Ordres pour un tournoi, coloration de graphes cordés

1 Ordres pour un tournoi

1.1 Tournois

1. Pour tout $0 \leqslant i \neq j < n$, on a soit un arc de i à j, soit un arc de j à i. Il y a donc exactement $n(n-1)/2$ arcs (comme dans un graphe non-orienté complet d'ordre n).

1.2 Méthode de Copeland

2. Le joueur 4 est en tête (quatre victoires), suivent les joueurs 0, 2 et 5 (trois victoires), et les joueurs 1 et 3 (une seule victoire). $4,0,2,5,1,3$ est une possibilité, de même que $4,5,2,0,1,3$, etc. Il y a douze possibilités au total.

3. Rien de bien compliqué ici, on construit un tableau de la bonne taille (attention, `n` est à définir !) initialisé à 0 qui contiendra les scores, et on lit le tableau de booléens pour le remplir :

```
let calc_scores t =
  let n = Array.length t in
  let scores = Array.make n 0 in
    for i = 0 to n-1 do
      for j = 0 to n-1 do
        if t.(i).(j) (* Une victoire pour i (sur j) *)
          then scores.(i) <- scores.(i)+1
      done;
    done;
  scores
```

On peut aussi ne se servir que de la partie supérieure droite (ou inférieure gauche), mais la fonction reste de complexité en $O(n^2)$:

```
let calc_scores t =
  let n = Array.length t in
  let scores = Array.make n 0 in
    for i = 0 to n-1 do
      for j = i+1 to n-1 do
        if t.(i).(j) (* Une victoire pour i (sur j) *)
          then scores.(i) <- scores.(i)+1
          else scores.(j) <- scores.(j)+1
      done;
    done;
  scores
```

4. Attention, le but n'est pas ici de trier les *scores* mais les entiers de 0 à $n-1$. Pour deux entiers x et y, on utilisera l'ordre $x \preceq y \iff \text{score}(x) \leqslant \text{score}(y)$.

Après avoir construit un tableau avec les entiers de 0 à $n-1$, on peut utiliser par exemple un tri par insertion :

```
let order_Copeland t =
  let n = Array.length t in
  let scores = calculer_scores t
  and ordering = Array.init n (fun i->i)   (* [| 0; 1; 2; ...; n-1 |] *)
  for i=1 to n-1 do                        (* Tri par insertion :         *)
    let j = ref i in                       (* insertion de classement.(i) *)
      while !j > 0
              && scores.(ordering.(!j-1)) < scores.(ordering.(!j)) do
        let tmp = ordering.(j) in
          ordering.(j) <- ordering.(j-1);
          ordering.(j-1) <- tmp
        j := !j - 1
      done;
  done;
  ordering
```

Le calcul des scores est en $O(n^2)$, le tri en $O(n^2)$, et donc la fonction également. Notons que l'on peut effectuer un tri en $O(n\log n)$ ou même $O(n)$ ici, mais cela ne permet pas de gagner du temps à cause du calculs des scores. Pour les curieux, l'idée (dans les grandes lignes) est de compter le nombre de joueurs ayant k points pour tout $0 \leqslant k \leqslant n-1$, et d'en déduire la position d'un joueur à k points *en partant de la fin* en comptant combien de joueurs ont fait « pire » (le dernier a une position 1, etc.). Pour gérer les ex-æquos, on fait décroitre la position chaque fois qu'on place un joueur avec k vitoires.

```
let classement_Copeland t =
  let n = Array.length t in
  let scores = calc_scores t
  and ordering = Array.make n 0
  and position = Array.make (n-1) 0 in
  for i=0 to n-1 do
    position.(scores.(i)) <- position.(scores.(i))+1
  done;
  for i=1 to n-2 do
    position.(i) <- position.(i)+position.(i-1)
  done;
  for i=0 to n-1 do
    ordering.(n-position.(scores.(i))) <- i;
    position.(scores.(i)) <- position.(scores.(i))-1
  done;
  ordering
```

Notons que si l'on se permet d'utiliser toute la bibliothèque standard OCaml, on peut faire appel à `Array.sort` et simplement écrire :

```
let order_Copeland t =
  let order = Array.init n (fun i->i) in  (* [| 0; 1; 2; ...; n-1 |] *)
  Array.sort (fun i j -> compare order.(i) order.(j)) order;
  order
```

1.3 Valeur de Slater d'un classement

5. Slater$(T_5, \sigma_5) = 4$ (le joueur 2 est battu par les joueurs 0 et 1, le joueur 3 bat les joueurs 0 et 4, les autres résultats sont « conformes » au classement).

6. Slater(T_6, σ_6) dépend du classement choisi...

- Slater$(T_6, \sigma_6) = 4$ pour $\sigma_6 = 4,0,2,5,1,3$;
- Slater$(T_6, \sigma_6) = 3$ pour $\sigma_6 = 4,5,2,0,1,3$...

7. On compte 1 pour chaque $i < j$ tels que le i^e joueur au classement n'a pas battu le j^e joueur au classement :

```
let val_Slater t sigma =
  let n = Array.length sigma and slater = ref 0 in
    for i = 0 to n-2 do
      for j = i+1 to n-1 do
        if not t.(sigma.(i)).(sigma.(j)) then slater := !slater + 1;
      done;
    done;
  !slater
```

8. La fonction précédente a une complexité quadratique en temps $(O(n^2))$, en raison des deux boucles imbriquées dont le corps est en temps constant.

1.4 Indice de Slater d'un tournoi

9. Aucun classement ne permet d'avoir une valeur de Slater de 0 (0 gagne contre 3, qui gagne contre 2, qui gagne contre 0). Aussi $s(T_4) = 1$ car c'est la valeur de Slater pour le classement $3,1,2,0$ (qui est donc un classement de Slater).

10. L'indice de Slater est supérieur ou égal à m. En effet, chaque circuit dans le graphe contribue au moins pour 1 dans le calcul de la valeur de Slater, puisque si l'on « suit » le circuit dans le classement, on se déplace au moins une fois vers la gauche, ce qui implique qu'un joueur a battu un joueur mieux classé. S'il y a m circuits deux à deux arcs-distincts, cela signifie qu'il y a au moins m arcs de ce genre dans le graphe.

11. L'indice de Slater étant inférieur ou égal à m et en même temps supérieur ou égal à m, m est donc nécessairement l'indice de Slater du tournoi,et σ est un classement de Slater.

12. On peut trouver deux circuits deux à deux arcs-disjoints, dont l'indice de Slater est supérieur ou égal à 2. Le classement $\sigma = 2,4,0,5,1,3$ a une valeur de Slater de 2, donc l'indice de Slater est bien 2, et σ est un classement de Slater.

13. Il y a cinq triangles : $(0,1,3)$, $(0,2,3)$, $(0,2,4)$, $(1,2,4)$ et $(1,3,4)$. Le premier et le troisième $((0,1,3), (0,2,4))$ sont par exemple arcs-distincts. Le second et le quatrième également $((0,2,3), (1,2,4))$, ou bien encore le second et le cinquième $((0,2,3), (1,3,4))$.

14. Pour tout (i,j,k) avec $i < j$ et $i < k$, on regarde si le circuit $i \mapsto j \mapsto k \mapsto i$ existe dans le graphe. Si c'est le cas, on compte ce triangle. Pour éviter de compter deux fois le même arc, on supprime les arcs du graphe. Afin de ne pas modifier l'argument, on copie la matrice d'adjacence avant de commencer !

Petit intermède : la fonction `Array.copy` permet de copier un tableau. Mais, pour la même raison que `.copy()` en Python, la fonction ne copie pas correctement un tableau à deux dimensions (les lignes du tableau résultat correspondent aux mêmes lignes que le tableau original). On écrit donc une fonction copiant une matrice (tableau à deux dimensions) :

```
let matrix_copy t =
  let copy = Array.make (Array.length t) [||] in
    for i=0 to (Array.length t - 1) do copy.(i) <- Array.copy t.(i) done;
    copy
```

On aurait aussi également pu simplement utiliser « `Array.map Array.copy` » Revenons à notre fonction. On n'a pas besoin de vérifier $j \neq k$ puisque l'on a des `false` sur la diagonale. On peut même se passer de $i < j$ et $i < k$ puisque la suppression des arcs fait que l'on ne peut pas compter deux fois le même triangle.

```
let count_triangles t =
  let n = Array.length t in
  let nb_tri = ref 0 and t2 = matrix_copy t in
    for i=0 to n-3 do
      for j=i+1 to n-1 do
        for k=i+1 to n-1 do
          if t2.(i).(j) && t2.(j).(k) && t2.(k).(i) then
          begin
            nb_tri:= !nb_tri + 1;
            t2.(i).(j) <- false; t2.(j).(k) <- false; t2.(k).(i) <- false;
          end;
        done;
      done;
    done;
    !nb_tri
```

Notons que l'on peut « sortir » de la (seconde) boucle sur j dès que `t.(i).(j)` est `false`, mais cela n'améliore pas la complexité. Attention, si l'on déplace les tests sur `t2` dès qu'ils sont possibles (`t2.(i).(j)` avant la boucle sur `k` en particulier), il faut prendre garde à s'assurer que l'on ne sélectionne pas plusieurs triangles avec l'arc (i,j) !

1.5 Méthode de Slater

15. On considère les permutations une à une (que l'on mémorise dans un tableau `sigma`) par ordre lexicographique croissant, et on détermine la valeur de Slater de chacune. `meilleur_score` contiendra la plus petite valeur observée, et `classement` le classement qui lui correspond.

```
let order_Slater t =
  let n = Array.length t in
  let sigma = Array.init n (fun i->i) in          (* La première permutation *)
  let best_score = ref (valeur_Slater t sigma)     (* Meilleur score trouvé *)
  and ordering = ref (Array.copy sigma) in (* Meilleure permutation trouvée *)
    while next_permutation sigma do     (* Tant qu'il rest des permutations *)
      let s = val_Slater t sigma in      (* Si valeur_Slater t σ est *)
        if s < !best_score then                 (* la plus faible observée, *)
        begin
          best_score := s;                      (* on la mémorise ainsi que *)
          ordering := Array.copy sigma       (* le classement correspondant *)
        end;
    done;
    !ordering
```

Attention, on est bien obligé de *recopier* le classement à chaque fois que l'on trouve une meilleure solution, car prendre une référence ne suffit pas : comme `sigma` subit des mutations, une référence sur `sigma` reflète ces mêmes mutations !

2 Coloration optimale d'un graphe triangulé

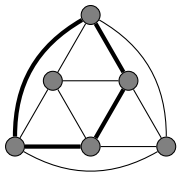
2.3 Premiers exemples

1. Supposons qu'il existe une corde $v_i \triangleright v_j$ dans le chemin $c = v_1 \triangleright \ldots \triangleright v_i \triangleright \ldots \triangleright v_j \triangleright \ldots \triangleright v_k$.

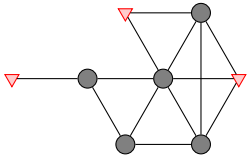
Le chemin $c' = v_1 \triangleright \ldots \triangleright v_i \triangleright v_j \triangleright \ldots \triangleright v_k$ où l'on a emprunté la corde $v_i \triangleright v_j$ plutôt que le chemin original de v_i à v_j est un chemin strictement plus court de v_1 à v_k passant uniquement par des sommets de c.

On peut donc raccourcir tout chemin de la sorte, tant qu'il contient une corde. Puisque les longueurs des chemins successifs forment une suite strictement décroissante dans $\mathbb{N}$, le processus va nécessairement s'arrêter, et le chemin obtenu, sans corde, convient.

2. Le premier graphe est triangulé, mais pas le second, qui contient notamment des cycles de longueur 4 sans corde tels que celui-ci :



3. Le second graphe n'a pas de sommets simpliciaux. Le premier en a plusieurs, a, c et f, marqués par des triangles ci-dessous :



2.4 Identification des sommets simpliciaux

4. Conformément à la définition d'un graphe triangulé, considérons un cycle de longueur $\geqslant 4$ sur G. Deux cas sont envisageables :

- ce cycle ne passe pas par v, c'est donc aussi un cycle dans G′, et puisque G′ est triangulé, il possède une corde ;
- ce cycle passe par v, et comme v est simplicial, le sommet précédent v et le sommet suivant v dans le cycle, tous deux voisins de v, sont adjacents, le cycle a donc également une corde.

Tout cycle de longueur $\geqslant 4$ sur G admet donc une corde, donc G est triangulé.

5. Considérons un sommet sommet quelconque $v \in V$. G étant une clique, tous les sommets sont adjacents deux à deux. Par conséquent, les voisins de v sont adjacents deux à deux, donc v est simplicial.

6. Si y est simplicial dans G′, tous les voisins de y dans G′ sont adjacents deux à deux. Les voisins de y dans G soit les voisins de y dans G′ (adjacents deux à deux) et x (adjacent à tous les sommets de G), donc tous les voisins de y sont bien adjacents deux à deux. Par conséquent, y est simplicial dans G.

7. Si y est simplicial dans G, il l'est de façon évidente dans G′, car si deux voisins v et v' de y dans G sont adjacents et que $v \neq x$ et $v' \neq x$, ils le restent dans G′ (et tout voisin de y dans G′ est un voisin de y dans G).

8. On a $T = \{a, b, f, h\}$ (sommets à distance 3 de c), H = T (puisque T est connexe), $U = \{e, g\}$ et $Q = \{c, d\}$.

9. Soit d la distance maximale d'un sommet à x. Si $d = 0$ ou $d = 1$, le cas a déjà été traité.

Les sommets de H sont à une distance d de x, ceux de U sont à une distance $d - 1$.

Pour montrer que U est une clique, il suffit de montrer que pour toute paire de sommets distincts v et v' de U, il existe une arête les liant (si $|U| = 1$, c'est évidemment une clique).

Soient donc $v \neq v'$ deux sommets de U. Il existe un chemin v à v' passant uniquement par des sommets de H (H est connexe et U contient les voisins de H). De même, il existe un chemin de v à v' ne passant que par des sommets de Q (a minima, il existe un chemin de v à x ne passant que par des sommets de Q, et un chemin similaire de x à v', ce qui permet de construire un chemin de v à v'...).

Supposons par l'absurde que v et v' ne sont pas voisins.

Comme pour la première question, on peut extraire du chemin de v à v' ne passant que pardes sommets de H un chemin dépourvu de cordes de v à v' ne passant que par des sommets de H (qui

ne peut être le chemin direct si v et v' ne sont pas voisins). De même, on peut obtenir un chemin dépourvu de cordes de v' à v ne passant que par des sommets de Q (là aussi, ce n'est pas un chemin direct).

En chaînant ces deux chemins, on obtient un cycle de longueur supérieure ou égale à 4 (v, v' et au moins un sommet dans H et dans Q) qui ne contient aucune corde entre des éléments de Q (par construction), de H (par construction) ou entre des éléments de Q et de H (différence de distance à x supérieure ou égale à deux). Ni entre v ou v' et des sommets de H ou Q qui ne leur sont pas directement adjoints dans le cycle (par construction, encore), ni entre v et v' par hypothèse. Le graphe n'est donc pas triangulé, ce qui est contraire aux hypothèses.

On a donc bien nécessairement une arête entre v et v'.

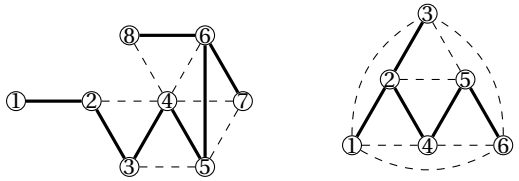
10. Tout d'abord, notons que tous les voisins d'un sommet dans H se trouvent soit dans H, soit dans U. Donc si un sommet de H est simplicial dans le graphe induit sur $\mathrm{U} \cup \mathrm{H}$, il l'est dans G.

Si le graphe induit sur $\mathrm{U} \cup \mathrm{H}$ est une clique, alors on en a terminé (tout sommet de H est simplicial). Sinon, considérons les sommets $u \in \mathrm{U}$.

Si tous les sommets $u \in \mathrm{U}$ sont voisins de tous les sommets $h \in \mathrm{H}$, alors il existe forcément un sommet simplicial dans H (En effet, pour qu'un sommet $h \in \mathrm{H}$ ne soit pas simplicial, il faut qu'il ait au moins deux voisins dans H qui ne sont pas voisins entre eux (car les voisins de h dans U sont voisins entre eux, puisque U est une clique, et un voisin de H et un voisin de U sont également voisins entre eux, par hypothèse). On a donc au moins un cycle de sommets de h de longueur $\geqslant 4$ sans corde, ce qui est contraire à l'hypothèse de triangulation.)

Sinon, enfin, il existe un sommet u qui ne soit pas voisin de tous les sommets de $\mathrm{U} \cup \mathrm{H}$. On itère alors le procédé sur le graphe restreint à $\mathrm{U} \cup \mathrm{H}$ en partant de ce sommet u. Les sommets les plus éloignés de u sont un sous-ensemble strict de H (u a au moins un voisin dans H). On forme ainsi une suite strictement décroissante de sous-ensembles de sommets, qui finira soit par une clique, soit dans le cas précédent où tout sommet de H est voisin de tous les sommets de U, et donc par trouver un sommet simplicial.

11. Pour l'ordre de parcours et les arbres couvrants, par exemple (il y a de très nombreuses possibilités) :



12. Soit $f : \mathrm{S} \mapsto [\![1 \,..\, n]\!]$, une numérotation des sommets selon leur ordre de traitement par l'algorithme MCS. On suppose qu'il existe un (ou plusieurs) chemin(s) sans cordes $c = u, v_1, v_2, \ldots, v_k, w$ tel(s) que $\forall i \in [\![1 \,..\, k]\!]$, $f(u) < f(v_i)$ et $f(w) < f(v_i)$. Parmi ces chemins, on considère ceux tels que $f(u) < f(w)$ sans perte de généralité, et on choisit parmi ceux-ci un chemin où $f(w)$ est minimal.

Considérons l'étape de l'algorithme qui traite u. On a $\ell(v_1) = \ell(v_1) + 1$ mais pas $\ell(w) = \ell(w) + 1$ (pas de corde entre w et u).

w étant traité avant v_1, il existe donc au moins un sommet v traité avant w qui soit voisin de w mais pas de v_1.

Si $\{(v, v_i)\} \wedge \mathrm{E} \neq \emptyset$, soit $j = \min\{i \mid (v, v_i) \in \mathrm{E}\}$. Le chemin $u, v_1, \ldots v_j, v$ est sans corde.

Sinon, on considère le chemin $c = u, v_1, v_2, \ldots, v_k, w, v$, qui est alors également sans corde.

Mais $f(v) < f(w)$, ce qui contredit l'hypothèse de minimalité de $f(w)$.

13. Soit v le dernier sommet visité par l'algorithme. Pour tout u et w voisins de v, $f(u) < f(v)$ et $f(w) < f(v)$, donc le chemin u, v, w a une corde. Donc u et w sont voisins. v est donc simplicial.

14. On a vu que si le graphe privé d'un sommet est triangulé et que ce sommet est simplicial, alors le graphe dans son ensemble est triangulé.

On sait que, si le graphe est triangulé, le dernier sommet visité par l'algorithme est simplicial. Pour vérifier qu'un graphe est triangulé, il suffit donc de vérifier cette condition, et que le graphe privé du dernier sommet est triangulé, ce qui peut être fait récursivement.

Comme l'ordre des sommets dans le graphe privé du dernier sommet est le même que dans le graphe complet (une fois retiré le dernier sommet), il suffit de considérer les sommets fournis par l'algorithme dans l'ordre inverse, vérifier qu'ils sont simpliciaux, les retirer du graphe, et continuer jusqu'à ce que le graphe soit réduit à un seul sommet.

15. On effectue le coloriage dans l'ordre des sommets fourni par l'algorithme, en choisissant à chaque fois le plus petit entier possible comme couleur. On a bien coloriage optimal, car à chaque étape de l'algorithme, le sommet considéré est simplicial pour le graphe réduit aux sommets visités, donc son voisinage est une clique. Si l'on introduit une couleur supplémentaire, c'est que l'on ne pouvait pas faire autrement, les couleurs étant déjà toutes utilisées sur une clique.

16. D'après l'algorithme de coloration optimale précédent, le nombre de couleurs utilisées (donc le nombre chromatique) correspond exactement à la taille de la clique de taille maximale dans le graphe triangulé.