Projet Info LDD2 – TD 7

Objectifs du TD : Longueur de chemins.

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 un 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.

⌜ **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. ⌟

Algorithm 1 Algorithme de Dijkstra

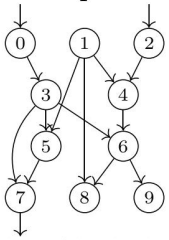
```
function DIJKSTRA(src, direction=None)   ▷ src est le noeud duquel
                                         ▷ on va calculer les distances
                                         ▷ direction ∈ {None, −1, 1}
    Q ← [src]
    dist ← {src: 0}
    prev ← {}
    while Q ≠ [] do
        u ← node id in 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 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. ⌟

Example. Dans :



,

la méthode de l'exercice 3 appliqué à 5 et 8 doit donner $\{0 : (2, 3),\ 3 : (1, 2),\ 1 : (1, 1)\}$.