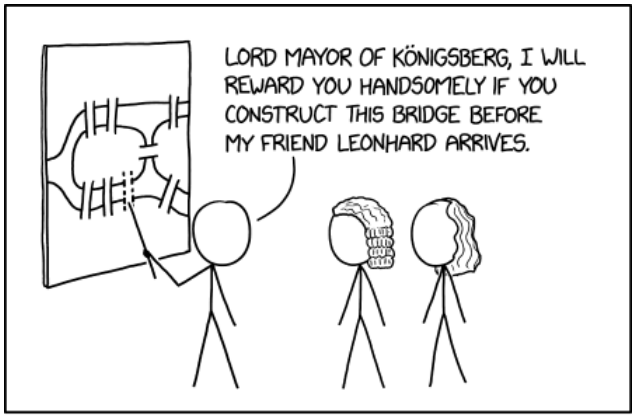
MPI* Info

TD Graphes

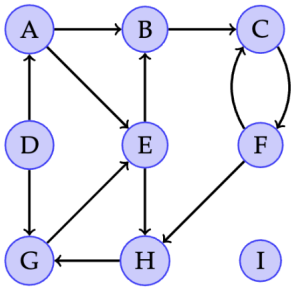


I TRIED TO USE A TIME MACHINE TO CHEAT ON MY ALGORITHMS FINAL BY PREVENTING GRAPH THEORY FROM BEING INVENTED.

Olivier Caffier

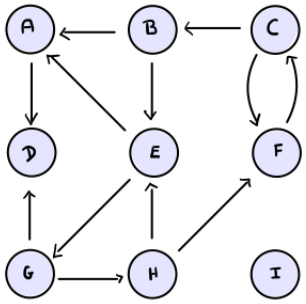
1 Kosaraju

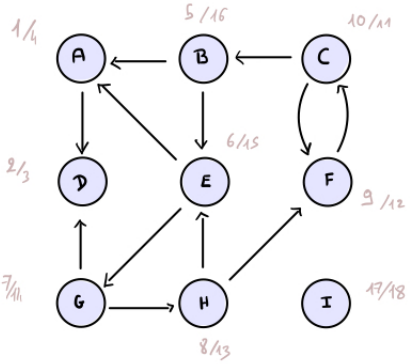
On considère le graphe suivant :



→ Dérouler l'algorithme de Kosaraju-Sharir 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 profondeurs 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. **Corrigé :**

$\vec{G}$:





Liste triée par ordre décroissant de fin de traitement

L = [~~I~~ ; ~~B~~ ; ~~E~~ ; ~~G~~ ; ~~H~~ ; ~~F~~ ; ~~C~~ ; A ; D]

I → (I)

B → B ; C ; F ; H ; G ; E

A → A

D → D

2 Composantes connexes par Unir et Trouver

On considère un graphe $G = (S, A)$ avec $S = [\![0; n-1]\!]$. On représente un graphe non-orienté par la liste de ses arêtes (chaque arête est représentée exactement une fois).

```
type graph = {nb_sommets : int; aretes : (int * int) list} ;;
```

On considère le type suivant pour représenter une partition avec l'algorithme unir et trouver :

```
type partition = {pere : int array; rang : int array} ;;
```

où `rang.(i)` représente la hauteur de l'arbre enraciné en `i` et cette valeur n'est effectivement garantie à jour que lorsque `i` est bien son propre père.

1. Écrire la fonction `creer_singletons : int -> partition` qui initialise une partition de $[\![0; n-1]\!]$ avec n singletons.
   Corrigé :

```
let creer_singleton (n : int) : partition =
  { pere = Array.init n (fun i -> i);
    rang = Array.make n 0 }
```

2. Écrire la fonction `trouver : partition -> int -> int -> unit` qui réalise l'union par rang. On admet qu'en utilisant ces deux heuristiques, la complexité d'une séquence de n opérations `creer_singletons`, trouver et unir est en $\mathcal{O}(n\alpha(n))$ où α est une fonction à croissance extrêmement lente.
   Corrigé :

```
let rec trouver (struct : partition) (n : int) : int =
  let p = struct.pere.(n) in
  if p = n then n
  else begin
    let r = trouver struct p in
    struct.pere.(n) <- r ;
    r
  end
```

3. Écrire une fonction `composantes_connexes : graph -> partition` qui calcule la partition représentant les composantes connexes. Quelle est la complexité de cette approche ?
   Corrigé :

```
let composantes_connexes (g : graph) : partition =
  let res = creer_singletons (g.nb_sommets) in
  let rec aux (li : (int * int) list ) : unit =
    match li with
    |[] -> ()
    |(x,y)::xs -> unir res x y ; aux xs
  in
  aux g.aretes
```

   On obtient alors une complexité en $\mathcal{O}(|S| + |A|\alpha(|S|))$

4. Quelle autre approche pourrait-on utiliser pour calculer les composantes connexes et quelle en serait la complexité ?
   Corrigé :

   On pourrait utiliser un parcours en profondeur mais cela n'est pas du tout pratique avec cette représentation imposée : le parcours serait en $\mathcal{O}(|A||S|)$. On pourait modifier cette représentation du graphe pour améliorer la dite complexité.

5. On suppose que l'on ajoute une nouvelle arête au graphe (en déclarant le champ mutable ou en renvoyant un nouveau graphe avec une arête de plus). Que faudrait-il faire pour mettre à jour les composantes connexes du graphe pour chacune des deux approches ? Discuter des avantages et inconvénients de ces dernières.
   Corrigé :

   — 1ère approche : un seul appel à `unir` (avec les deux extrémités de l'arête) → $\mathcal{O}(\alpha(|S|))$.
   — 2ème approche : modifier la liste d'adjacence, on aurait un coût linéaire dans ce cas.

3 Diamètre d'un graphe

Dans cet exercice, on considère des graphes non-orientés connexes.
Les sommets d'un graphe à $n \in \mathbb{N}^*$ sommets sont numérotés de 0 à $n-1$. On suppose qu'aucune arête ne boucle sur un même sommet.
Un chemin de longueur $p \in \mathbb{N}$ d'un sommet a vers un sommet b dans un graphe G est un chemin de longueur minimale parmi tous les chemins de a vers b, sa longueur est noté $d_G(a,b)$.

Définition - Diamètre d'un graphe

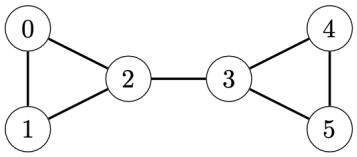
Le *diamètre d'un graphe G*, noté $\text{diam}(G)$ vaut le maximum des longueurs des plus courts chemins entre deux sommets du graphe G.
Autrement dit,

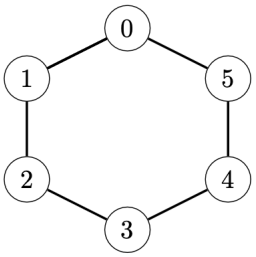
$$\text{diam}(G) = \max_{a,b \in S} d_G(a,b)$$

On dit alors qu'un chemin de plus court chemin de G de longueur $\text{diam}(G)$ est un *chemin maximal*.

3.1 Exemples de graphes

1. Donner sans justification le diamètre et les chemins maximaux pour chacun des deux graphes G_1 et G_2 ci-dessous.





Corrigé :

$$\text{diam}(G_1) = 3 \text{ et } \text{diam}(G_2) = 3$$

```
type graphe = int list array ;;
```

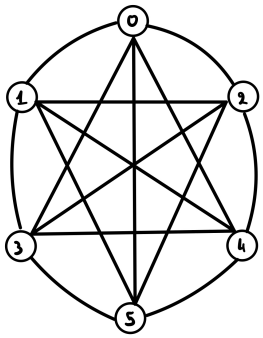
2. Graphes de diamètre maximal.
   (a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus grand possible.
   Corrigé :

   (b) Écrire une fonction `diam_max : int -> graphe` qui renvoie un graphe à $n \in \mathbb{N}^*$ sommets de diamètre maximal.
   Corrigé :

```
let diam_max (n : int) : graphe =
  Array.init n (fun i -> [ (i+1) mod n ; (i-1 + n) mod n])
```

3. Graphes de diamètre minimal.

   (a) Dessiner sans justification un graphe à 5 sommets ayant un diamètre le plus petit possible.
   Corrigé :



   (b) Écrire une fonction `diam_min : int -> graphe` qui renvoie un graphe à $n \in \mathbb{N}^*$ sommets de diamètre minimal.
   Corrigé :

```
let diam_min (n : int) : graphe =
  let rec aux (count : int) (i : int) : int list =
    match count with
    |x when x < 0 -> []
    |x when x = i -> aux (count -1) i
    |_ -> count::(aux (count-1) i)
  in
  Array.init n (fun i -> aux (n-1) i)
```

3.2 Algorithme de calcul du diamètre

Dans cette partie, on suppose que les graphes sont représentés par listes d'adjacence.

1. Donner l'entrée et la sortie de l'algorithme de Dijkstra. Comment cet algorithme permet-il de calculer le diamètre d'un graphe ?
   Corrigé :

   ➢ **Entrée** : un graphe $G = (S, A, p)$ un graphe orienté pondéré tq $p(A) \subset \mathbb{R}_+$, un sommet $s \in S$
   ➢ **Sortie** : δ un tableau de distance les plus courtes par rapport à s tq. $\forall x \in S, \delta(x) = d_G(s,x) \in \mathbb{R}_+ \cup \{+\infty\}$

   Le lien avec d_G nous indique une méthode pour calculer le diamètre du graphe, il suffit juste de parcourir δ pour trouver son maximum.
2. Quel parcours de graphe peut être utilisé pour le calcul du diamètre ?
   Corrigé :

   Un parcours en largeur semble être une option envisageable.
3. Laquelle des deux méthodes est la mieux adaptée pour calculer le diamètre d'un graphe ?
   Corrigé :

   ➢ Parcours en largeur : $\mathcal{O}((|S|+|A|)|S|) = \mathcal{O}(|S|^2 + |S||A|)$
   ➢ Dijkstra : $\mathcal{O}((|S|+|A|\log|S|)|S|) + \underbrace{\mathcal{O}(|S|^2)}_{\text{recherche du max.}} = \mathcal{O}(|S|^2 + |A||S|\log|S|)$

   Dijkstra semble donc moins adapté…