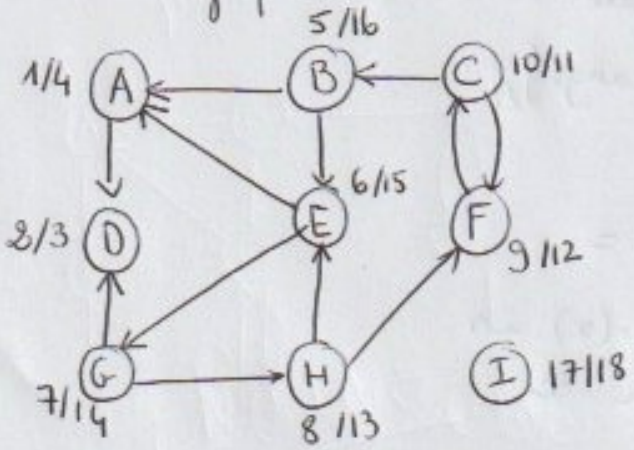
Correction du concours Banc.

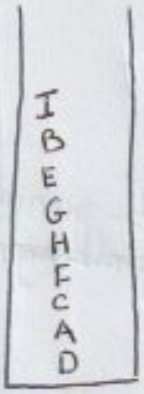
① Graphes

1.1 graphe miroir



On fait un pp à partir de A et on note les dates de découverte / fin de traitement.

La pile des sommets par ordre de fin de traitement obtenue est



On fait donc un parcours en profondeur du graphe initial en commençant par I :

visiter I donne CFC de I : {I}
visiter B donne CFC de B : {B, C, F, H, G, E}
visiter A donne CFC de A : {A}
visiter D donne CFC de D : {D}

1.2.

① let creer_singletons n =

```
let peres = Array.init n (fun i -> i) in
let rangs = Array.make n 0 in
{ pere = peres ; rang = rangs };;
```

② let rec trouver p x =

```
let parent = p.pere.(x) in
if (parent = x) then x
else
  begin
    let res = trouver p parent in
    p.pere.(x) <- res;
    res
  end;;
```

③ let composantes_connexes g =

```
let part = creer_singletons (g.nb_sommets) in

let rec parcourir l = match l with
    | [] -> ()
    | (x,y)::suite -> unir part x y ;
                      parcourir suite

in
parcourir g.aretes ;
part;;
```

La complexité est $O(n + m\,\alpha(n))$
où n est le nb de sommets
et m le nb d'arêtes.

④ On peut utiliser un parcours pour trouver les composantes connexes, par exemple un parcours en profondeur. Ici la représentat° du graphe n'est pas des plus confortable : pour chaque sommet on doit considérer ses voisins et pour cela parcourir toutes les arêtes ce qui donne une complexité en $O(n\,m)$. Transformer la représentation actuelle du graphe donnerait une complexité en $O(n+m)$ plus le coût de la transformation qui serait en $O(n+m)$ (en effet, on parcourt la listes des arêtes une fois pour ajouter ce qu'il faut dans les listes d'adjacence) et donc $O(n+m)$ au total.

⑤ Si on ajoute une arête alors avec la structure union et trouver, il suffit de faire un appel à la fonction trouver avec les extrémités de l'arête.

Dans la version avec un parcours, les composantes connexes sont obtenues sous forme de listes et l'ajout d'une arête nécessite de chercher dans quelles listes se trouvent les extrémités afin d'éventuellement concaténer leurs listes respectives si nécessaire. Le coût de recherche est $O(n)$ (la somme des longueurs des listes vaut n) puis la concaténat° peut avoir un coût $O(n)$ (suivant la longueur des listes à concaténer). Ainsi la mise à jour a un coût linéaire ici alors qu'avec la première version on a un coût amorti en $O(\alpha(n))$.

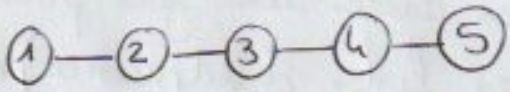
1.3.

① G_1 : $diam(G_1) = 3$ avec 0 2 3 4
0 2 3 5
1 2 3 4
1 2 3 5
chemins maximaux.

$diam(G_2) = 3$ avec 0 5 4 3
5 4 3 2
5 0 1 2 etc... comme chemins maximaux.

②

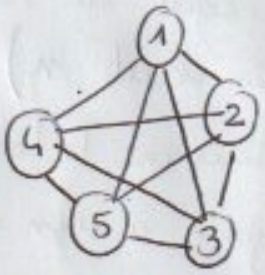
(a)



(b)

```
let diam_max n =
    let g = Array.make n [] in
    for i = 1 to (n-2) do
        g.(i) ← [(i+1) ; (i-1)]
    done; g.(0) ← [1] ; g.(n-1) ← [n-2];
    g;;
```

③ (a)



(b)

```
let diam_min n =
    let g = Array.make n [] in
    for i = 0 to n-1 do
        g.(i) ← ltsaufi i n;
    done;
    g;;
```

où la fonction ltsaufi i n renvoie une liste contenant tous les entiers entre 0 et n-1 sauf i

```
let toutsaufi i n =
    let rec aux k = matth k with
        |k when k=n -> []
        |k when k=i -> aux (k+1)
        |_ -> k::(aux(k+1))
    in aux 0;;
```

1.3.2.

① L'algorithme de Dijkstra prend en entrée un graphe pondéré et un sommet s et calcule le poids d'un plus court chemin de s à chacun des sommets du graphe. En utilisant une pondération des arêtes constante égale à 1, on obtient la distance de chaque sommet à s. En appliquant Dijkstra sur chaque sommet, on obtient les distances pour tous les couples de sommets et on peut en déduire le diamètre en cherchant un maximum.

② parcours en largeur.

③ parcours en largeur : $O(n+m) \times n$ pour chaque sommet

$+ O(n^2)$ pour recherche du max.

on obtient $O(n^2 + nm)$

Dijkstra : $O((n+m\log n) \times n + O(n^2) = O(n^2 + nm(\log n))$

c'est légèrement plus coûteux.