TD 3: NP-Completeness and Graph algorithms

1 NP-Completeness

Exercise 1. *Vertex Cover*

Let $G = (V, E)$ be a graph and k be an integer. A *vertex cover* is a subset $V' \subseteq V$ such that for all $\{u, v\} \in E$, $u \in V'$ or $v \in V'$. `VERTEX-COVER` is the problem to know whether there exists a vertex cover of size less than or equal to k in G or not. We want to show that `VERTEX-COVER` is NP-complete by doing a reduction from `3-SAT`. We take a `3-SAT` formula F and we transform it into an equivalent `VERTEX-COVER` problem.

1. Show that if you take a graph composed of n disjoint pairs of vertices and m disjoint triangles then the size of the minimum vertex cover is $n + 2m$.
2. Prove that `VERTEX-COVER` is NP-complete

Culture : `EDGE-COVER` is like `VERTEX-COVER` but we look for a subset $E' \subseteq E$ such that every vertex of V is incident to at least one vertex in E'. Yet it is not NP-Complete (we know a polynomial algorithm).

2 Graph algorithms

Exercise 2. *BFS*

What is the complexity of the `BFS` (Breadth-First-Search) algorithm ?

Exercise 3. *Bipartite Graphs*

A graph is said *bipartite* if the set of vertices can be partionned into two disjoint sets U and V such that every edge connects a vertex of U with a vertex of V.

✎ Show that the following propositions are equivalent :
- **(i)** G is bipartite.
- **(ii)** G does not contain an odd cycle.
- **(iii)** G is 2-colorable (*i.e.* we can color any vertex with either blue or green such that every edge does not connect two vertices with the same color).
- **(iv)** When doing a `BFS` (Breadth-First-Search) of G there is no edges inside a same distance layer.

Culture : Thus we have that `2-COLORING` (the problem to know whether a given graph is 2-colorable or not) can be solved in polynomial time, but `3-COLORING` is NP-Complete (the proof is close to the reduction for `VERTEX-COVER`).

Exercise 4. *Bellman Ford*

The goal of this exercise is to prove the correction of the `BELLMAN-FORD` algorithm. In the following we note $\delta(u, v)$ for the weight of the shortest path from u to v.

1. Show that at any time of the algorithm, for any vertex v, $d[v] \geq \delta(s, v)$.

2. Show that if G contains no negative-weight cycles that are reachable from s then at the end of the algorithm we have $d[v] = \delta(s, v)$ for each vertex v.
3. Show that the algorithm returns `false` if and only if G contains a negative-weight cycle that is reachable from s.
4. What is the complexity of the algorithm ?

Exercise 5. *All-to-all : Floyd Warshall*

Let $G = (V, E, W)$ be a weighted directed graph with no negative cycles. We want to calculate the shortest paths between all couples of vertices.

1. With the algorithms seen in course, what complexity would you achieve ?
2. Let us write $V = \{1, .., n\}$. For any i, j, k in $\{1, .., n\}$, let $d_{i,j}^{(k)}$ be the weight of the shortest path between i and j going through only the vertices $\{1, .., k\}$. Try to find a recurrence relation between $d_{i,j}^{(k)}$ and the $d_{i',j'}^{(k-1)}$'s.
3. Deduce from the previous question an algorithm to calculate all the shortest paths in the graph.
4. What is the complexity of your algorithm ? Is it better than the ones obtained in the first question ?
5. How would you adapt the algorithm to be able to recover the shortest path between any couple of vertices in $O(n)$?

Exercise 6. *How to dress well ?*

If we have some tasks, they might have some dependencies, meaning that task B can be executed only after task A for instance. Thus we can represent them using a directed graph (called a *task graph*).

1. Let's work with an example. You want to dress in the morning, you have to put on your socks, your underpants, your t-shirt, your pullover, your shoes and your trousers. Draw the task graph associated with those tasks.
2. Show that you can execute the tasks if and only if the task graph is acyclic (meaning that it does not contain any cycle).
3. Explain on the example from question 1 how we can use the `DFS` algorithm to :
   — Know whether you can execute the tasks or not.
   — Know in which order you can execute them if it is possible.