IS 1

22 septembre-durée 1h

1. Donner la formule des poignées de main qui relie la somme des degrés d'un graphe non orienté à son nombre d'arêtes.

Dans $G=(S,A)$. On a

$$\sum_{s \in S} \deg(s) = 2|A|$$

2. Dans quelle famille d'algorithmes se trouve l'algorithme de Dijkstra? Sous quelle représentation est-il préférable de prendre le graphe en entrée pour une implémentation efficace de cet algorithme? Quelle est sa complexité si on utilise une file de priorité sous forme de tasmin et une bonne représentation du graphe?

C'est un algo glouton.
Il est préférable d'avoir des listes d'adjacence.
Dans ce cas, la complexité est

$$O(|S| + |A| \log |S|)$$

3. Quelle est la complexité d'un parcours en largeur quand le graphe passé en entrée est donné sous forme de matrice d'adjacence ?

$$O(|S|^2)$$

4. Ecrire une fonction en Caml `pp : int list array-> int array` qui prend en entrée un graphe `g` sous forme de listes d'adajacence et qui renvoie un tableau `cc` indexé par les sommets de `g` tel que deux sommets x et y seront dans une même composante connexe de `g` ssi `cc.(x)=cc.(y)`. On impose d'utiliser une stratégie de parcours en **profondeur**. On pourra, pour faire simple, choisir n'importe quel sommet d'une classe d'équivalence pour être un représentant de celle-ci.

```
let pp g =
  let n = Array.length g in
  let vu = Array.make n false in
  let cc = Array.make n (-1) in
  let rec pp-rec num s =
      if (not vu.(s)) then
      begin vu.(s) <- true
            cc.(s) <- num
            List.iter (pp-rec num) g.(s) end
  in for i = 0 to n-1 do (pp-rec i i) done; cc
```

5. En admettant l'existence des fonctions `creer_file : unit-> file`, `enfiler : file-> int -> unit` et `defiler : file ->int`, écrire en Caml une fonction `pl_arbo : int list array-> int -> int array` qui prend en entrée un graphe `g` sous forme de listes d'adajacence et un sommet d'origine `s` et qui renvoie un tableau `peres` tel que, pour chaque sommet `x`, `peres.(x)` est le père de `x` dans l'arborescence d'un parcours en **largeur** de `g` depuis `s`.

```
let pl-arbo g s =
  let n = Array.length g in
  let vu = Array.make n false in
  let peres = Array.make n (-1) in
  let f = creer-file () in
  enfiler f s;
  vu.(s) <- true;
  while (not (est-vide f)) do
      let x = defiler f in
      List.iter (fun v -> if (not (vu.(v))) then
                          begin
                          vu.(v) <- true;
                          peres.(v) <- x;
                          enfiler f v;
                          end) g.(x)
  done;
  peres
```

6. Définir ce qu'est une heuristique admissible dans le cadre de l'algorithme A*. Que peut-on dire de A* quand il est exécuté avec une heuristique admissible ?

h est admissible ssi $\forall x \in S$, $h(x) \leq d(x,t)$ = la distance dans G de x à t avec t la destination.

Dans ce cas, si A* termine alors il renvoie bien $d(s,t)$

7. Définir ce qu'est une heuristique monotone dans le cadre de l'algorithme A*. Que peut-on dire de A* quand il est executé avec une heuristique monotone ?

$\forall (x,y) \in S^2$, $h(x) \leq h(y) + p(x,y)$.

Garanti terminaison et complexité polynomiale

8. Traduire en terme de date de découverte et de date de fin de traitement le fait qu'un sommet x est descendant d'un sommet y dans un parcours en profondeur.

$$dec(y) < dec(x) < fin(x) < fin(y)$$

9. Expliquer brièvement une stratégie permettant d'obtenir un tri topologique dans un graphe orienté acyclique à l'aide d'un parcours en profondeur.

On fait un pp et on récupère les elts dans l'ordre inverse de leurs dates de fin de traitement. car si $x \to_G y$ alors $fin(y) \leq fin(x)$

10. Donner la définition formelle des composantes fortement connexes d'un graphe orienté.

Ce sont les classes d'équivalence de la relat° suivante : $\forall (x,y) \in S^2 : x \sim y$ ssi $x \rightsquigarrow_G y$ et $y \rightsquigarrow_G x$

les graphes induits par

11. Soit $G = (S, A)$ un graphe orienté. Donner la définition de son graphe miroir.

$\overline{G} = (S, A')$ avec $A' = \{(x,y) : (y,x) \in A\}$

12. Ecrire en Caml une fonction `miroir : int list array->int list array` qui prend en entrée un graphe `g` sous forme de listes d'adjacence et renvoie une représentation de son graphe miroir.

```
let miroir g =
    let n = Array.length g in
    let gm = Array.make n [] in
    for i = 0 to n-1 do
      List.iter (fun v -> gm.(v) <- i :: gm.(v)) g.(i)
    done;
    gm
```

13. Expliquer les étapes de l'algorithme de Kosaraju.

$G = (S, A)$

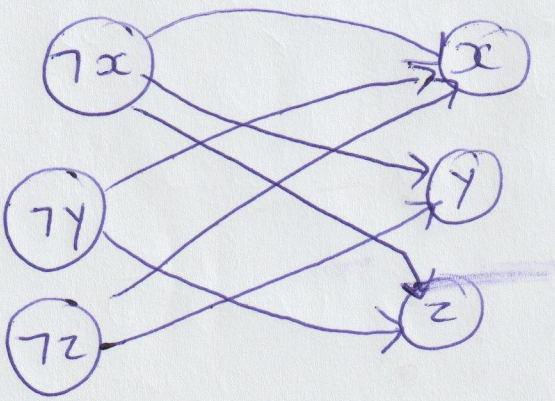
1) on calcule le graphe miroir de G, noté $\overline{G}$

2) on fait un pp de $\overline{G}$ pour récupérer les sommets dans l'ordre inverse de leurs dates de fin de traitement.

3) on fait un pp de G où la boucle principale lance successivement les parcours récursifs sur les sommets dans l'ordre obtenu précédemment. Chaque parcours rec visite une CFC exactement.

14. Représenter le graphe d'implications et déterminer la satisfiabilité de la formule suivante. On donnera un modèle de φ s'il en existe un.

$\varphi = (x \vee z) \wedge (y \vee x) \wedge (z \vee y) \wedge x \wedge (y \vee z)$

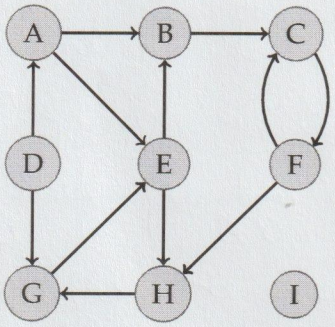


Il y a 6 CFC

$\{\{x\}, \{y\}, \{z\}, \{\neg x\}, \{\neg y\}, \{\neg z\}\}$ qui ne contiennent jamais un littéral et sa négation donc φ est satisfiable.

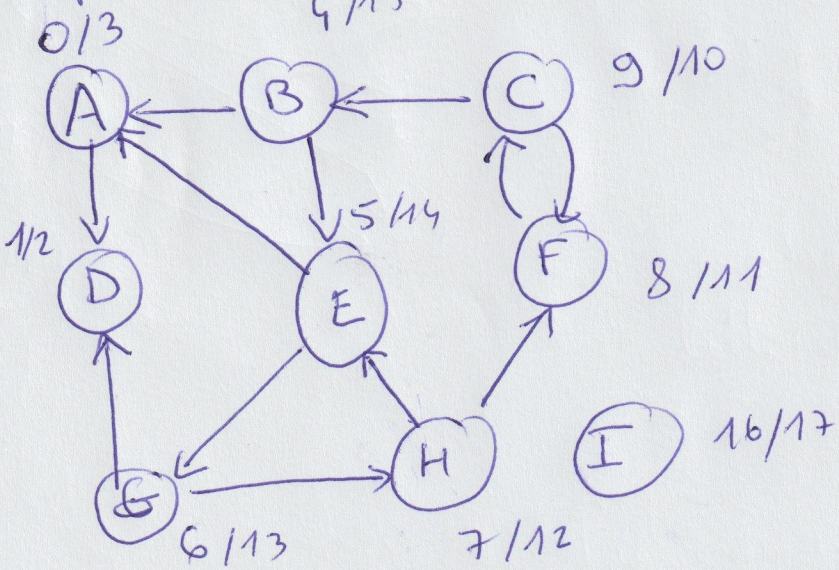
Par exemple, $\sigma(x) = \sigma(y) = \sigma(z) = \top$ vérifie bien $\sigma \models \varphi$.

On considère le graphe suivant :

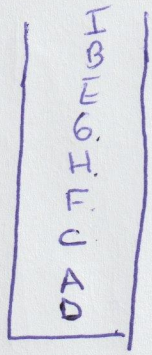


15. Dérouler l'algorithme de KOSARAJU sur le graphe ci-dessus, pour en trouver ses composantes fortement connexes. Dessiner et annoter le graphe miroir ainsi que le graphe initial avec les temps de visite des parcours en profondeur utilisés. Lorsque plusieurs choix seront possibles on utilisera toujours l'ordre alphabétique. Représenter sous la forme d'une pile l'ordre dans lesquels les sommets vont être considérés pour le second parcours.

G et parcours :



Pile obtenue



Parcours de G : on commence par

pp-rec I : {I}

pp-rec B : {B, C, F, H, G, E}

pp-rec A : {A}

pp-rec D : {D}