Spanning trees and complexity

Guillaume Dalle (ENPC, CERMICS)

REOP - Class 4 (20/10/2021)

Contents

Reminder: there will be a 1h exam on November 10th about everything you will have seen then.

1 Homework solutions

You will find solutions for this week's homework (exercises 6.10 and 6.13) in the complete lecture notes on Educnet.

2 Minimum spanning trees

2.1 Definitions

A spanning subgraph of $\mathcal{G} = (V, E)$ is a subgraph $\mathcal{H} = (V, F)$ whose set of edges is incident to every vertex of V. If $\mathcal{H}$ is a also a tree (connected & without cycles), we talk of a spanning tree.

Minimum spanning tree (MST) problem:

- Input: An undirected connected graph $\mathcal{G} = (V, E)$, a weight function $c : E \to \mathbb{R}$
- Output: A spanning tree $\mathcal{T} = (V, T)$ of minimum weight $\sum_{e \in T} c(e)$

2.2 Greedy construction

2.2.1 Kruskal's algorithm

1. Sort E by increasing edge weight: $E = \{e_1, ..., e_m\}$ with $c(e_i) \leq c(e_{i+1})$
2. Start with $F_0 = \emptyset$
3. For $i = 1, ..., n$: if $F_{i-1} \cup \{e_i\}$ has no cycles, set $F_i = F_{i-1} \cup \{e_i\}$, otherwise set $F_i = F_{i-1}$.
4. Return $\mathcal{T} = (V, F_m)$.

Exercise 4.2: applying Kruskal's algorithm

2.2.2 Correctness argument

At any time during the algorithm, $\mathcal{F}_i = (V, F_i)$ is a forest because we ensure that it remains without cycles. So $\mathcal{T}$ is a forest.

If $\mathcal{T} = (V, F_m)$ is not connected (in particular, if it has isolated vertices), then let V_1 and V_2 be two connected components in $\mathcal{T}$. Since $\mathcal{G}$ is connected, there is an edge $e \in E$ between V_1 and V_2: this edge does not belong to $\mathcal{T}$ and does not create a cycle, hence it should have been added at some point in the algorithm. We obtain a contradiction: $\mathcal{T}$ is a connected forest on V, aka a spanning tree.

To prove that $\mathcal{T}$ has minimum weight among all spanning trees, we can use the following loop invariant: *for all i, there is a minimum spanning tree $\mathcal{T}_i = (V, T_i)$ such that $\mathcal{F}_i = (V, F_i)$ is a subgraph of $\mathcal{T}_i$*. This will allow us to conclude that $\mathcal{T} = \mathcal{T}_m$ (since both are trees with the same number of vertices).

> Exercise: Prove that the loop invariant stays true during iteration i. To do that, try to build T_i from T_{i-1} by considering all possible cases. You will need to justify the following statements, in order:
>
> 1. We only need to consider the case where $e_i = (u_i, v_i)$ is added to F_{i-1} but doesn't belong to T_i.
> 2. In this case, let X be the connected component of $\mathcal{F}_{i-1}$ containing u_i: X does not contain v_i
> 3. T_{i-1} contains exactly one path between u_i and v_i, which crosses the cut $\delta(X, V \backslash X)$ using an edge $f \neq e_i$.
> 4. The cut $\delta(X, V \backslash X)$ is disjoint from F_{i-1}
> 5. $\delta(X, V \backslash X)$ does not contain any e_j with $j < i$
> 6. $T_i := T_{i-1} \backslash \{f\} \cup \{e_i\}$ satisfies $c(T_i) \leq c(T_{i-1})$
> 7. T_i is still a spanning tree

Here are the justifications:

1. If we don't add edge e_i to F_{i-1}, nothing happens, and since $F_i = F_{i-1} \subset T_{i-1}$, we can take $T_i := T_{i-1}$ to be a MST containing F_i. If we add e_i to F_{i-1} and e_i is already in T_{i-1}, we can also take $T_i := T_{i-1}$. If we add e_i to F_{i-1} but e_i is not in T_{i-1}, then we must be more clever since the spanning tree containing F_i changes.
2. $v_i \notin X$ because $e_i \notin F_{i-1}$ and adding e_i does not create a cycle, so there is no way to get from u_i to v_i in F_{i-1}.
3. There is only one path from u_i to v_i because T_{i-1} is a tree. The edge of this path crossing the cut is different from e because $e_i \notin T_{i-1}$.
4. If we had $\delta(X, V \backslash X) \cap F_{i-1} \neq \emptyset$ then the connected component X could be extended and would not be maximal.
5. An edge e_j with $j < i$ is either in F_{i-1} (which means it is not in the cut), or it creates a cycle in F_{i-1}, but this would require some edge of $\delta(X, V \backslash X)$ to be in F_{i-1} (which is impossible).
6. We have $f = e_j$ with $e_j > e_i$, which means $c(f) \geq c(e_i)$. As a consequence, if we define $T_i := T_{i-1} \backslash \{f\} \cup \{e_i\}$, then we have $c(T_i) = c(T_{i-1}) + c(e_i) - c(f) \leq c(T_{i-1})$.
7. $T_i \cup \{e_i\}$ has exactly one cycle containing e_i and f, which means removing f preserves the spanning property and the connectedness, while making the subgraph acyclic again.

2.2.3 Remarks

Kruskal's algorithm is greedy, but surprisingly it returns an optimal solution! Prim's algorithm is another example of greedy algorithm for the MST: instead of uniting forests, it grows a single tree.

Implementing Kruskal's algorithm is non-trivial: naively checking whether e_i creates a cycle in F_{i-1} would be very inefficient. Standard implementations use a specific data structure called "Union-Find" to store the disjoint forests and merge them.

2.3 Spanning tree polytope

The MST problem can be formulated as a Mixed Integer Linear Program:

$$\min \sum_{e \in E} c(e) x_e \quad \text{s.t.} \begin{cases} \sum_{e \in E} x_e = |V| - 1 & \\ \sum_{e \in E[X]} x_e \leq |X| - 1 & \forall X \subset V, X \notin \{\emptyset, V\} \\ x_e \in \{0, 1\} & \forall e \in E \end{cases}$$

Given an optimal solution x^* to this program, we can construct a subgraph (V, E_{x^*}) with $E_{x^*} = \{e \in E : x^*_e = 1\}$.

The second constraint imposes the absence of cycle among any subset of the vertices, which ensures we have a forest. The first constraint tells us that the forest is maximal, i.e. a (spanning) tree. The objective function ensures minimality of the weight.

However, this MILP is not easy to solve, for two reasons:

- The integrality constraint $x_e \in \{0, 1\}$. This is addressed by Theorem 4.4: we can replace the constraint by $x_e \geq 0$ and still be sure that basic optimal solutions (returned by the simplex) are binary-valued.
- The exponential number of constraints. This is taken care of by Exercise 4.4, which shows how to decide efficiently whether any constraint is broken. Counterintuitively, this means we can solve the LP efficiently, at least in theory.

  Exercise 4.4: The separation problem for the spanning tree polytope is polynomial.

  For Q2, we assume a feasible solution x is given. - Given a set of vertices X, show that we can find a cut $\varphi(X)$ with capacity $|X| + \sum_{e \notin E(X)} x_e$. - Given a minimum cut B on the graph defined Q1, show that we can find a cut B' of the same capacity and a set of vertices X such that $B' = \varphi(X)$.

In the standard setting, there is no need for this LP since Kruskal's algorithm is faster. However, it is very useful when additional constraints or objectives are involved.

3 Introduction to complexity theory

See the introduction class for the intuitive definitions of "problem" and "algorithm."

There are undecidable problems, such as the halting problem (deciding whether an algorithm will terminate or not on a given input). Here we focus on decidable problems and study their complexity.

3.1 Decision and optimisation problems

Formally, anything we can give to a computer can be described as a word x from a language X. A decision problem is a couple (X, Y) where

- X is a language called the input ("decidable" in polynomial time)
- Its elements $x \in X$ are called instances
- $Y \subset X$ contains all instances for which the answer is "yes"
- $X \backslash Y$ contains all instances for which the answer is "no"

A solution algorithm is a function $f : X \to \{\texttt{yes}, \texttt{no}\}$ such that $f(Y) = \{\texttt{yes}\}$ and $f(X \backslash Y) = \{\texttt{no}\}$. The algorithm f runs in polynomial time if there is a polynomial P such that its number of steps in its execution on instance x can be bounded by $P(\texttt{size}(x))$. Here $\texttt{size}(x)$ is the length of the word x in binary (example: integer factorization).

3.2 Complexity classes

3.2.1 Why is it important?

The complexity class determines the method we can try:

- Fast exact algorithms for "easy" problems
- Fast approximate or heuristic algorithms for "hard" problems

Easy	Hard
Eulerian cycle	Hamiltonian cycle
Shortest path	Longest path
2-SAT	3-SAT
Minimum spanning tree	Steiner tree
Train shunting (night)	Train shunting (day)

3.2.2 The classes P and NP

The class **P** contains all decision problems for which there is a polynomial algorithm that returns a solution.

The class **NP** (for "Nondeterministic Polynomial") contains all decision problems for which there is a polynomial algorithm that verifies a solution, using a so-called certificate. We don't need to be able to generate a certificate in polynomial time, just to check it against the instance.

Exercise : prove that $\mathbf{P} \subset \mathbf{NP}$.

Remarks:

- Why polynomial time (Cobham's thesis)? Independent of the underlying computer model.
- Not independent of the encoding: binary vs unary.
- Why no mention of memory? On a Turing machine, memory is bounded by time.

Exercise: Devise a dynamic programming algorithm for the knapsack problem. Is it enough to prove that this problem is in **P**?

3.2.3 NP-completeness

A reduction of a problem $P = (X, Y)$ to $P' = (X', Y')$ is a mapping $r : X \to X'$ such that $x \in Y \iff r(x) \in Y'$. The reduction is called polynomial if r can be computed in polynomial time.

A decision problem P is **NP**-complete if

1. P is in **NP**
2. Any decision problem P' in **NP** reduces polynomially to P

Since mathematicians have already compiled a large list of **NP**-complete problems, to prove the second point we only need to reduce a known **NP**-complete problem to P.

A decision or optimisation problem is **NP**-hard if any decision problem P' in **NP** reduces polynomially to P.

3.2.4 The 1M dollar hypothesis

One of the seven Millenium Prize Problems listed by the Clay Mathematics Institute. Still unsolved, although no one expects **P** to be equal to **NP**.

If $\mathbf{P} \neq \mathbf{NP}$, then both **P** problems and **NP**-complete problems are proper subsets of **NP** problems, which is itself a proper subset of **NP**-hard problems. More surprisingly, this would also mean that there are some problems which are neither polynomial nor **NP**-complete (graph isomorphism).

There are many more complexity classes with exotic properties: feel free to visit the complexity zoo if you want to have a laugh.

3.2.5 Optimisation problems

If we can solve the optimisation problem

$$\min_x c(x) \quad \text{s.t.} \quad x \in X$$

we can solve its decision version

$$\exists ?\ x \quad \text{s.t.} \quad x \in X \quad \text{and} \quad c(x) \leq c_0$$

To prove that an optimisation problem O is **NP**-hard, it is enough to reduce a known **NP**-complete decision problem to its decision version $\text{dec}(O)$.

> Exercise: Prove that the shortest path problem is **NP**-hard in the general case. To do that, use a reduction from Hamiltonian path to longest simple path, followed by a second reduction from longest simple path to shortest simple path.
>
> Exercise: The 3-SAT problem can be described as follows:
>
> - Input: A boolean formula $F(x)$ on n binary variables $x_1, ..., x_n$ in conjunctive normal form (i.e. a conjunction $\wedge$ = `AND` of $\vee$ = `OR` clauses) with at most three literals by clause (a literal can be x_i or $\neg x_i$). Example: $F(x) = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.
> - Output:
> - `yes` if there is an $x \in \{0,1\}^n$ satisfying F, and `no` otherwise.
>
> By the Cook-Levin theorem, 3-SAT is NP-complete. Show that this implies the maximum clique problem is NP-hard. Starting with an instance of 3-SAT, you can build an instance of CLIQUE using the graph shown on the Wooclap slide.