Correction du l'Interrogation 4 du Lundi 8 Décembre

Exercice 1 (Tri topologique d'un graphe orienté)

1. $6-8-3-0-1-2-4-7-5$ ou $0-8-1-7-3-4-2-5-6$.
2. Raisonnons par l'absurde en supposant que G admet un cycle $s_1 s_2 ... s_{k-1} s_1$, k étant un entier supérieur ou égal 3.

   Suivant le tri topologique de G :
   — comme l'arc (s_1, s_2) appartient à A, s_1 est rangé avant s_2,
   — comme l'arc (s_2, s_3) appartient à A, s_2 est rangé avant s_3,
   — ...
   — comme l'arc (s_{k-2}, s_{k-1}) appartient à A, s_{k-2} est rangé avant s_{k-1},
   — comme l'arc (s_{k-1}, s_1) appartient à A, s_{k-1} est rangé avant s_1,
   ce qui est impossible.

   Donc G est sans cycle.
3. (a) Considérons un chemin C dans G de longueur maximale noté $s_0 s_1 ... s_k$.

   Montrons que s_0 est alors de degré entant 0 dans G :
   — Soit le sommet s_0 admet un prédécesseur p n'appartenant pas à $\{s_1, ..., s_k\}$ et dans ce cas, le chemin $p s_0 s_1 ... s_k$ est un chemin de G de longueur strictement supérieure à la longueur de C, chemin de longueur maximale : CONTRADICTION !
   — Soit le sommet s_0 admet un prédécesseur p appartenant à $\{s_1, ..., s_k\}$ et dans ce cas, le chemin $p s_0 s_1 ... s_i (= p)$ est un cycle de G : CONTRADICTION !

   (b) Prenons le sommet s_0 de degré entrant 0 comme le premier sommet dans le tri topologique.

   Le graphe G_1 obtenu à partir du graphe G privé du sommet s_0 est toujours sans cycle et possède également un sommet s_1 de degré entrant 0, que nous prendrons en deuxième position.

   Nous répétons cette opération jusqu'à ce que tous les sommets de G aient été placés.

   Et nous avons ainsi construit un tri topologique du graphe G.
4. `let g = [|[1; 7]; [2; 7]; [5]; [2; 4]; [5]; []; []; []; [7]|]`
5. (a)
```
let rec suppression x l = match l with
  | [] -> []
  | t::q when t=x -> suppression x q
  | t::q -> t::(suppression x q)
;;
```
   (b) `let suprime_arc g a b = g.(a) <- suppression b g.(a)`

   (c) La fonction `aux` qui retourne 1 si x appartient à une liste l, 0 sinon :

```
let rec aux x l = match l with
  | [] -> 0
  | t::q when t=x -> 1
  | t::q -> aux x q
;;
```

   Il reste à parcourir toutes les listes d'adjacence du graphe G et à sommer le nombre d'occurence de x :

```
let degre g a = let n = Array.length g and d = ref 0 in
  for k = 0 to n-1 do
    d := !d + aux a g.(k)
  done;
  g
;;
```

(d) On créé un tableau de booléen pour ne pas considérer plusieurs fois les sommets de degré entrant 0. A chaque sommet a de degré entrant 0, on ajoute a à la liste l et on supprime tous les arcs sortants de a. On recommence jusqu'à avoir traité tous les sommets.

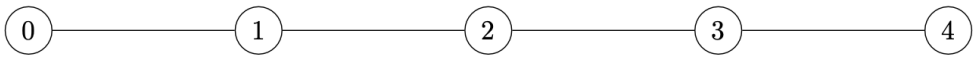
```
let tri_topologique g = let n = Array.length g and l = ref [] in
  let t = Array.make n true in
  for k = 0 to n-1 do
    for a = 0 to n-1 do
      if t.(a) && (degre g a = 0) then
        begin
          l := a::( !l);
          t.(a) <- false;
          for b = 0 to n-1 do
            supprime_arc g a b
          done
        end
    done;
  done;
  List.rev !l
;;
```

Exercice 2 (Diamètre d'un graphe)

1. G_1 a pour diamètre 3 et pour chemins maximaux $0-2-3-4$, $0-2-3-5$, $1-2-3-4$ et $1-2-3-5$.

   G_2 a pour diamètre 3 et pour chemins maximaux $0-1-2-3$, $0-5-4-3$, $1-2-3-4$, $1-0-5-4$, $2-3-4-5$, $2-1-0-5$, $3-4-5-0$, $3-2-1-0$, $4-5-0-1$, $4-3-2-1$, $5-0-1-2$ et $5-4-3-2$.

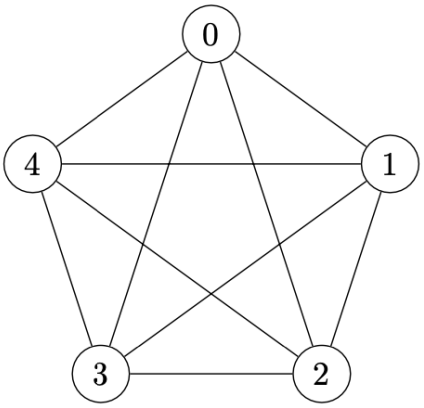
2. (a) Voici le graphe demandé :



(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;
    if n > 1 then
      begin
        g.(0) <- [1];
```

```
        g.(n-1) <- [n-2]
      end
    g
;;
```

3. (a) Un graphe complet (où toutes les arêtes possibles sont présentes) donne un diamètre de 1, qui est bien minimum :



(b)
```
let diam_min n =
  let g = Array.make n [] in
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        if i <> j then g.(i) <- j::g.(i)
      done;
    done;
    g
;;
```

4. Entrée : un graphe G (orienté ou non) pondéré dont tous les poids sont positifs et un sommet s de G.

   Sortie : la distance pondérée de s à chaque sommet de G (par exemple sous forme d'un tableau).

   Étant donné un graphe non pondéré, on peut mettre un poids de 1 sur chaque arête et la distance pondérée est alors égale à la distance définie par l'énoncé. Il suffit ensuite d'appliquer une fois l'algorithme de Dijkstra depuis chaque sommet du graphe : la distance maximum trouvée est alors le diamètre.

5. On peut utiliser un parcours en largeur depuis chaque sommet du graphe, ce qui permet aussi d'obtenir toutes les distances donc le diamètre.

6. Etant donné un graphe à n sommets et p arrêtes, on sait d'après le cours que la complexité de l'algorithme de Dijkstra ($O(p\ln(n))$) est plus élevée que celle du parcours en largeur ($O(n+p)$). Comme dans les deux cas on applique l'algorithme autant de fois qu'il y a de sommets, il est préférable d'utiliser des parcours en largeur.

7. `let a = Noeud (0, Noeud (1, Noeud (2, Noeud (4, Feuille, Feuille), Feuille), Noeud (3, Noeud (5, Feuille, Feuille), Noeud (6, Feuille, Feuille))), Feuille) ;;`

   Le diamètre de G_A est 4 et ses chemins maximaux sont $4-2-1-3-6$ et $4-2-1-3-5$.

8. Soit A un arbre binaire enraciné en s et dont l'ensemble des noeuds est N. La fonction qui à chaque noeud v de $N\backslash\{s\}$ associe l'arc aboutissant en v est une bijection de $N\backslash\{s\}$ vers l'ensemble des arcs de A. On en déduit que $r = n - 1$.
9. ```
   let rec nb_noeuds a = match a with
     | Feuille -> 0
     | Noeud(r , g , d) -> 1 + nb_noeuds g + nb_noeuds d
   ;;
   ```
10. ```
    let numerotation a =
      let c = ref (-1) in
        let rec aux arb = match arb with
          | Feuille -> Feuille
          | Noeud(_, g, d) -> c := !c + 1;
                              let r = !c in
                                Noeud(r, aux g, aux d)
                              in
        aux a
    ;;
    ```
11. On parcourt l'arbre en reliant chaque noeud avec son père :

    ```
    let arbre_vers_graphe a =
      let ga = Array.make (nb_noeuds a) [] in
        let rec aux p arb = match arb with
          | Feuille -> ()
          | Noeud(r, g, d) -> if p <> -1 then
                                begin
                                  ga.(r) <- p::ga.(r);
                                  ga.(p) <- r::ga.(p)
                                end
                                aux r g;
                                aux r d
                              in
        aux (-1) a;
        ga
    ;;
    ```
12. On peut numéroter les n sommets de l'arbre avec `numerotation`, le transformer en graphe avec `arbre_vers_graphe`, puis calculer son diamètre en utilisant des parcours en largeur.

    `numerotation` et `arbre_vers_graphe` parcourent chaque sommet de l'arbre une fois en faisant un nombre constant d'itérations, donc sont en $O(n)$. Comme le nombre d'arête de l'arbre est $n - 1$, la méthode pour calculer le diamètre dans le graphe obtenu est en $O(n^2)$.

    D'où la complexité totale $O(n) + O(n) + O(n^2) = O(n^2)$.
13. On note $|C|$ la longueur d'un chemin C. Si C est vide (c'est-à-dire qu'il ne contient aucun sommet) on définit $|C| = -1$. On pose aussi $h(\text{Feuille}) = 0$ (non défini par l'énoncé).

    Soit A un arbre et C un chemin maximal de G_A. Supposons que C passe par la racine de A.

Soit C_g la partie de C dans G_{A_g} et $\overrightarrow{C_g}$ le chemin correspondant dans A_g. Montrons que $|C_g| = h(A_g) - 1$.
$\overrightarrow{C_g}$ est un plus long chemin de la racine de A_g à un noeud de A_g (sinon, on pourrait remplacer C_g dans C par un chemin plus long, ce qui contredirait la maximalité de C). Donc

$$|C_g| = |\overrightarrow{C_g}| = h(A_g) - 1.$$

Remarquons que cette formule reste vraie si A_g est une feuille.
On raisonne de même pour la partie C_d de C dans G_{A_d} d'où

$$|C| = |C_g| + 2 + |C_d| = h(A_g) + h(A_d)$$

(on rajoute 2 pour les arêtes sortantes de la racine de A).

14. Le diamètre est obtenu récursivement en remarquant qu'un chemin de longueur maximum est soit entièrement dans A_g (donc de longueur égale au diamètre de A_g), soit entièrement dans A_d (donc de longueur égale au diamètre de A_d) soit passe par la racine (donc de longueur $h(A_g) + h(A_d)$, d'après la question précédente).
On a besoin à la fois du diamètre et de la hauteur dans cette formule de récurrence, il est donc judicieux d'utiliser une fonction auxiliaire qui renvoie les deux informations :

```
let diam_arbre arb =
  let rec aux a = match a with
    | Feuille -> (-1, 0)
    | Noeud(_, g, d) -> let dg, hg = aux g in
                        let dd, hd = aux d in
                        (max (max dg dd) (hg + hd), 1 + max hg hd) in
    fst (aux arb)
;;
```

`aux a` effectue un appel récursif pour chaque noeud de a et chacun de ces appels effectue un nombre constant d'opérations (en dehors des appels récursifs) donc la complexité de cette fonction est bien linéaire en le nombre de noeuds.