Lecture on Graphs and Algorithms (Master 1)

Nicolas Nisse[*]

Abstract

These are lecture notes of the course I gave, at Master 1 level. The main topic is the design of algorithms for solving NP-hard problems, mostly in graphs.

Contents

[*]Université Côte d'Azur, Inria, CNRS, I3S, France

Part I
Introduction

1 Informal Introduction

The main goal of this lecture is to present some ways/techniques/methods to design **efficient** algorithms (and their analysis) to solve optimization problems (mainly in the graph context). Note that Section 8 is clearly beyond the scope of this lecture but aims at going further into algorithmic graph theory.

This introduction is only an intuition of what will be addressed in the lecture (most of the concepts will be more formally defined and examplified later).

Tradeoff between time complexity and quality of the solution. What is an *efficient* algorithm? Some problems have a unique solution (e.g., sorting a list of integer), some other problems have several *valid* ($\approx$ correct) solutions but only some of them are optimal (e.g. finding a shortest path between two vertices in a graph: there may be many paths, but only few of them may be shorter).

Here, we measure the *efficiency* as a tradeoff between the "**time**" to get a valid/correct solution (time-complexity) and the **"quality"** of a valid solution (how "far" is it from an optimal solution?).

Difficult vs. Easy Problems. We assume here that readers are familiar with basics on time complexity of algorithms. If not, see [3] or here (in french, on polynomial-time algorithms) for prerequisite background.

Very informally, problems may be classified into

P. Class of problems for which we know a **P**olynomial-time algorithm (polynomial in the size of the input) to solve them.

NP. Class of problems for which we know a **N**on-deterministic **P**olynomial-time algorithm to solve them. Equivalently, it can be checked in (deterministic) polynomial-time whether a solution to such problem is valid/correct. (Clearly, $P \subseteq NP$)

NP-hard. Class of problems that are "as hard as the hardest problems in NP". I don't want to give a formal definition of it here. Informally, you should understand this class of problems as the ones for which **nobody currently knows** a deterministic polynomial-time algorithm to solve them (related to the question whether $P = NP$, a question that worths 1.000.000 dollars). Intuitively (not formally correct), the best known algorithms for solving such problems consist of trying all possibilities...

In what follows, I refer to problems in P as "easy" and to NP-*hard* problems as "difficult". The main question I try to address in this lecture is how to deal with difficult problems. We probably (unless $P = NP$) cannot solve them "efficiently" (in polynomial time)... so, should we stop trying solving them? NO !!! there are many ways to tackle them and **the goal of this lecture is to present some of these ways**. Roughly, we will speak about:

1. **Better exponential exact algorithms.** "Try all possibilities in a more clever way" [6]
2. **Approximation algorithms.** Design poly-time algo. for computing solution (not necessarily optimal) with "quality's guaranty" ("not far from the optimal") [8]
3. **Restricted graph classes.** "Use specifities of inputs" [4, 8]
4. **Parameterized algorithms.** (formal definition will be given later) [4]

First, I want to give some basics on graph theory. The main reasons for it is that: (1) graphs are a natural (and nice) mathematical model to describe many real-life problems, and (2), we will then mainly consider graph problems as examples (so, we need a common background on graphs structural results and algorithms). Then, this lecture will try to address several techniques (mentioned above) to deal with difficult problems (mostly in graphs).

2 Basics on Graphs [2, 5]

A graph G is a pair $G = (V, E)$ where V is a set[1] of elements and $E \subseteq V \times V$ is a relationship on V. Any element of V is called vertex. Two vertices $u, v \in V$ are "linked" by an edge $\{u, v\}$ if $\{u, v\} \in E$, in which case u and v are said adjacent or neighbors. So V is the set of vertices and E is the set of edges.[2]

Intuition. It can be useful (actually, IT IS!!) to draw graphs as follows: each vertex can be depicted by a circle/point, and an edge between two vertices can be drawn as a curve (e.g., a (straight) line) linking the corresponding circles/points.
Graphs are everywhere. As examples, let us consider a graph where vertices are: cities, proteins, routers in the Internet, people,... and where two vertices are linked if they are: linked by a road (road networks), by some chemical interaction (biological networks), by optical fiber (computer networks/Internet), by friendship relationship (social networks: Facebook, Twitter...).

Notation. For $v \in V$, let $N(v) = \{w \in V \mid \{v, w\} \in E\}$ be the *neighborhood* of v (set of its neighbors) and $N[v] = N(v) \cup \{v\}$ be its *closed neighborhood*. The degree of a vertex $v \in V$ is the number $deg(v) = |N(v)|$ of its neighbors. Given a graph G, if V and E are not specified, let $E(G)$ denote its edge-set and let $V(G)$ denote its vertex-set.

Proposition 1 *Let $G = (V, E)$ be any simple graph: $|E| \leq \frac{|V|(|V|-1)}{2}$ and $\sum_{v \in V} deg(v) = 2|E|$.*

[1] In what follows, we always assume that V is finite, i.e., $|V|$ is some integer $n \in \mathbb{N}$.

[2] **Technical remark.** Unless stated otherwise, in what follows, we only consider simple graphs, i.e., there are no loops (i.e., no vertex is linked with itself, i.e., $\{v, v\} \notin E$ for every $v \in V$) nor parallel edges (i.e., there is at most one edge between two vertices).

Proof. We prove that $\sum_{v \in V} deg(v) = 2|E|$ by induction on $|E|$. If $|E| = 1$, then G must have two vertices with degree 1, and all other with degree 0. So the result holds. Assume by induction that the result holds for $|E| \leq k$ and let assume that $|E| = k+1$. Let $\{a,b\} \in E$ and let $G' = (V, E \setminus \{a,b\})$ be the graph obtained from G by removing the edge $\{a,b\}$. By induction, $\sum_{v \in V} deg_{G'}(v) = 2|E(G')| = 2(|E|-1)$. Since $\sum_{v \in V} deg_G(v) = \sum_{v \in V} deg_{G'}(v) + 2$ (because the edge $\{a,b\}$ contributes for 2 in this sum), the result holds. Let $n = |V|$.

Since each vertex has degree at most $n-1$, $2|E| = \sum_{v \in V} deg(v) \leq \sum_{v \in V} (n-1) = n(n-1)$. ■

A Walk in a graph $G = (V,E)$ is a sequence $(v_1, \cdots, v_k)$ of vertices such that two consecutive vertices are adjacent (i.e., for every $1 \leq i < k$, $\{v_i, v_{i+1}\} \in E$). A Trail is a walk where no edges are repeated. A trail is a Tour if the first and last vertex are equal. A tour is Eulerian if it uses **each edge of G exactly once**.

A Path is a walk with no repeated vertex. Finally, a Cycle is a tour with no repeated vertex (except that the first and last vertices are equal). A cycle is Hamiltonian if it meets **every vertex of G exactly once**.

Note that, a path is a trail, and a trail is a walk (but not all walks are trails, not all trails are paths). Similarly, a cycle is a tour, and a tour is a walk (not all walks are tours, not all tours are cycles).

Exercise 1 *Give examples of graphs that are*

- *Eulerian (that admits a Eulerian tour) AND Hamiltonian (admits an Hamiltonian cycle)*[3]*;*
- *not Eulerian AND Hamiltonian;*
- *Eulerian AND not Hamiltonian;*
- *not Eulerian AND not Hamiltonian.*

At a first glance, the problem of deciding whether a graph is Eulerian and the problem of deciding whether a graph is Hamiltonian look very similar. From the complexity point of view it seems they are quite different.

2.1 P vs. NP-hard, a first example: Eulerian vs. Hamiltonian

A graph $G = (V,E)$ is connected if, for every two vertices $u, v \in V$, there is a path connecting u to v. Note that, to admit a Eulerian or Hamiltonian cycle, a graph must be connected. So, in what follows, we only focus on connected graphs. Given $v \in V$, the connected component of G containing v is the graph $(V_v, E \cap (V_v \times V_v))$ where V_v is the set of all vertices reachable from v in G.

The following algorithm decides whether a graph is Hamiltonian. For this purpose, it considers one after the other every permutation of V (all possible ordering of the n vertices) and checks if this permutation corresponds to a cycle.

There are $n!$ permutations of V[4] so, in the worst case, there are $n!$ iterations of the for-loop. At each iteration, the algorithm checks n edges to verify if the current permutation is a cycle. Overall, the time-complexity is then $O(n \cdot n!)$. In this course, we will see other (much better)

[3]Formally, a graph reduced to a single vertex is a pathological case of a graph with both a Eulerian cycle and an Hamiltonian cycle (both reduced to this single vertex). Try to find examples with more vertices.

[4]For any set with n elements, there are $n!$ orderings of these elements. Indeed, there are n choices for the first element, $n-1$ for the 2^{nd} one, $n-2$ for the 3^{rd} one, and so on, so $n(n-1)(n-2)\cdots 2 \cdot 1 = n!$ in total.

Algorithm 1 Naive algorithm for Hamiltonicity

Require: A connected graph $G = (V, E)$.

Ensure: Answers Yes is G is Hamiltonian and No otherwise.

```
1: for each permutation of $V$ do
2:   if the permutation corresponds to a cycle then
3:     return $Yes$.
4:   end if
5: end for
6: return $No$
```

algorithms for this problem, but all have at least exponential time-complexity. Roughly, the only way we know is to try all possibilities. Indeed, it can be proved (I will not do it here) that the problem of deciding whether a graph is Hamiltonian is NP-hard! [7]

The other problem is very different.

Theorem 1 *[***Euler 1736***] A graph admits an Eulerian cycle iff all vertices have even degree.*

Before proving this theorem, let us look at its consequence.

Algorithm 2 Algorithm for deciding of a graph is Eulerian

Require: A connected graph $G = (V, E)$.

Ensure: Answers Yes is G is Eulerian and No otherwise.

```
1: for every $v \in V$ do
2:   if $v$ has odd degree then
3:     return $No$.
4:   end if
5: end for
6: return $Yes$
```

There are n iterations and each of them just checks the degree of one vertex. Overall, the complexity is $O(\sum_{v \in V} deg(v))$ which is $O(|E|)$ by proposition above. Therefore, the problem of deciding whether a graph has an Eulerian cycle is in P.

Proof.[Sketch of proof of Th. 1] Assume that G is Eulerian and let $v \in V$. Each time the cycle reaches v by some edge, it must leave by another (not used yet) edge. Hence, v has even degree.

Now, let us assume that every vertex has even degree. The proof is constructive (it produces a Eulerian cycle). Let us sketch a recursive algorithm that computes an Eulerian cycle.

1. First, start from any vertex v and greedily progress along the edges. Each time a new vertex is reached, it is possible to leave it since its degree is even (and so it remains at least one non-used edge). Since a graph is finite, eventually, this greedy walk reaches a vertex that has already been visited and so, we found a cycle $\mathcal{C} = (v_1, \cdots, v_r)$.

2. Let G' be the graph obtained by removing from G every edge of $\mathcal{C}$. Let $G'_1, \cdots, G'_k$ be the connected components of G'. Removing the edges of $\mathcal{C}$, every vertex of $\mathcal{C}$ has its degree reduced by exactly two. So, for every $i \leq k$, every vertex of G'_i has even degree and, by induction on the number of edges, G'_i has a Eulerian cycle. By applying recursively the algorithm on G'_i, let $\mathcal{C}'_i$ be the Eulerian cycle obtained for G'_i.

3. A Eulerian cycle of G is obtained by starting from v_1, following $\mathcal{C}$ and, each time it meets a vertex v_j ($j \leq r$), it follows the Eulerian cycle of the connected component of G' that contains v_j (if not yet met). *Prove that it is actually a Eulerian cycle.*

Note that this algorithm has roughly time-complexity $O(|E||V|)$ (finding a cycle takes time $O(|V|)$ and, in the worst case, has size 3 and so decreases by 3 the number of edges).

■

Note that, previous proof shows that, not only it can be decided if a graph is Eulerian in polynomial-time, but also an Eulerian cycle (if any) can be found in polynomial-time.

2.2 Trees, subgraphs and spanning trees (Kruskal's algorithm)

A Tree is any acyclic (with no cycle) connected graph.

A Subgraph of G is any graph that can be obtained from G by removing some edges and some vertices. Hence, a subgraph of $G = (V, E)$ is any graph $H = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. If $V' = V$, then H is a spanning subgraph. A spanning tree of G is a spanning subgraph which is a tree.

Exercise 2 *Let $G = (V, E)$ be any graph. Show that:*

- *if G is a tree, there is a unique path between any two vertices of G;*
- *if G is a tree, then $|E| = |V| - 1$;*
- *G admits a spanning tree if and only if G is connected;*
- *deduce from previous items that, G is connected $\Rightarrow |E| \geq |V| - 1$;*
- *if G is acyclic and $|E| = |V| - 1$, then G is a tree;*
- *if G is connected and $|E| = |V| - 1$, then G is a tree;*

Given $X \subseteq V$, the subgraph (of G) induced by X is the subgraph $G[X] = (X, E \cap (X \times X))$. If $F \subseteq E$, the graph induced by F is $G[F] = (\bigcup_{\{u,v\} \in F} \{u, v\}, F)$.

Let $w : E \to \mathbb{R}^*_+$ be a weight function on the edges. The weight of a subgraph $G[F]$ induced by $F \subseteq E$ is $\sum_{e \in F} w(e)$. The goal of this section is the computation of a connected spanning subgraph with minimum weight.

Exercise 3 *Let (G, w) be a graph with an edge-weight function. Show that any minimum-weight spanning connected subgraph is a spanning tree.*

Proof. Let H be a minimum spanning connected subgraph. If H contains no cycle, it is a tree. Otherwise let C be a cycle in H and let $e \in E(C)$ be any edge of E. Show that $(V(H), E(H) \setminus \{e\})$ is a connected spanning subgraph of G. Conclusion? ■

Above exercise somehow justifies the interest of minimum spanning tree in a practical point of view. For instance, assume you want to connect some elements of a network (cities connected by roads, buildings connected by electrical cables, etc.) and that weights on the links represent the price of building them. Then, a minimum spanning tree will be the cheapest solution.

Let's compute it!

The first step (ordering the edges) takes time $O(m \log m)$[5]. Then there are at most m iterations of the for-loop. Each iteration takes time $O(1)$ (by using suitable data structure such as Union-Find). Overall, the time-complexity of the algorithm of Kruskal is $O(m \log m)$.

Theorem 2 *The Kruskal's algorithm returns a minimum spanning tree of G.*

[5]Recall that ordering a set of n elements takes time $O(n \log n)$ (e.g., merge-sort)

Algorithm 3 : Kruskal's Algorithm (1956)

Require: A (non-empty) connected graph $G = (V, E)$ and $w : E \to \mathbb{R}^*_+$.
Ensure: A minimum spanning tree T of G.

```
1:  Order $E$ in non decreasing order: $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.
2:  Let $v \in V$ and $T = (\{v\}, \emptyset)$.
3:  for $i$ from 1 to $m$ do
4:    if If $T$ is connected and spanning then
5:      return $T$.
6:    end if
7:    Let $e_i = \{u, v\}$.
8:    if $(V(T) \cup \{u, v\}, E(T) \cup \{e_i\})$ is acyclic then
9:      $T \leftarrow (V(T) \cup \{u, v\}, E(T) \cup \{e_i\})$.
10:   end if
11: end for
```

Proof. Let T be the spanning tree computed by the algorithm. Let $(e_{i_1}, e_{i_2}, \cdots, e_{i_{n-1}})$ be the edges of T ordered in non-decreasing order of their weights. Let T^* be a minimum spanning tree minimizing $|E(T^*) \cap E(T)|$. If $T^* = T$, we are done. Otherwise, let $j < n$ be the minimum index such that $e_{i_j} \in E(T) \setminus E(T^*)$ (why it exists?). Note that $E(T^*) \cap \{e_1, e_2, \cdots, e_{i_j}\} = \{e_{i_1}, e_{i_2}, \cdots, e_{i_{j-1}}\}$ by def. of j and of the algorithm. Let $e_{i_j} = \{u, v\}$ and let P be the unique path from u to v in T^* (why it exists?). Show that P contains an edge $f \notin E(T)$ such that $w(f) \geq w(e_{i_j})$. Show that $(T^* \setminus \{f\}) \cup \{e_{i_j}\}$ is a minimum spanning tree of G. Conclusion? ■

As we will see in the following of the lecture, computing a minimum spanning tree of a graph is one important basic blocks of many graph algorithms!!

2.3 Matching and Vertex Cover in graphs

Graphs are a very useful tool to deal with allocation problems. For instance, consider a set of students that have to choose an internship among a set of proposals. Each student and proposal may be modeled as vertices of a graph and a student is linked to a proposal if it has some interest for it. How to assign internships to students so that at most one student is assigned to each proposal and every student gets an internship that interests him/her? Say differently, how to *match* internships and students? This is the topic of this subsection.

Let $G = (V, E)$ be a graph. A matching in G is a set $M \subseteq E$ of edges such that $e \cap f = \emptyset$ for every $e \neq f \in M$. That is, a matching is a set of edges pairwise disjoint.

A matching M is perfect is all vertices are matched, i.e., for every vertex v, there is an edge $e \in M$ such that $v \in e$.

Exercise 4 *Show that a graph has a perfect matching only if $|V|$ is even.*
Give a connected graph with $|V|$ even but no perfect matching.
Show that, for any matching M, $|M| \leq \lfloor \frac{|V|}{2} \rfloor$.

A matching M in G is maximum if there are no other matching M' of G with $|M'| > |M|$. Let $\mu(G) = |M|$ be the size of a maximum matching M in G. A matching M in G is maximal if there is no edge $e \in E$ such that $M \cup \{e\}$ is a matching.

Exercise 5 *Show that every maximum matching is maximal.*
Give examples of maximal matchings that are not maximum.

Algorithm 4 Algorithm for Maximal Matching

```
Require: A graph $G = (V, E)$.
Ensure: A maximAL matching $M$ of $G$.
1: $G' \leftarrow G$ and $M \leftarrow \emptyset$.
2: while $E(G') \neq \emptyset$ do
3:     Let $e = \{u, v\} \in E(G')$                // so, $e$ is any (arbitrary) edge of $G'$
4:     $M \leftarrow M \cup \{e\}$ and $G' \leftarrow G'[V(G') \setminus \{u, v\}]$.
5: end while
6: return $M$.
```

Exercise 6 *Prove that above algorithm computes a maximal matching in time $O(|E|)$.*

Proof. Three things to be proved: M is a matching, M is maximal, and the time-complexity. ■

Now, we focus on computing maximUM matchings in graphs. Let $G = (V, E)$ be a graph and $M \subseteq E$ be a matching in G. A vertex $v \in V$ is covered by M if there is $e \in M$, $v \in e$ (i.e., if v "touches" one edge of the matching). Let $k \geq 2$. A path $P = (v_1, \cdots, v_k)$ is M-alternating if, for any two consecutive edges $e_{i-1} = \{v_{i-1}, v_i\}$ and $e_i = \{v_i, v_{i+1}\}$ $(1 < i < k)$, exactly one of e_{i-1} and e_i is in M. The path P is M-augmenting if P is alternating and v_1 and v_k are not covered by M. Note that, in that case, $v_2, \cdots, v_{k-1}$ are all covered by M and k is even.

Theorem 3 **(Berge 1957)** *Let $G = (V, E)$ be a graph and $M \subseteq E$ be a matching in G. M is maximum matching if and only if there are no M-augmenting paths.*

Proof. First, let us assume that there is an M-augmenting path P. Show that $M' = (M \setminus E(P)) \cup (E(P) \setminus M)$ ("switch" the edges in P) is a matching and that $|M'| = |M| + 1$ and so, M is not maximum. For this purpose, first show that M' is a matching. Then, show that P has odd length, i.e., $2k + 1$ edges, and that k edges of P are in M and $k + 1$ are not in M. Conclude.

Now, assume that there are no M-augmenting paths. Recall that the *symmetric difference* $A \Delta B$ between two sets A and B equals $(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$. Let M^* be a maximum matching in G and let $X = G[M \Delta M^*]$. So, X is the subgraph of G induced by the edges that are in M or M^* but not in both.

Show that every vertex has degree at most two (in X) in any connected component of X. Deduce that the connected components of the graph X consist of paths and cycles (we say that X is the *disjoint union* of paths and cycles).

So the connected components of X consist of cycles $C_1, \cdots, C_k$ and paths $P_1, \cdots, P_\ell$. Show that, for every $i \leq k$, C_i has even size. Deduce that $|E(C_i) \cap M| = |E(C_i) \cap M^*|$. Let $j \leq \ell$. Show that, because there are no M-augmenting path, $|E(P_j) \cap M^*| \leq |E(P_j) \cap M|$.

Therefore, $|M^*| = |M \cap M^*| + \sum\limits_{i=1}^{k} |E(C_i) \cap M^*| + \sum\limits_{j=1}^{\ell} |E(P_j) \cap M^*| \leq |M \cap M^*| + \sum\limits_{i=1}^{k} |E(C_i) \cap M| + \sum\limits_{j=1}^{\ell} |E(P_j) \cap M| = |M|$. So $|M^*| \leq |M|$ and M is a maximum matching (since M^* is maximum). ■

Theorem 3 suggests (actually proves) that, to compute a maximum matching in a graph, it is sufficient to follow the following greedy algorithm. The key point is that the order in which augmenting paths are considered is not relevant! (see [Bensmail *et al.*'17] for different behavior).

Algorithm 5 Algorithm for Maximum Matching

```
Require: A graph $G = (V, E)$.
Ensure: A maximUM matching $M$ of $G$.
1: $M \leftarrow \emptyset$.
2: while there is an $M$-augmenting path $P$ do
3:     $M \leftarrow (M \setminus E(P)) \cup (E(P) \setminus M)$.
4: end while
5: return $M$.
```

The time-complexity of previous algorithm relies on the time needed to find an M-augmenting path (condition in the While-loop). This can actually be done in polynomial-time using the *Blossom algorithm* [Edmonds 1965]. This algorithm has been improved and a maximum matching in any graph $G = (V, E)$ can be found in time $O(\sqrt{|V|}|E|)$ [Micali,Vazirani 1980]. Computing matching in graphs is a basic block of many graph algorithms as we will see below.

In what follows, we focus on a restricted graphs' class, namely *bipartite* graphs.

2.3.1 Matchings in bipartite graphs (Hall's theorem, Hungarian's method)

Given a graph $G = (V, E)$, a set of $I \subseteq V$ is an independent set (or stable set) if, $\{u, v\} \notin E$ for every $u, v \in I$ (the vertices of I are pairwise not adjacent in G). A graph $G = (V, E)$ is bipartite if V can be partitioned into two stable sets A and B.

Exercise 7 *Show that any tree is bipartite.*

Show that a graph G is bipartite iff G has no cycle of odd size.

Proof. Show that, if there is an odd cycle in G, then it is not bipartite.

Otherwise, let v be any vertex and consider a Breadth First Search (BFS[6]) from v. Set A to be the vertices at even *distance* from v and $B = V \setminus A$. ■

Let $G = (A \cup B, E)$ be a bipartite graph[7]. W.l.o.g., $|A| \leq |B|$. Clearly (prove it), a maximum matching M in G is such that $|M| \leq |A|$. A set $S \subseteq A$ is saturated by a matching M of G is all vertices of S are covered by M (in which cas $|M| = |A|$). For any graph $G = (V, E)$, given $S \subseteq V$, let us denote $N(S) = \{u \in V \setminus S \mid \exists v \in S, \{u, v\} \in E\} = \bigcup_{v \in S}(N(v) \setminus S)$. That is, $N(S)$ is the set of vertices not in S that are neighbor of some vertex in S.

Theorem 4 (Hall 1935) *Given a bipartite graph $G = (A \cup B, E)$, $|A| \leq |B|$, there is matching saturating A iff, for all $S \subseteq A$, $|S| \geq |N(S)|$.*

Proof. If there is $S \subseteq A$ such that $|S| > |N(S)|$ then no matching can saturates S. The reverse implication can be proved by induction on $|A|$. Algorithm 6 is a constructive proof.

Prove the correctness of this algorithm (In particular, prove that, in the last Else case, $|X| = |N(X)| + 1$). ■

Note that above algorithm, known as the Hungarian method [Kuhn 1955], can be used to find augmenting paths in polynomial-time in bipartite graphs (just try every uncovered vertex as starting point), and so it allows to compute a maximum matching in bipartite graphs.
Understanding why this algorithm requires the graph to be bipartite would be an instructive exercise.

[6]Please see [3] or [5] if you don't know what a BFS is.
[7]Implicitly (or say, by notation), A and B are stable sets.

Algorithm 6 : Hungarian method [Kuhn 1955]

```
Require: A bipartite graph $G = (A \cup B, E)$.
Ensure: A matching $M$ saturating $A$ or a set $S \subseteq A$ such that $|S| > |N(S)|$.
1: $M \leftarrow \emptyset$.
2: while $A$ is not saturated by $M$ do
3:   Let $a_0 \in A$ be any vertex not covered by $M$. Set $X = \{a_0\}$.
4:   Let $Continue = True$.
5:   while $N(X)$ saturated by $M$ and $Continue$ do
6:     $Y \leftarrow \{a_0\} \cup \{a \mid \exists b \in N(X), \{a,b\} \in M\}$.
7:     if $X \subset Y$ then
8:       $X \leftarrow Y$
9:     else
10:      $Continue = False$.
11:    end if
12:  end while
13:  if $\exists b_0 \in N(X)$ not covered by $M$ then
14:    Let $P$ be an $M$-augmenting path between $a_0$ and $b_0$;
15:    $M \leftarrow (M \setminus E(P)) \cup (E(P) \setminus M)$.
16:  else
17:    return $X$
18:  end if
19: end while
20: return $M$.
```

2.3.2 Vertex Cover in graphs (König's theorem and 2-approximation)

On the "practical" point of view, consider a city (buildings are vertices that are linked with streets/edges). We aim at placing, say, as few as possible (because it is expansive) fire-stations in buildings so that each building it adjacent to at least one fire-station.

This problem is modeled as follows. Given a graph $G = (V, E)$, a set $K \subseteq V$ is a vertex cover if $K \cap e \neq \emptyset$ for every $e \in E$. Let $vc(G)$ be the smallest size of a vertex cover in G. The problem of computing a minimum vertex cover in a graph is NP-hard in general graphs [7].

When you are facing an NP-hard problem (or, even, any problem), you must have the **reflex** to think to any "naive" algorithm to solve it (e.g., trying all feasible solutions and keep a best one). Precisely, imagine any feasible solution (e.g., prove that V is a vertex cover for any graph $G = (V, E)$) and try to improve it.

Algorithm 7 Naive Algorithm for Minimum Vertex Cover

```
Require: A graph $G = (V, E)$.
Ensure: A minimum Vertex Cover of $G$.
1: $K \leftarrow V$.
2: for every $S \subseteq V$ do
3:   if $S$ is a vertex cover of $G$ and $|S| < |K|$ then
4:     $K \leftarrow S$.
5:   end if
6: end for
7: return $K$
```

In the above algorithm, there are $2^{|V|}$ iterations of the For-loop (number of subsets of V), and each iteration requires to check if a set of vertices is a vertex cover (check if all edges are touched), i.e., each iteration requires time $O(|E|)$. Overall, for any $G = (V, E)$, the problem of computing $vc(G)$ can be solved in time $O(|E| \cdot 2^{|V|})$.

Again, the goal of this lecture is to learn how/when to solve such a problem (that, unless $P = NP$, cannot be solved in polynomial-time) in more efficient ways...

Now, we show that the Min. Vertex Cover problem is "easy" (can be solved in polynomial-time) in bipartite graphs. Then we show that finding a vertex cover that is "not too large" is not difficult in any graph.

Lemma 1 *Let K be any vertex cover and M be any matching of any graph G. Then $|M| \leq |K|$.*

Proof. For every edge $e \in M$, $K \cap e \neq \emptyset$ by definition of a vertex cover. Moreover, for every $v \in K$, there is at most one edge $e \in M$ such that $v \in e$ (by definition of a matching). So $|M| \leq |K|$ (each edge of M is touched by at least one vertex of K and each vertex of K touches at most one edge of M). ■

Theorem 5 (König-Egerváry 1931) *For any bipartite graph $G = (A \cup B, E)$, the size of a minimum vertex cover $vc(G)$ equals the size of a maximum matching $\mu(G)$.*

Proof. The fact that $\mu(G) \leq vc(G)$ follows from Lemma 1.

Let M be a maximum matching of G, i.e., M is a matching of G and $|M| = \mu(G)$. If A is saturated by M then $|M| = |A|$ and, because G is bipartite, A is a vertex cover and the result follows. Assume A is not saturated by M and let $U \subseteq A$ be the uncovered vertices in A. Let X be the set of vertices linked to some vertex in U by a M-alternating path. Let $X_A = X \cap A$ and $X_B = X \cap B$. Note that X_B is saturated (since otherwise, there is a M-augmenting path contradicting that M is maximum by Th. 3). Moreover, $X_A = N(X_B)$. Prove that $Y = X_B \cup (A \setminus X_A)$ is a vertex cover of size $|M|$ (take any edge $e \in E$ and show that at least one of its ends is in Y and prove that $|Y| = |M|$). So, $vc(G) \leq |Y| = |M| = \mu(G)$. ■ Note that the proof of Theorem 5 is constructive: it allows to compute, from a maximum matching in a bipartite graph, a minimum vertex cover in polynomial-time.

Theorem 6 *Let M be any maximal matching of a graph G. Then, $|M| \leq vc(G) \leq 2|M|$.*

Proof. The left inequality follows from Lemma 1.

Let M be a maximal matching. Then $Y = \bigcup_{\{u,v\} \in M} \{u, v\}$ is a vertex cover. Indeed, if not, there is $f = \{x, y\} \in E$ such that $Y \cap \{x, y\} = \emptyset$, and so $M \cup \{f\}$ is a matching, contradicting that M is maximal. Hence, $vc(G) \leq |Y| = 2|M|$. ■

Recap. on Min. Vertex Cover. Let us consider the problem that takes a graph $G = (V, E)$ as input and aims at computing a set $K \subseteq V$, which is a vertex cover of minimum size in G. As mentioned above the corresponding decision problem is NP-complete in general graphs and can be (naively) solved in time $O(|E| \cdot 2^{|V|})$. The goal of this lecture is to explain how/when we can improve this time-complexity. In the introduction, we mentioned four possible ways to go through the "P *vs.* NP barrier". In the current subsection, we gave two different answers in the specific case of the Min. Vertex Cover Problem (MVCP):

Restrict inputs: By Th. 5 and algorithm of [Micali,Vazirani 1980], the MVCP can be solved in time $O(\sqrt{|V|}|E|)$ in bipartite graphs, i.e., MVCP is in P when restricted to bipartite graphs!!

First Approximation Algorithm: Consider the following algorithm: compute M a maximal matching of G (can be done in time $O(|E|)$ by Alg. Maximal Matching) and return $V(M) = \{v \mid \exists e \in M, v \in M\}$. By the proof of Th. 6, $V(M)$ is a vertex cover of G. Moreover, $|V(M)| = 2 \cdot |M| \leq 2 \cdot vc(G)$ (because $|M| \leq vc(G)$ by Th. 6). So, in polynomial-time $O(|E|)$, it is possible to compute a vertex cover X of any graph G that is "not too bad" ($|X|$ is at most twice the size of a minimum vertex cover).

Exercise 8 *Give an example of graph for which the algorithm of previous paragraph gives a vertex cover of size twice the optimal.*

Note that last proposition relies on the following facts. We are able to compute in polynomial-time some feasible solution (here M) and to bound on both sides (lower and upper bound) the size of any optimal solution in the size of M!!

Roughly, these are the key points of approximation algorithms as we formalize them in next section.

Part II
Approximation Algorithms [8]

3 Introduction to Approximation Algorithms: Load balancing

In this section, we formally define the notion of approximation algorithm (efficient algorithm that computes a "not too bad" solution) and exemplify this notion via the Load Balancing problem. For this problem we design and analyze two approximation algorithms, the second algorithm improving the first one.

3.1 Definition of an Approximation Algorithm

The following definition is not really formal since I do not want to give the formal definition of an optimization problem here. I think/hope this will be sufficient for our purpose.

Let us consider an *optimization problem* Π and let w be the function evaluating the quality (or cost) of a solution. Let $k \geq 1$. An algorithm $\mathcal{A}$ is a *k-approximation* for Π if each of the following three conditions is satisfied. For any input I of Π:

- $\mathcal{A}$ computes an output O in **polynomial-time** (in the size of I);
- this output O is a **valid** solution for I, i.e., satisfies the constraints defined by Π, and
- - if Π is a *minimization problem* (aims at finding a valid solution with minimum cost), then $w(O) \leq k \cdot w(O^*)$ where O^* is an optimal (i.e., with minimum cost) valid solution for I.
  - if Π is a *maximization problem* (aims at finding a valid solution with maximum cost), then $w(O^*) \leq k \cdot w(O)$ where O^* is an optimal (i.e., with maximum cost) valid solution for I.

For instance, if Π is the minimum vertex cover studied in previous section, the goal is, given a graph $G = (V, E)$, to compute a set $K \subseteq V$ with the constraint that it is a vertex cover (i.e., every edge is touched by some vertex in K), and the cost function is the size $|K|$ of the set.

Exercise 9 *Show that the algorithm that consists in computing a maximal matching M in G and returning $V(M)$ is a 2-approximation for the minimum vertex cover problem.*

Remark. In previous definition, we have assumed that k is a constant. This definition can be generalized by considering k as a function f of the size of the input. For instance, a $f(n)$-approximation is an algorithm that, given an input of size n, computes in polynomial-time a valid solution of cost at most $f(n)$ times the cost of an optimal solution for a minimization problem (resp., of cost at least $\frac{1}{f(n)}$ times the cost of an optimal solution for a maximization problem).

3.2 The Load Balancing Problem

For once, let us not speak about graphs...

Let us consider a set of $m \in \mathbb{N}$ identical processors, and a set of $n \in \mathbb{N}$ jobs described by the n-tuple $(t_1, \cdots, t_n)$ where t_i is the time required by a processor to execute job i, for every $1 \leq i \leq n$. Each processor can treat only one job at a time. Moreover, we assume that each job is unbreakable, i.e., it cannot be shared between several processors. The goal of the *Load Balancing* Problem is to assign each job to one processor in order to minimize the overall completion time.

More formally, the goal of the Load Balancing problem is to compute a partition $\mathcal{S} = \{A_1, \cdots, A_m\}$ of the jobs, i.e., $\bigcup_{1 \leq i \leq m} A_i = \{1, \cdots, n\}$ and $A_i \cap A_j = \emptyset$ for every $1 \leq i < j \leq m$ (this corresponds to an assignment of the jobs to the processors. Note that any A_i may be $\emptyset$) minimizing $\max_{1 \leq i \leq m} \sum_{j \in A_i} t_j$. Given an assignment $\mathcal{S} = \{A_1, \cdots, A_m\}$, the load $L_i(\mathcal{S})$ of the processor i ($1 \leq i \leq m$) is $\sum_{j \in A_i} t_j$ and $\max_{1 \leq i \leq m} L_i(\mathcal{S})$ is called the makespan of $\mathcal{S}$. So the Load Balancing problem consists in finding an assignment minimizing the makespan.

Load Balancing Problem (LBP):

Input: $m \in \mathbb{N}$ and $(t_1, \cdots, t_n) \in (\mathbb{R}^+)^n$.

Output: A partition $\{A_1, \cdots, A_m\}$ of $\{1, \cdots, n\}$ such that $\max_{1 \leq i \leq m} \sum_{j \in A_i} t_j$ is minimum.

The LBP is a classical NP-complete problem [7].

As usual, let us first think to what could be a (naive) algorithm: try all possibilities!! Each of the n jobs can be assigned to any of the m processors, which give m^n possibilities :(

Hence, we aim at designing approximation algorithms for solving it. An approximation algorithm requires to be able to evaluate the quality of the solution it returns with respect to the quality OPT of an optimal solution. Since, in general, we don't know the cost of an optimal solution, it is important to have reliable lower (in case of minimization problems) bounds on the cost of an optimal solution (resp., upper bound for maximization problems). Indeed, consider a minimization problem (the case of a maximization problem is similar, prove it). If we know a lower bound $LB \leq OPT$ of the cost of an optimal solution and that we can relate the cost c of any solution computed by our algorithm to LB (e.g., say $c \leq k \cdot LB$ for some constant k), this will be an indirect way to relate c and OPT: $LB \leq OPT \leq c \leq k \cdot LB \leq k \cdot OPT$.

3.2.1 Greedy 2-Approximation (least loaded processor)

For the LBP, there are two easy lower bounds:

Lemma 2 *Let $(m,(t_1,\cdots,t_n))$ be an instance of the LBP and let OPT be the minimum makespan over all assignments. Then $\max_{1\leq i\leq n} t_i \leq OPT$ and $\frac{1}{m}\sum_{1\leq i\leq n} t_i \leq OPT$.*

Proof. The first statement is obvious. The 2^{nd} statement follows the pigeonhole principle. ■

Exercise 10 *Prove that, if $n \leq m$, an optimal solution is to assign each job to exactly one processor, and that $\max_{1\leq i\leq n} t_i = OPT$.*

Hence, let us assume that $n > m$. Let us consider the intuitive simple greedy algorithm[8]. Here, let us assign the jobs sequentially (in any order) to a least loaded processor.

Algorithm 8 : 2-Approximation for Load Balancing Problem

```
Require: $n > m \in \mathbb{N}$ and $(t_1,\cdots,t_n) \in (\mathbb{R}^+)^n$.
Ensure: A partition $\{A_1,\cdots,A_m\}$ of $\{1,\cdots,n\}$.
1: $\{A_1,\cdots,A_m\} \leftarrow \{\emptyset,\cdots,\emptyset\}$.
2: for $i = 1$ to $n$ do
3:     $A_1 \leftarrow A_1 \cup \{i\}$.
4:     Reorder the $A_i$'s such that $\sum_{j\in A_1} t_j \leq \sum_{j\in A_2} t_j \leq \cdots \leq \sum_{j\in A_m} t_j$.
5: end for
6: return $\{A_1,\cdots,A_m\}$.
```

Theorem 7 *Algorithm 8 is a 2-approximation for the LBP, with time-complexity $O(n \log m)$.*

Proof. There are 3 properties to be proved! First, it returns a valid solution, i.e., here the computed solution $\{A_1,\cdots,A_m\}$ is a partition of $\{1,\cdots,n\}$. This is obvious (I hope?).

Second, it has polynomial-time complexity. Indeed, there are n iterations of the For-loop, and each of them requires to re-sort an already sorted list of m elements where only one has changed, which takes $O(\log m)$ time per iteration (see, e.g., [3]).

The third property, i.e., that the makespan $L = \max_{1\leq i\leq m}\sum_{j\in A_i} t_j$ of the computed partition $\{A_1,\cdots,A_m\}$ is at most twice the optimal makespan OPT, is the most "difficult" to prove. For any $i \leq m$, let $L_i = \sum_{j\in A_i} t_j$.

Let $x \leq m$ be such that $L = L_x = \max_{1\leq i\leq m} L_i$ (i.e., Processor x is one of the most loaded in the computed solution). Note that $OPT \leq L$.

Let $j \leq n$ be the last job assigned to A_x, i.e., j is the maximum integer in A_x. Note that $t_j \leq OPT$ by the first statement of Lemma 2.

For every $1 \leq y \leq m$, let L'_i be the load of Processor i when t_j is assigned to P_x (i.e., after iteration $j-1$ of the For-loop). Then, $L_x - t_j \leq L'_i$ for every $i \leq m$ (indeed, it is obvious for $i = x$, and if there is $y \in \{1,\cdots,m\} \setminus \{x\}$ with $L_x - t_j > L'_y$, the job j would have been assigned to Processor y by the definition of Algorithm 8). Since, for every $1 \leq i \leq m$, the load of Processor i is not decreasing during the algorithm, this implies that $L_x - t_j \leq L_i$ for every $i \leq m$.

Therefore, $m(L_x - t_j) \leq \sum_{1\leq i\leq m} L_i = \sum_{1\leq i\leq n} t_i$. Hence, $L_x - t_j \leq \frac{1}{m}\sum_{1\leq i\leq n} t_i \leq OPT$ where the last inequality is given by the second statement of Lemma 2.

[8]Roughly, an algorithm is *greedy* if each of its steps is guided by a simple local (i.e., depending on the current state) rule.

Hence, by previous paragraphs, we have $t_j \leq OPT$ and $L_x - t_j \leq OPT$.
To conclude, $L = L_x = (L_x - t_j) + t_j \leq OPT + OPT = 2 \cdot OPT$. ■

Note that, sometimes, the algorithm is better than we can expect from its analysis. For instance, previous theorem proves that Algorithm 8 is a 2-approximation for the LBP. The next question is whether Algorithm 8 is a c-approximation for the LBP for some $c < 2$. Note that the latter is not true if we can find an instance for which Algorithm 8 computes a solution of cost exactly twice the optimal.

Exercise 11 *Let $m \in \mathbb{N}$ and $n = (m-1)m+1$. Apply Algorithm 8 to the input $(m, (t_1, \cdots, t_n))$ where $t_1 = \cdots, = t_{n-1} = 1$ and $t_n = m$. Conclusion (let m tend to ∞)?*

Hence, to expect a c-Approximation for the LBP with $c < 2$, we need another algorithm.

3.2.2 Greedy $\frac{3}{2}$-Approximation (least loaded processor and ordered tasks)

Algorithm 9 : $\frac{3}{2}$-Approximation for Load Balancing Problem

```
Require: $n > m \in \mathbb{N}$ and $(t_1, \cdots, t_n) \in (\mathbb{R}^+)^n$.
Ensure: A partition $\{A_1, \cdots, A_m\}$ of $\{1, \cdots, n\}$.
1: Order the jobs in non decreasing order of their completion time, i.e. $t_1 \geq \cdots \geq t_n$.
2: $\{A_1, \cdots, A_m\} \leftarrow \{\emptyset, \cdots, \emptyset\}$.
3: for $i = 1$ to $n$ do
4:     $A_1 \leftarrow A_1 \cup \{i\}$.
5:     Reorder the $A_i$'s such that $\sum_{j \in A_1} t_j \leq \sum_{j \in A_2} t_j \leq \cdots \leq \sum_{j \in A_m} t_j$.
6: end for
7: return $\{A_1, \cdots, A_m\}$.
```

Note that Algorithm 9 only differs from Algorithm 8 by its first step that consists in ordering the jobs by non-decreasing completion time. However, it offers us the opportunity to use a new lower bound (proved by pigeonhole principle).

Lemma 3 *Let $n > m$, $(m, (t_1, \cdots, t_n))$ be an instance of the LBP with $t_1 \geq \cdots \geq t_n$ and let OPT be the minimum makespan over all assignments. Then $OPT \geq 2 \cdot t_{m+1}$.*

Theorem 8 *Algorithm 9 is a $\frac{3}{2}$-approximation for the LBP, with time-complexity $O(n \log n)$.*

Proof. Again, Algorithm 9 computes a valid solution (a partition) $\{A_1, \cdots, A_m\}$. Moreover, the bottleneck for the time-complexity is the first step that takes time $O(n \log n)$. It only remains to prove the approximation ratio.

For any $i \leq m$, let $L_i = \sum_{j \in A_i} t_j$ and let $x \leq m$ be such that $L = L_x = \max_{1 \leq i \leq m} L_i$ (i.e., Processor x is one of the most loaded in the computed solution). Note that $OPT \leq L$.

Let $j \leq n$ be the last job assigned to A_x, i.e., j is the maximum integer in A_x. Then, $j \geq m+1$ and $2 \cdot t_j \leq OPT$ by Lemma 3 and because $t_j \leq t_{m+1}$. As in the proof of previous theorem, $L_x - t_j \leq OPT$. Hence, $L_x = (L_x - t_j) + t_j \leq OPT + OPT/2 = \frac{3}{2} \cdot OPT$. ■

4 Traveling Salesman Problem (TSP)

4.1 Different variants and exact dynamic programming algorithm

A salesman has to visit several cities but aims at minimizing the length (or cost) of his journey. The TSP aims at computing a best possible route for the salesman.

More formally, let K_n be the complete graph (or clique) with n vertices, i.e., the graph with all possible edges between the n vertices. Recall that the weight of a subgraph H is $\sum_{e \in E(H)} w(e)$. Given the clique K_n with edge-weight $w : E(K_n) \to \mathbb{R}^+$, the Traveling Salesman Problem (TSP) aims at computing a Hamiltonian cycle of minimum weight in (K_n, w).

Note that the problem is restricted to complete graphs to ensure that there exists a Hamiltonian cycle. However, the TSP is equivalent to the problem of computing a minimum-weight walk[9] passing through all vertices and going back to the initial vertex in any graph. Indeed, let $G = (V, E)$ be a graph with edge-weight $\ell : E \to \mathbb{R}^+$. The distance between 2 vertices $u, v \in V$, denoted by $dist_G(u, v)$, is the minimum weight of a path from u to v in (G, ℓ). Let $w : V \times V \to \mathbb{R}^+$ that assigned to every pair (u, v) of vertices the weight $w(u, v) = dist_G(u, v)$.

Exercise 12 *Show that there exists a Hamiltonian cycle in (K_n, w) of weight $\leq k \in \mathbb{R}^+$ if and only if there exists a walk passing through all vertices and going back to the initial vertex, in G, with weight $\leq k$.*

Given (K_n, w) and $k \in \mathbb{R}^+$, the problem of deciding whether there exists a Hamiltonian cycle of weight at most k is NP-complete [7]. The algorithm proposed in Section 2.1 to decide if a graph has a Hamiltonian cycle can be directly adapted to solve the TSP in time $O(n \cdot n!)$. Here, we show how to improve the time complexity by presenting a dynamic programming algorithm.

Let $v_1 \in V(K_n)$ and let $S \subseteq V(K_n) \setminus \{v_1\}$ be a non-empty set of vertices and $v \in S$. Let $OPT[S, v]$ denote the minimum weight of a path that starts in v_1 and visits exactly all vertices in S, ending in v. Clearly, for every $v \neq v_1$, $OPT[\{v\}, v] = w(v_1, v)$.

Lemma 4 *Let $S \subseteq V(K_n) \setminus \{v_1\}$ with $|S| \geq 2$ and let $v \in S$.*

$$OPT[S, v] = \min_{u \in S \setminus \{v\}} OPT[S \setminus \{v\}, u] + w(u, v).$$

Proof. The proof is by double inequalities.

First, let P be a path from v_1 to v, with $V(P) = S \cup \{v_1\}$ and with weight $OPT[S, v]$. Let x be neighbor of v in P and let P' be the subpath of P from v_1 to x. Then, $OPT[S, v] = w(P') + w(x, v) \geq OPT[S \setminus \{v\}, x] + w(x, v) \geq \min_{u \in S \setminus \{v\}} OPT[S \setminus \{v\}, u] + w(u, v)$.

Conversely, let $x \in S \setminus \{v_1\}$ be such that $OPT[S \setminus \{v\}, x] + w(x, v)$ is minimum. Let P' be a path from v_1 to x, with $V(P) = (S \setminus \{v\}) \cup \{v_1\}$ and with weight $OPT[S \setminus \{v\}, x]$. Let P be the path from v_1 to v obtained from P' by adding to it the edge $\{x, v\}$. Then, $\min_{u \in S \setminus \{v\}} OPT[S \setminus \{v\}, u] + w(u, v) = OPT[S \setminus \{v\}, x] + w(x, v) = w(P) \geq OPT[S, v]$. ■

Theorem 9 (Bellman-Held-Karp 1962) *Algorithm 10 computes the minimum weight of a Hamiltonian cycle (and can be easily adapted to compute a minimum-weight Hamiltonian cycle) in time $O(n^2 \cdot 2^n)$.*

[9]The weight of a walk $(v_1, \cdots, v_k)$ is $\sum_{1 \leq i < k} w(v_i, v_{i+1})$, i.e., the multiplicity of edges is taken into account.

Algorithm 10 : Dynamic Programming for TSP **[Bellman-Held-Karp 1962]**

```
Require: Complete graph $K_n = (V, E)$ with $w : E \to \mathbb{R}^+$.
Ensure: The minimum weight of a Hamiltonian cycle.
 1: Let $v_1 \in V$.
 2: for $v \in V \setminus \{v_1\}$ do
 3:     $OPT[\{v\}, v] = w(v_1, v)$.
 4: end for
 5: for $S \subseteq V \setminus \{v_1\}$ with $|S| \geq 2$ in non decreasing size order do
 6:     for $v \in V \setminus \{v_1\}$ do
 7:         $OPT[S, v] = \min_{u \in S \setminus \{v\}} OPT[S \setminus \{v\}, u] + w(u, v)$.
 8:     end for
 9: end for
10: return $\min_{v \in V \setminus \{v_1\}} OPT[V \setminus \{v_1\}, v] + w(v, v_1)$.
```

Proof. The time complexity comes from the fact that there are $O(2^n)$ sets S to be considered. For each set S, and for each of the $O(n)$ vertices in $V \setminus \{v_1\}$, the algorithm must find a minimum value among $O(n)$ values, so each iteration of the main For-loop takes time $O(n^2)$.

The correctness follows the fact that, after the last "EndFor", the values $OPT[V \setminus \{v_1\}, v]$ are known for every $v \in V \setminus \{v_1\}$ (by Lemma 4). To conclude, let C^* be an optimal Hamiltonian cycle and let OPT be its weight. The proof is by double inequalities.

Let x be a neighbor of v_1 in C^* and let P be the path obtained from C^* by removing the edge $\{v_1, x\}$. Then, $OPT = w(P) + w(v_1, x) \geq OPT[V \setminus \{v_1\}, x] + w(x, v_1) \geq \min_{v \in V \setminus \{v_1\}} OPT[V \setminus \{v_1\}, v] + w(v, v_1)$. Finally, let $x \in V \setminus \{v_1\}$ minimizing $OPT[V \setminus \{v_1\}, x] + w(x, v_1)$ and let P be a spanning path from v_1 to x with weight $OPT[V \setminus \{v_1\}, x]$. Then adding the edge $\{x, v_1\}$ to P leads to a Hamiltonian cycle C and so, $\min_{v \in V \setminus \{v_1\}} OPT[V \setminus \{v_1\}, v] + w(v, v_1) = OPT[V \setminus \{v_1\}, x] + w(x, v_1) = w(C) \geq OPT$. ■

No algorithm for solving TSP in time $O(c^n)$ is known for any $c < 2$. Moreover,

Theorem 10 *If $P \neq NP$, there is no c-approximation algorithm for solving TSP, for any constant $c \geq 1$.*

So the TSP problem is difficult and it is even difficult to approximate it. To overcome this difficulty, there may be several options. Here, we discuss two of them: simplifying the problem and/or restricting the instances. To simplify the problem, we may allow repetitions of vertices and edges. So, let TSP^r be the problem that, given a weighted K_n, must compute a closed[10] walk, passing through all vertices, and with minimum weight. On the other hand, we may restrict the instances of the TSP. Since considering a complete graph is important because it ensures the existence of a Hamiltonian cycle, let us restrict the weight function. A weight function $w : V \times V \to \mathbb{R}^+$ satisfies the triangular inequality if, for every $a, b, c \in V$, $w(a, b) \leq w(a, c) + w(c, b)$. Let TSP_{ti} be the problem that, given a weighted (K_n, w) where w satisfies the triangular inequality, must compute a Hamiltonian cycle with minimum weight. Finally, let TSP^r_{ti} be the problem that, given a weighted (K_n, w) where w satisfies the triangular inequality, must compute a closed walk, passing through all vertices, and with minimum weight.

Lemma 5 *Any c-approximation algorithm for one problem in $\{TSP^r, TSP_{ti}, TSP^r_{ti}\}$ can be turned into a c-approximation algorithm for any problem in $\{TSP^r, TSP_{ti}, TSP^r_{ti}\}$.*

[10] "Closed" means that the starting and final vertices are the same.

Proof. Any solution for TSP_{ti} is a solution for TSP^r_{ti} (with same weight). Similarly, any solution for TSP^r is a solution for TSP^r_{ti} (with same weight).

Let $W = (v_1, \cdots, v_k)$ be a solution for TSP^r_{ti}. If no vertex is repeated, then W is a solution of TSP_{ti}. Otherwise, let us assume that there are $1 \leq i < j < k$ such that $v_i = v_j$, then $W' = (v_1, \cdots, v_{j-1}, v_{j+1}, \cdots, v_k)$ is a solution and the weight of W' is not larger than the weight of W by the triangular inequality. Repeating this process sequentially until no vertex is repeated leads to a solution of TSP_{ti} (without increasing the weight).

Finally, let $(K_n = (V, E), w)$ be an instance of TSP^r. Let $dist_{K_n} : E \to \mathbb{R}^+$ be the distance function with respect to w. Prove that $dist_{K_n}$ satisfies the triangular inequality. Then, $(K_n, dist_{K_n})$ is an instance of TSP_{ti}. Following Exercise 12, any solution of $(K_n, dist_{K_n})$ for TSP_{ti} leads to a solution (with same weight) of (K_n, w) for TSP^r.

Prove that above arguments allow to prove the lemma. ■

4.2 2-Approximation (using minimum spanning tree)

Let us now show that these problems (TSP^r, TSP_{ti}, TSP^r_{ti}) admit "good" approximation algorithms. Precisely, let us first consider the problem TSP^r. Since repetitions of edges/vertices are allowed, we may consider any connected graph (rather than complete graphs) since closed walk passing through every vertex always exists in any connected graph. Recall the notion of Depth First Search (DFS) of a tree [3].

As usual, the design (and analysis) of an approximation algorithm for solving some optimization problem Π requires some lower bound (if possible, that can be computed in polynomial time) on the quality of an optimal solution of Π.

Lemma 6 *Let $G = (V, E)$ be a connected graph and $w : E \to \mathbb{R}^+$. Let w^* be the minimum weight of a closed walk passing through all vertices in V. Let t^* be the minimum weight of a spanning tree in G. Then, $t^* \leq w^*$.*

Proof. Let W be any closed walk passing through all vertices in V and with minimum weight $w(W) = w^*$. Then, $E(W)$ induces a connected spanning subgraph H of G with weight $w(H) \leq w(W)$ (the difference between $w(H)$ and $w(W)$ is that, in $w(H)$, each edge is counted only once). Let T be any minimum spanning tree of G. By Exercise 3, the weight $t^* = w(T)$ of T is at most the weight of any connected spanning subgraph. Hence, $t^* = w(T) \leq w(H) \leq w(W) = w^*$. ■

Algorithm 11 : 2-approximation for TSP^r

```
Require: A connected graph $G = (V, E)$ with $w : E \to \mathbb{R}^+$.
Ensure: A closed walk passing through every vertex in $V$.
1: Let $T$ be a minimum spanning tree of $G$.
2: return the closed walk defined by any DFS-traversal of $T$.
```

Theorem 11 *Algorithm 11 is a 2-approximation algorithm for the problem TSP^r.*

Proof. The fact that Algorithm 11 returns a valid solution is trivial. Its time-complexity follows from the one of the problem of computing a minimum spanning tree which can be done in polynomial-time (Th. 2). Finally, the weight of the computed walk W is twice the minimum weight t^* of the computed spanning tree T (because W follows a DFS of T, each edge of T is crossed exactly twice in W). Hence, $w(W) = 2t^* \leq 2w^*$ where w^* is the weight of an optimal closed spanning walk (by Lemma 6). ■

4.3 $\frac{3}{2}$-Approximation (Christofides'algorithm)

Let us conclude this section by an even better approximation algorithm. To simplify the presentation, let us consider the TSP_{it}. The next approximation algorithm relies on a new lower bound.

Lemma 7 *Let K_n with $w : E \to \mathbb{R}^+$ satisfying the triangular inequality, and let w^* be the minimum weight of a Hamiltonian cycle. Let $V' \subseteq V(K_n)$ with $|V'|$ even. Finally, let M be a minimum weight perfect matching of V'. Then, $w(M) \leq w^*/2$.*

Proof. Let C be an optimal Hamiltonian cycle of K_n and let C' be the cycle spanning V' obtained by short-cutting C. By triangular inequality, $w(C') \leq w(C) = w^*$. Moreover, $E(C')$ can be partitioned into two perfect matchings M_1 and M_2 of V'. Since $w(C') = w(M_1)+w(M_2)$, w.l.o.g., $w(M_1) \leq w(C')/2$. Finally, $w(M) \leq w(M_1) \leq w(C')/2 \leq w(C)/2 = w^*/2$. ∎

Algorithm 12 : 3/2-approximation for TSP_{ti} **[Christofides 1976]**

```
Require: A complete graph $K_n = (V, E)$ with $w : E \to \mathbb{R}^+$ satisfying the triangular inequality.
Ensure: A Hamiltonian cycle.
1: Let $T^*$ be a minimum spanning tree of $G$.
2: Let $O$ be the set of vertices with odd degree in $T^*$.
3: Let $M$ be a perfect matching, with minimum weight, between the vertices in $O$.
4: Let $H$ be the graph induced by $E(T^*) \cup M$ (possibly, $H$ has parallel edges) and $C$ be an
   eulerian cycle in $H$.
5: return the Hamiltonian cycle obtained by considering the vertices in the order they are
   met for the first time in $C$.
```

Theorem 12 (**Christofides 1976**) *Algorithm 12 is a 3/2-approximation algorithm for the problem TSP_{ti}.*

Proof. Note that, by Proposition 1, O has even size and then, it is easy to see that M is well defined (because $|O|$ is even and we are in a clique) and can be computed in polynomial time. By construction, every vertex has even degree in H and so C is well defined and can be computed in polynomial time (Th. 1). Finally, the Hamiltonian cycle returned by the algorithm has weight at most $w(C)$ by the triangular inequality. Hence, Algorithm 12 computes, in polynomial time, a Hamiltonian cycle with weight at most $w(C) = w(T^*) + w(M)$. The result follows from Lemmas 6 and 7. ∎

5 Set Cover

To continue with approximation algorithms, let us consider that a new problem that is not (directly) related to graphs.

The Set Cover problem takes as inputs a ground set (a *universe*) $U = \{e_1, \cdots, e_n\}$ of elements, a set $\mathcal{S} = \{S_1, \cdots, S_m\} \subseteq 2^U$ of subsets of elements and $k \in \mathbb{N}$. The goal is to decide if there exists a set $K \subseteq \{1, \cdots, m\}$ such that $\bigcup\limits_{j \in K} S_j = U$ and $|K| \leq k$. In "optimization" words, the Set Cover problem aims at computing a minimum number of sets in $\mathcal{S}$ covering all elements in U.

As an example, consider a set of persons, each one speaking only its own language (English, French, Spanish...) and a set of translators, each ones speaking several langages (the first translator knows French, Chinese and Russian, the second one knows French and Spanish...). What is the minimum number of translators required to be able to communicate with all persons?

Exercise 13 *Formalize the above paragraph in terms of Set Cover problem (define U and $\mathcal{S}$). Invent another application of the Set Cover problem.*

Without surprise (given the topic of this course), the Set Cover problem is NP-complete [7].

5.1 Relationship with Graphs: Dominating Set and Vertex Cover

Before trying to solve the Set Cover problem, let us discuss its relationship with graphs. This subsection aims at getting a better understanding of Set Cover problem by considering it by different points of view (related to graphs).

Given a graph $G = (V, E)$, a dominating set $D \subseteq V$ is a set of vertices such that $N[D]$[11] $= V$. The minimum size of a dominating set in G is denoted by $\gamma(G)$.

Exercise 14 *Show that, for any graph $G = (V, E)$, $\gamma(G) \leq vc(G)$ (recall that $vc(G)$ is the minimum size of a vertex cover in G). That is, prove that any vertex cover is a dominating set.*

Give a graph G in which $\gamma(G) < vc(G)$.

The minimum Dominating Set (MDS) problem takes a graph $G = (V, E)$ and $k \in \mathbb{N}$ as inputs, and asks whether $\gamma(G) \leq k$. The MDS problem is actually "related" to the Set Cover problem. Precisely, we show below that, any polynomial-time algorithm solving the MDS problem in bipartite graphs may be used to solve the Set Cover problem in polynomial-time.

Let $(U = \{e_1, \cdots, e_n\}, \mathcal{S} = \{S_1, \cdots, S_m\}, k)$ be an instance of the Set Cover problem. Let us define the bipartite graph $G(U, \mathcal{S}) = (A \cup B, E)$ as follows. Let $A = U \cup \{r\}$ and $B = \mathcal{S} \cup \{r'\}$. Let us add an edge between r and every vertex in B, and an edge $\{r, r'\}$. Then, for every $u \in U = A \setminus \{r\}$ and $s \in B = \mathcal{S}$, there is an edge $\{u, s\} \in E$ if and only if $u \in s$.

Lemma 8 *Let $k \in \mathbb{N}$. There exists a Set Cover of $(U, \mathcal{S})$ of size $\leq k$ iff there exists a dominating set of $G(U, \mathcal{S})$ of size $\leq k + 1$.*

Proof. Let $K \subseteq \mathcal{S}$ be a set cover of $(U, \mathcal{S})$. Then, it is easy to check that $K \cup \{r\}$ is a dominating set in $G(U, \mathcal{S})$.

Let $D \subseteq V(G(U, \mathcal{S}))$ be a dominating set in $G(U, \mathcal{S})$. Prove that either r or r' belongs to D, and that, if $r' \in D$, then $(D \setminus \{r'\}) \cup \{r\}$ is a dominating set with size no larger than $|D|$. Hence, we may assume that $r \in D$ and $r' \notin D$. Now, if there is $u \in D \cap U$, let $s \in \mathcal{S}$ such that $\{u, s\} \in E(G(U, \mathcal{S}))$ (i.e., $u \in s$). Show that $(D \setminus \{u\}) \cup \{s\}$ is a dominating set with size no larger than $|D|$. Hence, we may assume that $r \in D$ and $D \subseteq \mathcal{S} \cup \{r\}$. Finally, show that $D \setminus \{r\}$ is a set cover of $(U, \mathcal{S})$. ■

On the other hand, Set Cover is "related" to the Vertex Cover problem. Precisely, any polynomial-time algorithm solving the Set Cover problem can be used to solve the Vertex Cover problem in polynomial-time. For every graph $G = (V, E)$ and, for every $v \in V$, let $E_v = \{uv \in E \mid u \in V\}$ be the set of edges adjacent to v.

Exercise 15 *Let $G = (V, E)$ be a graph. For any $K \subseteq V$, $\{E_v \mid v \in K\}$ is a Set Cover of $(E, \{E_v \mid v \in V\})$ if and only if K is a vertex cover of G.*

[11] In a graph $G = (V, E)$ and given $X \subseteq V$, $N(X) = \{u \in V \setminus X \mid \exists v \in X, \{u, v\} \in E\}$ and $N[X] = N(X) \cup X$.

Remark. The following goes beyond this course (since, I voluntary do not want to go into more details about NP-hardness). However, let us mention that above paragraphs actually consist of reductions leading to hardness proofs. If you already know what it is about, there is no need for more details, otherwise let us just state the following consequence.

Since the Set Cover problem is NP-hard and, since building $G(U, \mathcal{S})$ can be done in polynomial-time, Lemma 8 leads to the following corollary.

Corollary 1 *The Minimum Dominating Set problem is NP-hard even if the input graphs are restricted to the class of bipartite graphs.*

Similarly, since the Vertex Cover problem is NP-hard and, since building $(E, \{E_v \mid v \in V\})$ can be done in polynomial-time, Exercise 15 leads to the following corollary.

Corollary 2 *The Set Cover problem is NP-hard even if the input $(U, \mathcal{S})$ is such that every element of U is in at most 2 sets in $\mathcal{S}$.*

5.2 Greedy $O(\log n)$-approximation

Let $(U = \{e_1, \cdots, e_n\}, \mathcal{S} = \{S_1, \cdots, S_m\} \subseteq 2^U)$ be an instance of the Set Cover problem. Moreover, we consider a cost function $c : \mathcal{S} \to \mathbb{R}^+$ over the sets. This section is devoted to the computation of a minimum-cost solution.

Exercise 16 *Give a "naive" algorithm that computes a set $K \subseteq \{1, \cdots, m\}$ such that $\bigcup_{j \in K} S_j = U$ and of minimum cost in time $O^*(2^m)$*[12].

Next, let us present a greedy algorithm for the Set Cover problem that, while very simple, appears to be an approximation algorithm with best asymptotic approximation ratio. More precisely, the greedy algorithm sequentially adds to the set cover, while the set cover does not cover all elements, a new set with minimum *effective cost* defined as follows. Given $F \subseteq \{1, \cdots, m\}$, $X_F = \bigcup_{i \in F} S_i$ and $j \in \{1, \cdots, m\} \setminus F$ such that $S_j \setminus X_F \neq \emptyset$, let the effective cost of S_j, denoted by $c_{eff}(S_j, F)$, be $\frac{c(S_j)}{|S_j \setminus X_F|}$ (i.e., the cost of S_j is shared among the elements that are not covered by F). Note that, if $c(S_i) = 1$ for all $i \leq m$, a set S_j has minimum effective cost with respect to F iff S_j is a set covering the maximum number of elements uncovered by F.

The intuition behind this algorithm can be stated as follows. At each iteration, when a new set S_i is added in the solution, its effective cost is equally distributed to each element that S_i allows to cover. Precisely, for every element $e \in U$, let us assume that e is covered for the first time when a set S_i is added to the **current** solution K (i.e., consider the value of K before S_i is added to it). Let us say that this element e receives $price(e) = c_{eff}(S_i, \bigcup_{j \in K} S_j) = \frac{c(S_i)}{|S_i \setminus \bigcup_{j \in K} S_j|}$.

Claim 1 *Let K be the solution computed by Algorithm 13.*
Then, $\sum_{e \in U} price(e) = \sum_{j \in K} c(S_j)$.

Let us now prove the main theorem of this section.

Theorem 13 (Chvátal 1979) *Algorithm 13 is a $O(\log n)$-approximation algorithm for the Set Cover problem (with n being the size of the ground-set/universe U).*

[12]A function $f(n) = O^*(g(n))$ if there exists $c > 0$ such that $f(n) = O(g(n)n^c)$, i.e., O^* omits polynomial factors.

Algorithm 13 : Greedy $O(\log n)$-approximation for Set Cover. **[Chvátal 1979]**

```
Require: $(U, \mathcal{S} = \{S_1, \cdots, S_m\} \subseteq 2^U)$.
Ensure: $K \subseteq \{1, \cdots, m\}$ such that $\bigcup_{j \in K} S_j = U$.
1: Let $K = \emptyset$.
2: while $\bigcup_{j \in K} S_j \neq U$ do
3:    Let $i \in \{1, \cdots, m\} \setminus K$ such that
        • $S_i \setminus \bigcup_{j \in K} S_j \neq \emptyset$, and
        • $c_{eff}(S_i, \bigcup_{j \in K} S_j) = \frac{c(S_i)}{|S_i \setminus \bigcup_{j \in K} S_j|}$ is minimum.
4:    $K \leftarrow K \cup \{i\}$.
5: end while
6: return $K$.
```

Proof. Algorithm 13 clearly returns a valid solution in polynomial-time. Let us focus on the approximation ratio.

Let $K = \{j_1, \cdots, j_k\} \subseteq \{1, \cdots, m\}$ be a solution returned by above algorithm. For every $1 \leq i \leq k$, let $X_i = \bigcup_{\ell < i} S_{j_\ell}$ be the set of elements already covered before the i^{th} iteration of the While-loop.

Let K^* be an optimal solution for the Set Cover problem and let $OPT = \sum_{j \in K^*} c(S_j)$. For every $1 \leq i \leq k$, let $F_i \subseteq K^*$ such that, for every $j \in F_i$, S_j is needed to cover some vertex in $E \setminus X_i$ (i.e., for all $j \in F_i$, $\bigcup_{\ell \in F_i \setminus \{j\}} S_\ell$ does not cover $U \setminus X_i$). By the pigeonhole principle:

Claim 2 *For every $1 \leq i \leq k$, there is $j \in F_i$ such that $c_{eff}(S_j, X_i) \leq \frac{OPT}{n - |X_i|}$.*

Proof of Claim. First, let us show that the total cost $\sum_{j \in F_i} c(S_j) \leq OPT$ of the sets in F_i is distributed to the elements in $U \setminus X_i$ (by considering the effective costs of these sets and the prices of elements in $U \setminus X_i$ as in above claim).

Precisely, let us set $\{S_j \mid j \in F_i\} = \{S^*_1, \cdots, S^*_{|F_i|}\}$ and assume that these sets are "added" to the optimal solution in this order. Hence, for all $j \leq |F_i|$, the effective cost $c_{eff}(S^*_j, X_i \cup \bigcup_{\ell < j} S^*_\ell) = \frac{c(S^*_j)}{|S^*_j \setminus (X_i \cup \bigcup_{\ell < j} S^*_\ell)|}$ (Note that, because K^* is an optimal solution, for every $j \in F_i$, there is at least one element covered only by S_j, and so this effective cost is well defined) is equally distributed among the elements in $S^*_j \setminus (X_i \cup \bigcup_{\ell < j} S^*_\ell)$. Hence, the total cost $\sum_{j \in F_i} c(S_j) = \sum_{j \leq |F_i|} c(S^*_j) \leq OPT$ is distributed over the elements of $U \setminus X_i$.

By the pigeonhole principle, some element must receive a price at most $\frac{\sum_{j \in F_i} c(S_j)}{|U \setminus X_i|} \leq \frac{OPT}{|U \setminus X_i|}$. Hence, the set S^*_j, $j \leq |F_i|$, associated to this element has effective cost at most $\frac{OPT}{|U \setminus X_i|} = \frac{OPT}{n - |X_i|}$.
◇

Let us go back to the solution computed by Algorithm 13. Let us assume that all elements of U are covered in the following order $(e_1, \cdots, e_n)$.

Let $j \leq n$ and assume that the element $e_j \in U$ is covered when the set S_{j_i} (for some $i \leq k$) is added to the solution. By claim above, there is $t \in F_i$ such that $c_{eff}(S_t, X_i) \leq \frac{OPT}{n - |X_i|}$. By the definition of the algorithm, this implies that $c_{eff}(S_{j_i}, X_i) \leq c_{eff}(S_t, X_i) \leq \frac{OPT}{n-|X_i|}$. Moreover, $X_i \subseteq \{e_1, \cdots, e_{j-1}\}$ (since e_j is not covered before S_{j_i} is added and so $e_j, \cdots, e_n \notin X_i$) and, therefore, $|X_i| \leq j-1$ and so, $price(e_j) = c_{eff}(S_j, X_i) \leq \frac{OPT}{n-j+1}$.

It follows that $\sum\limits_{j \in K} c(S_j) = \sum\limits_{1 \leq j \leq n} price(e_j) \leq \sum\limits_{1 \leq j \leq n} \frac{OPT}{n-j+1} = OPT \cdot \sum\limits_{1 \leq j \leq n} \frac{1}{j} = O(\log n) \cdot OPT$.

■

It can be proved that no $o(\log n)$-approximation algorithm exists (unless $P = NP$) for the Set Cover problem [8]. That is, Algorithm 13 is asymptotically optimal in terms of approximation ratio (as a function of n). However, if the input $(U, \mathcal{S})$ is such that every element of U appears in at most $f \geq 2$ sets in $\mathcal{S}$, better performances may be achieved (e.g., Vertex Cover, where $f = 2$). For instance, rounding of linear programming relaxation allows the design of a $O(f)$-approximation (e.g., see here).

6 Knapsack and (F)PTAS

The Knapsack problem takes a set of integers $\mathcal{S} = \{w_1, \cdots, w_n\}$ and an integer b as inputs. The objective is to compute a subset $T \subseteq \{1, \cdots, n\}$ of items such that $\sum\limits_{i \in T} w_i \leq b$ and $\sum\limits_{i \in T} w_i$ is maximum. That is, we want to fill our knapsack without exceeding its capacity b and putting the maximum total weight in it.

6.1 (Pseudo-polynomial) Exact Algorithm via dynamic programming

Recall that Dynamic Programming is a generic algorithmic method that consists in solving a problem by combining the solutions of sub-problems.

As an example, the Simple Knapsack Problem consists in computing an optimal solution for an instance $\mathcal{S} = \{w_1, \cdots, w_n\}$ and an integer b. Let $OPT(\mathcal{S}, b)$ denote such a solution. We will compute it using solutions for sub-problems with inputs $\mathcal{S}_i = \{w_1, \cdots, w_i\}$ and $b' \in \mathbb{N}$, for any $i \leq n$ and $b' < b$. That is, we will compute $OPT(\mathcal{S}, b)$ from all solutions $OPT(\mathcal{S}_i, b')$ for $i \leq n$ and $b' < b$.

Algorithm 14 Dynamic programming algorithm for Knapsack

```
Require: A set of integers $\mathcal{S} = \{w_1, \cdots, w_n\}$ and $b \in \mathbb{N}$.
Ensure: A subset $OPT \subseteq \{1, \cdots, n\}$ of items such that $\sum_{i \in T} w_i \leq b$
 1: For any $0 \leq i \leq n$ and any $0 \leq b' \leq b$, let $OPT[i, b'] = \emptyset$;
 2: For any $0 \leq i \leq n$ and any $0 \leq b' \leq b$, let $opt\_cost[i, b'] = 0$;
 3: for $i = 1$ to $n$ do
 4:   for $b' = 1$ to $b$ do
 5:     if $\max\{opt\_cost[i-1, b'-w_i] + w_i, opt\_cost[i-1, b']\} = opt\_cost[i-1, b']$ then
 6:       $OPT[i, b'] = OPT[i-1, b']$
 7:       $opt\_cost[i, b'] = opt\_cost[i-1, b']$
 8:     else
 9:       $OPT[i, b'] = OPT[i-1, b'-w_i] \cup \{i\}$
10:       $opt\_cost[i, b'] = opt\_cost[i-1, b'-w_i] + w_i$
11:     end if
12:   end for
13: end for
14: return $OPT = OPT[n, b]$
```

Theorem 14 *Algorithm 14 computes a optimal solution for the Knapsack problem in time* $O(n \cdot b)$.

Proof. Algorithm 14 consists in two imbricated loops, the first one with $O(n)$ iterations and the second one with $O(b)$ iterations. "Inside" the second loop, there are a constant number of operations (tests, comparisons, arithmetical operations). Hence, its time-complexity is $O(nb)$.

To prove the correctness of Algorithm 14, let us first understand the meaning of $OPT(\mathcal{S}_i, b')$ ($i \leq n, b' \leq b$). The set $OPT(\mathcal{S}_i, b')$ is a combination (a choice/a subset) of elements in $\{1, \cdots, i\}$ that maximizes the weight of the chosen elements such that it does not exceed b'. That is $OPT(\mathcal{S}_i, b') \subseteq \{1, \cdots, i\}$ is such that $opt(\mathcal{S}_i, b') = \sum\limits_{j \in OPT(\mathcal{S}_i, b')} w_j \leq b'$ and, for every $T \subseteq \{1, \cdots, i\}$ with $w(T) = \sum\limits_{j \in T} w_j \leq b'$, then $w(T) \leq opt(\mathcal{S}_i, b')$. The key point is that,

Claim 3 *For every* $1 \leq i \leq n$ *and* $b' \leq b$, $opt(\mathcal{S}_i, b') = \max\{opt(\mathcal{S}_{i-1}, b'), w_i + opt(\mathcal{S}_{i-1}, b' - w_i)\}$.

Proof of Claim. Clearly, $OPT(\mathcal{S}_i, b')$ is obtained either by not taking the i^{th} element, in which case a solution is $OPT(\mathcal{S}_{i-1}, b')$, or by taking the i^{th} element (with weight w_i) and adding to it $OPT(\mathcal{S}_{i-1}, b' - w_i)$. Formally prove this claim by "mimicking" the proof of Theorem 4. $\diamond$

Then, the correctness easily follows by induction. Indeed, by Lines 1-2, $OPT[0, b'] = OPT(\mathcal{S}_0, b') = \emptyset$ and $opt\_cost[0, b'] = opt(\mathcal{S}_0, b') = 0$ for every $b' \leq b$ (setting $\mathcal{S}_0 = \emptyset$). Then, by induction on $i \leq n$, let us assume that, for every $b' \leq b$, $OPT[i, b'] = OPT(\mathcal{S}_i, b')$ and $opt\_cost[i, b'] = opt(\mathcal{S}_i, b')$. Then, by Lines 5-10, $opt\_cost[i+1, b'] = \max\{opt\_cost[i, b'], w_i + opt\_cost[i, b' - w_i]\}$. By the induction hypothesis, $opt\_cost[i, b'] = opt(S_i, b')$ and $opt\_cost[i, b' - w_i] = opt(S_i, b' - w_i)$. By the claim, $opt(\mathcal{S}_{i+1}, b') = \max\{opt(\mathcal{S}_i, b'), w_i + opt(\mathcal{S}_i, b' - w_i)\}$. Hence, $opt\_cost[i+1, b'] = opt(\mathcal{S}_{i+1}, b')$ and similarly, $OPT[i+1, b'] = OPT(\mathcal{S}_{i+1}, b')$.

So, the algorithm returns $OPT[n, b] = OPT(\mathcal{S}_n, b)$ which is, by definition, an optimal solution. ■

Exercise 17 *Explain that we may assume that* $\max_i w_i \leq b$ *and* $b \leq \sum_i w_i$ *since, otherwise, the instance may be simplified.*

Prove that, if $\max_i w_i \leq b \leq \sum_i w_i$, *Algorithm 14 proceed in polynomial-time if* $\max_i w_i$ *is polynomial in* n *but exponential if* $\max_i w_i$ *is exponential in* n.

Actually, the KNAPSACK Problem is an example of *Weakly NP-hard* (roughly, it can be solved in polynomial-time if the weights are polynomial). Typically (informally), a weakly NP-hard problem takes a set of n integers as inputs and can be solved in time polynomial in the number of integers (n) and in the maximum value of the integers (**pseudo-polynomial** algorithm) but, if the values of the integers are exponential in the number n of integers, we do not know any polynomial-time (in n) algorithm to solve it.

6.2 Greedy 2-Approximation and PTAS

Note that Algorithm 15 proceeds in a greedy way: it takes one by one the possible items (in non increasing order of their weights) and simply add them if they fit in the sack.

Theorem 15 *Algorithm 15 is a* 2*-approximation algorithm for the* KNAPSACK *problem.*

Proof. In line 3, there is a sorting of n integers (time $O(n \log n)$), then there is a loop with n iterations with, for each iteration, a constant number of operations. Hence, the algorithm

Algorithm 15 Greedy 2-approximation for KNAPSACK

```
Require: A set of integers $\mathcal{S} = \{w_1, \cdots, w_n\}$ and $b \in \mathbb{N}$.
Ensure: A subset $T \subseteq \{1, \cdots, n\}$ of items such that $\sum_{i \in T} w_i \leq b$
 1: $T = \emptyset$
 2: $total\_weight = 0$
 3: Sort $\mathcal{S}$ such that $w_1 \geq w_2 \geq \cdots \geq w_n$.
 4: for $i = 1$ to $n$ do
 5:   if $total\_weight + w_i \leq b$ then
 6:     Add $i$ to $T$
 7:     Add $w_i$ to $total\_weight$
 8:   end if
 9: end for
10: return $T$
```

has complexity $O(n \log n)$. Moreover, it clearly computes a valid solution. Hence, it only remains to prove the approximation ratio. Let $OPT = \sum_{i \in S^*} w_i$ be the value of an optimal solution $S^* \subseteq \{1, \cdots, n\}$. Note that, in contrast with previous examples, this is a maximization problem. Hence, we aim at proving that $\frac{OPT}{2} \leq ValueOfComputedSolution \leq OPT$.

To prove the approximation ratio, let $T \subseteq \{1, \cdots, n\}$ be the computed solution and let $\sum_{i \in T} w_i = SOL$ be its value. Clearly, $\sum_{1 \leq i \leq j} w_i \leq SOL \leq OPT \leq b$. Let $j \geq 1$ be the smallest integer such that $j+1$ is NOT in T. By definition of the algorithm, $\sum_{1 \leq i \leq j+1} w_i = w_{j+1} + \sum_{1 \leq i \leq j} w_i > b$ and, because the w_i's are ordered, $w_{j+1} \leq \min_{1 \leq i \leq j} w_i = w_j$. Finally, $\min_{1 \leq i \leq j} w_i \leq \frac{\sum_{1 \leq i \leq j} w_i}{j}$ (because the average of $w_1, \cdots, w_j$ cannot be less than the minimum w_i).

It follows that $\sum_{1 \leq i \leq j} w_i \leq SOL \leq OPT \leq b < w_{j+1} + \sum_{1 \leq i \leq j} w_i \leq (1 + 1/j) \sum_{1 \leq i \leq j} w_i \leq (1 + 1/j) SOL \leq 2SOL$ (because $j \geq 1$) and obviously $2SOL \leq 2OPT$.

Summing up, $OPT \leq 2SOL \leq 2OPT$, i.e., $\frac{OPT}{2} \leq SOL \leq OPT$. ■

A **polynomial-time approximation scheme (PTAS)** is a family of algorithms which take an instance of an optimization problem and a parameter $\epsilon > 0$ and, in polynomial time in the size of the instance (not necessarily in ϵ), produces a solution that is within a factor $1 + \epsilon$ of being optimal.

That is, when ϵ tends to 0, the solution tends to an optimal one, while the complexity increases (generally, the complexity is of the form $O(n^{f(1/\epsilon)})$ for some function f).

We now present a PTAS algorithm for the Knapsack Problem. Algorithm 16 generalizes the previous greedy algorithm. Instead of computing a greedy solution "from scratch", Algorithm 16 depends on some fixed integer $k = \lceil 1/\epsilon \rceil$. For every subset X of size at most k, Algorithm 16 starts from X and uses the greedy algorithm to complete (try to improve) X. Then, Algorithm 16 keeps the best solution that it met in this way. Intuitively, Algorithm 16 aims at using the greedy algorithm but starting from a "best" partial solution (since it checks all subsets of size $\leq k$, in particular, there is one iteration for which it will consider the heaviest k elements of an optimal solution, and it will only need to improve this "already good" partial solution "not too badly"). Hence, the larger k (the smaller ϵ), the best will be the obtained solution (the approximation ratio) but the higher will be the time-complexity (since we need at least to check all subsets of size at most k).

Algorithm 16 PTAS for the KNAPSACK PROBLEM

```
Require: A set of integers $\mathcal{S} = \{w_1, \cdots, w_n\}$, $b \in \mathbb{N}$ and a real $\epsilon > 0$.
Ensure: A subset $T \subseteq \{1, \cdots, n\}$ of items such that $\sum_{i \in T} w_i \leq b$
 1: $best = \emptyset$
 2: $best\_cost = 0$
 3: $k = \lceil 1/\epsilon \rceil$
 4: Sort $\mathcal{S}$ such that $w_1 \geq w_2 \geq \cdots \geq w_n$.
 5: for Any subset $X \subseteq \mathcal{S}$ of size at most $k$ do
 6:                                   //Complete $X$ using the Greedy Algorithm. That is:
 7:   Let $T = X$ and $k' = |X| \leq k$.
 8:   Let $total\_weight = \sum_{i \in X} w_i$
 9:   Up to renaming the elements in $\mathcal{S} \setminus X$, let us assume that $\mathcal{S} \setminus X = \{w_1, \cdots, w_{n-k'}\}$ with
      $w_1 \geq w_2 \geq \cdots \geq w_{n-k'}$.
10:   for $i = 1$ to $n - k'$ do
11:     if $total\_weight + w_i \leq b$ then
12:       Add $i$ to $T$
13:       Add $w_i$ to $total\_weight$
14:     end if
15:   end for
16:   if $total\_weight > best\_cost$ then
17:     Replace $best$ by $T$
18:   end if
19: end for
20: return $T$
```

Theorem 16 *Algorithm 16 is a PTAS for the* KNAPSACK *problem.*

Proof. Algorithm 16 computes a valid solution in time-complexity $O(n^{\lceil 1/\epsilon \rceil + 1})$. Indeed, there are $O(n^k)$ subsets of size at most k in a ground-set with n elements.

Then, Algorithm 16 is a $(1+\epsilon)$-approximation algorithm for the KNAPSACK problem. Indeed, consider an optimal solution OPT and let $X^* = \{i_1, \cdots, i_k\}$ be the k items with largest weight in OPT (show that, if OPT consists of less than k items, then Algorithm 16 computes an optimal solution). Consider the iteration of Algorithm 16 when it considers X^*. The proof is a (not difficult) adaptation of the proof of Theorem 15. ■

Actually, we can do better. Indeed, the KNAPSACK Problem admits a **fully polynomial-time approximation scheme (FPTAS)** algorithm, that is an algorithm that computes a solution that is within a factor $1 + \epsilon$ of being optimal **in time polynomial both in the size of the instance AND in** $1/\epsilon$.

Part III
Parameterized Algorithms [4]

7 Introduction to Parameterized Algorithms (with Vertex Cover as toy example)

Until now, we have evaluated the "quality/efficiency" of algorithms (resp., the "difficulty" of problems we have met) in function of the size s (generally, in graph's problems, the number of vertices and/or edges) of the instances. Very roughly, a problem is considered as “easy” if there exists an algorithm for solving it in time polynomial in s. If no such algorithm is known (all known algorithms are exponential in s), the problem is said “difficult” (NP-hard).

On the other hand, we have seen that problems that are “difficult” in general may be “easy” in some particular classes of instances. For instance, the Vertex Cover problem is NP-hard in general graphs but can be solved in polynomial-time when restricted to bipartite graphs (Theorem 5).

Very roughly, the Parameterized Complexity aims at evaluating the complexity (here we only focus on time-complexity) of an algorithm/a problem not only as a function of the size of the input but also as a function of other parameters of the instance/problem. For instance, in graphs, appropriated parameters may be the diameter, the maximum degree, the minimum vertex cover (i.e., in the case of the Vertex Cover problem, **the size of the solution itself**), etc.

7.1 First Approach: deciding if $vc(G) \leq k$?

Recall that a vertex cover $K \subseteq V$ of a graph $G = (V, E)$ is a subset of vertices such that $K \cap e \neq \emptyset$ for all $e \in E$. Moreover, recall that $vc(G)$ denotes the minimum size of a Vertex Cover in G. In Section 2.3, we have already seen that the following algorithm computes a minimum-size Vertex Cover in time $O^*(2^{|V|})$.

Algorithm 7 Naive Algorithm for Minimum Vertex Cover (reminder)

```
Require: A graph $G = (V, E)$.
Ensure: A minimum Vertex Cover of $G$.
1: $K \leftarrow V$.
2: for every $S \subseteq V$ do
3:   if $S$ is a vertex cover of $G$ and $|S| < |K|$ then
4:     $K \leftarrow S$.
5:   end if
6: end for
7: return $K$
```

Let us (slightly) simplify the question. Let $k \in \mathbb{N}$ be a **fixed** parameter. Instead of looking for $vc(G)$ (or a Vertex Cover of minimum size), let us “only” ask whether G has some Vertex Cover of size $\leq k$ (we may also ask to compute a minimum Vertex Cover of G given that we already know that $vc(G) \leq k$).

Algorithm 17 1^{st} Algorithm to decide if $vc(G) \leq k$, where $k \in \mathbb{N}$ is a fixed parameter.

```
Require: A graph $G = (V, E)$.
Ensure: A minimum Vertex Cover of $G$ (if $vc(G) \leq k$) or $V$ (if $vc(G) > k$).
1: $K \leftarrow V$.
2: for every $S \subseteq V$, $|S| \leq k$ do
3:   if $S$ is a vertex cover of $G$ and $|S| < |K|$ then
4:     $K \leftarrow S$.
5:   end if
6: end for
7: return $K$
```

Exercise 18 *Show that Algorithm 17 has time-complexity $O^*(|V|^k)$.*

Compare $O^*(|V|^k)$ and $O^*(2^{|V|})$ when, for instance, $|V| = 10^4$ and $k = 10$.

Hence, if we *a priori* know that the graph into consideration has "small" vertex cover (at most k), the above algorithm is much more efficient than the previous one. We show below that even better algorithm can be designed. For this purpose, we need the following lemma:

Lemma 9 *Let $G = (V, E)$ be a graph and $\{x, y\} \in E$. $vc(G) = \min\{vc(G \setminus x), vc(G \setminus y)\} + 1$*
Intuition: for any edge xy, any minimum vertex cover contains at least one of x or y

Proof. Let $S \subseteq V$ be any vertex cover of $G \setminus x$. Then $S \cup \{x\}$ is a vertex cover of G. Hence $vc(G) \leq vc(G \setminus x) + 1$ (symmetrically for $G \setminus y$)

Let $S \subseteq V$ be any vertex cover of G. At least one of x or y is in S. If $x \in S$ then $S \setminus x$ vertex cover of $G \setminus x$. Hence $vc(G \setminus x) \leq vc(G) - 1$. Otherwise, if $y \in S$, then $S \setminus y$ vertex cover of $G \setminus y$ and $vc(G \setminus y) \leq vc(G) - 1$. ■

Algorithm 18 Branch & Bound Algo. to decide if $vc(G) \leq k$, where $k \in \mathbb{N}$ is a fixed parameter.

```
Require: A graph $G = (V, E)$ and an integer $\ell \leq k$.
Ensure: The minimum size of a Vertex Cover of $G$ if $vc(G) \leq \ell$ or $\infty$ otherwise.
1:  if $\ell = 0$ and $|E| > 0$ then
2:    return $\infty$
3:  else
4:    if $|E| = 0$ then
5:      return 0
6:    else
7:      Let $\{u, v\} \in E$ be any edge
8:      Let $x =$ Algorithm 18$(G \setminus u, \ell - 1)$ and $y =$ Algorithm 18$(G \setminus v, \ell - 1)$
9:      return $\min\{x, y\} + 1$
10:   end if
11: end if
```

Exercise 19 *Using Lemma 9, prove the correctness of Algorithm 18.*
Show that the recursive depth is at most k. Deduce that Algorithm 18 has time-complexity $O(2^k|E|)$.

Adapt Algorithm 18 to return a minimum vertex cover of G if $vc(G) \leq k$.

Lemma 10 *Let $G = (V, E)$ be any simple graph. If $vc(G) \leq k$, then $|E| \leq k(|V| - 1)$.*

Proof. Let $K \subseteq V$ be a vertex cover with $|K| \leq k$. Note that $G - K$ induces a stable set. Hence, every edge of G is incident to a vertex in K (it is the definition of a vertex cover). Finally, each vertex in K is adjacent to at most $|V| - 1$ edges. ■

It follows that:

Corollary 3 *Algorithm 18 has time-complexity $O(k2^k \cdot |V|)$.*

Hence, Algorithm 18 is linear in the order of the graph! Note that Vertex Cover is NP-hard, but we proved that the combinatorial complexity does not come from the order of the graph but from its structure. That is, the Vertex Cover problem can be solved in linear time (in the size of the input) in the class of graphs with bounded (fixed) minimum size of a vertex cover. Compare the complexity of Algorithm 18 with the complexity of Algorithm 17 when, for instance, $|V| = 10^4$ and $k = 10$.

The key difference between the time-complexity of Algorithm 17, $O(|E||V|^k)$, and the one of Algorithm 18, $O(k2^k \cdot |V|)$, that are both polynomial in $|V|$ and exponential in k, is that, in the latter case, the polynomial on $|V|$ does not depend on k. That is, in Algorithm 18, the dependencies in k and $|V|$ are "separated". Such an algorithm is called a *Fixed Parameter Tractable (FPT)* algorithm, parameterized by the size of the solution (the size of the vertex cover).

7.2 Fixed Parameter Tractable (FPT) Algorithms and Kernelization

We don't aim at giving a formal definition of parameterized complexity and refer to, e.g., [4] for more precise definition. As usual, we moreover focus on graphs.

Let $\mathcal{G}$ be the set of all graphs. A *graph parameter* is any function $p : \mathcal{G} \to \mathbb{N}$ (e.g., diameter, maximum degree, minimum size of a vertex cover, minimum size of a dominating set, etc.).

Roughly, a parameterized problem (Π, p) is defined by a problem Π (here on graphs) and a parameter p. The key point is to understand the behaviour of the time-complexity of an algorithm for solving Π not only as a function of n, the size of the instance, but also as a function of the value of the parameter p.

An algorithm $\mathcal{A}$ for solving (Π, p) is said Fixed Parameter Tractable (FPT) if there exists a computable function $f : \mathbb{N} \to \mathbb{N}$, such that the time-complexity of $\mathcal{A}$ can be expressed as $O(f(p(G))n^{O(1)})$, where n is the size of the input graph G. Note that the polynomial in n is independent on $p(G)$ and f depends only on the parameter $p(G)$. A parameterized problem (Π, p) is FPT if it admits a FPT algorithm, parameterized by p, for solving it.

For instance, Algorithm 18 shows that the Vertex Cover problem is FPT when parameterized by the size of the solution. Another example is any FPTAS algorithm, that can be seen as a FPT (approximation) algorithm parameterized by $1/\epsilon$.

To conclude this subsection, let us present a particular kind of FPT algorithms called Kernelization algorithms. A natural way to tackle difficult problems is to try to reduce the size of the input (e.g., in the case of graphs, to "limit" the problem to the connected components of a graph, or to its 2-connected components...).

Precisely, given a parameterized problem (Π, p), a kernelization algorithm replaces an instance $(I, p(I))$ by a "reduced" instance $(I', p'(I'))$ (called problem kernel) such that

1. $p'(I') \leq g(p(I))$, $|I'| \leq f(p(I))$ for some computable functions f and g **only** depending on $p(I)$ (not on $|I|$)[13],

2. $(I, p(I)) \in \Pi$ **if and only if** $(I', p'(I')) \in \Pi$, and

3. reduction from $(I, p(I))$ to $(I', p'(I'))$ has to be computable in polynomial time (in $|I| + p(I)$).

Hence, if a parameterized problem (Π, p) admits a Kernelization algorithm that transforms a n-node graph G and parameter $p(G) = k$ into an equivalent graph G' with size $f(n)$ (for some computable function f) and parameter $k' = p'(G') \leq g(k)$ (for some computable function g) in time $(n + p(G))^{O(1)}$, then (Π, k) admits a FPT algorithm with time complexity $n^{O(1)} + O(2^{f(g(k))})$: first reduce G to G' and then exhaustive search in G' with parameter $k' \leq g(k)$. Note the difference between this complexity and the one of Algorithm 18 (also FPT): in the latter one, the terms depending on k and n are multiplied, while here it is a sum. More generally, it is easy to prove (while a bit counter-intuitive) that:

[13] A kernel G' is said linear (resp., quadratic, single exponential...) if $|G'| = O(k)$ (resp., $|G'| = O(k^2)$, $|G'| = O(2^k)$,...).

Theorem 17 (Bodlaender *et al.* 2009) *A parameterized problem is FPT if and only if it is decidable and has a kernelization algorithm.*

7.3 A first Kernelization Algorithm for Vertex Cover

We aim at improving Algorithm 18.

Lemma 11 *Let $G = (V, E)$ be a graph and $v \in V$ with degree $> k$. Then v belongs to every vertex cover K of size at most k*

Proof. Indeed, if $v \notin K$, all its neighbors must belong to it and $|K| > k$. ■

Lemma 12 *Let $G = (V, E)$ be a graph. If $vc(G) \leq k$ and no vertex of G has degree $> k$. Then $|E| \leq k^2$*

Proof. Each of the $\leq k$ vertices of a Vertex Cover covers $\leq k$ edges (see proof of Lem. 9). ■

The following algorithm to decide if $vc(G) \leq k$ proceeds as follows. While there is a "high" degree node, add it to the solution. When there are no such nodes, either it remains too much edges to have a small vertex cover. Otherwise, apply brute force algorithm (e.g., Algorithm 18) to the remaining "small" graph.

Algorithm 19 Kernelization Alg. to decide if $vc(G) \leq k$, where $k \in \mathbb{N}$ is a fixed parameter.

```
Require: A graph $G = (V, E)$ and an integer $\ell \leq k$.
Ensure: The minimum size of a Vertex Cover of $G$ if $vc(G) \leq \ell$ or $\infty$ otherwise.
 1: Let $I \subseteq V$ be the set of isolated vertices in $G$. Remove $I$ from $G$
 2: if $|E| = 0$ then
 3:   return 0
 4: else
 5:   if $\ell = 0$ then
 6:     return $\infty$
 7:   else
 8:     if $G$ has no vertex of degree $> \ell$ and $|E| > \ell^2$ then
 9:       return $\infty$
10:     else
11:       if $G$ has no vertex of degree $> \ell$ then
12:         return Algorithm 18$(G, \ell)$
13:       else
14:         Let $v$ be a vertex of degree $> \ell$
15:         return Algorithm 19$(G \setminus v, \ell - 1) + 1$
16:       end if
17:     end if
18:   end if
19: end if
```

Exercise 20 *Using Lemmas 11 and 12, prove the correctness of Algorithm 19.*
Show that Algorithm 19 has time-complexity $O(2^k k^2 + k|V|)$. Adapt Algorithm 19 to return a minimum vertex cover of G if $vc(G) \leq k$.

Note that previous algorithm relies on a quadratic kernel. Compare the complexity of Algorithm 19 with the complexity of Algorithm 18 when, for instance, $|V| = 10^4$ and $k = 10$.

8 Toward tree-decompositions, Graph Minor Theory and beyond

In this last section, we focus on particular graph classes. Precisely, we will start with trees, then generalize the proposed method to *k-trees* and then to graphs with bounded *treewidth*. We will then conclude with *planar* graphs and beyond. Along the way, we will continue to use our favorite problem, namely Vertex Cover, as an illustrative example.

8.1 Minimum (weighted) Vertex Cover in trees

Let us start with the problem of computing a minimum-size vertex cover (as it has been studied many times above). That is, given a tree $T = (V, E)$, the goal is to compute a set $K \subseteq V$, such that $\forall e \in E$, $e \cap K \neq \emptyset$, and $|K|$ is minimum subject to this property. Recall that $vc(T)$ denotes the minimum size of a vertex cover in T.

Prove that any tree is a bipartite graph (e.g., use a BFS). So, by König's theorem (Th. 5) and, e.g., the Hungarian method (Th. 4), $vc(T)$ can be computed in polynomial time in any tree T. We aim at giving here a simpler algorithm in the case of trees. It is based on the following lemma whose proof is left to the reader.

Lemma 13 *Let $G = (V, E)$ be any graph with a vertex v of degree 1. Let u be the unique neighbor of v and let K be a vertex cover of G.*
Then, $K \cap \{u, v\} \neq \emptyset$, and $K' = (K \setminus \{v\}) \cup \{u\}$ is a vertex cover of G such that $|K'| \leq |K|$.

Notation. Let $T = (V, E)$ be a tree and $r \in V$ be any vertex. Let us denote by T_r the tree T rooted in r. For any $v \in V$, the children of v in T_r are the neighbors of v whose distance to r is greater than $dist(r, v)$. The parent of v in T_r (if v is not the root r) is the unique neighbor of v that is not a children of v. The descendants of v in T_r are all vertices w such that v is an internal vertex of the path between r and w in T (such a path is unique by Exercise 2). Finally, the subtree T_v rooted at v in T_r is the subtree induced by v and its descendants. A leaf in a rooted tree is any vertex $v \neq r$ with degree 1.

Algorithm 20 Greedy Algorithm for Minimum Vertex Cover in trees

```
Require: A tree $T = (V, E)$ rooted in any arbitrary vertex $r \in V$.
Ensure: A minimum Vertex Cover of $T$.
1: if $E = \emptyset$ then
2:     return $\emptyset$
3: else
4:     Let $v \in V$ be any non-leaf vertex maximizing the distance with $r$ (possibly $r = v$).
5:     Let $T'$ be the tree (rooted in $r$) obtained from $T$ by removing $v$ and all its children.
                    // Prove that children of $v$ are leaves and so that $T'$ is a tree (rooted in $r$)
6:     return $\{v\} \cup Algorithm\ 20(T')$.
7: end if
```

Theorem 18 *Algorithm 20 computes a minimum Vertex Cover of any tree T in linear time.*

Proof. Let us first prove its correctness. Let T_r be a rooted tree, v be a non-leaf maximizing the distance with r (possibly $v = r$) and T' be defined as in Algorithm 20 (i.e., T' is the tree rooted in r obtained from T by removing v and all its leaves neighbors). We claim that $vc(T) = 1 + vc(T')$. Indeed, if K' is a minimum vertex cover of T', then $K' \cup \{v\}$ is a vertex cover of T of size $|K'|+1 = vc(T')+1 \geq vc(T)$. On the other hand, let K be a minimum vertex cover of T. By Lemma 13, we may assume that $v \in K$. Hence, $K' = K \setminus \{v\}$ is a vertex cover of T' and so $|K'| = vc(T) - 1 \geq vc(T')$. By induction on $|V(T)|$, Algorithm 20 returns a vertex cover of size $1 + vc(T')$. By the claim, it is then an optimal vertex cover of size $vc(T)$. Hence, Algorithm 20 is correct.

For the time-complexity, there at most $|V(T)|$ recursive calls (since $|V(T')| < |V(T)|$). The main (most time consuming) step at each call consists in finding the vertex v. It can be done in constant time by, for instance (without more details), using a pre-processing that orders the vertices of T_r by non-increasing distance from r, e.g., by a "BFS-like" ordering: first the leaves, then the parents of only leaves, then the parents of only parents of only leaves and so on (this is called a topological ordering of the vertices of T_r). ∎

Hence, minimum (size) Vertex Cover is almost trivial in trees, so let us "complexify" a bit the problem.

Minimum weighted Vertex Cover in trees. Given a tree $T = (V, E)$ with weight function $w : V \to \mathbb{R}^+$ on the vertices, the goal is to compute a set $K \subseteq V$, such that $\forall e \in E$, $e \cap K \neq \emptyset$, and $w(K) = \sum\limits_{v \in K} w(v)$ is minimum subject to this property. Let $vc(T, w)$ denote the minimum weight of a vertex cover in (T, w).

Exercise 21 *Give an example of a weighted tree (you may restrict your example to be a star, i.e., a tree with at most one vertex with degree > 1) such that $vc(T) = 1$ (minimum size) and the number of vertices in a minimum weighted Vertex Cover is arbitrary large.*

In what follows, we present a linear-time dynamic programming algorithm to compute $vc(T, w)$ and a vertex cover with this weight.

Let (T_r, w) be a weighted rooted tree (not reduced to a single vertex) and let $r_1, \cdots, r_d$ be the children of r. For every $1 \leq i \leq d$, let T_i be the subtree of T_r rooted in r_i. The proof of the next lemma is left to the reader.

Lemma 14 *Let K be any vertex cover of T_r. Either $r \in K$ and then, for every $1 \leq i \leq d$, $K \cap V(T_i)$ is a vertex cover of T_i. Or $r \notin K$ and then, for every $1 \leq i \leq d$, $K \cap V(T_i)$ is a vertex cover of T_i with $r_i \in K \cap V(T_i)$.*

Previous lemma suggests that, given a weighted rooted tree $(T_r = (V, E), w)$, our dynamic programming algorithm will proceed bottom-up (from leaves to root) by, for every vertex $v \in V$, keeping track of $vc(T_v, w)$ (the minimum weight of a vertex cover of (T_v, w)) but also of $vc'(T_v, w)$ defined as the minimum weight of a vertex cover of T_v containing v (i.e., $vc'(T_v, w)$ is the minimum weight of any vertex cover among all vertex covers of (T_v, w) containing v).

From Lemma 14 and by induction on $|V(T)|$, the proof of next theorem easily follows.

Theorem 19 *Algorithm 21 computes a minimum weight Vertex Cover of any vertex-weighted tree (T, w) in linear time.*

8.2 2-trees

In the sequels, we will extend Algorithm 21 to some graph class generalizing trees (graphs with bounded *treewidth*). To make the next algorithms easier to understand, let us first go one step further. For this purpose, a key notion is the one of *separators* in graphs. Given a graph $G = (V, E)$ and $X, Y \subseteq V$, $X \cap Y = \emptyset$, a set $S \subseteq V \setminus (X \cup Y)$ is a (X, Y)-separator in G if, for

Algorithm 21 Dynamic Programming Algorithm for Minimum weight Vertex Cover in trees

```
Require: A weighted tree $(T_r = (V,E), w : V \to \mathbb{R}^+)$ rooted in any arbitrary vertex $r \in V \neq \emptyset$.
Ensure: $(K, K')$ where $K$ is a minimum Vertex Cover of $(T_r, w)$ (of weight $vc(T,w)$) and $K'$
    is a minimum Vertex Cover of $(T_r, w)$ containing $r$ (of weight $vc'(T_r, w)$)
 1: if $V = \{r\}$ then
 2:   return $(\emptyset, \{r\})$                    // of weight respectively $0$ and $w(r)$
 3: else
 4:   Let $r_1, \cdots, r_d$ be the children of $r$ and, for every $1 \leq i \leq d$, let $T_i$ be the subtree of $T_r$
      rooted in $r_i$.
 5:   for $i = 1$ to $d$ do
 6:     Let $(K_i, K'_i) = Algorithm\ 21(T_i, w_{|V(T_i)})$
              //$K_i$ is a minimum weight vertex cover of $T_i$, i.e., of weight $vc(T_i, w_{|V(T_i)})$
              //$K'_i$ is a minimum weight vertex cover of $T_i$ containing $r_i$, i.e., of weight
        $vc'(T_i, w_{|V(T_i)})$
 7:   end for
 8:   Let $K' = \{r\} \cup \bigcup_{1 \leq i \leq d} K_i$.      // Show it is a min. weight VC of $(T,w)$ containing $r$
 9:   Let $K'' = \bigcup_{1 \leq i \leq d} K'_i$.      // Show it is a min. weight VC of $(T,w)$ not containing $r$
10:   Let $K \in \{K', K''\}$ such that $w(K) = \min\{w(K'), w(K'')\}$.
                                          // Show it is a min. weight VC of $(T,w)$
11:   return $(K, K')$.
12: end if
```

every $u \in X$ and $v \in Y$, every u,v-path goes through a vertex in S (equivalently, u and v are in distinct connected components of $G[V \setminus S]$). A set $S \subset V$ is a separator in G if there exist $u, v \in V \setminus S$ such that S is a $(\{u\},\{v\})$-separator (or u,v-separator).

The class of 2-trees is defined recursively as follows. A complete graph (clique) K_3 with 3 vertices (*triangle*) is a 2-tree. Given any 2-tree H with some edge $\{u,v\} \in E(H)$, the graph obtained from H by adding a new vertex x adjacent to u and v is a new 2-tree. The recursive construction of a 2-tree naturally leads to the definition of a corresponding "building tree".

Given a 2-tree $G = (V,E)$, a tree-decomposition $(T, \mathcal{X} = \{X_t \mid t \in V(T)\})$ of G is defined recursively as follows. If $G = K_3$ is reduced to a clique with vertices u, v and w, its corresponding tree-decomposition consists of a tree T reduced to a single node t_0 and $\mathcal{X} = \{X_{t_0} = \{u,v,w\} \subseteq V\}$. If G is obtained from a 2-tree H, with $\{u,v\} \in E(H)$, by adding a vertex x adjacent to u and v, a tree-decomposition $(T_G, \mathcal{X}_G = \{X^G_t \mid t \in V(T_G)\})$ of G is obtained from any tree-decomposition $(T_H, \mathcal{X}_H = \{X^H_t \mid t \in V(T_H)\})$ of H by considering any node $s \in V(T_H)$ such that $u, v \in X^H_s$ (show that such a node exists by induction on $|V|$) and build T_G from T_H by adding a new node n_x adjacent to s, and $\mathcal{X}_G = \mathcal{X}_H \cup \{X_{n_x} = \{u,v,x\} \subseteq V\}$. Note that a 2-tree may admit several decompositions.

Intuitively, each node of T_G corresponds to a subset of vertices of G inducing a maximal clique (triangle) of G. Hence, $(T_G, \mathcal{X}_G)$ "organizes" the triangles of G in a tree-like fashion while satisfying some "connectivity properties" as described by the first 2 items of Lemma 15.

Lemma 15 *Let $G = (V,E)$ be a 2-tree, $(T, \mathcal{X})$ be a tree-decomposition of G and $e = \{u,v\} \in E(T)$. Let T_u (resp. T_v) be the subtree of $T \setminus e = (V(T), E(T) \setminus \{e\})$ containing u (resp., v) and $G_u = G[\bigcup_{t \in V(T_u)} X_t]$ and $G_v = G[\bigcup_{t \in V(T_v)} X_t]$. Then,*

- *$S = X_u \cap X_v = \{x,y\}$ where $\{x,y\} \in E(G)$;*

- *S separates $V(G_u) \setminus S$ from $V(G_v) \setminus S$ (every path from a vertex in $V(G_u) \setminus S$ to a vertex in $V(G_v) \setminus S$ goes through S);*

  // in particular, there are no edges between $V(G_u) \setminus S$ and $V(G_v) \setminus S$.

Moreover, let $Q \subseteq S \setminus \{\emptyset\}$. Prove that, because S is a $(V(G_u) \setminus S, V(G_v) \setminus S)$-separator:

- *If K is a vertex cover of G with $K \cap S = Q$, then $K' = K \cap V(G_u)$ is a vertex cover of G_u with $K' \cap S = Q$.*
- *If K is a vertex cover of G_u with $K \cap S = Q$, then there exists a vertex cover K' of G such that $K = K' \cap V(G_u)$ (and, in particular, $K' \cap S = Q$).*

The key points are that Vertex Cover is a "local" problem and that 2-trees have small-size separators. Intuitively, extending a vertex cover K of G_u to some vertex cover of G does not depend on the whole K but only on $K \cap S$ where S is a separator between G_u and the reminding of G. When $|S|$ is "small", this can be done efficiently.

Algorithm 22 Dynamic Programming Algorithm for Minimum size Vertex Cover in 2-trees

```
Require: A 2-tree $G = (V, E)$ and a tree-decomposition $(T_r, \mathcal{X})$ of $G$ rooted in some vertex $r \in V(T_r)$.
Ensure: For every $Q \subseteq X_r$, a minimum-size Vertex Cover $K_Q$ of $G$ such that $K_Q \cap X_r = Q$ and $K_Q = \infty$ if no vertex cover $K$ of $G$ is such that $K \cap X_r = Q$.
1: if $V(T_r) = \{r\}$ and $X_r = \{u, v, w\}$ then
2:   return $(K_Q)_{Q \subseteq X_r}$ with $K_Q = Q$ if $|Q| > 1$ and $K_Q = \infty$ otherwise.
3: else
4:   Let $r_1, \cdots, r_d$ be the children of $r$ and, for every $1 \leq i \leq d$, let $T_i$ be the subtree of $T_r$ rooted in $r_i$. Let $\mathcal{X}_i = \{X_t \in \mathcal{X} \mid t \in V(T_i)\}$ and let $G_i = G[\bigcup_{X \in \mathcal{X}_i} X]$ be the subgraph induced by the vertices in the sets in $\mathcal{X}_i$.
5:   for $i = 1$ to $d$ do
6:     Let $(K^i_{Q'})_{Q' \subseteq X_{r_i}} = Algorithm\ 22(G_i, (T_i, \mathcal{X}_i))$
7:   end for
8:   for $Q \subseteq X_r$ do
9:     if $|Q| \leq 1$ then
10:      Let $K_Q = \infty$
11:    else
12:      for $i = 1$ to $d$ do
13:        Let $Q_i \subseteq X_{r_i}$ be such that $Q_i \cap X_r = Q \cap X_{r_i}$ and, among such sets, $|K^i_{Q_i}|$ is minimum.   //abusing notations, $|K| = \infty$ if $K = \infty$
14:      end for
15:      Let $K_Q = Q \cup \bigcup_{i=1}^{d} K^i_{Q_i}$.
16:    end if
17:  end for
18:  return $(K_Q)_{Q \subseteq X_r}$.
19: end if
```

Theorem 20 *Algorithm 22 is correct and has time-complexity $O(n)$ where n is the number of vertices of G.*

Proof. The proof of correctness is by induction on the number of nodes of T. It is clearly true if $|V(T)| = 1$ by lines 1-3 of the algorithm. If $|V(T)| > 1$ then r has $d \geq 1$ children and, by the induction hypothesis, for every $1 \leq i \leq d$ and $Q' \subseteq X_{r_i}$, $K^i_{Q'}$ is a minimum vertex cover of G_i containing Q' (or $K^i_{Q'} = \infty$ if Q' is not a vertex cover of $G[X_{r_i}]$).

If $Q \subseteq X_r$ is such that $|Q| \leq 1$, then Q cannot cover all edges of X_r and so, there are no vertex cover K of G with $K \cap X_r = Q$, and so $K_Q = \infty$ (lines 9-10).

Otherwise, Q is a vertex cover of X_r and so there are vertex cover K of G such that $K \cap X_r = Q$ (e.g., $(V \setminus X_r) \cup Q$). Let K^*_Q be any minimum vertex cover of G such that $K^*_Q \cap X_r = Q$. For every $1 \leq i \leq d$, by Lemma 15, $K^* \cap V(G_i)$ is a minimum vertex cover (containing $Q \cap X_{r_i}$) of G_i. Lines 12-15 precisely consider such sets and so, the set K_Q is a minimum vertex cover of G such that $K^*_Q \cap X_r = Q$.

For the complexity, Lines 5-7 needs (by induction) $\sum\limits_{i=1}^{d} O(|E(T_i)|)$ operations. Then, because $|X_t| = 3$ for all $t \in V(T)$, then the number of sets $Q \subseteq X_r$ is $2^3 = O(1)$, and then the loop in Line 8 has $O(1)$ iterations and the computation of the minima in Line 13 takes constant time. Overall, the complexity is then $O(|E(T)|)$.

Since any tree-decomposition $(T, \mathcal{X})$ of a 2-tree G with n vertices has $O(n)$ nodes/edges (prove it), this leads to an overall complexity of $O(n)$. ∎

From previous Theorem and Algorithm, we easily get the linear-time Algorithm 23 that computes a minimum vertex cover in any 2-tree.

Algorithm 23 Minimum size Vertex Cover in 2-trees

Require: A 2-tree $G = (V, E)$ and a tree-decomposition $(T_r, \mathcal{X})$ of G rooted in some vertex $r \in V(T_r)$.

Ensure: A minimum-size vertex cover K of G

```
1: $(K_Q)_{Q \subseteq X_r} = Algorithm\ 22(G, (T_r, \mathcal{X}))$.
2: $K = V$
3: for $Q \subseteq X_r$ do
4:   if $|K_Q| \leq |K|$ then
5:     $K \leftarrow K_Q$                //abusing notations, $|K| = \infty$ if $K = \infty$
6:   end if
7: end for
8: return $K$
```

8.3 k-trees

Let us go a step further. Let k be any integer ≥ 1.

The class of k-trees is defined recursively as follows. A complete graph (clique) K_{k+1} with $k+1$ vertices is a k-tree. Given any k-tree H with some clique Q of size k in H, the graph obtained from H by adding a new vertex x adjacent to every vertex in Q is a new k-tree.

Given a k-tree $G = (V, E)$, a tree-decomposition $(T, \mathcal{X} = \{X_t \mid t \in V(T)\})$ of G is defined recursively as follows. If $G = K_{k+1}$ is reduced to one clique Q, its corresponding tree-decomposition consists of a tree T reduced to a single node t_0 and $\mathcal{X} = \{X_{t_0} = Q \subseteq V\}$. If G is obtained from a k-tree H, with a clique Q of size k in H, by adding a vertex x adjacent to all vertices in Q, a tree-decomposition $(T_G, \mathcal{X}_G = \{X^G_t \mid t \in V(T_G)\})$ of G is obtained from any tree-decomposition $(T_H, \mathcal{X}_H = \{X^H_t \mid t \in V(T_H)\})$ of H by considering any node $s \in V(T_H)$ such that $V(Q) \subseteq X^H_s$ (show that such a node exists by induction on $|V|$) and build T_G from

T_H by adding a new node n_x adjacent to s, and $\mathcal{X}_G = \mathcal{X}_H \cup \{X_{n_x} = Q \cup \{x\} \subseteq V\}$. Note that a k-tree may admit several decompositions.

Intuitively, each node of T_G corresponds to a subset of vertices of G inducing a maximal clique of G. Hence, $(T_G, \mathcal{X}_G)$ "organizes" the maximal cliques of G in a tree-like fashion while satisfying some "connectivity properties" as described by the first 2 items of Lemma 16.

Lemma 16 *Let $G = (V,E)$ be a k-tree, $(T,\mathcal{X})$ be a tree-decomposition of G and $e = \{u,v\} \in E(T)$. Let T_u (resp. T_v) be the subtree of $T \setminus e = (V(T), E(T) \setminus \{e\})$ containing u (resp., v) and $G_u = G[\bigcup_{t \in V(T_u)} X_t]$ and $G_v = G[\bigcup_{t \in V(T_v)} X_t]$. Then,*

- *$S = X_u \cap X_v$ is a clique of size k in G;*
- *S separates $V(G_u) \setminus S$ from $V(G_v) \setminus S$ (every path from a vertex in $V(G_u) \setminus S$ to a vertex in $V(G_v) \setminus S$ goes through S);*

  // in particular, there are no edges between $V(G_u) \setminus S$ and $V(G_v) \setminus S$.

Moreover, let $Q \subseteq S \setminus \{\emptyset\}$. Prove that, because S is a $(V(G_u) \setminus S, V(G_v) \setminus S)$-separator:

- *If K is a vertex cover of G with $K \cap S = Q$, then $K' = K \cap V(G_u)$ is a vertex cover of G_u with $K' \cap S = Q$.*
- *If K is a vertex cover of G_u with $K \cap S = Q$, then there exists a vertex cover K' of G such that $K = K' \cap V(G_u)$ (and, in particular, $K' \cap S = Q$).*

The key points are that Vertex Cover is a "local" problem and that k-trees have small-size separators (of size k). Intuitively, extending a vertex cover K of G_u to some vertex cover of G does not depend on the whole K but only on $K \cap S$ where S is a separator between G_u and the reminding of G. When $|S|$ is "small", this can be done efficiently.

Theorem 21 *Algorithm 24 is correct and has time-complexity $O(2^k n)$ where n is the number of vertices of any k-tree G.*

Proof. The proof of correctness is similar to the one of Algorithm 22.

For the complexity, Lines 5-7 needs (by induction) $\sum_{i=1}^{d} O(|E(T_i)|)$ operations. Then, because $|X_t| = k+1$ for all $t \in V(T)$, then the number of sets $Q \subseteq X_r$ is $2^{k+1} = O(2^k)$, and then the loop in Line 8 has $O(2^k)$ iterations and the computation of the minima in Line 13 takes time $O(2^k)$. Overall, the complexity is then $O(2^k|E(T)|)$.

Since any tree-decomposition $(T,\mathcal{X})$ of a k-tree G with n vertices has $O(n)$ nodes/edges (prove it), this leads to an overall complexity of $O(2^k n)$. ■

From previous Theorem and Algorithm, we easily get the Algorithm 25 that computes, in time $O(2^k n)$ a minimum vertex cover in any n-node k-tree.

8.4 Brief introduction to treewidth and tree-decompositions

Hopping to lead to a better intuition, we first give a definition of the treewidth (and tree-decomposition) following the previous sub-sections. Then, we will give an equivalent but more technical (?) definition that is (maybe) easier to work with.

Let $k \in \mathbb{N}$. A graph is a partial k-tree iff it is a subgraph of a k-tree. The *treewidth* of a graph $G = (V,E)$, denoted by $tw(G)$, equals the minimum integer k such that G is a partial

Algorithm 24 Dynamic Programming Algorithm for Minimum size Vertex Cover in k-trees

```
Require: A $k$-tree $G = (V, E)$ and a tree-decomposition $(T_r, \mathcal{X})$ of $G$ rooted in some vertex $r \in V(T_r)$.
Ensure: For every $Q \subseteq X_r$, a minimum-size Vertex Cover $K_Q$ of $G$ such that $K_Q \cap X_r = Q$ and $K_Q = \infty$ if no vertex cover $K$ of $G$ is such that $K \cap X_r = Q$.
 1: if $V(T_r) = \{r\}$ and $X_r = \{u, v, w\}$ then
 2:   return $(K_Q)_{Q \subseteq X_r}$ with $K_Q = Q$ if $|Q| > k-1$ and $K_Q = \infty$ otherwise.
 3: else
 4:   Let $r_1, \cdots, r_d$ be the children of $r$ and, for every $1 \leq i \leq d$, let $T_i$ be the subtree of $T_r$ rooted in $r_i$. Let $\mathcal{X}_i = \{X_t \in \mathcal{X} \mid t \in V(T_i)\}$ and let $G_i = G[\bigcup_{X \in \mathcal{X}_i} X]$ be the subgraph induced by the vertices in the sets in $\mathcal{X}_i$.
 5:   for $i = 1$ to $d$ do
 6:     Let $(K^i_{Q'})_{Q' \subseteq X_{r_i}} = \mathit{Algorithm}\ 22(G_i, (T_i, \mathcal{X}_i))$
 7:   end for
 8:   for $Q \subseteq X_r$ do
 9:     if $|Q| \leq k-1$ then
10:       Let $K_Q = \infty$
11:     else
12:       for $i = 1$ to $d$ do
13:         Let $Q_i \subseteq X_{r_i}$ be such that $Q_i \cap X_r = Q \cap X_{r_i}$ and, among such sets, $|K^i_{Q_i}|$ is minimum.        //abusing notations, $|K| = \infty$ if $K = \infty$
14:       end for
15:       Let $K_Q = Q \cup \bigcup_{i=1}^{d} K^i_{Q_i}$.
16:     end if
17:   end for
18:   return $(K_Q)_{Q \subseteq X_r}$.
19: end if
```

Algorithm 25 Minimum size Vertex Cover in k-trees

```
Require: A $k$-tree $G = (V, E)$ and a tree-decomposition $(T_r, \mathcal{X})$ of $G$ rooted in some vertex $r \in V(T_r)$.
Ensure: A minimum-size vertex cover $K$ of $G$
 1: $(K_Q)_{Q \subseteq X_r} = \mathit{Algorithm}\ 24(G, (T_r, \mathcal{X}))$.
 2: $K = V$
 3: for $Q \subseteq X_r$ do
 4:   if $|K_Q| \leq |K|$ then
 5:     $K \leftarrow K_Q$        //abusing notations, $|K| = \infty$ if $K = \infty$
 6:   end if
 7: end for
 8: return $K$
```

k-tree (note that any n-node graph is a partial $(n-1)$-tree as subgraph of K_n and so this parameter is well defined).

Following previous sub-sections, a first way to define a tree-decomposition of a graph is as follows. Let $G = (V, E)$ be a partial k-tree, i.e., a subgraph of a k-tree $H = (V_H, E_H)$. Let $(T, \mathcal{X} = (X_t)_{t \in V(T)})$ be a tree-decomposition of H (as defined in previous sub-section). Then, $(T, \mathcal{X} \cap V = (X_t \cap V)_{t \in V(T)})$ is a tree-decomposition of G of width at most k. Roughly, the difference with tree-decompositions of k-trees is that, in a tree-decomposition of a k-tree, the sets X_t (called *bags*) consist of cliques of size $k+1$, while, in the case of partial k-trees, they are subgraphs of size at most $k+1$. However, as we will see below they share the same connectedness properties. First, let us mention the algorithmic applications of tree-decompositions.

Theorem 22 *There exists an algorithm that, given any n-node graph G of treewidth $tw(G)$ and a tree-decomposition of G of width at most $tw(G)$, computes a minimum vertex cover of G in time $O(2^{tw(G)} \cdot n)$.*

Proof. Such an algorithm (almost) directly follows Algorithm 25 by modifying the Algorithm 24 in the following way. In Line 2, $|Q| > k-1$ is replaced by "Q is a vertex cover of X_r", and in Line 9, $|Q| \leq k-1$ is replaced by "Q is not a vertex cover of X_r". ■

The algorithm described in the proof of Theorem 22 is an FPT algorithm to compute a minimum Vertex Cover when the parameter is the treewidth (in contrast with previous FPT algorithms we have seen so far where the parameter was always the size of the solution, namely, the size k of a vertex cover).

To further exemplify the algorithmic applications of tree-decompositions, let us mention (without any explanation) the celebrated Courcelle's meta theorem.

Theorem 23 (Courcelle 1990) *Every graph property P definable in the monadic second-order logic*[14] *of graphs can be decided in linear time on graphs of bounded treewidth. That is, there is a function f_P such that P can be decided in time $O(f_P(k) \cdot n)$ in the class of n-node graphs with treewidth at most k.*

Complexity of treewidth. Theorem 22 (and most of the dynamic programming algorithms using tree-decompositions) explicitly requires a "good" (with small width) tree-decomposition as input (Theorem 23 actually requires it implicitly). Unfortunately, the problem of deciding whether $tw(G) \leq k$ is NP-complete[15]. On the positive side, this problem is FPT (with parameter the width itself)[16] and there exists a $\sqrt{\log k}$-approximation algorithm for the problem[17]. On a practical point of view, it is an important research topic to design efficient approximation algorithms or heuristics that compute "good" tree-decompositions of graphs. The problem can be solved more "efficiently" in particular graphs classes. For instance, there is a cubic 3/2-approximation algorithm in planar graphs[18], however, the complexity of the problem in planar graphs is still open...

[14]See Chapter 7.4 of [4] for an intuitive definition of MSOL. Examples of such problems include Vertex Cover, Dominating Set, 3-Colouring...

[15]Stefan Arnborg, Derek G. Corneil, Andrzej Proskurowski: Complexity of finding embeddings in a k-tree. SIAM J. of Discrete Maths 8(2): 277-284 (1987)

[16]Hans L. Bodlaender, Ton Kloks: Efficient and Constructive Algorithms for the Pathwidth and Treewidth of Graphs. J. Algorithms 21(2): 358-402 (1996)

[17]Uriel Feige, Mohammad Taghi Hajiaghayi, James R. Lee: Improved Approximation Algorithms for Minimum Weight Vertex Separators. SIAM J. Comput. 38(2): 629-657 (2008)

[18]Paul D. Seymour, Robin Thomas: Call Routing and the Ratcatcher. Combinatorica 14(2): 217-241 (1994)

Second definition of treewidth and tree-decomposition. So far, we have given a first definition of tree-decomposition and treewidth in terms of partial k-trees because we hope that it is a bit more intuitive. Let us now give a more technical definition that does not rely on a k-tree supergraph. Let $G = (V, E)$ be a graph. A tree-decomposition[19] of G is a pair $(T, \mathcal{X})$ where $T = (V(T), E(T))$ is a tree and $\mathcal{X} = \{X_t \mid t \in V(T)\}$ is a family of subsets (called *bags*) of V such that:

1. $\bigcup\limits_{t \in V(T)} X_t = V$;

2. for every $e = \{u, v\} \in E$, there exists $t \in V(T)$ such that $u, v \in X_t$;

3. for every $v \in V$, the set $\{t \in V(T) \mid v \in X_t\}$ induces a subtree of T.

The *width* of $(T, \mathcal{X})$ equals $\max\limits_{t \in V(T)} |X_t| - 1$ and the *treewidth, tw(G)*, of G is the minimum width of a tree-decomposition of G (Note that, for any graph G, there is a trivial tree-decomposition consisting of a tree with a single note t such that $X_t = V$). The -1 in the definition of the width is only there to ensure that $tw(G) = 1$ if and only if G is a *forest* (i.e., all connected components of G are trees).

Exercise 22 *Prove that the above definitions of tree-decomposition and treewidth are equivalent to the ones given in terms of partial k-tree.*

Let us note that, if a graph $G = (V, E)$ admits a tree-decomposition $(T, \mathcal{X})$ of width k, then G has an infinity of such decompositions. Indeed, for any $t \in V(T)$, let T' be the tree obtained from T by adding a (leaf) vertex t' adjacent to t and $\mathcal{X}' = \mathcal{X} \cup \{X_{t'} = X_t\}$, then prove that $(T', \mathcal{X}')$ is a tree-decomposition of G with width k. To avoid this pathological cases (and simplify next proofs), let us first show that we can always restrict ourself to tree-decomposition $(T, \mathcal{X})$ such that, for every $t, t' \in V(T)$, $X_t \setminus X_{t'} \neq \emptyset$.

Given a graph $G = (V, E)$ and an edge $uv \in E$, let G/uv be the graph obtained by contracting the edge uv defined as $V(G/uv) = (V \setminus \{u, v\}) \cup \{x\}$ and $E(G/uv) = (E \setminus \{e \in E \mid e \cap u \neq \emptyset \textit{ or } e \cap v \neq \emptyset\}) \cup \{xw \mid w \in N(u) \cup N(v)\}$ (roughly, u and v are identified).

Exercise 23 *Let $G = (V, E)$ be a graph and $(T, \mathcal{X})$ be tree-decomposition of G with width k. Let $tt' \in E(T)$ such that $X_{t'} \subseteq X_t$. Let $T' = T/tt'$ (with x being the new vertex resulting from the identification of t and t') and $\mathcal{X}' = (\mathcal{X} \setminus \{X_t, X_{t'}\}) \cup \{X_x = X_t\}$. Show that $(T', \mathcal{X}')$ is a tree-decomposition of G with width k.*

From now on, every considered tree-decomposition $(T, \mathcal{X})$ will be assumed to satisfy that for every $t, t' \in V(T)$, $X_t \setminus X_{t'} \neq \emptyset$. Note that, for such a tree-decomposition $(T, \mathcal{X})$ of a graph $G = (V, E)$, for every $t \in V(T)$ leaf of T, there exists $v \in V$ such that $v \in X_t$ and $v \notin X_{t'}$ for every $t' \in V(T) \setminus \{t\}$.

The next exercise probably describes the main property of tree-decompositions.

Exercise 24 *Let $G = (V, E)$ be a graph and $(T, \mathcal{X})$ be tree-decomposition of G.*

- *Let $t \in V(T)$ and let $T_1, \cdots, T_d$ be the components of $T \setminus \{t\}$. For every $1 \leq i < j \leq d$, X_t separates $\bigcup\limits_{t' \in T_i} X_{t'} \setminus X_t$ and $\bigcup\limits_{t' \in T_j} X_{t'} \setminus X_t$ in G.*

[19] Neil Robertson, Paul D. Seymour: Graph minors. IV. Tree-width and well-quasi-ordering. J. Comb. Theory, Ser. B 48(2): 227-254 (1990)

- *Let $e = tt' \in E(T)$ and let T_1 (resp., T_2) be the component of $T \setminus \{e\}$ containing t (resp., containing t'). $X_t \cap X_{t'}$ is a $(\bigcup_{h \in T_1} X_h \setminus X_t, \bigcup_{h \in T_2} X_h \setminus X_t)$ separator.*

Before going further, let us define an important notion. A graph H is a *minor* of a graph G, denoted by $H \preceq G$, if H is a subgraph of a graph G' that is obtained from G by a sequence of contraction(s) of edges. In other words, H is a minor of G if it can be obtained from G by sequentially removing vertices, edges and/or contracting edges.

Exercise 25 *Prove that,*

- *if $H \preceq G$, then $tw(H) \leq tw(G)$;* // *treewidth is minor-closed*
- *$tw(C) = 2$ for any cycle C;*
- *$tw(G) = 1$ if and only if G is a forest;*
- *$tw(K_n) = n - 1$ for any $n \geq 1$;*
- *Let G be a graph containing a clique K as subgraph. Then, for every tree-decomposition $(T, \mathcal{X})$ of G, there exists $t \in V(T)$ with $K \subseteq X_t$, and so $tw(G) \geq |K| - 1$;*
- *Let $G_{n \times m}$ be the $n \times m$ grid. Then, $tw(G_{n \times m}) \leq \min\{n, m\}$.*

Above, we tried to make it clear that graphs with bounded treewidth have "simple" structure that can be efficiently used for algorithmic purposes. Several "real-life" graphs have bounded treewidth[20], but, unfortunately, it is not true in many important fields of applications (Internet, road networks...). Therefore, it is natural to ask what is the structure of graphs with large treewidth. Such a "dual" structure for graphs with large treewidth would also be useful for proving lower bounds for the treewidth of graphs (e.g., we will use it to give the exact value of treewidth for grids). One important result of Robertson and Seymour in their Graph Minor theory (see below for more details) is the characterization of such an obstruction for small treewidth.

Given a graph $G = (V, E)$ and $X, Y \subseteq V$, X and Y are touching if $X \cap Y \neq \emptyset$ or if there are $x \in X$ and $y \in Y$ such that $xy \in E$. A bramble $\mathcal{B}$ in G is a family of subsets of V pairwise touching (i.e., for every $B, B' \in \mathcal{B}$, B and B' are touching). The order of $\mathcal{B}$ is the minimum size of a transversal of $\mathcal{B}$, i.e., the minimum size of a set $T \subseteq V$ such that $T \cap B \neq \emptyset$ for all $B \in \mathcal{B}$. The bramble number, $BN(G)$, of a graph G is the maximum order of a bramble in G.

Theorem 24 (Seymour and Thomas 1993) [21] *For any graph G, $tw(G) = BN(G) - 1$.*

An intuitive way to understand (and prove) the above theorem is by considering the equivalence between tree-decompositions and graph searching games[22].

As a consequence of previous theorem, let us show the following lemma.

Lemma 17 *Let $G_{n \times m}$ be the $n \times m$ grid. Then, $tw(G_{n \times m}) = \min\{n, m\}$.*

[20]e.g., Mikkel Thorup: All Structured Programs have Small Tree-Width and Good Register Allocation. Inf. Comput. 142(2): 159-181 (1998)

[21]Paul D. Seymour, Robin Thomas: Graph Searching and a Min-Max Theorem for Tree-Width. J. Comb. Theory, Ser. B 58(1): 22-33 (1993)

[22]See Section 4.1 in Nicolas Nisse: Network Decontamination. Distributed Computing by Mobile Entities 2019: 516-548

Proof. Let us assume that $n \leq m$. The upper bound follows from Exercise 25. For the lower bound, by Theorem 24, let us exhibit a bramble of order $n+1$. Given a grid, a *cross* consists of the union of any row plus any column. Let G' be the subgrid obtained from $G_{n\times m}$ by removing its first row and its first column. The desired bramble consists of the first row, the first column minus its vertex in the first row, and all crosses of G'. ■

Intuitively, a bramble with large order in a graph G may be seen as a large grid or as a large clique minor in G. The following result shows that any planar graph[23] has a large treewidth if and only if it admits a large grid as minor.

Theorem 25 (Grid Theorems) *[Robertson and Seymour 1986[24], Kawarabayashi and Kobayashi 2012[25], Chuzhoy and Tan 2019[26]]*
Any planar graph G with treewidth $\Omega(k)$ has an $k \times k$ grid as minor.
Any graph G with treewidth $\Omega(k^9 poly\log(k))$ has an $k \times k$ grid as minor.
There are graphs with treewidth $\Omega(k^2\log(k))$ without any $k \times k$ grid as minor.

One first interesting application of previous theorem is the framework of bidimensionality theory that we present with an example below.

Bidimensionality. Let us consider a function $f_P : \{graphs\} \to \mathbb{N}$ and consider the problem P that, given a graph G and an integer k, aims at deciding whether $f_P(G) \leq k \leq |V(G)|$. Let us assume that P is closed under taking minor, i.e., $f_P(H) \leq f_P(G)$ for every $H \preceq G$, that the problem can be decided in time $O(2^{tw(G)}n)$ and that $f_P(G_{n\times n}) = \Omega(n^2)$ where $G_{n\times n}$ is the grid of side n. The Vertex Cover problem is an example of such a problem.

Theorem 26 (Demaine and Hajiaghayi 2008) [27] *Such a problem P can be solved in sub-exponential time $O(2^{\sqrt{n}}poly(n))$ in the class of n-node planar graphs.*

Proof. (Sketch) Consider the following algorithm to decide whether $f_P(G) \leq k$. First, if $tw(G) = O(\sqrt{k})$, which can be decided (and a corresponding tree-decomposition can be computed) in time $O(2^{\sqrt{k}}n)$ [28], then by the second property of P, then the solution can be computed in time $O(2^{\sqrt{k}}n)$. Otherwise, by the Grid theorem, G has a $\sqrt{k} \times \sqrt{k}$-grid H as minor. Since $f_P(H) = \Omega(k)$ and P is closed under taking minor, then $f_P(G) = \Omega(k)$.

Finally, since $k \leq n$, the result follows. ■

The above theorem has been generalized for larger classes of sparse graphs such as bounded genus graphs and even graphs excluding some fixed graph as minor.

[23] A graph is planar if it can be drawn on the sphere without crossing edges.

[24] Neil Robertson, Paul D. Seymour: Graph minors. V. Excluding a planar graph. J. Comb. Theory, Ser. B 41(1): 92-114 (1986)

[25] Ken-ichi Kawarabayashi, Yusuke Kobayashi: Linear min-max relation between the treewidth of H-minor-free graphs and its largest grid. STACS 2012: 278-289

[26] Julia Chuzhoy, Zihan Tan: Towards Tight(er) Bounds for the Excluded Grid Theorem. SODA 2019: 1445-1464

[27] Erik D. Demaine, MohammadTaghi Hajiaghayi: The Bidimensionality Theory and Its Algorithmic Applications. Comput. J. 51(3): 292-302 (2008)

[28] e.g., Hans L. Bodlaender, Pâl Gronas Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, Michal Pilipczuk: A c^k n 5-Approximation Algorithm for Treewidth. SIAM J. Comput. 45(2): 317-378 (2016),
Hans L. Bodlaender: A linear time algorithm for finding tree-decompositions of small treewidth. STOC 1993: 226-234

A third definition of treewidth. For completeness (and to conclude this brief introduction to treewidth), let us give another definition of treewidth. A graph is *chordal* if it has no induced cycle of length at least 4 as subgraph. Equivalently, a graph is chordal if it is the intersection graph of a family of subtrees of a tree. Given a graph G, let $\omega(G)$ be the maximum size of a clique in G. The treewidth of a graph G can be defined as the minimum $\omega(H)-1$ among all chordal supergraphs H of G. Note that there is a close relationship between tree-decompositions of a graph G and the clique trees of its chordal supergraphs[29].

8.5 Graph Minor theory

To conclude this section, let us try to sketch the main reason why Robertson and Seymour introduced tree-decompositions and treewidth. Note that there are very nice surveys on this topic[30]. Recall that a partial order is called Well Quasi Ordered (WQO) if it admits no infinite antichain (i.e., no infinite sequence of elements that are pairwise incomparable). The Wagner's conjecture (1970) asked whether the minor ordering is WQO over the set of graphs. Along a serie of 20 papers (with overall about 500 pages) from 1983 to 2004, Robertson and Seymour answered this question (and many fundamental others) through what is now called the Graph Minor theory (interestingly, the order of publication of these papers does not necessarily corresponds to the order of the results).

Theorem 27 (Robertson and Seymour 2004) [31] *The minor relationship is WQO.*

Before giving a very rough idea of its proof, let us show the algorithmic consequences of the above theorem. A class of graph $\mathcal{G}$ is minor-closed if, for every $H \preceq G$, $G \in \mathcal{G}$ implies that $H \in \mathcal{G}$. Given a graph class $\mathcal{G}$, let the set of obstructions $Obs(\mathcal{G})$ be the set of minor-minimal graphs not in $\mathcal{G}$, i.e., the set of graphs H such that $H \notin \mathcal{G}$ and $H' \in \mathcal{G}$ for all $H' \prec H$.

Corollary 4 *Let $\mathcal{G}$ be a minor-closed class of graphs. Then $Obs(\mathcal{G})$ is finite.*

Proof. Otherwise, by Theorem 27, there would be two graphs G, G' in $Obs(\mathcal{G})$ such that $G \prec G'$, a contradiction. ■

As an example, note first that any minor of a planar is also planar. Hence, the class $\mathcal{P}$ of planar graphs is minor-closed.

Theorem 28 (Wagner 1937) *A graph is planar if and only if it has no K_5 nor $K_{3,3}$ (the complete bipartite graph with 3 vertices in each part) as minor, i.e., $Obs(\mathcal{P}) = \{K_5, K_{3,3}\}$.*

To understand the importance of Corollary 4, let us do a short detour to vertex-disjoint paths in graphs. Given a graph $G=(V,E)$ and two disjoint subsets $X, Y \subset V$ with $|X| = |Y| = k$, the problem of deciding whether there are k vertex-disjoint paths between X and Y (and compute such paths) can be solved in polynomial-time (e.g., using flow algorithm or the proof of Menger's theorem). In contrast, given a graph $G=(V,E)$ and two disjoint subsets $X = \{s_1, \cdots, s_k\}, Y = \{t_1, \cdots, t_k\} \subset V$ with $|X| = |Y| = k$, the problem of deciding whether there are k vertex-disjoint paths $P_1, \cdots, P_k$, where P_i is a path between s_i and t_i for all $i \leq k$, (and compute such paths) is NP-complete [7]. To see the difference between the two problems,

[29]Philippe Galinier, Michel Habib, Christophe Paul: Chordal Graphs and Their Clique Graphs. WG 1995: 358-371

[30]See the survey of Lovász here and the survey of Robertson and Seymour themselves (1985) here.

[31]Neil Robertson, Paul D. Seymour: Graph Minors. XX. Wagner's conjecture. J. Comb. Theory, Ser. B 92(2): 325-357 (2004)

consider a cycle with vertices (s_1, s_2, t_1, t_2) (in this order): clearly, there are 2 vertex-disjoint paths from $X = \{s_1, s_2\}$ to $Y = \{t_1, t_2\}$, but 2 vertex-disjoint paths P_1 from s_1 to t_1 and P_2 from s_2 to t_2 do not exist.

One of the numerous fundamental contributions of Robertson and Seymour along their Graph Minor serie is the proof that, when k is fixed, the latter problem (k-linkage), is FPT in k[32]. This allowed them to show that, given a fixed graph H, the problem that takes an n-node graph G as input and asks whether $H \preceq G$ (G admits H as minor) can be solved in time $O(n^3)$ where the "big O" hides a constant depending on H (this result has been improved to an $O(n^2)$-time algorithm since then).

Theorem 29 (Kawarabayashi, Kobayashi and Reed 2012) *Let H be a fixed graph. The problem that takes an n-node graph G as input and decides if $H \preceq G$ can be solved in time $O(n^2)$.*

To give an intuition of the relationship between the minor containment problem and the k-linkage problem, let us give the very sketchy following process (whose time-complexity is much worst than the one announced in previous theorem but still polynomial for fixed H). First, we can "guess" the vertices of G that correspond to vertices of H (by trying the $O(n^{|V(H)|})$ possibilities). For each choice of $|V(H)|$ vertices in G, then, we have to recover the $|E(H)|$ edges of H as $|E(H)|$ vertex-disjoint paths in G (with sources and terminal the vertices we have guessed).

Now, we are ready to give the main algorithmic consequence of Robertson and Seymour's theorem.

Theorem 30 *Let $\mathcal{G}$ be any minor-closed graph class. The problem that takes a graph G as input and asks whether $G \in \mathcal{G}$ is in P.*

Proof. The algorithm is as follows. For each $H \in Obs(\mathcal{G})$ (there are a finite number of such graph by Corollary 4), decide if $H \preceq G$ (can be done in polynomial-time by Theorem 29). If $H \preceq G$ for some $H \in Obs(\mathcal{G})$ then $G \notin \mathcal{G}$, else $G \in \mathcal{G}$. ■

Note that previous theorem is only an existential result since it requires the knowledge of $Obs(\mathcal{G})$ for the considered graph class $\mathcal{G}$. Unfortunately, as far as I know, the set of obstructions is known for very few graph classes. For instance, the full set of obstructions of the class of graphs with genus 1 (that can be embedded without crossing edges on a "doughnut") is still unknown.

"Proof" of Robertson and Seymour's theorem. To conclude this section, let us give a very very very sketchy (and probably a bit wrong, sorry) idea of the proof of Theorem 27. Roughly, the guideline is to prove that the minor relationship is WQO in graph classes that are more and more large.

Theorem 31 (Kruskal 1960) *The minor relationship is WQO in the class of trees.*

The next step is naturally the class of graphs with bounded treewidth.

Theorem 32 (Robertson and Seymour 1990) [33] *The minor relationship is WQO in the class of graphs with bounded treewidth.*

[32]Neil Robertson, Paul D. Seymour: Graph Minors XIII. The Disjoint Paths Problem. J. Comb. Theory, Ser. B 63(1): 65-110 (1995)

[33]Neil Robertson, Paul D. Seymour: Graph minors. IV. Tree-width and well-quasi-ordering. J. Comb. Theory, Ser. B 48(2): 227-254 (1990)

Intuitively, let $(G_1, \cdots)$ be an infinite sequence of graphs with treewidth at most k, and let $(T_i, \mathcal{X}_i)$ be a tree-decomposition of width k of G_i. The sequence $(T_1, \cdots)$ is an infinite sequence of trees and, by Threorem 31, we can extract an infinite sequence $(T_{i_1} \preceq T_{i_2} \preceq \cdots)$. Because the graphs $(G_{i_1}, G_{i_2}, \cdots)$ have bounded treewidth, the trees $(T_{i_1}, T_{i_2}, \cdots)$ can be seen as trees with labels of bounded length on their vertices. The result follows (after some work).

Next, the case of planar graphs arises.

Theorem 33 (Robertson and Seymour 1986) [34] *The minor relationship is WQO in the class of planar graphs.*

Indeed, very intuitively, let us consider an infinite sequence $\mathcal{S}$ of planar graphs. If infinitely of them have bounded treewidth, then the result follows previous theorem. Otherwise, by the grid Theorem 25, they have arbitrary large grids as minors. Note that, for any planar graph G, there exists a grid Gr such that $G \preceq Gr$. Overall, it is possible to find $G, G' \in \mathcal{S}$ such that G' has a sufficiently large grid Gr as minor such that $G \preceq Gr$. Hence, $G \preceq Gr \preceq G'$.

Previous result can then be extended to bounded genus graphs. Roughly, a surface has (orientable) genus at most g if it can be obtained from a sphere by adding to it g *handles*. A graph has genus g if it can be embedded without crossing edges on a surface with genus g (planar graphs are graphs with genus 0, graphs with genus ≤ 1 are the ones that can be embedded on a doughnut...). See [1] for more formal definitions.

Theorem 34 (Robertson and Seymour 1990) [35] *The minor relationship is WQO in the class of graphs with bounded genus.*

We now can "conclude". Let $(G_1, \cdots)$ be an infinite sequence of graphs. For every $k \geq 2$, $G_1 \not\preceq G_k$ (since otherwise we are done). Hence, the graphs $G_2, \cdots$ are all excluding G_1 as minor. A key contribution of Robertson and Seymour is the structural characterization of the graphs excluding a fixed graph H as minor. Namely, given a fixed graph H, they show that any H-minor free graph (i.e., excluding H as minor) admits a particular decomposition[36] that we try to sketch below.

Very very very roughly (sorry again), a H-minor free graph G admits a tree-decomposition $(T, \mathcal{X})$ such that

- for every $uv \in E(T)$, $|X_u \cap X_v| \leq 3$ (this bound is actually due to Demaine *et al.*);
- for every $v \in V(T)$, the bag X_v induces a graph G_v that is obtained from: a graph G'_v that has bounded (in terms of $|H|$) genus, to which it can be added a bounded (in terms of $|H|$) number of vortices (subgraphs of bounded (in terms of $|H|$) "pathwidth" that may be "glued" along non-contractible cycles of G'_v) and then a bounded (in terms of $|H|$) number of apices can be added (vertices that can be adjacent to any vertex).

The proof of Theorem 27 then follows from Robertson and Seymour's decomposition and previous theorems (bounded treewidth, bounded genus...).

[34] Neil Robertson, Paul D. Seymour: Graph minors. V. Excluding a planar graph. J. Comb. Theory, Ser. B 41(1): 92-114 (1986)

[35] Neil Robertson, Paul D. Seymour: Graph minors. VIII. A kuratowski theorem for general surfaces. J. Comb. Theory, Ser. B 48(2): 255-288 (1990)

[36] Neil Robertson, Paul D. Seymour: Graph Minors. XVI. Excluding a non-planar graph. J. Comb. Theory, Ser. B 89(1): 43-76 (2003)

References

[1] B. Mohar, C. Thomassen *Graphs on Surfaces.* Johns Hopkins University Press Books, 2001, ISBN: 978-0-80186-689-0.

[2] A. Bondy, M.S.R. Murty. *Graph Theory.* Graduate texts in maths, Springer 2008, ISBN 978-1-84628-969-9.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press 2009, ISBN 978-0-262-03384-8.

[4] M. Cygan, F.V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, Ma. Pilipczuk, Mi. Pilipczuk, S. Saurabh. *Parameterized Algorithms.* Springer 2015, ISBN 978-3-319-21274-6

[5] R. Diestel: *Graph Theory.* Graduate texts in mathematics 173, Springer 2012, ISBN 978-3-642-14278-9

[6] F.V. Fomin, D. Kratsch: *Exact Exponential Algorithms.* Texts in Theoretical Computer Science. An EATCS Series, Springer 2010, ISBN 978-3-642-16532-0

[7] M.R. Garey, D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness* W. H. Freeman & Co, 1979, ISBN:0716710447

[8] V. V. Vazirani: *Approximation algorithms.* Springer 2001, ISBN 978-3-540-65367-7