


Projet Info LDD2 – TD 7

Renaud Vilmar -- vilmar@lsv.fr

Objectifs du TD : Longueur de chemins, profondeur, tri topologique.

 Ce TD est noté, il est donc à finir, et l'état du projet à déposer, avant la séance du TD 8, sur eCampus (bien zipper tout le projet, pas seulement un fichier). La notation sera sur les exos de ce TD exclusivement. Comme précédemment, une attention particulière sera portée à la documentation aux commentaires et à la lisibilité.

Un chemin dans un circuit booléen correspond au trajet pris par l'information. Ainsi des notions comme le plus court ou plus long chemin, la profondeur d'un noeud, de (plus petit) ancêtre commun, de (plus grand) descendant commun, etc... ont toutes une signification dans ce domaine.

La longueur du plus court chemin orienté de u à v représente le temps minimal pour que l'information (ou une partie de cette information) en u à un instant donné atteigne v . La longueur du plus long chemin au contraire peut nous dire à partir de quel moment l'information de u à un instant t cesse d'influencer celle de v . Je vous laisse le soin de trouver une signification aux notions d'ancêtres et de descendants communs.

La notion de profondeur est notamment très importante : la profondeur du circuit (qu'on définira plus proprement dans la suite) représente en un sens sa complexité. C'est une métrique qu'on aimerait pouvoir minimiser (une autre est plus simplement le nombre de noeuds dans le graphe).

Un algorithme important qui permet de simplifier beaucoup de ces calculs à lui seul est l'algorithme de Dijkstra. Une variante habituelle de celui-ci calcule l'arbre des chemins les plus courts à partir d'un noeud donné.

Nous allons, nous, l'appliquer souvent à des graphes dirigés acycliques, mais pas toujours. On veut donc une variante de cet algorithme qui en plus permet de prendre en compte l'orientation des arêtes du graphe. L'algorithme adapté à nos besoins est donné dans le bloc 1, en pseudo-code.

Ici, **direction** fixe la notion de voisinage. **None** signifie qu'on cherche à la fois dans les parents et les enfants, **-1** seulement dans les parents, et **1** seulement dans les enfants. Ainsi, appeler l'algorithme sur u avec **direction=-1** doit parcourir seulement les "ancêtres" de u .

▮ **Exercice 1** (Tests Requis) : Implémenter l'algorithme de Dijkstra pour les **open_digraph**.

Astuce : pour la recherche du min, on peut utiliser `min(l , key= f)` qui retourne un $u \in l$ tel que $f(u) \leq f(v)$ pour tout $v \in l$; i.e. qui retourne le (un) u qui minimise f . ▮

Si on cherche plus simplement la distance (et le chemin le plus court) entre deux noeuds donnés, on peut faire s'arrêter l'algorithme plus tôt.

Algorithm 1 Algorithme de Dijkstra

```
function DIJKSTRA(src, direction=None)  ▷ src est le noeud duquel
                                          ▷ on va calculer les distances
                                          ▷ direction ∈ {None, -1, 1}

  Q ← [src]
  dist ← {src: 0}
  prev ← {}
  while Q ≠ [] do
    u ← node id in Q with min dist[u]
    remove u from Q
    neighbours ← the neighbours of u subject to direction
    for all v ∈ neighbours do
      if v ∉ dist then
        add v to Q
      end if
      if v ∉ dist or dist[v] > dist[u] + 1 then
        dist[v] ← dist[u] + 1
        prev[v] ← u
      end if
    end for
  end while
  return dist, prev
end function
```

「 **Exercice 2 :** Modifier l'algorithme en rajoutant `tgt=None` en argument, pour que si ce dernier est spécifié, on retourne directement *dist* et *prev* dès qu'on est sûr du chemin le plus court de *src* à *tgt*.

Utiliser cette méthode pour implémenter `shortest_path` qui calcule le chemin le plus court de *u* vers *v*. ┘

「 **Exercice 3 :** Implémenter une méthode qui étant donnés deux noeuds renvoie un dictionnaire qui associe à chaque ancêtre commun des deux noeuds sa distance à chacun des deux noeuds. Par exemple, dans le graphe donné dans la suite, l'algorithme appliqué sur les noeuds 5 et 8 doit renvoyer :

$$\{0 : (2, 3), 3 : (1, 2), 1 : (1, 1)\}.$$

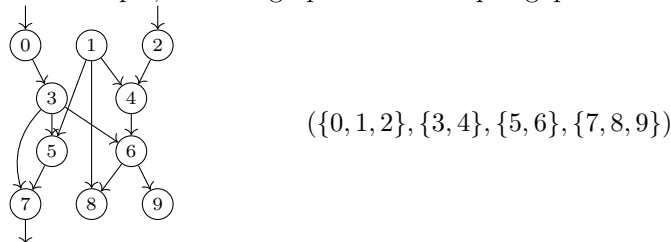
On va a priori se servir de l'algorithme de Dijkstra. ┘

La notion de chemin le plus long est un tout petit peu plus subtile. En particulier, dans les graphes cycliques, il faut faire attention au nombre d'occurrences des arêtes dans les chemins, sinon on pourrait utiliser les cycles pour augmenter indéfiniment leur taille. Même en faisant attention à cet aspect, le problème reste compliqué à résoudre efficacement (en fait il est NP-dur, i.e. on ne sait pas le résoudre plus efficacement qu'avec un algorithme exponentiel).

Heureusement, le problème devient simple dans les graphes acycliques. Une façon de procéder est de d'abord réaliser un *tri topologique*. Un tri topologique

est un ordre (\prec) donné sur les noeuds du graphes tel que si (u, v) est une arête (orientée) du graphe, alors $u \prec v$. Une telle notion n'existe pas s'il y a des cycles, et un même graphe peut souvent donner plusieurs tris topologiques. On propose ici de calculer un tri topologique "compressé vers le haut". On peut représenter ce tri par une séquence d'ensembles $(\ell_i)_i$, tel que $i < j \implies (\forall u \in \ell_i, \forall v \in \ell_j, u \prec v)$, et tel que tout noeud de ℓ_{i+1} a au moins un parent dans ℓ_i . Les ensembles ℓ_i doivent partitionner les noeuds du graphe.

Par exemple, voici un graphe et le tri topologique obtenu :



(On n'a pas représenté les noeuds qui servent d'entrées ou de sorties. On les ignorera ici : on fera comme s'ils n'existaient pas.)

On peut faire cela assez simplement, en réutilisant des idées du test de cyclicité. Le premier ensemble est en fait composé des toutes les "co-feuilles" du graphe (i.e. les noeuds sans parents). Ensuite, si on ôte ces co-feuilles, l'ensemble suivant sera composé des co-feuilles du nouveau graphe, etc...

▮ **Exercice 4 :** Implémenter une méthode qui implémente ce tri topologique. On peut par ailleurs ce faisant détecter si le graphe est cyclique (ce qui arrive s'il n'y a plus de co-feuilles mais que le graphe est non-vide). Renvoyer alors une erreur. ▮

On a choisi particulièrement ce tri topologique car il permet facilement d'obtenir une notion de profondeur des noeuds du graphe et du graphe lui-même. Si $u \in \ell_i$, sa profondeur est i . La profondeur du graphe est le maximum des profondeurs de ses sommets, c'est donc plus simplement le nombre des ensembles ℓ_i .

▮ **Exercice 5 :** Implémenter une méthode qui retourne la profondeur d'un noeud donné dans un graphe. Implémenter ensuite une méthode qui calcule la profondeur d'un graphe. ▮

Le dernier exercice va consister à (enfin) calculer le plus long chemin d'un noeud vers un autre (en considérant qu'on est dans un graphe acyclique).

Attention : il ne suffit pas de faire la différence de profondeur entre les deux noeuds. Par exemple, dans le graphe ci-dessus, la longueur du chemin le plus long de 1 à 5 est bien 1 et non pas 2.

La méthode qu'on va utiliser utilise le tri topologique (et fonctionnerait aussi avec une petite variation pour le calcul du chemin le plus court).

Supposons qu'on veuille calculer le chemin de u vers v dans un graphe dont on connaît un tri topologique $(\ell_i)_i$. On cherche déjà ℓ_k tel que $u \in \ell_k$. Ensuite, pour tous les noeuds w dans ℓ_{k+1} , puis ℓ_{k+2} , puis ..., et tant que $w \neq v$, on

va remplir $dist[w]$ et $prev[w]$ en fonction de leur valeur pour les parents de w .
I.e. si aucun des parents de w n'est dans $dist$, ça veut dire que u n'est pas un ancêtre de w , donc on peut ne rien changer. Sinon, on choisit le parent de $dist$ maximum, on affecte cette valeur $+1$ à $dist[w]$, et on stocke dans $prev[w]$ le parent en question.

▮ **Exercice 6 :** Implémenter une méthode qui calcule le chemin et la distance maximaux d'un noeud u à un noeud v . ▮