING, we have a graph and we want to modify (add/remove) a minimum number of edges so that each connected component of G is a clique, which form what are called clusters.

CLUSTER-EDITING
Input : graph $G = (V, E)$
Parameter : k, the number of edges to modify
Output : a list of edges E^+ to add and a list of edges E^- to remove such that $|E^+| + |E^-| \leq k$ and such that $G' = (V, (E \cup E^+) \setminus E^-)$ has only cliques as connected components.

We will develop a recursion tree with $O(3^k)$ nodes with the following property (to be proved in exercise).

Theorem 22. *Each connected component of a graph G is a clique if and only if for any $u, v, w \in V$, $uv \in E$ and $vw \in E$ implies that $uw \in E$.*

Note that if $uv \in E, vw \in E$ but $uw \not\in E$, the vertices u, v, w form a path with three vertices, which is called a P_3. CLUSTER-EDITING is thus equivalent to modifying a minimum of edges so that the graph does not have a P_3.

We will find conflicting u, v, w (which form a P_3), and branch into ways of fixing the problem. We note that if $uv \in E$ and $vw \in E$ but $uw \notin E$, we have three ways to satisfy the theorem:
– remove uv
– remove vw
– add uw.

Each case modifies one edge and reduces k by 1.

fonction *clusterEditing*$(G = (V, E), k, E^+, E^-)$
  if $k < 0$ **then**
    return *null*
  end
  if G *has no* P_3 **then**
    return (E^+, E^-)
  end

  Find u, v, w that form a P_3
  Let G_1 be obtained from G by removing uv
  $(E_1^+, E_1^-) = clusterEditing(G_1, k-1, E^+, E^- \cup \{uv\})$
  Let G_2 be obtained from G by removing vw
  $(E_2^+, E_2^-) = clusterEditing(G_2, k-1, E^+, E^- \cup \{vw\})$
  Let G_2 be obtained from G by adding uw
  $(E_3^+, E_3^-) = clusterEditing(G_3, k-1, E^+ \cup \{uw\}, E^-)$

  if *one of the calls did not return null* **then**
    Return the solution not *null*
  else
    return *null*
  end

Each call takes time $O(n^3)$ because we have to find u, v and w (and it is not trivial to do better than watching each triplet). This algorithm creates a recursion tree where each node has three children, and the depth is k or less. The complexity is therefore $O(3^k n^3)$.

9.3 More Intelligent Branching

If we dig a little deeper, we can often refine our branching tree to reduce the number of node children or the depth (ironic that digging reduces the depth!). The price to pay is that we have to work a little harder and analyze more complex recurrences. Let's go back to VERTEX-COVER.

In the algorithm we saw earlier, we were branching into two cases on an uv edge:
– include u;
– include v.

An alternative way to see this is to branch in two cases:

– include u;
– do not include u.

But if we decide that u is not in the solution, it is not able to cover its edges. Let's say that u's neighbors are $v_1, \ldots, v_l$. If we don't include u, we have to include v_1 to cover uv_1, and we have to include v_2 to cover uv_2, and so on. So, choosing not to include u forces us to include $deg(u)$ vertices. If $deg(u) \geq 2$, this will allow us to reduce k by more than 1 and thus limit the depth of the recursion tree.

Let's write the algorithm first. We want to branch on u such that $deg(u) \geq 2$. If such a u does not exist, all vertices have a degree of 0 or 1 and it is trivial to decide if there is a solution of size k or less (exercise!).

```
fonction vc2(G = (V, E), k, X)
    if k < 0 then
        return null
    end
    if G has no edge then
        return X
    end
    if G has only vertices of degree 0 or 1 then
        Find the optimal vertex cover X'
        If |X'| ≤ k, return X' ∪ X, otherwise return null
    end

    Let u be a vertex of degree at least 2
    Obtain G_u by removing u and its edges from G
    X_1 = vc2(G_u, k − 1, X ∪ {u})

    Obtain G* by removing {u} ∪ N(u) and their edges
    X_2 = vc2(G*, k − |N(u)|, X ∪ N(u))

    if one of X_1 or X_2 is not null then
        return the non-null vertex cover
    else
        return null
    end
```

This algorithm is correct: when we consider a u, when we include it in

our solution, its edges are covered. On the other hand, if we don't include u, we must include all its neighbors.

For complexity, we will focus only on the number of nodes in the branch tree. Let $t(k)$ be the number of such nodes on input k. When recursive calls are made by the algorithm, we have

$$t(k) = t(k-1) + t(k - |N(u)|)$$

The larger $N(u)$ is, the smaller $t(k)$ will be. Since we have $|N(u)| \geq 2$, we can assume that in the worst case,

$$t(k) = t(k-1) + t(k-2)$$

Even without knowing the initial conditions of this recurrence, it is possible to deduce that

$$t(k) \in O(1.618^k)$$

In fact, $t(k)$ grows exactly like the Fibonacci function.

The above algorithm therefore takes time $O(1.618^k \cdot (n+m))$, a significant improvement over the 2^k we had earlier.

9.4 How to solve recurrences?

The simple answer to this question is to use software. Few algorithm designers analyze recurrences manually, except when they are too complicated for a program. But it is important to know how to use the output of software.

The most frequent recurrences are called *homogeneous linear recurrence*. They have the form

$$t(k) = a_1 t(k-1) + a_2 t(k-2) + \ldots + a_d t(k-d)$$

where d is a constant, and the a_i are also constants.

The idea is that these recurrences are $O(c^k)$ for a constant c that needs to be determined. To find this c, we will simply assume that $t(k) = c^k$ and solve. This is not always true because constants are ignored, but in terms of O, everything remains valid.

So, assuming that $t(k) = c^k$, we have

$$\begin{aligned} t(k) &= a_1 t(k-1) + a_2 t(k-2) + \ldots a_d t(k-d) \\ c^k &= a_1 c^{k-1} + a_2 c^{k-2} + \cdots a_d c^{k-d} \end{aligned}$$

By putting everything on the same side, we have

$$c^k - a_1 c^{k-1} - a_2 c^{k-2} - \cdots a_d c^{k-d} = 0$$

We can divide everything by c^{k-d} and get

$$c^d - a_1 c^{d-1} - a_2 c^{d-2} - \cdots a_d c^0 = 0$$

This is a polynomial of degree d with c as variable. It is called the characteristic polynomial.

Since the degree is d, there are d possible roots (i.e. d values of c such that the polynomial gives 0). Today's software can output all roots. Some of them are sometimes complex (in the sense of complex numbers), but it turns out that we can take the largest real root to get c. That is, if the real roots of the polynomial are $r_1, r_2, \ldots, r_l$, the number of nodes of our tree is $O(\max\{r_1, \ldots, r_l\}^k)$.

For example, the VERTEX-COVER case above gave

$$t(k) = t(k-1) + t(k-2)$$

which becomes

$$\begin{aligned} c^k &= c^{k-1} + c^{k-2} \\ c^k - c^{k-1} - c^{k-2} &= 0 \\ c^2 - c - 1 &= 0 \end{aligned}$$

Wolfram tells us that the roots are about 1.618 and -0.618. We take the larger $c = 1.618$ and the number of nodes in the recursion tree is $O(1.618^k)$.

The recipe for linear homogeneous recurrences is therefore as follows:

1. Obtain $t(k)$, the recurrence for the number of recursion tree nodes;
2. Assume that $t(k) = c^k$;
3. Write the characteristic polynomial;
4. Find the roots of the polynomial;
5. Let r be the largest real root;
6. The recursion tree has $O(r^k)$ nodes.

9.5 An improved 3-HITTING-SET

We are going to improve our 3-HITTING-SET algorithm stated above by digging a bit more into our branching cases. The idea is the following: let's take two sets S_i and S_j such that their intersection $S_i \cap S_j$ is maximum. Suppose that $S_i = \{x, y, z\}$ and $S_j = \{x, y, a\}$. We imagine what we need to include in our solution X to cover S_i and S_j. There are three possibilities: include x, include y, or include a and b. The first two cases reduce k by 1, but the last case reduces k by 2. We get the recurrence

$$t(k) = 2t(k-1) + t(k-2)$$

It is possible that $S_i = \{x, y, z\}$ and $S_j = \{x, a, b\}$. One can include x, or include one of y, z and one of a, b. The possible choices are therefore to include $\{x\}$, $\{y, a\}, \{y, b\}, \{z, a\}$, or $\{z, b\}$. The first decreases k by 1, the other four decrease k by 2. This case leads to the recurrence

$$t(k) = t(k-1) + 4t(k-2)$$

It is also possible that all S_i and S_j have no intersection. This case must be handled separately, but this is simple. If the S_i do not share an element, at least one element from each S_i must be included. So, if $|S_i| > k$, we return *null*, and otherwise we return one element per S_i. The pseudo-code is described below. Note that instead of making the recursive calls explicit, we just list the cases we are branching into. We also do not specify the return behavior (return the non *null* solution if there is one, or *null* if not). This is common in FPT — this allows to shorten the presentation of standard branching calls.

fonction *3hitsandImproved*(S, U, k, X)
  if $k < 0$ **then**
    return *null*
  end
  if S *is empty* **then**
    return X
  end
  if $S_i \cap S_j = \emptyset$ *for all distinct* $S_i, S_j \in S$ **then**
    if $|S| > k$ **then**
      return *null*
    else
      Add to X an item from each S_i
      return X
    end
  end
  Let S_i, S_j be distinct such that $|S_i \cap S_j|$ is maximum
  if $|S_i \setminus S_j| = 2$ **then**
    Let $S_i = \{x, y, z\}$ and $S_j = \{x, y, a\}$
    Branch recursively into the following cases:
    – Add x to X, reduce k by 1
    – Add y to X, reduce k by 1
    – Add z and a to X, reduce k by 2
  else if $|S_i \cap S_j| = 1$ **then**
    Let $S_i = \{x, y, z\}$ and $S_j = \{x, a, b\}$
    Branch recursively into the following cases:
    – Add x to X, reduce k by 1
    – Add y and a to X, reduce k by 2
    – Add y and b to X, reduce k by 2
    – Add z and a to X, reduce k by 2
    – Add z and b to X, reduce k by 2
  end

We are not going to dwell on the fact that this algorithm is correct. Let's focus on complexity. We have two possible situations and therefore two recurrences. Which one do we take? As we always do in algorithms: we take the worst case! One of the two cases of the algorithm should be worse than the other. We will therefore assume that it is always the worst case that occurs.

In the case where $|S_i \cap S_j| = 2$, we had

$$t(k) = 2t(k-1) + t(k-2)$$

with the characteristic polynomial

$$c^2 - 2c - 1 = 0$$

and by finding the zeros, we have $c \simeq 2.4143$.
in the case where $|S_i \cap S_j| = 1$, we had

$$t(k) = t(k-1) + 4t(k-2)$$

with the characteristic polynomial

$$c^2 - c - 4 = 0$$

and we have $c \simeq 2.5616$.

So we can assume that in the worst case, the branch shaft a $O(2.5616^k)$ knots. Each call can be implemented in time $O(m)$, and by rounding, the complexity is therefore $O(2.57^k \cdot m)$.

9.6 The consensus sequence

We end this chapter with an example where it is not trivial to limit the number of cases to branch into.

Let S be a sequence of characters. We write $S[i]$ for the character at the position i of S. The Hamming distance $d(S,T)$ between two S and T sequences of the same length is defined as the number of different characters per position. That is,

$$d(S,T) = |\{i : S[i] \neq T[i]\}|$$

CONSENSUS-SEQUENCE
Input : sequences of characters $S_1, S_2, \ldots, S_n$, each of length ℓ
Parameter : distance d
Output : a sequence S such that $d(S, S_i) \leq d$ for any S_i, or *null* if non-existent.

Before thinking about an FPT algorithm, one must analyze the problem

and derive some observations. Note that if all sequences have the same c character at the p position, then the consensus sequence will have the c character at the p position. We can therefore ignore this position and assume that for each p position, there are S_i, S_j such that $S_i[p] \neq S_j[p]$. This is not fundamental, but does show a basic reduction rule.

Another observation is that if we have S_i and S_j too distant, there can be no solution. One way to see this is to see that d satisfies the triangular inequality, and that a consensus sequence must be "in between" sequences.

Lemma 12. *If there is S_i, S_j such that $d(S_i, S_j) > 2d$, then there is no consensus sequence at distance d from both S_i and S_j.*

Proof. Suppose there is a sequence S such that $d(S, S_i) \leq d$ and $d(S, S_j) \leq d$. Let $P = \{p_1, \ldots, p_{2d+1}\}$ be a set of $2d+1$ positions on which S_i and S_j differ. Since S differs from S_i to d position or less, there must be at least $d+1$ position of P where S and S_i are identical. This means that S differs from S_j at these $d+1$ position, a contradiction. □

The branching strategy will be as follows. We will start with the sequence S_1 (this choice is arbitrary). By the above lemma, we know that $d(S_1, S_j) < 2d$, and therefore that each S_j differs from S_1 by at most $2d$ positions. If $d(S_1, S_j) \leq d$ for every S_j, we are finished. Otherwise, let's take S_j such that $d(S_1, S_j) > d$. Let P be the positions where S_1 and S_j differ. If there is a consensus sequence S, at least one of the positions $i \in P$ must be such that $S[i] = S_j[i]$. We know that there are at most $2d$ such positions, so we branch in each of them. This results in $2d$ branches, each giving a sequence S' with a position different from S_1 in 1 place. We repeat with S', but since each change takes us away from S_1, we cannot repeat more than d times. If we reach a point where $d(S', S_j) > 2d$ for a certain S_j, we know that it is impossible to apply d modifications to S' to reach a consensus sequence at a distance at most d from S_j. In this case, we can exit.

The pseudo-code that describes this procedure can be found below. We keep the original d, but we retain the number of modifications still allowed with d_rest. We also keep the current consensus candidate sequence S'. Initially, $d\_rest = d$ and $S' = S_1$.

```
fonction seqConsensus(S_1, ..., S_n, d, d_rest, S')
    if d_rest < 0 then
        return null
    end
    if d(S', S_j) > 2d for a certain S_j then
        return null
    end
    if d(S', S_j) ≤ d for any S_j then
        return S'
    end
    Let S_j such that d(S', S_j) > d
    Let P be the positions where S' and S_j differ
    foreach i ∈ P do
        S'' = S'
        S''[i] = S_j[i]
        S* = seqConsensus(S_1, ..., S_n, d, d_rest − 1, S'')
        if S* ≠ null then
            return S*
        end
    end
    return null
```

It is not trivial to guarantee that this algorithm works, i.e. it always returns a solution if there is one, and *null* if not. The main ideas have been stated above, and we leave it to you to demonstrate this fact. The complexity can be analyzed as follows. We branch in $|P|$ possible cases, and we know that $|P| \leq 2d$. The depth of the tree is bounded by d, so the recursion tree has $O((2d)^d)$ nodes. Each call takes a time $O(n^2)$, most of the time being in distance calculations with S'. The complexity is therefore $O((2d)^d \cdot n^2)$.

Chapter 10

Kernelisation

Kernelization aims at transforming a given instance into another equivalent but smaller instance. The size of the equivalent instance must in fact be $O(f(k))$ for an arbitrary f function. For example, suppose we are given an instance $G = (V, E)$ of VERTEX-COVER with n vertices. Let's say that we manage to transform G into a G' graph such as :

- $|V(G')| \leq 2k$;
- G has a VERTEX-COVER of size k if and only if G' has a VERTEX-COVER of size k.

We can therefore work on the G' graph, because the existence of a solution on G' is equivalent to the existence of a solution on G'. Moreover, since $|V(G')| \leq 2k$, one can make a simple brute force on all subsets of $V(G')$, which would list $O(2^{2k})$ subsets. Assuming that the transormation takes a polynomial time, we thus have an FPT algorithm.

If such a G' exists, it is said to form a *kernel*. The important thing is that a kernel is equivalent to the original instance, and that it is of a size bounded by $f(k)$. Once a kernel is found, we know that our problem is FPT because a brute force on the kernel will take an FPT.

In practice, it is common to combine techniques to speed up algorithms. First we transform our instance into a kernel, then we execute for example a branching algorithm, but on the kernel. Whenever possible, this results in much more efficient algorithms in practice (though not in theory most of the time).

10.1 Defining a kernel

Let P be an algorithmic problem and let (I, k) be a parameterized instance of P. Another instance (I', k') of P is said to be a *kernel* of (I, k) if the following conditions are satisfied:

1. $|I'| \in O(f(k))$ for a function f, where $|I'|$ is the size of the I' instance;
2. $k' \in O(g(k))$ for a function g;
3. (I, k) admits a solution if and only if (I', k') admits a solution;
4. it is possible to transform (I, k) into (I', k') in time $O(|I|^c)$, where c is a constant.

In words, we want to transform I into an instance I' of size bounded by a function of k, allowing the brute force on I' in FPT time. Note that the time required to perform the transformation must not be exponential in k. The spirit of kernelization is to provide a pre-processing procedure that can be followed by an FPT algorithm. Having an exponential time in k to create an equivalent instance would still result in an FPT algorithm, but the instance would not be considered as a kernel.

Note also that one can modify the k' parameter associated with I', as long as k' is also bounded by a k function. Finally, we ask that (I', k') be equivalent to (I, k). A counter-intuitive fact is that the solutions (I, k) and (I', k') may have nothing to do with each other. Kernelization thus allows us to decide whether (I, k) admits a solution or not, but a solution for (I', k') may not be a solution for (I, k). Sometimes it is necessary to apply an inverse transformation to the solution for (I', k'). However, this is not required, because only the equivalence between the existence of solutions is required.

How to find a kernel? With reduction rules!

The construction of the kernel depends on the k parameter. Most of the time, we take our I instance and start an argument of the style

"If it's true that there is a k size solution, then we can't have more than [...] in our instance."

or

"If it's true that there is a solution of k size, then we can eliminate ... from our instance."

and all that remains to be done is to fill in the [...]. Another way of presenting these arguments is also to contrast:

"If we have more than [...] in our instance, then there is no solution of size k and we can return *null*".

The general idea is therefore to assume that there is a solution of size k, and to identify the important properties of our instance. These properties should then allow us to reduce our instance. This often takes the form of a set of **reduction rules**, which are statements like "if I has X property, then delete Y". Of course, we need to show that our reduction rules are valid, i.e. that they preserve equivalence.

All this may sound very abstract, so let's move on to examples. As usual, we'll use VERTEX-COVER to begin.

10.2 Kernelization of VERTEX-COVER

Let $G = (V, E)$ an instance of VERTEX-COVER with parameter k, the size of the solution. We start by asking if some operations are forced when we suppose that there is a cover of k vertices.

If we think about it long enough, we notice the following observation.

Observation 1. *If there is $u \in V$ such that $|N(u)| > k$, then any VERTEX-COVER of size k or less contains u.*

Proof. If a solution X does not contain u, then we must include all the vertices of $N(u)$ to cover the edges incident to u. It is then impossible that $|X| \leq k$ because $|N(u)| > k$ and $N(u) \subseteq X$. □

We can deduce from this a very simple rule, which is completely deterministic and which avoids branching into to several cases.

Rule 1. If there is $u \in V$ such that $|N(u)| > k$, delete u and its edges by G and reduce k by 1. .

This rule reduces the number of vertices of G and also reduces k. Of course, it must be argued that by applying the rule, one obtains an equivalent instance. When this is the case, we say that the rule is *safe*.

Lemma 13. *Rule 1 is safe.*

Proof. Let G be a graph and let G' be the graph obtained after applying rule 1 by deleting u. It must be shown that G has a VERTEX-COVER of size k if and only if G' has a VERTEX-COVER of size $k-1$.

If G has a VERTEX-COVER X of size k, then according to the above observation, we know that $u \in X$. Therefore, $X \setminus \{u\}$ must cover all non-incident edges at u, and therefore $X \setminus \{u\}$ is a VERTEX-COVER of size $k-1$ of G'.

In the other direction, that is X' is a VERTEX-COVER of size $k-1$ of G'. All edges of G are covered by X', except perhaps some incidental to u. Adding u to X' gives a VERTEX-COVER of size k or less. □

To create our kernel, we apply rule 1 until it is no longer possible. This results in a G' graph in which each vertex u satisfies $|N(u)| \leq k'$, where k' is the resulting parameter after applying the rules. To simplify, we will assume that our graph is called G and that its parameter is k, and that rule 1 is no longer applicable.

Knowing that each $u \in V(G)$ has at most k neighbors, we observe that if we include u in X, then u can cover only k edges. This leads to a second observation.

Observation 2. *Suppose Rule 1 does not apply to G. Then if $|E(G)| > k^2$, there is no solution of size k.*

Proof. If rule 1 does not apply, then each u in a X solution covers at most k edges. But if $|E(G)| > k^2$, k vertices will not be enough to cover all edges, because together they cover a maximum of k^2 edges. □

We can therefore assume that $|E(G)| \leq k^2$. What about the number of vertices? There is no limit, because it is possible that G contains many isolated vertices. It is not difficult to see that we can derive another sound rule.

Rule 2. If G contains $u \in V(G)$ such that $|N(u)| = 0$, then remove u from G.

Knowing that G has at most k^2 edges and no isolated vertex, we can limit the number of vertices. It can be show that the maximum number of vertices with k^2 edges is reached if each vertex has only one neighbor. This happens when the graph is in fact a matching, in which case it has $2k^2$ vertices. We deduce the following result.

Theorem 23. *VERTEX-COVER admits a kernel with at most $2k^2$ vertices and at most k^2 edges.*

Note that the X set is not used. However, it contains all the vertices that should be part of a cover and are no longer in the kernel. The purpose of X is to rebuild a concrete solution. Once a X' solution for the kernel has been found, we know that $X' \cup X$ is a solution for the original G. The X is therefore optional.

10.3 A trivial kernel for MAX-3-SAT

Recall the MAX-3-SAT problem, where the goal is to maximize the number of satisfied clauses of size 3.

MAX-3-SAT
Input : clauses $C_1, \ldots, C_m$ with three variables each, on variables $x_1, \ldots, x_n$
Parameter : k, the number of clauses satisfied
Output : an assignment of x_i such that at least k clauses are satisfied, or *null* if non-existent

We remember the probabilistic algorithm for MAX-3-SAT that returned an assignment that, in expectation, satisfied at least $7m/8$ clauses. Our derandomization procedure such an assignment in a deterministic way.

We can use this for the parameterized version of MAX-3-SAT. If $k \leq 7m/8$, we automatically know that an assignment exists and we know how to find one. If $k > 7m/8$, then $m < 8k/7$ and the number of clauses is bounded by a k function. We still have to make sure that the number of variables is also bounded. This is simple: since each clause has 3 literals, we know that the number of variables is at most $3m < 24k/7$ (we can eliminate useless variables). So we deduce an almost trivial kernel from the following algorithm:

```
fonction max3satKernel(C_1, ..., C_m, k)
    if k ≤ 7m/8 then
        Return the result of the malfunction for MAX-3-SAT
    else
        Eliminate variables that do not appear in any clause
        return C_1, ..., C_m
    end
```

The discussion preceding the algorithm gives us :

Theorem 24. *The max3satKernel algorithm gives a kernel for MAX-3-SAT with at most* $8k/7$ *clauses and at most* $24k/7$ *variables.*

10.4 A kernel for MAX-SAT

Let's now consider the general version of MAX-SAT, where the number of variables per clause is arbitrary. We know that by choosing a random assignment, each clause is satisfied with probability at least $1/2$ (the worst case being a clause with only one variable). We can therefore design a probabilistic algorithm that returns an assignment that satisfies $m/2$ clauses in expectation. This algorithm can be derandomized to return such an assignment deterministically.

We can thus assume that if $k \leq m/2$, we always return an assignment satisfying at least k clauses. This can be expressed with a rule.

Rule 1. If $k \leq m/2$, return an assignment that satisfies $m/2$ clauses (with a derandomization algorithm, for example).

Now assume that $k > m/2$. So we have $m < 2k$. Now we have to limit the number of variables.

If each clause had say k variables, we could limit the number of useful variables by $2k \cdot k$. On the other hand, some clauses may have many variables. Let C^p be clauses with k variables or less, and C^g be clauses with more than k variables (p for "petit" and g for "gros"). We would like to get rid of C^g.

If $|C^g| > k$, it is not difficult to see that we can satisfy k clauses of C^g. Just choose k clauses of C^g and choose a different variable for each $C_i \in C^g$ (this is possible because they all have more than k variables). We assign each chosen variable to satisfy its corresponding clause, and voilà!

Rule 2. If $|C^g| > k$, return an assignment that satisfies each clause of C^g (for example by choosing one variable per clause).

So we can assume that $|C^g| < k$. Note that it is possible to satisfy all $|C^g|$ clauses of C^g with $|C^g|$ variables (as above, we choose one variable per clause). Could we simply eliminate the "big" clauses by assuming that we are going to satisfy them with $|C^g|$ variables? As it turns out, yes.

Rule 3. Remove clauses C^g and reduce k by $|C^g|$

The idea of rule 3 is that if we can satisfy $k - |C^g|$ "small" clauses, we need at most $k - |C^g|$ variables to do so. The $|C^g|$ other variables can be used to satisfy C^g. This rule is non-trivial, and we still need to demonstrate that it works.

Lemma 14. *If rules 1 and 2 have been applied, then rule 3 is safe. That is, one can satisfy k clauses of $C_1, \ldots, C_m$ if and only if one can satisfy $k - |C^g|$ clauses of C^p.*

Proof. ($\Rightarrow$) suppose that one can satisfy k clauses among $C_1, \ldots, C_m$. There must be $k - |C^g|$ clauses among C^p that are satisfied.

($\Leftarrow$) Let us suppose that we can satisfy $l = k - |C^g|$ clauses among C^p. Let $C_1, \ldots, C_l$ be these satisfied clauses. Note that it is possible to satisfy them by assigning only l variables (because only one variable is enough to satisfy a clause). It is possible to retrieve l such variables if we already have an assignment, but this is not necessary for the purposes of the lemma. If we know these $l = k - |C^g|$ variables used to satisfy $C_1, \ldots, C_l$, we can then use $|C^g|$ additional variables to satisfy all C^g clauses. □

We deduce a kernel of quadratic size.

Theorem 25. *Rules 1,2 and 3 lead to a kernel for MAX-SAT with $O(k)$ clauses and $O(k^2)$ variables.*

Proof. We assume that $k > m/2$, otherwise we return the result of an approximation algorithm (rule 1). So $m < 2k \in O(k)$. Then, if rule 2 applies, we solve the problem trivially. If not, we apply rule 3. Thus we have a set of C^p clauses of size at most $2k$, and each clause has at most k variables. The number of variables involved in total is therefore $O(k^2)$. □

10.5 A kernel for EDGE-CLIQUE-COVER

We will see that cores can sometimes have an exponential size. In the EDGE-CLIQUE-COVER problem, we want to cover all the edges with cliques.

EDGE-CLIQUE-COVER
Input : a graph $G = (V, E)$.
Parameter : k, the number of cliques
Output : a set of k cliques $C_1, \ldots, C_k$ such as for any $uv \in E$, there is C_i such that $u \in C_i$ and $v \in C_i$.

Note that there could be less than k cliques covering each edge. In this case, one can return exactly k cliques by repeating the same click several times.

Let's start with a very simple rule that is easily demonstrable as sound.

Rule 1. If there is $u \in V(G)$ such that $|N(u)| = 0$, then remove u from G.

Another simple rule is that if a connected component is a clique, we can't do better than including that clique to cover its edges.

Rule 2. If there is a connected component C of G such that C is a clique, then include C in our solution and decrement k by 1.

Two vertices u and v are *twins* if $N(u) \cup \{u\} = N(v) \cup \{v\}$. In particular, twins must be neighbors. For all intents and purposes, twins are "identical" and we can keep only one.

Rule 3. If rule 2 does not apply to G and G contains two twins u and v, then remove u from G.

To see that this rule is safe, we see that we can always put u and v in the same cliques. The formal proof is left in exercise. **As observed in class**, we note that it is necessary to apply rule 2 before rule 3. Otherwise, with a connected component that is a clique, we would eliminate all its twins and we would end up with a single vertex. The latter would be eliminated and we would never have counted the clique.

Perhaps surprisingly, that's all it takes to have a kernel.

Theorem 26. *Let G' be the graph obtained from G after applying rules 1,*

2 and 3 until it is no longer possible. Assume that G' admits an edge clique cover of size at most k. Then G' has at most 2^k vertices.

Proof. Let $C_1, \ldots, C_k$ be a set of k cliques that cover each edge of G'. Let $u \in V(G')$. We associate to u a vector b_u of k bits, where the i bit of b_u is 1 if $u \in C_i$, and 0 otherwise. Note that there are 2^k bit vectors possible. We want to show that all vertices have a different bit vector. Let $u, v \in V(G')$ be two distinct vertices and suppose that $b_u = b_v$. If b_u (or b_v) has only 0, then u has no neighbors and rule 1 should have been applied. Otherwise, the fact that $b_u = b_v$ implies that u and v are neighbors because they are in a common clique. Also, any neighbor of u appears in one of the cliques and so does v. Moreover, these neighbors appear in the same cliques. It follows that u and v are twins and one of them should have been eliminated by rule 3, a contradiction.

We deduce that each vertex of G' has a different vector of bits, so there are at most 2^k vertices. □

A corollary of the previous theorem is that if there are more than 2^k vertices in $V(G')$, we can immediately return *null*. Concretely, the kernelization algorithm is as follows.

fonction *eccKernel*$(G = (V, E), k)$
  if G *has a vertex* u *of degree* 0 **then**
    return $eccKernel(G - u, k)$
  else if G *has a connected component* C *that is a clique* **then**
    return $eccKernel(G - C, k - 1)$
  else if G *has distinct twins* u *and* v **then**
    return $eccKernel(G - u, k)$
  else if $|V| > 2^k$ **then**
    return *null*
  else
    return G
  end

Note that under certain assumptions of complexity (the Strong Exponential Time Hypothesis, for the connoisseurs), it has been shown that it is impossible to obtain a kernel smaller than 2^k. So there are strong reasons to believe that some kernels cannot be of polynomial size.

10.6 A kernel for VERTEX-COVER based on LPs

Let's go back to our favorite problem, VERTEX-COVER. We got a kernel of size $O(k^2)$, but it turns out that it is possible to get one of size $2k$ using an LP. Let's recall the LP of VERTEX-COVER

Minimize

$$\sum_{v_i \in V} x_i$$

Subject to

$$\begin{aligned} x_i + x_j &\geq 1 && \text{for each } v_i v_j \in E \\ 0 \leq x_i &\leq 1 && \text{for every } v_i \in V \end{aligned}$$

Suppose we have solved this LP and have obtained a solution $\boldsymbol{x}^* = \{x_1^*, \ldots, x_n^*\}$. Since the LP gives a lower bound on the size of the solution, we can immediately use the following rule.

Rule 1. If $\sum_{v_i \in V} x_i^* > k$, return *null*. .

To go a little further, we can separate our vertices into three sets:

$$\begin{aligned} V_0 &= \{v_i \in V : x_i^* < 1/2\} \\ V_{\frac{1}{2}} &= \{v_i \in V : x_i^* = 1/2\} \\ V_1 &= \{v_i \in V : x_i^* > 1/2\} \end{aligned}$$

Intuitively, the vertices of V_1 have the most weight and it is reasonable to believe that they should be in an optimal cover. Similarly, V_0 have a small weight and should not have to be included. It turns out that this intuition is true.

Lemma 15. *There is a vertex cover $X \subseteq V$ of minimum size such that $V_1 \subseteq X \subseteq V_{\frac{1}{2}} \cup V_1$.*

In words, the lemma says that we don't need the V_0 vertices in our solution. Moreover, we can assume that all the vertices of V_1 are in the solution. We are not going to prove this lemma - a proof can be found in Cygan's book & al. However, we can extract a very simple rule from it.

Rule 2. Remove V_0 from G, add V_1 to the solution, and reduce k by $|V_1|$. .

Lemma 16. *Rule 2 is safe.*

Proof. Let G' be the graph obtained after applying rule 2. It must be shown that G has a vertex cover of size k if and only if G' has a vertex cover of size $k - |V_1|$.

($\Rightarrow$) Let X be a vertex cover of G of size k. By the lemma 15, we can assume that $V_1 \subseteq X$. So there must be $k - |V_1|$ vertices in X to cover the remaining edges in G'.

($\Leftarrow$) Let X' be a vertex cover G' of size $k-|V_1|$. In G, only the incidental edges of V_0 and V_1 remain to be covered. Let $X = X' \cup V_1$. Then the edges incident to V_1 are covered. Let $v_i \in V_0$ and $v_iv_j \in E$. Since $v_i \in V_0$, we have $x_i^* < 1/2$ and we need $x_j^* > 1/2$ to satisfy the LP constraint. So $v_j \in V_1$ and X covers v_iv_j. □

With the right analysis, this leads directly to a kernel.

Theorem 27. *Let G' be the graph obtained after applying rules 1 and 2. Then G' has at most $2k$ vertices.*

Proof. Since rule 2 does not apply, $x_i^* = 1/2$ for any $v_i \in V(G')$. We have

$$|V(G')| = |V_{\frac{1}{2}}| = \sum_{v_i \in V_{\frac{1}{2}}} 2x_i^* \leq 2 \sum_{v_i \in V(G)} x_i^* \leq 2k$$

where we used rule 1 to deduce that $\sum_{v_i \in V(G)} x_i^* \leq k$. □

10.7 Do all FPT problems have a kernel?

It is possible to demonstrate that a problem is in FPT if and only if it admits a kernel. For some, FPT could even have been defined by the existence of a kernel, and the aspect of parameterized complexity would have been only a consequence.

Theorem 28. *A P problem is in FPT if and only if P admits a kernel.*

Proof. ($\Leftarrow$) Let's start with the easy direction. Let (I, k) be an instance of P and suppose that we can find a kernel of size $O(f(k))$ with parameter $k' \in O(g(k))$ in time $O(|I|^c)$. A brute force on the kernel will give an algorithm $O(h(k', f(k)))$ for a certain function h. Whatever h, we will have $h \in O(h_2(k))$ for a certain function h_2. The algorithm that creates the kernel and executes the brute force takes a time $O(h_2(k) + |I|^c)$, which is FPT.

($\Rightarrow$) Let's assume that P is FPT and that we can solve any instance (I, k) in time $O(f(k)|I|^c)$ with a certain algorithm A. We build a kernel with an alternative B algorithm. Initially, B executes A during $d|I|^{c+1}$ instructions, where d is the hidden constant in the O of $O(f(k)|I|^c)$, and looks to see if A has finished. If so, then B returns the result of A, which took B a polynomial time (so no kernel is needed).

If A has not finished, then the time required by A, which is $df(k)|I|^c$, is greater than $d|I|^{c+1}$. So,

$$df(k)|I|^c > d|I|^{c+1}$$

which implies that

$$f(k) > |I|$$

Since $|I| < f(k)$, the size of our instance is bounded by k and is therefore a kernel. □

Note that the construction of a kernel in the above proof is not very useful in practice. It only executes a known FPT algorithm, so what is the use of the kernel of this algorithm if it can already solve the instance? The real use of kernels is usually to allow a simple and fast reduction of an instance, and then apply a more sophisticated FPT algorithm.

Chapter 11

Tree decomposition and *treewidth*

The *treewidth* is a parameter on graphs that determines how "close" the graph is to being a tree. Trees admit polynomial time algorithms over a wide range of problems. By generalizing, one can expect that a graph close to a tree admits algorithms close to being polynomial. The tree algorithms use, for the most part, a dynamic programming approach. We will start with some examples on trees, then generalize to graphs with bounded treewidth.

11.1 Dynamic programming on trees

Let $T = (V, E)$ a rooted tree. Recall that a tree is a connected graph without a cycle. An equivalent definition of a tree is a connected graph with exactly $n - 1$ edges. We denote the root by $r(T)$. For $v \in V$, the sub-tree rooted at v is denoted by $T[v]$.

Dynamic programming on a tree usually proceeds as follows. We need to optimize a certain criterion, and we determine a certain set of information $I(v)$ to be computed at each node $v \in V$, where $I(v)$ allows us to compute an optimal solution on T if it is known at each vertex (or at least at the root).

The information $I(v)$ must be computable from the $I(v_1), \ldots, I(v_k)$ of the children $v_1, \ldots, v_k$ of v. Often, $I(v)$ is expressed by a recurrence function that depends on the $I(v_i)$.

The general form of dynamic programming on a tree is therefore as follows:
– for each $v \in V$ knot in a post-order traversal – – let $v_1, \ldots, v_k$ be the

children of v
– – calculate $I(v)$ using the values $I(v_1), \ldots, I(v_k)$

Most of the time, calculating $I(v)$ on a v leaf is straightforward. It is at the internal nodes that there is sometimes a complication. More concretely, we usually have a recursive procedure:

```
fonction progDynTree(T = (V, E), v)
    //v is the current node
    if v is a leaf then
        Calculate I(v) trivially
    else
        foreach child v_i of v do
            progDynTree(T, v_i)
        end
        Calculate I(v) (knowing the I(v_i))
    end
```

The initial call is made with $v = r(T)$, the root. It is also necessary to specify how the optimal solution is calculated from $I(v)$ (which is usually done outside the algorithm).

Let's see some examples.

11.1.1 Independent set in a tree

Recall that a set $X \subseteq V$ is independent if $uv \notin E$ for any $u, v \in X$. We are looking for the maximum size independent set. This is of course NP-complete, but not on trees.

Suppose we get a rooted tree $T = (V, E)$. For each $v \in V$, we will calculate the following information:

- $M_0[v]$: the size of a maximum independent set X of $T[v]$, with the constraint that v is *not* in X;
- $M_1[v]$: the size of a maximum independent set X of $T[v]$, with the constraint that v *must be* in X.

Suppose we know $M_0[v]$ and $M_1[v]$ for each $v \in V$. Then the optimal solution on the whole tree is given by $\max(M_0[r(T)], M_1[r(T)])$. This is

because there are two possibilities for the independent set on $T = T[r(T)]$: either the root is in the set or not, and we know the optimal in both cases.

But how to compute $M_0[v]$ and $M_1[v]$? If v is a leaf, this is easy. We have

$$M_0[v] = 0 \qquad M_1[v] = 1$$

that correspond to not include or include v.

If v is an internal node, if we decide not to include v, then we can take the optimal solutions from the child sub-trees and take the union. We will always have an independent set because we are not going to include two nodes that share an edge in the union. The optimal solution in subtree $T[w_i]$ is given by $\max(M_0[w_i], M_1[w_i])$. So we have

$$M_0[v] = \sum_{w_i \in child(v)} \max(M_0[w_i], M_1[w_i])$$

If we decide to include v, then we cannot include a child of v, otherwise we would not have an independent set. We must therefore count 1 for the addition of v, plus the optimal of the child subtrees that do not include a child. So we have

$$M_1[v] = 1 + \sum_{w_i \in child(v)} M_0[w_i]$$

The complete algorithm is therefore as follows.

```
fonction maxIndSet(T = (V, E), v)
    //v is the current node
    if v is a leaf then
        M_0[v] = 0
        M_1[v] = 1
    else
        M_0[v] = 0
        M_0[v] = 1
        foreach child w_i of v do
            maxIndSet(T, w_i)
            M_0[v] += max(M_0[w_i], M_1[w_i])
            M_1[v] += M_0[w_i]
        end
    end
```

The initial call is made with $v = r(T)$, and we return $\max(M_0[r(T)], M_1[r(T)])$.

Note that the "if" was not necessary, because M_0 and M_1 are initialized the same way whether we have a sheet or not.

11.1.2 VERTEX-COVER on a tree

The complement of a maximum independent set is a VERTEX-COVER, so a simple algorithm to return the minimum size of a vertex cover is to run the above algorithm, get a value k, then reoturn $n - k$. One can also formulate dynamic programming using the same principles.

For $v \in V$, we write $M_0[v]$ for the size of a minimum vertex cover X such that $v \in X$, and $M_1[v]$ when $v \in X$. For v a leaf, we still have $M_0[v]$ =0 and $M_1[v]$ =1.

If v is an internal node, if we don't include v, we must include each child of v to cover the edges. So

$$M_0[v] = \sum_{w_i \in child(v)} M_1[w_i]$$

If we include v, we do what we want with the children, and therefore

$$M_1[v] = 1 + \sum_{w_i \in child(v)} \min(M_0[w_i], M_1[w_i])$$

(we take the min because here we minimize). The searched final value is $\min(M_0[r(T)], M_1[r(T)])$.

The algorithm is almost identical to the one above. In fact, this type of algorithm is often given only by recurrences, because they can be translated directly into the algorithm in a standard way. For the rest, we will therefore give the recurrences only. Let's see a last example.

11.1.3 Assigning characters in a phylogeny

We have a tree $T = (V, E)$ in which each leaf l is assigned to a character $c(l)$. The character is part of an alphabet Σ. We have a transformation cost function $f(a, b)$ which represents the cost of transforming the a character into b, where $a, b \in \Sigma$. We assume that $f(a, b) = f(b, a)$. The goal is to assign to each internal node a character so as to minimize the sum of the costs on each edge. Put another way, given $c(l)$ to the leaves, we try to assign $c(v)$ for each internal node, so as to minimize $\sum_{uv \in E} f(u, v)$. This

is useful in bioinformatics to reconstruct the evolution of ancestral species (the internal nodes) when only today's species (the leaves) are known.

For each node $v \in V$ and each character $a \in \Sigma$, we calculate $M_a[v]$ which is the minimum possible cost in $T[v]$ knowing that $c(v) = a$.

If v is a leaf, we have

$$M_a[v] = \begin{cases} 1 \text{ if } a = c(v) \\ \infty \text{ si } a \neq c(v) \end{cases}$$

Otherwise, if v is an internal node, we will take the minimum in the child subtrees by adding the cost of the edges. The main observation is that we don't need to test every possible combination in the children. We can take the minimum in each child independently. In short, we have

$$M_a[v] = \sum_{w_i \in child(v)} \min_{b \in \Sigma}(f(a,b) + M_b[w_i])$$

In the end, we return $\min_{a \in \Sigma} M_a[r(T)]$. In exercise, write the pseudo-code that implements this recurrence.

11.2 Tree decomposition and *treewidth*

We would like to use the techniques described above on a graph $G = (V, E)$, not just on a tree. To do so, we would have to find some kind of tree representation of G. After much research effort, a representation that fits very well with dynamic programming was found.

Given a $G = (V, E)$ graph, the idea is to

- define subsets of vertices $B_1, B_2, \ldots, B_p \subseteq V$, called *bags*. These sets can have a non-empty intersection, and p must be polynomial in n. We can even assume that $p \in O(n)$.
- build a tree $T = (V_T, E_T)$ in which $V_T = \{B_1, \ldots, B_p\}$.

The T tree must satisfy certain properties for dynamic programming to be applicable.

Let $G = (V, E)$ be a graph. It is assumed that G has no isolated vertex. Let $T = (V_T, E_T)$ a tree with $V_T = \{B_1, \ldots, B_p\}$. We say that T is a *tree decomposition of* $G = (V, E)$ if all the following conditions are satisfied:

1. for each $B_i \in V_T$, $B_i \subseteq V$ (the B_i are bags);

2. for each $uv \in E$, there is $B_i \in V_T$ such that $u \in B_i$ and $v \in B_i$ (each edge is in at least one bag);

3. for each $v \in V$, then the bags containing v form a connected subgraph of T. More formally, let $B_1^v, \ldots, B_q^v$ be the T bags that contain v. Then the subgraph of T on the vertices $B_1^v, \ldots, B_q^v$ is connected.

The *size* of the T decomposition is $\max_{B_i \in V_T} |B_i| - 1$. The *treewidth* of G is the minimum size of a decomposition of G.

In other words, there are many tree decompositions for a graph G. We are looking for the one in which the largest bag is the minimum size.

These conditions may seem abstract at the moment. It is difficult to give an intuition as to their purpose. The idea is that these conditions allow us to say that a node in the T decomposition tree is a bag B_i that separates the graph into two or more independent parts, where the parts do not share any edges except those in B_i. These conditions are necessary to ensure this, and play a role in dynamic programming.

11.3 Some examples

It is generally difficult to obtain a minimum decomposition of a graph. It is in fact NP-complete to calculate it. However, we can look at a few simple examples.

11.3.1 Tree decomposition of a tree

If $G = (V, E)$ is a tree, then $tw(G) = 1$. This is because we can take the decomposition $T = (V_T, E_T)$ in which $V_T = \{\{u, v\} : uv \in E\}$. It is assumed that G is rooted in a leaf ℓ (otherwise we reroot). Then we add an edge in E_T between $\{u, v\}$ and $\{v, w\}$ if and only if u is the parent of v and v is the parent of w. The size of this decomposition is $\max_{uv \in E} |\{u, v\}| - 1 = 1$.

It is clear that $\forall uv \in E$, there is a B_i containing u and v.

We must show that for any $v \in V$, the bags containing v form a connected subgraph of T. This is obvious for a leaf because only one bag contains it. Otherwise, for an internal node v, these bags are the edges containing v. There is the edge-bag $\{u, v\}$ where u is the parent of v, and the edge-bags $\{v, v_i\}$ for each $v_i \in child(v)$. In T, these bags are linked because the $\{v, v_i\}$ have an edge with $\{u, v\}$, so it's connected.

To show that T is a tree, we have to argue that T is connected and has no cycle. It is easy to see that T is related because G is related (to

be proved in exercise). Moreover, if we assume that T has a cycle, it has the form $(\{u_1, u_2\}, \{u_2, u_3\}, \ldots, \{u_{k-1}, u_k\}, \{u_k, u_1\})$, where each u_i is the parent of u_{i+1}. This means that in G, we have the cycle $(u_1, u_2, \ldots, u_k, u_1)$, contradicting that G is a tree.

11.3.2 Tree decomposition of a cycle

Suppose that $G = (V, E)$ is a cycle $(v_1, v_2, \ldots, v_n, v_1)$. We can show that $tw(G) = 2$. We can obtain a decomposition into a tree of size 2 (so with bags of size 3 or less).

First we choose v_1 arbitrarily. Let $G - v_1$, i.e. the graph obtained from G by removing v_1. We see that $G - v_1$ is a tree with vertices $v_2, \ldots, v_n$. As seen above, we can obtain a decomposition $T' = (V'_T, E'_T)$ for $G - v_1$ where the bags are all of size 2. Now, if we add v_1 to *all* the bags of T', we get a decomposition for G. This is because the edges containing v_1 will certainly be in one of the bags, and because the bags containing a v_i form a connected subgraph.

11.3.3 Tree decomposition of a clique

Let's assume that $G = (V, E)$ is a clique of size n. It turns out that $tw(G) = n - 1$, because essentially, the only way to decompose G is to have a single bag with V as a whole.

To demonstrate this, suppose that G can be decomposed into a tree $T = (V_T, E_T)$ where all bags of V_T have $n-1$ elements or less. Let B_r be the largest bag of T. We can assume that B_r is the root of T. Since $|B_r| \leq n-1$, there is $v \in V$ such that $v \notin B_r$. Now, let B_v be the closest descendant of B_r that contains v. Since the bags containing v form a connected graph, all bags containing v descend from B_v. Also, since B_r is maximum, there must be w such that $w \in B_r$ but $w \notin B_v$. Since bags containing w form a connected graph, no descendant of B_v contains w. This implies that there is no bag that contains both v and w.

It is therefore not a valid decomposition, because vw is an edge of the clique and each edge must be contained in a bag.

We deduce that there must be a bag of size n in any decomposition of a clique.

11.4 Basic results

In the cycle decomposition described above, a vertex v was removed, the decomposition of $G - v$ was obtained, and v was added to all bags. This procedure can be generalized to any graph and any vertices removed.

Theorem 29. *Let $G = (V, E)$ a graph and $X \subseteq V$. Then $tw(G) \leq tw(G - X) + |X|$.*

We are not going to prove this theorem. The idea is simple. If we take a minimum decomposition of $G - X$, we can add X to all the bags and prove that we satisfy all the conditions.

Theorem 30. *Let $G = (V, E)$ and X a clique with a maximum size of G. Then $tw(G) \geq |X|-1$.*

Proof. Suppose that $tw(G) < |X|-1$. Then there is a decomposition $T = (V_T, E_T)$ where each bag has at most $|X| - 1$ elements. By removing the vertices that are not part of X from these bags, we get a decomposition for X of size $|X| - 1$ or less. This contradicts the fact that the X clique has treewidth $|X|-1$. □

This fact can be very useful in practice. Several problems on graphs can be solved in time $O(c^{tw(G)} \cdot n^c)$. If the graphs in your application have a small treewidth, such an algorithm can be very efficient. On the other hand, if you have reason to believe that your graphs have very large cliques, you can immediately discard the option of using a tree decomposition.

A final result, possibly less useful in practice but still very interesting, is that $tw(G)$ is equivalent to the "cops vs. robbers" parameter.

In the game of **cops versus robbers**, we have a graph $G = (V, E)$ and a thief placed on a vertex of G. There are also k policemen who are in helicopters, and a helicopter can be either flying or parked on a vertex. The thieves and policemen take turns. In one turn, a thief can move to any accessible vertex without passing a vertex occupied by a policeman. Then, a policeman can either take off (if parked), or land at an arbitrary summit (if flying). The goal is to find out if it is possible for the police to get to a point where all the vertices neighboring the thief are occupied by police officers. The *cops vs robbers number* is the minimum k such that k policemen are enough to catch the thief.

It turns out that the cops vs robbers number in G is equal to $tw(G) + 1$. In a sense, the cops vs robbers problem is equivalent to calculating the treewith of a graph.

11.5 Algorithms on tree decomposition

For an algorithm designer, the most important element of the treewidth concept is the tree decomposition. A classical result in FPT states that if a graph has a treewidth k, then we can find a tree decomposition in time $O(k^{O(k^3)})$ (other approximation results of the treewidth in FPT time are also known). Moreover, the number of nodes in the decomposition is linear and the tree is binary (remember that binary means that each internal node has 2 children). The proof of this theorem is deep and we will use it in a black box for our purposes.

Theorem 31. *Let G be a graph such that $tw(G) = k$. In time $O(k^{O(k^3)})$, it is possible to find a tree decomposition of G width k with $O(|V|)$ nodes.*

We can assume that such an algorithm has been executed and that a decomposition is given to us. All that remains is to use it to solve our problems.

To do this, we do a bit like in dynamic programming on a tree, but knowing that the vertices represent bags, therefore subsets of V. The fundamental point is that the size of the bags is bounded by $tw(G) + 1$. When one reaches a B_i bag, one can thus store information on each subset of B_i, or on each permutation of B_i.

11.6 Maximum independent set

As a first example, let's consider MAX-INDSET parameterized by the treewidth. Recall that a set X is independent if $\forall u, v \in V$, $uv \notin E$.

MAX-INDSET
Input : graph $G = (V, E)$
Parameter : $tw(G)$
Output : an independent set $X \subseteq V$ maximum size.

Suppose we are given a tree decomposition $T = (V_T, E_T)$ of G. We can assume that T has $O(|V|)$ nodes. For $B_i \in V_T$ a bag, we recall that $B_i \subseteq V$ and that $|B_i| \leq tw(G) + 1$. We denote by $T[B_i]$ the sub-tree of T rooted at B_i. In addition, we denote by

$$V_i = \bigcup_{B_j \in V(T[B_i])} B_j$$

the set of G vertices that are present in $T[B_i]$.

We would like to store the information used to find a maximum independent set in $G[V_i]$, the subgraph induced by V_i. A classical way is to compute a table $M[S, B_i]$ for each $B_i \in V(T)$ and each $S \subseteq B_i$, such that $M[S, B_i]$ is the size of the maximum independent set X of $G[V_i]$. We can express $M[S, B_i]$ with a recurrence, but it becomes relatively complex.

Here, we prefer a more intuitive version that considers two-color colorings of the vertices of G. For $S \subseteq V$, a coloring of S is a function $c : S \to \{green, red\}$ which assigns a color $green$ or red to each vertex of S. We can see $green$ as "is in the set independent" and red as "is not in the independent set". We denote by

$$c^g$$

all the vertices colored in green.

We can see MAX-INDSET as the search for a coloring of V such that c^g is an independent set of maximum size. There are 2^n colorings, which is too much for an FPT algorithm, but the idea here is to try all possible colorings for each B_i bag.

Let $B_i \in V_T$ and let c_i a coloring of B_i. We define

$$M[c_i, B_i]$$

as the maximum size of an independent X set of $G[V_i]$, with the restriction that $X \cap B_i = c_i^g$. We define $M[c_i, B_i] = -\infty$ if such a set does not exist. This seems to represent a lot of concepts, but note that it is only a generalization of dynamic programming on a tree.

But how do you compute $M[c_i, B_i]$? If B_i is a leaf of T, it is easy to see that

$$M[c_i, B_i] = \begin{cases} |c_i^g| \text{ if } c_i^g \text{ is an independant set} \\ -\infty \text{ otherwise} \end{cases}$$

Let's assume that B_i is an internal node. The idea will be to compute $M[c_i, B_i]$ from the $M[c_j, B_j]$ of the children of B_i. Let B_j be a child of B_i and let c_j be a coloring of B_j. It is said that c_i and c_j are *compatible* if $c_i(u) = c_j(u)$ for any $u \in B_i \cap B_j$.

If c_i^g does not induce an independent set, we can put $M[c_i, B_i] = -\infty$ immediately. Otherwise, we can prove the following recurrence for $M[c_i, B_i]$:

$$M[c_i, B_i] = |c_i^g| + \sum_{B_j \in child(B_i)} \left[\max_{c_j \text{ compatible with } c_i} (M[c_j, B_j] - |c_i^g \cap c_j^g|) \right]$$

If we think about it long enough, this recurrence becomes intuitive. To find an independent set of $G[V_i]$ that contains c_i^g, one can combine independent sets found in child subtrees. On the other hand, before combining these independent sets, we want to make sure that their colorings are compatible with what is requested by c_i. The subtraction of $|c_i^g \cap c_j^g|$ is done to avoid counting a green vertex more than once.

This is just a hunch. Usually it is necessary to demonstrate that such a recurrence is correct, which often requires significant work.

Theorem 32. *The recurrence for $M[c_i, B_i]$ above is correct.*

Proof. To show that this recurrence is true, we will show that $M[c_i, B_i]$ is smaller than or equal to the given expression, and that $M[c_i, B_i]$ is larger than or equal to the given expression.

Let's start with the first statement. Let X be a maximum independent set of $G[V_i]$ such that $X \cap B_i = c_i^g$. Let B_j be a child of B_i in T. Let c_{j^*} be a coloring of B_j such that $c_{j^*}(u) = green$ if $u \in X$, and $c_{j^*}(u) = red$ otherwise. It is clear that c_i and c_{j^*} are compatible, because they come from the same X set. If we consider $X_j = X \cap V_j$, we have an independent set of $G[V_j]$ such that $X_j \cap B_j = c_{j^*}^g$. By definition, $|X_j| \leq M[c_{j^*}, B_j]$. So we have

$$\begin{aligned} M[c_i, B_i] &= |c_i^g| + \sum_{B_j \in child(B_i)} (|X_j| - |c_i^g \cap c_{j^*}^g|) \\ &\leq |c_i^g| + \sum_{B_j \in child(B_i)} (M[c_j, B_j] - |c_i^g \cap c_{j^*}^g|) \\ &\leq |c_i^g| + \sum_{B_j \in child(B_i)} \left[\max_{c_j} (M[c_j, B_j] - |c_i^g \cap c_j^g|) \right] \end{aligned}$$

(or subtraction is done to avoid double counting).

In the other direction, let $B_{j_1}, \ldots, B_{j_l}$ be the children of B_i. For each B_{j_h}, let X_{j_h} be an independent set of $G[V_{j_h}]$ such that the coloring c_{j_h} of B_{j_h} corresponding to X_{j_h} is compatible with c_i, and which maximizes $|X_{j_h}| - |c_i^g \cap c_{j_h}^g|$. We want to show that

$$X = c_i^g \cup X_{j_1} \cup \ldots \cup X_{j_l}$$

is an independent set. Note that since the coloring c_{j_h} are all compatible with c_i, $X \cap B_i = c_i^g$. Suppose there are $u, v \in X$ that share an edge. Then $u, v \in B_i$ is impossible because we first check that c_i^g induces an independent

set and $X \cap B_i = c_i^g$. So, $u \in V_{j_h}$ for a child B_{j_h} of B_i and $u \notin B_i$. Since bags containing u form a connected component of T, all occurrences of u are in B_{j_h} or below. Furthermore, since $uv \in E$, there must be a bag B_{uv} descending from B_{j_h} such that $u, v \in B_{uv}$. Thus, u and v both have occurrences in B_{j_h} or below. This would mean that $u, v \in X_{j_h}$, which is contradictory because X_{j_h} is supposed to be an independent set of V_{j_h}. So our X is indeed an independent set.

Note that the size of X is given by $|c_i^g| + \sum_{B_{j_h}} \left[|X_{j_h}| - |c_i^g \cap c_{j_h}^g| \right]$. This is to subtract the intersections of the X_{j_h} with c_i^g, and because the X_{j_h} have in common only elements of B_i (because of condition 3 of the decompositions). Since the $|X_{j_h}| - |c_i^g \cap c_{j_h}^g|$ are of maximum size, the size of X is less than $|c_i^g| + \sum_{B_j \in child(B_i)} \left[\max_{c_j}(M[c_j, B_j] - |c_i^g \cap c_j^g|) \right]$, as desired. □

11.7 Nice decompositions

As we have seen, recurrences on decompositions can become complex. One way to (somewhat) simplify these recurrences is to simplify the decomposition itself.

Let $G = (V, E)$ be a graph and $T = (V_T, E_T)$ be a tree decomposition of G. We say that T is a *nice* decomposition if:

1. each node B_i of T has $0, 1$ or 2 children;
2. if a node B_i has 2 children B_l and B_r, then $B_i = B_l = B_r$.

   In this case, B_i is called a Join Node.
3. if a node B_i has 1 child B_j, then one of these two cases occurs:
   - $B_i = B_j \cup \{v\}$ for a certain $v \in V$. In this case, B_i is called a Introduce Node.
   - $B_i = B_j \setminus \{v\}$ for a certain $v \in B_j$. In this case, B_i is called a Forget Node.

The advantage of nice decompositions is that the content of a node B_i compared to that of its children is simple, which allows us to simplify the definition of recurrences at nodes B_i. The disadvantage is that we have to deal with three cases for our recurrences: we have to describe a recurrence according to the type of the B_i node, which could be a Join node, an Introduce node or a Forget node.

An important fact is that we can always find a nice decomposition without changing the size of the decomposition.

Theorem 33. *Let $G = (V, E)$ a graph. If we are given a tree decomposition $T' = (V'_T, E'_T)$ of G size $tw(G)$, we can transform T' into a **nice** decomposition of size $tw(G)$ in polynomial time.*

11.8 MAX-INDSET and nice decomposition

Let's take the MAX-INDSET problem again, but suppose that $T = (V, T)$ is nice. We take again the notions of coloring and compatibility between colorings.

Recall that $M[c_i, B_i]$ denotes the maximum size of an independent set X of $G[V_i]$ such that $X \cap B_i = c_i^g$. If B_i is a leaf, the definition of $M[c_i, B_i]$ does not change. If B_i is an internal node, we check to see if the elements of c_i^g contain an edge. If so, then $M[c_i, B_i] = -\infty$. If not, we have three possible cases for $M[c_i, B_i]$:

1. B_i is a INTRODUCE NODE. Let B_j be the child of B_i and let $v \in V$ such that $B_i = B_j \cup \{v\}$. Since $B_j \subseteq B_i$, we note that there is only one coloring of B_j compatible with c_i, because we have to use the same colors no matter the color of v. Let c_j^* be the only coloring of B_j compatible with c_i.

   Then
   $$M[c_i, B_i] = M[c_j^*, B_j] + \begin{cases} 1 \text{ if } c(v) = green \\ 0 \text{ otherwise} \end{cases}$$

   This corresponds to taking an optimal independent set on V_j, and adding v if it has been colored green. It is important to understand why adding v cannot create an edge in the independent set.

2. B_i is a FORGET NODE. Let B_j be the child of B_i and let v be $B_i = B_j \setminus \{v\}$. We don't add any vertex at the B_i level, so we take an optimal solution at the B_j level.

   Then
   $$M[c_i, B_i] = \max_{c_j \text{ compatible with } c_i} M[c_j, B_j]$$

   Note that there are only two colorings of B_j compatible with c_i, depending on whether we add v or not.

3. B_i is a JOIN NODE. Let B_l and B_r be the children of B_i and recall that $B_i = B_l = B_r$. We can just take an independent set at the level B_l and join it with an independent set at the level B_r. In both cases, there is only one compatible coloring, and we must avoid double counting the green vertices.

   So
   $$M[c_i, B_i] = M[c_i, B_l] + M[c_i, B_r] - |c_i^g|$$

We are not going to demonstrate that these recurrences are correct. However, it is important to be convinced that the case of join nodes works. Taking the sum $M[c_i, B_l] + M[c_i, B_r]$ corresponds to taking a maximum independent set on each side and combining them. How do you know that you have not included two vertices sharing an edge by doing so? This is a consequence of the properties of decompositions, which ensure that B_i is a separator between these two sets.

11.9 MAX-CUT and nice decompositions

Recall that in MAX-CUT, we have a graph $G = (V, E)$ and we want a bipartition (V_1, V_2) of G such that $|E(V_1, V_2)|$ is maximum. We parameterize by $tw(G)$ and suppose that we have a nice decomposition $T = (V_T, E_T)$.

The idea of coloring is still useful here. Each vertex will have a color 1 or 2 representing its presence in V_1 or V_2. For a bag B_i, c_i represents a coloring of the vertices of B_i with 1 or 2. We writs $c_i^1 = \{v \in B_i : c(v) = 1\}$ and $c_i^2 = \{v \in B_i : c(v) = 2\}$. The notion of compatibility between colorings remains the same.

For $B_i \in V_T$ and c_i a coloring of B_i, we denote by $M[c_i, B_i]$ the maximum number of traversing edges in a bipartition (V_1', V_2') of $G[V_i]$ such that $c_i(v) = 1 \Rightarrow v \in V_1'$ and $c_i(v) = 2 \Rightarrow v \in V_2'$.

If B_i is a leaf, it is easy to see that

$$M[c_i, B_i] = |E(c_i^1, c_i^2)|$$

Otherwise, assume that B_i is an internal node. We have three cases.

- B_i is a INTRODUCE NODE. Let B_j be the child of B_i and v the new element of B_i. Let us denote by c_j^* the only coloring of B_i compatible with c_i. The number of traversing edges is the same as before, in addition to the new traversing edges that include v. By the decomposition properties, all neighbors of v in V_i must be in B_i.

So

$$M[c_i, B_i] = M[c_j^*, B_j] + \begin{cases} |N(v) \cap c_i^2| \text{ if } c_i(v) = 1 \\ |N(v) \cap c_i^1| \text{ if } c_i(v) = 2 \end{cases}$$

- B_i is a FORGET NODE. Let B_j be the child of B_i. In this case, we don't add a vertex to our bipartition and we can take the best solution at the level of B_i which is compatible (note that there are only two).

  So

  $$M[c_i, B_i] = \max_{c_j \text{ compatible with } c_i} M[c_j, B_j]$$

- B_i is a JOIN NODE. Let B_l and B_r be the children of B_i, remembering that $B_i = B_l = B_r$. It is enough to combine the bipartitions at the level of B_l and B_r, taking care not to count the same edge twice.

  Then

  $$M[c_i, B_i] = M[c_i, B_l] + M[c_i, B_r] - |E(c_i^1, c_i^2)|$$

Chapter 12

Conclusion

We have seen various algorithmic solutions applicable when faced with an NP-complete problem. One can either try to approximate it quickly, or to extract a small parameter on which the exponential complexity depends. Most of the problems encountered are toy problems. The usefulness of the approaches presented here consists in making links between theory and real-life computer problems.

An approximation is necessary when the data sets are very large, for example in the millions, because in this case there is very little chance of finding a small parameter. On the other hand, datasets in the order of thousands can often be handled using exact FPT methods. This course has been used to introduce you to the main techniques of algorithm development. The challenges that await you are to know **when** to apply these techniques. And if you ever undertake algorithmic research, a bigger challenge awaits you: creating *new* techniques.