TRAVAUX PRATIQUES XII

Algorithme de Kosaraju et résolution de 2-SAT

L'objectif de ce TP est d'implémenter l'algorithme de Kosaraju pour la recherche de composantes fortement connexes d'un graphe, et d'utiliser cet algorithme pour proposer une résolution en temps linéaire du problème 2-SAT.

Dans tout ce TP, on interdit l'utilisation de utop ! Il va falloir lire des données sur l'entrée standard (terminal). Le plus simple sera donc de compiler votre programme OCaml et de lui fournir une entrée dans le terminal à l'aide d'un **pipe**.

A Sérialisation d'un graphe dans un fichier

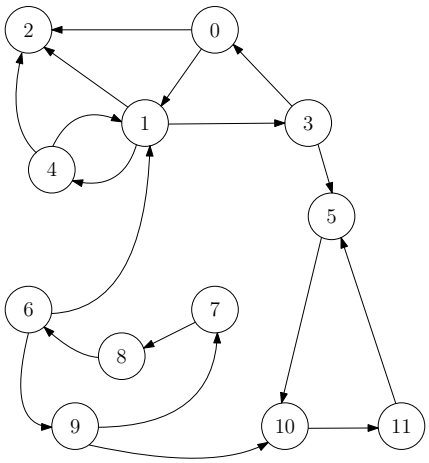
On définit le format suivant pour sérialiser un graphe orienté :

- la première ligne contient deux entiers n et p, séparés par une espace (n le nombre de sommets, p le nombre d'arcs) ;
- les p lignes suivantes contiennent chacune deux entiers séparés par une espace – une ligne `x y` signifie qu'il y a un arc du sommet x vers le sommet y.

On suppose qu'il n'y a pas de répétition d'arcs. Plusieurs graphes sont fournis dans des fichiers :

- Le fichier `exemple_cours.txt` correspond à la figure XII.1 du cours, c'est à dire la suivante :

FIGURE XII.1 – Le graphe G_1.



- Le graphe `arxiv.txt` a pour sommets des articles de recherche publiés sur ArXiv dans la rubrique *High-energy physics*, et possède un arc de x vers y si l'article x cite l'article y.

Une fonction `read_graph : unit -> graph` est fournie, qui lit un graphe sous ce format **sur l'entrée standard** (terminal). Cette fonction suppose que les sommets sont numérotés de 0 à $n-1$.

Remarque (HP) :

— La fonction `Scan.scanf` permet de lire sur l'entrée standard. Cette fonction prend en entrée une chaîne de format et une fonction qui est appliquée aux valeurs lues.

Exemple de format : `"%d %f"!` permet de lire un entier et un flottant séparés par un nombre quelconque d'espaces, de tabulations et de retours à la ligne. Ainsi, par exemple, l'appel `Scanf.scanf "%d %d" (fun x y -> x + y)"` lit deux entiers et renvoie leur somme.

Rappelons les commandes suivantes, qui seront utiles au cours de ce TP :

- `cat fichier` affiche le contenu de `fichier` sur l'entrée standard.
- Le *pipe* `e1 | e2` permet de donner la sortie de la commande `e1` en entrée de la commande `e2`.

Vous n'avez pas besoin de comprendre la fonction `read_graph`, mais le reste du fichier `kosaraju` fourni est essentiel.

1. Lisez le fichier `kosaraju.ml` et prenez le temps de le comprendre. N'hésitez pas à poser des questions !
2. Compilez et testez ce fichier ! Donnez lui d'abord un graphe de votre choix en le tapant vous-mêmes dans le terminal. Puis fournissez-lui un fichier en entrée à l'aide d'un *pipe* et de la commande `cat`. Par exemple, pour le fichier `arxiv.txt`, vous devriez obtenir l'affichage suivant :

Terminal
```
--- Test de lecture du graphe. ---
Le sommet d'indice 0 du graphe a les voisins suivants :
6 5 4 3 11 2 10 9 1 8 7
```

B Algorithme de Kosaraju

On travaille sur des graphes orientés et non pondérés, qui seront représentés **par listes d'adjacence** par le type OCaml suivant :

OCaml
```
type sommet = int

type graphe = sommet list array
```

Pour un graphe $G = (S, A)$, on notera sauf mention explicite du contraire $n = |S|$ et $p = |A|$.

À chaque fois qu'une question demande d'écrire une fonction, on précisera, en la justifiant, la complexité de cette fonction : je vous encourage fortement à rédiger ses justifications pour gagner en efficacité sur ce type de questions.

L'algorithme de Kosaraju a été expliqué en détail dans le cours, et un pseudo-code précis a été donné. Cette partie du TP sera donc assez directe : il s'agit de coder, en OCaml, l'algorithme de Kosaraju.

Il nous faut d'abord deux fonctions intermédiaires, une pour calculer le transposé d'un graphe, et une pour faire le parcours en profondeur initial (qui déterminera l'ordre dans lequel on traite les sommets) :

3. Écrire une fonction `transpose : graphe -> graphe` qui prend en entrée un graphe G et renvoie son graphe transposé (ou miroir) $G^{\leftarrow}$.
4. Écrire une fonction `dfs_post : graphe -> sommet list` qui effectue un parcours en profondeur complet d'un graphe et renvoie la liste de ses sommets par instant de fin de traitement décroissant.
   Indication : Comme d'habitude, on pourra représenter `vus` *par un tableau de booléens, tel que* `vus.(x)` *vaut* `true` *si et seulement si* $x \in$ `vus`.

Ici, on pourrait découper davantage Kosaraju avec des fonctions auxiliaires (notamment la fonction Ajouter) du cours. Pour coller fidèlement au pseudo-code donné dans le cours, nous allons maintenant directement coder l'algorithme de Kosaraju :

5. Écrire une fonction `kosaraju : graphe -> sommet list list` qui prend en entrée un graphe G et renvoie la liste de ses composantes fortement connexes $C_1, \dots, C_l$ (chaque C_i sera représentée par la liste de ses sommets)
6. Si ce n'est pas déjà le cas, modifiez votre fonction `kosaraju` pour qu'elle renvoie la liste des composantes dans l'ordre topologique de G_{CFC}. Autrement dit, la liste doit vérifier :

$$x \in C_i, y \in C_j, xy \in A \Rightarrow i \leqslant j$$

Remarques/aide :

— On a montré dans le cours que l'algorithme de Kosaraju calcule les composantes fortement connexes dans l'ordre topologique (à chaque étape, on calcule une composante **source** dans le graphe des sommets non marqués). Il suffit donc de les renvoyer dans cet ordre.

— ⚠Attention à ne **jamais** ajouter des éléments successivement en fin de liste ! Ces concaténations successives seraient en temps $O(|l|^2)$ avec l la liste dans laquelle on ajoute. On préférera toujours calculer la liste "à l'envers" en ajoutant bien en tête de liste, puis la renverser avec `List.rev` .

7. Est-il important que le parcours du graphe soit fait en profondeur :
- dans la fonction `dfs_post` ?
- dans la fonction `kosaraju` ?

Tests de Kosaraju

8. Écrire un programme OCaml qui lit un graphe sur l'entrée standard et affiche :
- son nombre de composantes fortement connexes ;
- la taille de sa plus grande composante fortement connexe.

9. Tester votre programme sur les fichier `exemple_cours.txt` et `arxiv.txt`.
Pour le graphe `arxiv.txt`, on doit trouver 21608 composantes fortement connexes, dont la plus grande contient 12711 sommets.

Félicitation, vous avez implémenté (et testé !) l'algorithme de Kosaraju !

C Problème 2-SAT

C.1 Définitions et types utilisés

Rappelons quelques définitions utilisées dans ce sujet :
- un **littéral** est soit une variable propositionnelle x_i, soit la négation $\neg x_i$ d'une variable ;
- si l est un littéral, on considère que $\neg l$ en est également un ($\neg l = x_i$ si $l = \neg x_i$) ;
- une **clause disjonctive** est une disjonction de littéraux ne contenant pas de variables répétées ;
- une formule en **forme normale conjonctive** (ou CNF) est une conjonction de clauses disjonctives ;
- le problème CNF-SAT consiste à déterminer si une formule en forme normale conjonctive est satisfiable (et éventuellement à trouver un témoin de satisfiabilité, c'est à dire une valuation qui la satisfait) ;
- le problème k-SAT est la restriction de CNF-SAT aux formules dans lesquelles chaque clause disjonctive est constituée d'exactement k littéraux.

On définit les types suivant. Remarquez qu'on n'utilise pas ici des entiers positifs ou négatifs pour représenter un littéral, mais un type somme [1].

OCaml
```
type litteral =
  | P of int (* occurrence positive *)
  | N of int (* occurrence negative *)

type clause = litteral * litteral
type deuxcnf = clause list
type valuation = bool array
```

On supposera pour les calculs de complexité que les variables présentes dans une formule sont numérotées consécutivement à partir de zéro.

C.2 Résolution en force brute

10. Écrire deux fonctions `eval_litt : litteral -> valuation -> bool`
et `eval : deuxcnf -> valuation -> bool` prenant en entrée respectivement un littéral ou une formule en 2-CNF, ainsi qu'une valuation, et renvoyant leur évaluation dans ce contexte.
On supposera que la valuation fournie est « assez grande ».

11. Écrire une fonction `incremente_valuation` qui prend en entrée une valuation $[v_0, \dots, v_{n-1}]$ et la modifie pour en faire la valuation suivante dans l'ordre lexicographique, ou lève l'exception `Last` si la valuation est déjà maximale (vaut `[true; ...; true]`).

1. Encore une fois, c'est bien de changer un peu de représentation.

OCaml
```
exception Last
val incremente_valuation : valuation -> unit
```

12. Écrire une fonction `brute_force : deuxcnf -> valuation option` qui prend en entrée une formule φ en 2-CNF et renvoie :

- `Some v`, où `v` est un témoin de satisfiabilité de φ, s'il en existe un ;
- `None` si φ est insatisfiable.

Indication : n'hésitez pas à utiliser des fonctions auxiliaires ! Comme d'habitude, une fonction pour trouver la plus grande variable de φ peut être utile, par exemple…

La complexité obtenue pour cette méthode étant exponentielle, on va maintenant en changer.

C.3 Réduction de 2-SAT à la recherche de CFC

Nous avons vu en TD-cours comment réduire 2-SAT à un problème de composantes fortement connexes dans un graphe orienté. Nous allons maintenant implémenter cette réduction et utiliser l'algorithme de Kosaraju pour conclure.

Graphe associé à une formule en 2-CNF

Rappelons la transformation utilisée :

Soit $\varphi = \bigwedge_{i=1}^{r} F_i$ une formule en 2-CNF (chaque F_i est donc de la forme $l \vee l'$, où l et l' ne contiennent pas la même variable) et $x_0, \dots, x_{n-1}$ les variables présentes dans φ. On définit le graphe $G_\varphi = (S_\varphi, A_\varphi)$ comme suit :

- il y a un sommet pour chaque littéral possible (autrement dit, $2n$ sommets $x_0, \neg x_0, \dots, x_{n-1}, \neg x_{n-1}$) ;
- pour chaque clause $l \vee l'$, il y a deux arcs $\neg l \to l'$ et $\neg l' \to l$.

13. On considère la formule suivante :

$$\varphi = (x_0 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_3 \vee \neg x_0)$$

Représenter son graphe G_φ.

14. Écrire une fonction `graphe_de_cnf : deuxcnf -> graphe` prenant en entrée une formule φ en 2-CNF et renvoyant le graphe G_φ associé. On pourra supposer qu'aucune clause ne contient deux fois le même littéral, ni un littéral et sa négation.

Indication : on numérotera les sommets de la manière suivante : pour chaque variable x_i de φ, le littéral x_i est représenté par le sommet $2i$, et le littéral $\neg x_i$ par le sommet $2i+1$.

Satisfiabilité

On a montré en TD-cours que si l' est accessible depuis l dans G_φ, alors $\varphi \vDash l \to l'$.
Rappel : le symbole de conséquence logique $P \vDash Q$, prononcé "P satisfait Q", signifie que toute valuation qui satisfait P satisfait aussi Q. Ici, on veut donc dire que si une valuation v satisfait φ, alors elle satisfait l'implication « l implique l' ».

On a en fait prouvé en TD-cours que φ est satisfiable si et seulement si aucune composante fortement connexe de G_φ ne contient à la fois un littéral et sa négation. On en déduit donc une méthode pour déterminer la satisfiabilité de φ en utilisant l'algorithme de Kosaraju.

15. Écrire une fonction `satisfiable : deuxcnf -> bool` déterminant si une formule φ en 2-CNF est satisfiable. On exige une complexité linéaire en la taille de la formule (nombre d'occurrences de variables).

Indication : dans un tableau, associer à chaque littéral le numéro de sa composante fortement connexe (on pourra les numéroter dans l'ordre topologique obtenu grâce à l'algorithme de Kosaraju).

16. Testez votre fonction sur la formule φ donnée plus haut. Pour vous aider :

OCaml
```
let exemple_f = [(P 0, N 2); (P 1, P 3); (N 1, P 2); (N 2, P 3); (P 3, N 0)]
```

C.4 Témoin de satisfiabilité

On souhaite plus que la simple information satisfiable / insatisfiable. Dans le cas où la formule φ est satisfiable, on aimerait fournir en plus une valuation qui la satisfait. On va appliquer la construction de valuation vue en TD-cours.

On considère une formule φ en 2-CNF et le graphe G_φ associé, dont l'on note $C_1, \dots, C_r$ les composantes fortement connexes, dans un ordre topologique, i.e. $C_1 \prec \cdots \prec C_r$. Pour un sommet $x \in S$, on note $C(x)$ la composante fortement connexe de G contenant x.

Si φ est satisfiable (aucune variable et sa négation n'apparaissent dans la même CFC), on définit la valuation suivante :

$$v : x \mapsto \begin{cases} \text{Vrai} & \text{si } C(\neg \text{x}) < C(\text{x}) \text{ (pour le tri topologique)} \\ \text{Faux} & \text{sinon, auquel cas } C(\text{x}) < C(\neg \text{x}) \end{cases}$$

17. En vous inspirant de la fonction `satisfiable`, écrire une fonction `temoin : deuxcnf -> valuation option` qui prend en entrée une formule φ dont les variables sont $x_0, \dots, x_{n-1}$ et renvoie :

- `None` si φ est insatisfiable ;
- `Some v`, où `v` est un tableau de taille n codant une valuation de $(x_0, \dots, x_{n-1})$ satisfaisant φ, sinon.

On exige une complexité linéaire en la taille de la formule.

D Pour aller plus loin

18. Écrire une fonction `lit_graphe : string -> graphe` qui prend en argument un nom de fichier et construit le graphe en lisant directement dans le fichier.

Indications :

- *Le cours sur la lecture de fichier en OCaml se trouve dans le TP « Algorithme de Huffman ».*
- *Vous pouvez utiliser la fonction* `split_on_char` *pour séparer votre chaîne de caractères (allez voir la doc).*
- *N'oubliez pas l'existence de la fonction* `int_of_string`.

19. Rédigez au propre toutes les preuves de complexité et montrez-les moi.

On définit le type OCaml suivant :

OCaml
```
type arcs = Arbre | Avant | Arriere | Lateral
```

20. Implémenter l'algorithme de classification des arcs vu en cours, pour un parcours en profondeur du graphe. On explorera les sommets dans l'ordre croissant.

Cette fonction prendra en entrée un graphe et renverra un tableau de listes d'adjacence tel que chaque liste d'adjacence contient le type de l'arc, en plus du sommet voisin. Autrement dit, si `g.(s)` contient v (i.e. v est un voisin de s dans le graphe), alors la liste d'adjacence renvoyée sera telle que `g.(s)` contient `(v, Arbre)` si $s \to v$ est un arc arbre, `(v, Avant)` si c'est un arc avant, etc.

`classification : graphe -> sommet * arcs list array`