TP 8 : Graphes

20 et 27 janvier 2025

1 Couplage dans un graphe biparti

Nous allons ici implémenter l'algorithme de recherche d'un couplage de cardinal maximum dans un graphe biparti vu en classe.

On utilisera les notations suivantes :

- $\mathrm{G} = (\mathrm{X} \sqcup \mathrm{Y}, \mathrm{E})$ désigne un graphe biparti - on note $V = X \sqcup \mathrm{Y}$ son ensemble de sommets;
- $X = \{0, \dots, n_1 - 1\}, Y = \{n_1, \dots, n - 1\}$ - toutes les arêtes relient donc un sommet d'indice strictement inférieur à n_1 à un sommet d'indice supérieur ou égal à n_1;
- on note $p = |E|$ le nombre d'arêtes du graphe;
- on note $A \oplus B = (A \cup B) \backslash (A \cap B)$ la différence symétrique de deux ensembles A et B - on rappelle que cette opération est associative et commutative.

On représente un graphe biparti par le type suivant :

```
type bipartite = {
n1 : int;
adj : int list array;
}
```

Un couplage est représenté par :

```
type matching = int option array
```

Pour un couplage `m`, on aura :

- `m.(i)= None` si le sommet `i` est libre;
- `m.(i)= Some j` et `m.(j)= Some i` si les sommets `i` et `j` sont appariés.

Un chemin $x_0, \dots, x_k$ sera simplement représenté par la liste $[x_0; \dots; x_k]$:

```
type path = int list
```

Remarque 1. *Mathématiquement, on verra parfois un chemin de longueur k comme une liste de k arêtes, parfois comme une liste de $k+1$ sommets. En revanche, il sera toujours représenté informatiquement par une liste de $k+1$ sommets.*

▷ **Question 1.** Écrire une fonction `print_matching : matching -> unit` qui prend en entrée un couplage et affiche les couples de sommets appariés : un couple par ligne. ◁

▷ **Question 2.** Écrire une fonction `correct_matching : bipartite -> matching -> bool` qui teste si un tableau correspond à un couplage correctement formé. ◁

▷ **Question 3.** Écrire une fonction `cardinal_matching : matching -> int` qui renvoie le nombre d'arêtes présentes dans le couplage passé en argument. ◁

▷ **Question 4.** Écrire une fonction `is_augmenting : matching -> path -> bool` prenant en entrée un couplage `m` et un chemin et indiquant si le chemin est augmentant. On supposera sans le vérifier que le chemin est élémentaire et que les entiers sont bien entre 0 et la taille du tableau `m`.
◁

Remarque 2. *Ces fonctions ne sont pas nécessaires pour la suite mais peut aider à identifier d'éventuelles erreurs dans le code.*

▷ **Question 5.** Écrire une fonction `delta : matching -> path -> unit` prenant en entrée un couplage `m` et un chemin `p` supposé augmentant pour `m` et effectuant l'opération $m \leftarrow m \oplus p$, où $\oplus$ représente la différence symétrique.
◁

Remarque 3. *Mathématiquement, on voit ici p et M comme des ensembles d'arêtes.*

▷ **Question 6.** Écrire une fonction `orient : bipartite -> matching -> int list array` prenant en entrée un graphe biparti `g` et un couplage `m` et renvoyant un graphe orienté G_m (sous forme d'un int list array) tel que :

- les sommets de G_m sont les mêmes que ceux de `g`;
- G_m contient exactement un arc orienté pour chaque arête $\{x, y\}$ (avec $x \in X$ et $y \in Y$) de `g` :
  - si $\{x, y\} \in m$, l'arc de G_m est (y, x);
  - si $\{x, y\} \notin m$, l'arc de G_m est (x, y).

Cette orientation est celle présentée dans le cours - on ne rajoute en revanche pas de sommet source ni de sommet puit.
◁

On définit l'exception suivante

```
exception Found of int
```

▷ **Question 7.** Écrire une fonction
`explore : bipartite -> matching -> int array-> int -> int-> unit` telle que
`explore g m peres p x` fait un parcours en profondeur récursif dans le graphe orienté `g` depuis le sommet `x` $\in X$ de père (dans l'arborescence de parcours) `p`, l'arborescence du parcours est stockée dans le tableau `peres` et le parcours s'arrête dès lors que l'on accède à un sommet de $y \in Y$ libre et lève l'exception `Found y` le cas échéant. Si aucun tel sommet n'est accessible depuis `x` alors la fonction ne fait rien. ◁

▷ **Question 8.** Écrire une fonction `reconstruct_path : int -> int array-> int list` telle que `reconstruct_path x peres` reconstruit le chemin entre `x` et la racine dans l'arborescence obtenue par un parcours en profondeur stockée dans le tableau `peres` passé en argument.
◁

▷ **Question 9.** Écrire une fonction `chemin_augmentant : bipartite->matching-> int list option` prenant en entrée un graphe biparti `g` et un couplage `m` et renvoyant :

- `None` si `m` n'admet pas de chemin augmentant;
- `Some p`, avec `p` un chemin augmentant, s'il en existe un.

◁

▷ **Question 10.** Écrire une fonction `get_maximum_matching : bipartite -> matching` prenant en entrée un graphe biparti `g` et renvoyant un couplage `m` de `g` de cardinalité maximale.
◁

▷ **Question 11.** Déterminer la complexité de la fonction précédente. ◁

▷ **Question 12.** Une série de fichiers nommés `graph_n1_n2_c.txt` sont fournis : chacun décrit un graphe biparti $G = (X \sqcup Y, E)$ avec $|X| = n_1, |Y| = n_2$ et c le cardinal maximal d'un couplage de `g`. Le format de fichier est très simple, mais il est inutile de s'y intéresser : une fonction `read_bipartite` est fournie. Elle prend en entrée un nom de fichier (ou plutôt un chemin d'accès) et renvoie un bipartite.

Écrire un programme, que l'on compilera et exécutera en ligne de commande, qui accepte en argument un nom de fichier et affiche le cardinal du couplage renvoyé par `get_maximum_matching` sur le graphe correspondant. Vérifier que vous obtenez bien les valeurs attendues. On rappelle que pour lire un argument de l'exécutable on utilise `Sys.argv.(1)`. ◁

2 KOSARAJU ET 2SAT

L'objectif de cette partie est d'implémenter le test de satisfiabilité d'une formule du calcul propositionnel sous forme 2SAT.

1. Rappeler la définition d'une formule sous forme 2SAT.

   Pour représenter une telle formule, on utilisera le type suivant :

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

   type clause = litteral * litteral
   type twocnf = clause list
   ```

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

   Soit $\phi = \bigwedge_{i=1}^{r} F_i$ 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 $G_\phi = (V_\phi, E_\phi)$ 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 $\neg l' \to l$.

2. On considère la formule suivante :

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

   Représenter son graphe G_ϕ.

3. Ecrire une fonction `max_var : twocnf->int` qui renvoie l'indice maximal d'une variable apparaissant dans la formule passée en entrée.

4. Écrire une fonction `graph_of_cnf: twocnf->graph` 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 son complémenté. Pour le type graphe, on utilisera `type graph = int list array`. Tester votre fonction sur le graphe `ex` fourni. (on doit obtenir `[|[6];[5];[4];[6];[6;0];[3];[];[1,5,2]|]`).

5. Rappeler la condition nécessaire et suffisante sur le graphe G_ϕ vue en cours qui caractérise le fait qu'une formule ϕ sous forme 2SAT est satisfiable.

6. Afin d'utiliser cette caractérisation, on va implémenter l'algorithme de Kosaraju.

   (a) Écrire une fonction `transpose` qui prend en entrée un graphe G et renvoie son graphe transposé (ou miroir).

   (b) Écrire une fonction `post_order` qui effectue un parcours en profondeur complet d'un graphe et renvoie la liste de ses sommets par instant de fin de traitement décroissant.

   (c) Compléter le code fourni pour obtenir une fonction `kosaraju : graph-> int list list` qui renvoie une liste contenant chaque composante fortement connexe du graphe passé en entrée. Chaque CFC est elle même une liste.

   (d) Tester votre fonction en vérifiant que le fichier `arxiv.txt` représente un graphe à 21608 composantes fortement connexes (une fonction de lecture du fichier vous est fournie).

7. Écrire une fonction `satisfiable` 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).