TP n°33 - Part I
Calcul de composantes fortement connexes d'un graphe orienté. Algorithme de Kosaraju-Sharir

Dans ce TP, on retravaillera avec des graphes que nous avons déjà utilisés comme exemples :

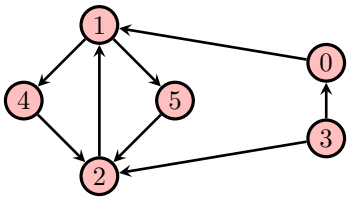


FIGURE 1 – Graphe g_2

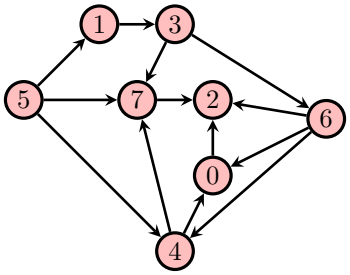


FIGURE 2 – Graphe g_3

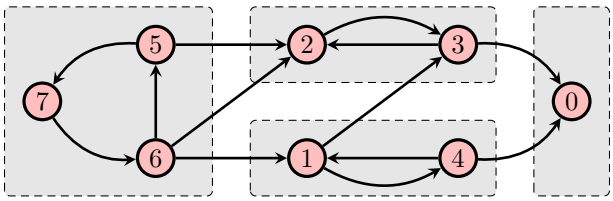


FIGURE 3 – Graphe g_4

Définition 1 (Composante fortement connexe)

> Une *composante fortement connexe* d'un graphe g **orienté** est un sous-ensemble S de ses sommets tel que pour tout couple de sommets $(u, v) \in S^2$, il existe un chemin de u à v dans g.

Le calcul des composantes fortement connexes d'un graphe g orienté peut se faire naïvement en testant l'existence d'un chemin dans le graphe pour chaque couple de sommets. Cette approche effectuant un parcours complet du graphe à chaque départ de sommet est trop coûteuse en temps. **D'autres approches, comme celle de Kosaraju-Sharir, font le même calcul en temps linéaire en la taille du graphe**. L'algorithme comporte deux étapes.

— Il effectue un parcours en profondeur récursif de g et relève, pour chaque nœud visité v_i, l'instant t_i de *fin de traitement* du nœud. Cet instant se situe à la fin de l'exécution de la fonction de parcours sur le nœud v_i, après le retour des appels récursifs sur ses successeurs dans le graphe.

— Il effectue un deuxième parcours en profondeur, cette fois du graphe g' obtenu en renversant le sens de toutes les arcs de g. Les différents sommets de départ du parcours sont choisis non pas dans un ordre arbitraire comme dans l'étape précédente mais dans l'ordre décroissant des instants t_i. Pour chaque sommet de départ v_i, l'ensemble des sommets visités par le parcours en profondeur de g' à partir de v_i forme une composante fortement connexe du graphe initial g.

Exercice 1 (Déroulé de l'algorithme de Kosaraju-Sharir).

On considère le graphe g_4 ci-dessus. On choisit le sommet d'étiquette 0 comme sommet de départ d'un parcours en profondeur.

1. En visitant les successeurs par ordre croissant d'indice, montrer que les instants de fin de traitement des sommets du graphe g_4 de la figure ci-dessus lors d'un parcours en profondeur sont dans l'ordre suivant.

$$t_0 < t_4 < t_2 < t_3 < t_1 < t_6 < t_7 < t_5 \tag{2}$$

2. En déduire la liste des sommets qui permet la mise en œuvre de la second étape de l'algorithme.

3. En déduire les composantes fortement connexes de g_4 dans l'ordre où l'algorithme les construit.

Exercice 2.

Une composante fortement connexe C de g est dite *subordonnée* à une autre composante fortement connexe C' s'il existe des sommets $v_i \in C$ et $v_{i'} \in C'$ et un chemin de $v_{i'}$ à v_i dans g.

Comme C et C' sont des composantes fortement connexes, cela revient à dire qu'il existe des chemins dans g de n'importe quel sommet de C' à n'importe quel sommet de C.

1. Montrer que la relation *être subordonnée à* est une relation d'ordre (pas forcément totale) sur l'ensemble des composantes fortement connexes de C.

2. À chaque composante fortement connexe C on associe un instant t_C de fin de traitement comme ceci

$$t_C = \max_{v_i \in C} t_i$$

où les t_i sont définis comme dans la première étape de l'algorithme. Par exemple, dans le cas de la figure 2 on a

$$t_0 = t_{\{v_0\}} < t_3 = t_{\{v_2,v_3\}} < t_1 = t_{\{v_1,v_4\}} < t_5 = t_{\{v_5,v_6,v_7\}}$$

Montrer que pour tous sommets $v_i \neq v_j$ tels qu'il existe un chemin de v_i à v_j dans le graphe et pas de chemin de v_j à v_i, on a $t_i > t_j$.

3. En déduire que l'ordre total sur les composantes fortement connexes de g défini par les instants de fin de traitement t_C est compatible avec l'ordre partiel *être subordonnée à*, c'est-à-dire que pour toutes composantes C et C' telles que C est subordonnée à C', on a $t_C \leq t_{C'}$.

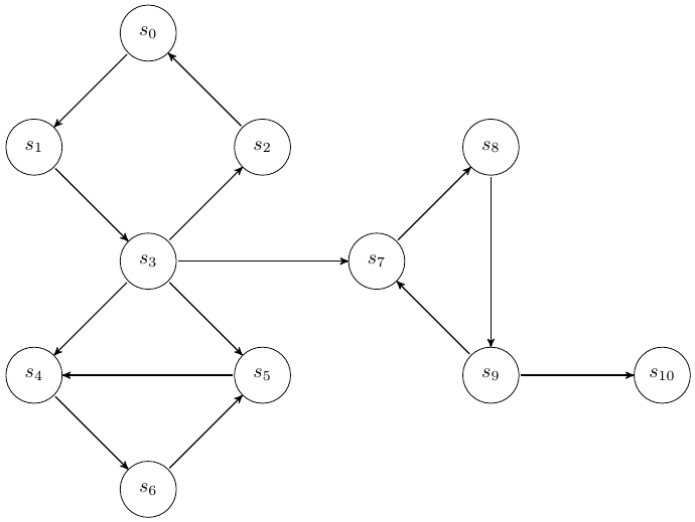
4. En utilisant les résultats des questions précédentes, montrer que le parcours en profondeur du graphe renversé g' lors de la seconde étape de l'algorithme extrait les composantes fortement connexes de g les unes après les autres, dans l'ordre des t_C décroissants. Pour cela, on pourra s'appuyer sur les observations simples suivantes sans les démontrer.

- Les composantes fortement connexes de G' sont les mêmes que celles de G.
- La relation d'ordre *être subordonnée à* dans G' est inversée par rapport à celle dans G.

Exercice 3 (Implémentation de l'algorithme de Kosaraju-Sharir).

Un fichier `cfc-ini.ml` est mis à disposition sur Moodle, avec les graphes ci-dessus déjà construits. On remet également à disposition le module `Wgraph` qui implémente une structure de graphe pondéré par listes d'adjacence.
Toutes les fonctions et tous les tests de cette première partie seront implémentés dans ce fichier OCaml que l'on aura pris soin de renommer, comme d'habitude, `NOM_cfc.ml`

1. Réécrire une fonction `dfs_sort_terminated : 'a Wgraph.t -> int Stack.t` qui prend en entrée un graphe et qui renvoie la pile des indices de ses sommets, empilés au fur et à mesure de leur terminaison lors d'un parcours en profondeur. L'algorithme doit parcourir tous les sommets donc si le sommet de départ ne permet pas de les atteindre tous, le parcours en profondeur doit être redémarrer à partir d'un nouveau sommet non encore visité.
2. Vérifiez sur le graphe g_4 que l'ordre d'empilement correspond bien au résultat trouvé à la main à l'exercice précédent. *Remarque : on a bien fait en sorte dans la construction des graphes que les successeurs soient classés par étiquette croissante, vous pouvez aller le vérifier.*
3. Justifier formellement la complexité linéaire en la taille du graphe de votre fonction `dfs_sort_terminated`.
4. Pour effectuer la seconde étape de l'algorithme, il faut d'abord renverser le graphe. Écrire une fonction `reverse : 'a Wgraph.t -> 'a Wgraph.t` qui prend en entrée un graphe orienté et qui renvoie un autre graphe dans lequel les sommets sont les mêmes et le sens de tous les arcs est inversé. La complexité de la fonction doit être linéaire en la taille du graphe fourni en entrée.
5. Écrire une fonction `strongly_connected_components: 'a Wgraph.t -> int array * int` qui code la seconde étape de l'algorithme. Elle prend en entrée un graphe orienté et renvoie un tableau de taille n_v tel que la case d'indice i contient le numéro (la couleur) de composante fortement connexe du sommet d'indice i. La fonction renvoie également le nombre de composantes fortement connexes. La complexité de la fonction doit être linéaire en la taille du graphe.
6. Tester votre fonction sur les graphes g_2, g_3 et g_4 et sur le graphe suivant, que l'on appellera, g_5 :



TP n°33 - Part 2
Problème 2-SAT - Étude des composantes fortement connexes du graphe d'implication.

On rappelle que toute formule logique φ définie sur n variables propositionnelles peut être mise sous *forme normale conjonctive* (CNF), c'est-à-dire sous forme d'une conjonction de clauses disjonctives (voir cours sur la logique propositionnelles et TP correspondant).

Définition 2 (Problème k-CNFSAT ou k-SAT)

> Le problème qui s'intéresse à la satisfiabilité des formules sous forme CNF dont toutes les clauses sont de longueur maximum k se nomme problème k-CNFSAT, ou, plus simplement, problème k-SAT.

Le problème 3-SAT joue un rôle majeur puisque de nombreux problèmes informatiques, à commencer par les problèmes SAT eux-mêmes, peuvent s'y ramener par ce qu'on appelle une *réduction*. Une conséquence immédiate de ce résultat est que la résolution efficace de 3-SAT entraînerait celle des autres problèmes. Mais 3-SAT est NP-complet et aucune solution efficace à ce problème existe ; on ne sait même pas s'il en existe une. [1]

En revanche, **2-SAT peut être résolu en temps polynomial**.

L'objet de cet exercice est d'établir ce résultat en **transformant une instance du problème 2-SAT sur une formule φ en un calcul de composantes fortement connexes dans un graphe, appelé graphe d'implication.**

Le principe de cette *transformation* repose sur l'observation que toute clause $(l_i \vee l_j)$ est logiquement équivalente à $(\neg l_i) \Rightarrow l_j$, elle même équivalente à sa contraposée $(\neg l_j) \Rightarrow l_i$. Lorsqu'une clause est formée d'un seul littéral l_i, il suffit de l'exprimer sous la forme équivalente $(l_i \vee l_i)$, ce qui donne $(\neg l_i) \Rightarrow l_i$, qui est sa propre contraposée.

Cette observation suggère la procédure suivante pour construire un graphe orienté g, appelé **graphe d'implication**, à partir d'une 2-CNF φ à r variables propositionnelles notées $p_1, \ldots, p_r$.

À chaque variable p_i, $i \in [\![1, r]\!]$, on associe **deux** sommets $v_{2(i-1)}$ et $v_{2(i-1)+1}$ de g :

- $v_{2(i-1)}$ représente le littéral p_i
- $v_{2(i-1)+1}$ représente le littéral $\neg p_i$.

On ajoute ensuite des arcs en fonction de la forme des clauses disjonctives à 1 ou 2 littéraux de la 2-CNF :

Chaque clause à un seul littéral de type (l_i) est équivalente à $(l_i \vee l_i)$.

- Si $l_i = p_i$, on a donc :

  $$p_i \equiv p_i \vee p_i \equiv \neg p_i \rightarrow p_i$$

  On ajoute alors un arc du sommet $v_{2(i-1)+1}$ au sommet v_{2i} pour représenter cette implication.
- Sinon, si $l_i = \neg p_i$, on a donc :

  $$\neg p_i \equiv \neg p_i \vee \neg p_i \equiv p_i \rightarrow \neg p_i$$

  On ajoute alors un arc du sommet $v_{2(i-1)}$ au sommet $v_{2(i-1)+1}$ pour représenter cette implication.

Pour chaque clause du type $(l_i \vee l_j)$ où les littéraux l_i, l_j contiennent des variables p_i, p_j distinctes, on ajoute deux arcs à g, choisis en fonction des cas suivants :

- Si les deux littéraux sont positifs $p_i \vee p_j$, alors :
  - on a $p_i \vee p_j \equiv \neg p_i \rightarrow p_j$ et on ajoute l' arc $(v_{2(i-1)+1}, v_{2(j-1)})$ pour représenter cette implication
  - mais on a aussi $p_i \vee p_j \equiv \neg p_j \rightarrow p_i$ (contraposée), que l'on représente dans le graphe en ajoutant l'arc $(v_{2(j-1)+1}, v_{2(i-1)})$

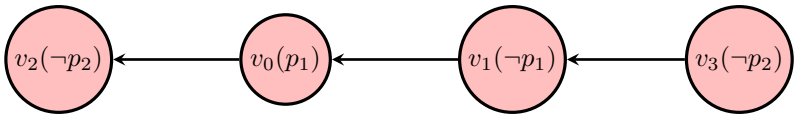
1. Toutes ces notions seront expliquées et approfondies l'an prochain

— Si le premier littéral est positif et pas le second $p_i \vee \neg p_j$, alors :
  — on a $p_i \vee \neg p_j \equiv \neg p_i \rightarrow \neg p_j$ et on ajoute l' arc $(v_{2(i-1)+1}, v_{2(j-1)+1})$ pour représenter cette implication
  — mais on a aussi $p_i \vee \neg p_j \equiv p_j \rightarrow p_i$ (contraposée), que l'on représente dans le graphe en ajoutant l'arc $(v_{2(j-1)}, v_{2(i-1)})$
— Si le premier littéral est négatif et le second positif, $\neg p_i \vee p_j$, alors :
  — on a $\neg p_i \vee p_j \equiv p_i \rightarrow p_j$ et on ajoute l' arc $(v_{2(i-1)}, v_{2(j-1)})$ pour représenter cette implication
  — mais on a aussi $\neg p_i \vee p_j \equiv \neg p_j \rightarrow \neg p_i$ (contraposée), que l'on représente dans le graphe en ajoutant l'arc $(v_{2(j-1)+1}, v_{2(i-1)+1})$
— Si les deux littéraux sont négatifs $\neg p_i \vee \neg p_j$, alors :
  — on a $\neg p_i \vee \neg p_j \equiv p_i \rightarrow \neg p_j$ et on ajoute l'arc $(v_{2(i-1)}, v_{2(j-1)+1})$ pour représenter cette implication
  — mais on a aussi $\neg p_i \vee \neg p_j \equiv p_j \rightarrow \neg p_i$ (contraposée), que l'on représente dans le graphe en ajoutant l'arc $(v_{2(j-1)}, v_{2(i-1)+1})$

Enfin, pour chaque clause du type $(l_i \vee l_j)$ où les littéraux l_i et l_j contiennent la même variable $p_i = p_j$, soit on élimine directement la clause si elle est de la forme $(p_i \vee \neg p_i)$ ou $(\neg p_i \vee p_i)$ (principe du tiers-exclu), soit on se ramène au cas d'une clause à un littéral (l_i) déjà vu.

Exemple 1

Avec la formule $p_1 \wedge (p_2 \vee \neg p_1)$, la procédure donne un graphe à quatre sommets v_0, v_1, v_2, v_3 et à trois arcs comme illustré sur la figure ci-dessous :



En effet :

- la clause p_1, a un seul littéral, correspond à :

$$p_1 \equiv p1 \vee p_1 \equiv \neg p_1 \rightarrow p_1$$

  et nous permet de tracer l'arc (v_1, v_0)
- la clause $p_2 \vee \neg p_1$, a un 2 littéraux, correspond à :

$$p_2 \vee \neg p_1 \vee p_1 \equiv p_1 \rightarrow p_2$$

  ou à la contraposée :

$$p_2 \vee \neg p_1 \vee p_1 \equiv \neg p_2 \rightarrow \neg p_1$$

  et nous permet de tracer les deux arcs (v_0, v_2) et (v_3, v_1).

Exercice 4 (Lien entre une formule 2-CNF et son graphe d'implication).

1. Dessiner le graphe d'implication de la formule logique suivante, qui est bien sous forme 2-CNF :

$$\varphi = p_3 \wedge (\neg p_1 \vee p_3) \wedge (p_1 \vee p_2)$$

2. Donner la formule logique associée au graphe g_4
3. Supposons que l'on s'intéresse à une 2-CNF φ satisfiable et soit v une valuation modèle, c'est-à-dire telle que $[\![\varphi]\!]_v = V$.
   On désigne par g le graphe d'implication associé à φ.
   Montrer que, pour tous les sommets v_k et v_l situés dans une même composante fortement connexe de g, les variables propositionnelles $p_{\lfloor k/2 \rfloor + 1}$ et $p_{\lfloor l/2 \rfloor + 1}$ associées à ces sommets vérifient :
   - $[\![p_{\lfloor k/2 \rfloor + 1}]\!]_v = [\![p_{\lfloor l/2 \rfloor + 1}]\!]_v$ si $(k - l)$ est pair
   - $[\![p_{\lfloor k/2 \rfloor + 1}]\!]_v = [\![\neg p_{\lfloor l/2 \rfloor + 1}]\!]_v$ si $(k - l)$ est impair.
4. En déduire que si φ est satisfiable alors il n'existe pas de variable p_i tels que les deux sommets correspondants $v_{2(i-1)}$ et $v_{2(i-1)+1}$ soient dans une même composante fortement connexe du graphe d'implication g.

Réciproquement, on peut montrer que, s'il n'existe pas de variable propositionnelle p_i de φ tels que les deux sommets correspondants $v_{2(i-1)}$ et $v_{2(i-1)+1}$ soient dans une même composante fortement connexe du graphe d'implication g, alors, en attribuant la même valeur de vérité à tous les littéraux impliqués dans une même composantes fortement connexe de g, on construit une valuation qui satisfait φ. Ceci fournit un critère simple pour décider de la satisfiabilité de φ. Nous allons utiliser ce résultat pour proposer une résolution d'une instance du problème 2-SAT en temps polynomial.

Exercice 5 (Implémentation d'un solver 2-CNFSAT).
On copiera le fichier `NOM_cfc.ml` dans un nouveau fichier `NOM_2sat.ml` et on implémentera les fonctions qui suivent à la suite de ce nouveau fichier.

1. On propose le type OCaml `cnf` suivant pour représenter une formule sous forme 2-CNF :

```
type clause = int list
type cnf = clause list
```

   Une formule CNF est donc représentée par une `int list list`.
   Expliquer ce type et les conventions sous-jacentes.
2. Implémenter une fonction `var_max: int list list -> int` qui renvoie l'indice maximale des variables propositionnelles utilisées dans une formule logique écrite sous forme 2-CNF. Tester votre fonction sur les formules de l'exemple et de l'exercice précédent.
3. Écrire une fonction `graph_of_2cnf: int list list -> float Wgraph.t` qui construit le graphe d'implication associé à une formule logique 2-CNF. Tester votre fonction sur la formule $p_1 \wedge (p_2 \vee \neg p_1)$ dont le graphe d'implication a été donné en exemple.
4. En utilisant toutes les fonctions précédentes, le travail effectué dans la partie I de ce TP et les résultats théoriques évoqués dans l'exercice précédent, écrire une fonction `is_2sat: int list list -> bool` qui indique si une formule logique 2-CNF est satisfiable ou non. Tester votre fonction sur des formules 2-CNF contradictoires ou satisfiables simple. Votre fonction doit répondre `true` sur $p_1 \wedge (p_2 \vee \neg p_1)$ et `false` sur la formule logique associée au graphe g_4 par exemple. Testez abondamment.
5. Vérifier que le solveur 2-CNFSAT implémenté a bien une complexité temporelle linéaire en le nombre de clauses de la formule 2-CNF.