INFO-F203 : Les bons comptes
Rapport

Carlos Requena López - *410031*
Pedro Filipe Nogueira Cabaço - *414153*

19 décembre 2016

Table des matières

1 Introduction

Ce projet a comme but faciliter la gestion de dettes entre amis (lors qu'ils organisent une fête ou un voyage par exemple). On doit donc pouvoir effectuer des manipulations sur un graphe orienté.

Pour atteindre cet objectif on doit mettre au point plusieurs algorithmes nous permettant de simplifier des dettes, d'identifier des communautés, entre autres.

2 Structure de données

La structure de données choisie est une liste d'adjacence. Cette ci utilise de la mémoire en fonction des arcs, tandis que la matrice d'adjacence (une autre structure de données valable pour répresenter les digraphes) utilise toujours $\mathcal{O}(n \times n)$. En outre, il est plus rapide de parcourir les voisins (ce qu'on fait très souvent) avec la liste d'adjacence.

3 Description des algorithmes

3.1 Simplification des dettes

3.1.1 Contexte

La simplification de dettes se déroule de la manière suivante : si une personne A doit de l'argent a une autre personne B, cette ci doit de l'argent a une autre personne C et cette dernière à première (A), alors les dettes peuvent être simplifiées de façon à ce que une des ces personnes ne doive rien à personne et une autre devra seulement payer mais pas recevoir. Voir fig. 1

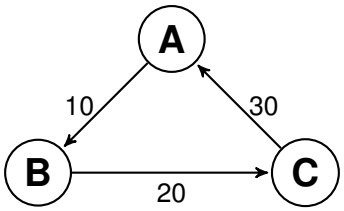

FIGURE 1 – Exemple simplification de dettes

Pour trouver tous les cycles du digraphe efficacement, les composantes fortement connexes (SSC [1]) sont tout d'abord isolées (fig 3) : un arc entre deux composantes fortement connexes (un pont) ne fera jamais partie d'un cycle et ce sera inutile de faire un DFS à travers ces arcs.

Donc, l'algorithme de Tarjan est utilisé en première instance et les SSC de seulement 1 noeud seront retirés.

Afin de trouver tous les cycles dans ces SSC, l'algorithme de Johnson est utilisé [2]. Contrairement à l'algorithme donné dans le syllabus du cours, celui-ci nous donne même les cycles imbriqués et autres.

Finalement, une fois tous les cycles ont été détectés, ils sont minimisés (simplifiés).

3.1.2 Étapes

Par conséquent, la simplification de dettes peut être vu comme un procès en 3 étapes :

— **Isolement des composantes fortement connexes :**

1. strongly connected components

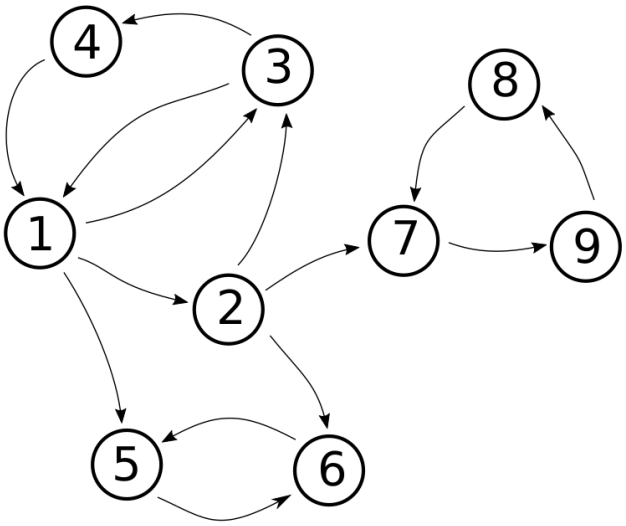

FIGURE 2 – Exemple de graphe dirigé

L'algorithme de Tarjan donné dans le syllabus du cours [1] nous suffit pour trouver ces composantes. Le vecteur `comp` contient le numéro de composante au quelle les noeuds sont attachés.
Il n'y aura pas de cycles en dehors de (entre) ces composantes.

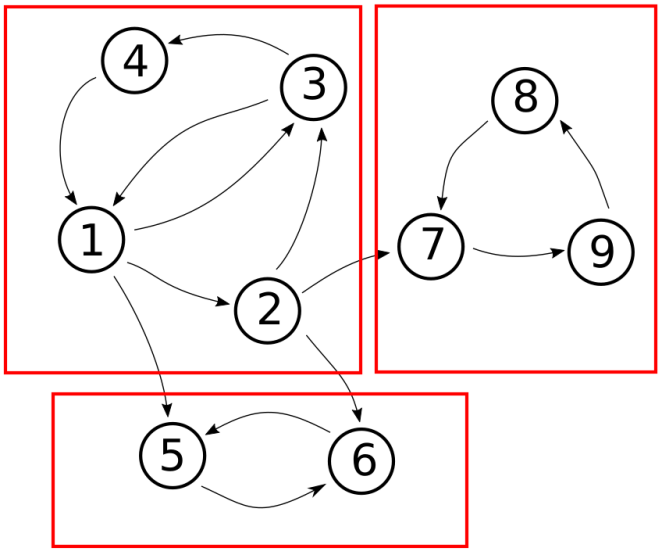

FIGURE 3 – Composantes fortement connexes dans le graphe

— **Identification des cycles - Algorithme de Johnson [2]**
L'algorithme de Johnson pour trouver les cycles d'un digraphe commence par choisir un noeud d'ordre minimale (il doit avoir un ordering interne : dans notre cas, la méthode `__lt__()` établit l'ordre en fonction des lettres des noeuds, par example `A < B`) et tous les noeuds d'un ordre majeur dans **une composante fortement connexe**.
Ce noeud d'ordre minimale sera retiré du graph une fois traité et le noeud de départ sera celui qui a le nouveau ordre minimale.
On parcourt les voisins de ce noeud *s* et les voisins des voisins (en DFS). Pour eviter les

cycles dupliqués, un voisin v est bloqué (on l'ajoute au vecteur `blocked`) et reste bloqué si pour tout chemin de retour de v au noeud de départ s, ceci croisse ou touche notre chemin de parcours originel (qu'on a ajouté dans un stack). Quand on arrive de nouveau à s, un cycle a été trouvé. Si v est bloqué ça veut dire aussi qu'il ne peut pas être utilisé deux fois dans un même cycle.

Notre pseudo-code est le suivant :

```
1  begin find_cycles(current, start):
2      found_cycle = False
3      cycles = []
4
5      stack.append(current)
6      blocked[current] = True
7
8      while next_node is not None:
9          if next_node is the start node:
10             add current stack to cycles array
11             found_cycle = True
12         elif next_node is not blocked:
13             if find_cycles(next_node, start): # DFS
14                 found_cycle = True
15         next_node = get next node
16
17     if found_cycle:
18         unblock(current)
19     else: # Otherwise add relation to blocked map: neighbour ->↩
           me
20         while next_node is not None:
21             if current not in b_map: # block map
22                 append current node to next_node blocked map
23         next_node = get next node
24
25     # Finish by removing 'current' from stack since we're done.
26     stack.remove(current)
27     return found_cycle
28 end find_cycles
29
30 begin unblock(node):
31     blocked[node] = False
32     Bnode = b_map[node]
33     while Bnode:
34         next_node = Bnode.pop(0)
35         if blocked[next_node]:
36             unblock(next_node)
37 end unblock
38
39 # entry point
40 begin get_elementary_cycles():
41     for ssc in tarjan.ssc():
42         for start_node in ssc:
43             least_node = min(ssc)
```

```
44              find_cycles(least_node, least_node)
45              ssc = tarjan.remove_useless_edges(ssc, least_node)
46      return cycles array
47  end get_elementary_cycles
```

— **Simplification des poids des arcs**
Finalement, le digraphe doit être modifié avec les poids reduits. Les étapes sont les suivantes :

— Pour chaque cycle :
— Chercher le poids minimum
— Faire la soustraction entre les poids des arcs concernés et ce poids minimum

3.1.3 Améliorations

D'une manière plus générale, cette simplification pourrait être faite avec une topologie comme celle montrée dans la fig. 4. Même si le cycle n'existe pas, la personne C pourrait payer seulement 20 unités (quelque soit la devise) à A et donner 30 unités a B. Le graphe deviendrait non-dirigé en ce qui concerne les cycles. Cette capacité n'a pas été implémentée dans ce projet mais peut être vu comme une amélioration.

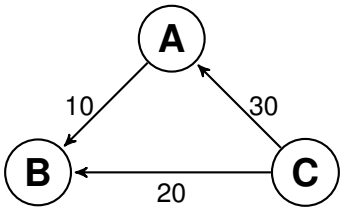

FIGURE 4 – Exemple simplification de 'transactions'

3.2 Identification des communautés

Cet algorithme permet d'identifier tous les communautés et les personnes appartenant à chaque une d'entre elles.

L'algorithme marche de la façon suivante :

Il commence par calculer le nombre de dettes de chaque personne, a fin de minimiser le nombre de fois qu'il doit parcourir la liste *identifier* pour mettre a jour les *id*'s. Il prend la personne qui a plus de dettes et parcourt tous les noeuds qu'il peut atteindre en les marquant comme visités et en les attribuent un même *id*. Si les noeuds n'ont pas encore tous un *id* (si il existe plus qu'une communauté par exemple), on prend la personne qui a plus de dettes et qui n'a pas encore été visité et ont répète le processus. Dans le cas où on tombe sur une personne déjà visité, on met à jour les *id*'s.

À la fin, tous les noeuds (personnes) d'une même composant connexe (communauté) auront le même *id* et on peut donc retourner une liste de communautés.

```
1  communitiesIdentification(g):
2      identifier = [-1]*size; succ = [0]*size; current_id
3      for each node
4          get descendant
5          while descendant
6              ++succ[i]
7              next descendant
8      while not every node visited
9          ++current_id
10         explore(node with most debts && not visited)
11     put nodes with same identifier in the same list
12     return result
13
14 explore(k):
15     if not visited
16         mark as visited
17         update id
18         get descendant
19         while descendant
20             explore(descendant index)
21             next descendant
22     elif node already visited but id needs to be updated
23         for each identifier
24             if needs an update
25                 update
```

3.3 Identification des hubs sociaux

L'algorithme d'identification des hubs sociaux permet d'identifier les points d'articulation d'un graphe tel que sans eux on obtient un plus grand nombre de composantes connexes d'au moins n noeuds.

Il commence par appeler une méthode de *graph* pour obtenir un graphe non dirigé. Ensuite, il fait un parcours en profondeur, marquant les noeuds déjà visites avec un id. Si on trouve un noeud déjà visité avec un id plus petit, on fait une mise à jour.

On sait qu'on a rencontré un point d'articulation quand le *id* de l'élément k est plus petit que le *low* de son fils. Dans ce cas, l'algorithme stocke la taille de la plus petite composante

connexe créée après la suppression de ce point.

```
hubsIdentification(g, n):
    g.makeUnordered
    for each node i:
            explore(i)
    for each articularion point found:
        if smallest community >= k:
            append to result
    return result

explore(i):
    ++current_id
    val[k] = current_id
    low = cid
    while descendant:
        if not visited:
            m = explore()
            if m < low:
                low = m
            if m >= val[i] and not starting node nor marked yet:
                articulation point found
                sol[i] = smallest community created
        else:
            if val[descendant] < low:
                low = val[decendant]
    return low
```

4 Conclusion

On a travaillé en groupe (binôme). Ensemble, on a écrit l'algorithme permettant la lecture d'un fichier, la création du graphe aussi bien que le rapport.
Néanmoins, certaines taches ont étés partages :
Carlos était en chargé de faire la simplification des dettes, tandis que Pedro s'a vu responsable de l'identification des communautés et de l'identification des hubs sociaux.

Avec ce projet on a eu l'opportunité de mettre en pratique une partie de ce qu'on a vu au cours théorique de M. Frédéric Servais.

Ça nous a permis non seulement de milleux comprendre certains algorithmes mais aussi d'approfondir nos connaissances d'algorithmique en général.

Références

[1] Olivier MARKOWITCH Bernard FORTZ. *Algorithmique 2*. Presses Universitaire de Bruxelles, 3éme edition, 2016.

[2] Donald B. JOHNSON. Finding all the elementary circuits of a directed graph. *SIAM J Comput*, 4(1), 3 1975.