TP #23 — Détection de cycle dans des graphes fonctionnels

Le but de ce TP est d'implémenter des algorithmes permettant de trouver un ou plusieurs cycles dans les *graphes fonctionnels* de fonctions finies.
Le premier exercice vise à implémenter un algorithme de génération de permutation uniformément aléatoire, puis à trouver les cycles de son graphe fonctionnel, et d'utiliser ces deux fonctions pour évaluer expérimentalement certaines propriétés statistiques sur les cycles des permutations uniformes.
Le second exercice demande d'implémenter un algorithme de détection de cycle dans le graphe fonctionnel d'une fonction finie quelconque, puis à l'appliquer à la recherche de collision dans une telle fonction.

Exercice 1. *Cycles de permutations*

On décrit l'algorithme suivant que vous connaissez normalement déjà pour tirer uniformément au hasard une permutation des entiers $0, \dots, n-1$ (c'est à dire une fonction bijective $\pi : [\![n]\!] \to [\![n]\!]$) :

- On initialise un tableau `p` de n entiers où chaque élément est égal à son index (commençant à 0) ; autrement dit, `p` est tel que `p[i] = i` pour `i` entier de 0 à $n-1$
- Pour `i` de 0 à $n-2$ (tous deux inclus) :
  - On tire uniformément un nombre j dans $[\![i, n-1]\!]$
  - On échange `p[i]` avec `p[j]` (on remarque que l'on peut avoir $i = j$; dans ce cas cet échange ne fait rien)
- On renvoie le tableau `p` ainsi obtenu, qui en `p[i]` vaut l'image de la permutation qu'il représente en `i`.

On prétend que si les tirages des indices sont uniformes, alors cet algorithme tire bien une permutation uniformément au hasard (c'est à dire que pour toute permutation, la probabilité qu'elle soit produite par l'algorithme est $1/n!$).

1. Écrivez une fonction C de signature
   ```
   void shuffle(size_t n, int p[n])
   ```
   qui implémente cet algorithme.
   Au besoin, allez (re-)consulter le poly pour une mini-documentation sur la génération de nombres pseudo-aléatoires.
2. Testez.
3. *(Optionnel.)* Prouvez que cet algorithme génère bien une permutation uniforme.

Un *cycle d'ordre k* d'une permutation π est une suite de k valeurs $c_0, \dots, c_{k-1}$ telles

que pour $0 \leq i < k$ l'on a $\pi(c_i) = c_{(i+1) \bmod k}$ (où $x \mod y$ désigne l'unique reste positif de la division entière de x par y).

4. Écrivez une fonction C de signature :
   ```
   void cycles(size_t n, int p[n], int c[n])
   ```
   qui prend en entrée un tableau `p` représentant une permutation de longueur `n` (de la même façon qu'aux questions précédentes) et modifie le tableau `c` en argument de façon à ce que pour tout `i` membre d'un même cycle l'on a `c[i]` égal à un même `r` égal à l'un des membres (quelconque) du cycle (sont représentant).
   Par exemple, pour :
   ```
   int p[10] = {7, 0, 2, 8, 9, 3, 4, 5, 1, 6};
   ```
   `cycles(10, p, c)` peut indifféremment modifier `c` en (un tableau de valeurs égales à celles que l'on obtiendrait par un initialiseur) :
   ```
   {0, 0, 2, 0, 4, 0, 4, 0, 0, 4}
   ```
   ou
   ```
   {7, 7, 2, 7, 9, 7, 9, 7, 7, 9}
   ```
   (D'autres possibilités existent encore.)
   Cette fonction devra être de **coût linéaire en** `n`.
5. Testez.
6. Écrivez une fonction C de signature :
   ```
   void cycle_structure(size_t n, int p[n], int cs[n+1])
   ```
   qui calcule la *structure de cycle* de son argument `p` : pour `p` comme précédemment, elle modifie `cs` de façon à ce que `cs[i]` soit égal au nombre de cycles de `p` d'ordre `i`.
   Par exemple, pour le tableau `p` donné en exemple à la question précédente l'on a après exécution que `cs` vaut (un tableau de valeurs etc.) :
   ```
   {0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0}
   ```
   Cette fonction devra être de **coût linéaire en** `n`.
7. Testez.
8. Écrivez une fonction C `stat_fixp` qui utilise vos fonctions précédentes pour vérifier expérimentalement le fait que le nombre espéré de *points fixes* (de cycles d'ordre 1) d'une permutation tirée uniformément au hasard est égal à 1.
9. Testez, avec des permutations de tailles variées.
10. *(Optionnel.)* Montrez qu'il existe $(n-1)!$ permutations de n éléments qui sont *cycliques*, c'est à dire formées d'un unique cycle d'ordre n.
11. Écrivez une fonction C `stat_cyclic` qui teste expérimentalement la propriété ci-dessus.
12. Testez, avec des permutations de tailles variées.
13. Écrivez une fonction C `stat_large` qui constate expérimentalement le fait que la probabilité qu'une permutation uniforme possède un cycle contenant

plus de la moitié des éléments (c'est à dire d'ordre supérieur à $n/2$ pour une permutation de n éléments) est *constante* en fonction de n (et donc en fait de toute autre quantité).

14. Testez.

Exercice 2. *Détection de cycle et collisions*

Soit une fonction $F : [\![n]\!] \to [\![n]\!]$, on définit son *graphe fonctionnel* comme le graphe orienté $G = S, A$ avec $S = [\![n]\!]$ et A tel qu'il existe un arc $i \to j$ ssi. $F(i) = j$. On dit d'un tel arc qu'il est *sortant* pour i et *entrant* pour j. F étant une fonction totale, un sommet a exactement un arc sortant, mais il peut avoir de zéro à n arcs entrants. On définit un cycle de F comme un cycle dans son graphe fonctionnel. (On pourra constater que cela coïncide avec la définition *ad hoc* donnée pour les permutations à l'exercice précédent.)

1. Montrez que F possède au moins un et au plus n cycles.

On définit une *collision* (non triviale) pour F comme une paire $i, j \neq i \in [\![n]\!]^2$ telle que $F(i) = F(j)$. F possède (au moins) une collisions ssi. elle n'est pas une permutation. On peut remarquer que F possède une collision ssi. elle possède au moins un point qui n'appartient pas à un cycle.

Si F possède une collision, celle-ci peut s'exprimer en terme du graphe fonctionnel comme un sommet possédant (au moins) deux arcs entrants. De façon plus spécifique, F contient alors au moins un sommet *appartenant à un cycle* (possiblement d'ordre 1) qui possède au moins deux arcs entrants, dont l'un est celui de son prédécesseur dans le cycle (possiblement lui-même).
En reformulant, si F possède une collision, alors elle contient un cycle $x_0 \to \cdots \to x_{k-1}$ tel qu'il existe x_i dans ce cycle et un certain y qui ne fait pas partie du cycle avec :

— $F(x_{(i-1) \bmod k}) = x_i$
— $F(y) = x_i$

La forme alors obtenue en considérant le cycle et la « queue » formée par y (et ses « prédécesseurs ») est souvent comparée à celle de la lettre ρ.
De tout ce qui précède, on peut déduire une stratégie pour chercher une collision dans une fonction en possédant une : on choisit un point de départ x_0, on itère F sur x_0 (autrement dit, on suit l'unique chemin du graphe fonctionnel défini par l'arc sortant de x_0, l'arc sortant de $F(x_0)$ etc.) et :

— si l'on atteint à nouveau x_0 la collision trouvée est triviale et l'on recommence avec un autre point de départ (ce cas est peu probable pour une fonction uniforme, qui est celui qui nous intéressera par la suite)
— si l'on est en mesure de détecter que ce chemin possède un cycle qui ne contient **pas** x_0, alors puisque nous avons abouti à ce cycle en partant de x_0, il *existe* un sommet du cycle possédant un arc entrant depuis un sommet qui

ne se trouve pas sur le cycle mais qui est un *successeur* de x_0. Soit k l'ordre du cycle, on a alors $F^o(x_0) = F^{k+o}(x_0)$ (où $F^i(x)$ dénote l'application de F composée i fois avec elle même, appliquée à x) pour un certain entier $o > 0$ (le cas $o = 0$ correspond au cas précédent où x_0 appartient à un cycle et où la collision trouvée est triviale), ce qui constitue une collision non triviale.

L'intérêt de cette stratégie est que l'on dispose d'algorithmes efficaces pour *détecter* un cycle (et calculer son ordre) dans le graphe fonctionnel d'une fonction F quelconque, et notamment d'algorithmes utilisant très peu de mémoire. Puisque les autres étapes de la stratégie ne nécessitent pas non plus de mémoire, cela implique que l'on peut trouver une collision d'une fonction qui en possède une (ou déterminer qu'elle n'en possède pas) en *stockant* seulement un **très petit nombre** de ses évaluations (ce qui n'est pas *a priori* évident).
On propose d'implémenter une telle stratégie en utilisant l'algorithme suivant pour détecter la présence d'un cycle :

- on fixe un point de départ `s` (par exemple `0`)
- on initialise une structure de donnée de pile à la pile vide
- tant qu'un cycle n'a pas été détecté :
  - on calcule `ns = F(s)`
  - on retire de la pile tous les éléments s'y trouvant éventuellement qui sont strictement plus grands que `ns`
  - si l'élément au sommet de la pile est maintenant égal à `ns`, on a détecté un cycle qui contient `ns`
  - sinon on ajoute `ns` en sommet de pile, on affecte `s = ns`, et l'on recommence

On peut remarquer que cet algorithme peut en principe utiliser beaucoup de mémoire, typiquement si les sommets du graphe fonctionnel sont visités par valeur croissante (ce que l'on ne peut pas éviter *a priori*). Cependant, si appliqué à une fonction dont toutes les images ont été tirées uniformément et indépendamment au hasard, le nombre *moyen* d'éléments stockés dans la pile restera « petit » (autrement dit, le coût mémoire *moyen* est faible). Il est aussi possible de garantir un bon comportement *espéré* pour n'importe quelle fonction en conjuguant celle-ci avec une permutation uniforme, ce que nous nous abstiendront de faire (ou d'analyser) ici (on entre alors dans le cadre des algorithmes probabilistes, mais la question du coût mémoire n'est pas abordée par la catégorisation Monte-Carlo ou Las-Vegas…).

2. Écrivez une fonction C de signature :
```
void unif(size_t n, int a[n])
```
qui remplit son argument `a` de valeurs aléatoires tirées uniformément et indépendamment dans $[\![n]\!]$.

3. Écrivez une fonction C de signature :
```
struct cycle_dat find_cycle(size_t n, int a[n])
```

qui prend en entrée une représentation de fonction telle que renvoyée par `unif` et qui implémente l'algorithme de détection de cycle ci-dessus. Cette fonction devra renvoyer un point `x` appartenant à un cycle, le nombre d'itération de la boucle qui a été effectué avant de détecter le cycle, et l'ordre du cycle (informations stockées dans un type `struct cycle_dat` adapté que vous définirez).

Conseil : la structure de pile utilisée par l'algorithme peut rester complètement interne à votre fonction `cycle_dat` ; il n'est pas nécessaire d'implémenter une structure complète, et surtout pas nécessaire d'utiliser une liste chaînée ! Une implémentation par tableau est ici la plus indiquée, et vous pouvez supposer que le nombre maximum d'éléments devant être stocké est logarithmique en `n` (et faire échouer votre fonction si votre pile est pleine).

4. Testez.
5. Écrivez une fonction C de signature :

   ```
   void col_find(size_t n, int a[n], int *x, int *y)
   ```

   qui implémente la stratégie de recherche de collision décrite ci-dessus, en utilisant votre fonction `find_cycle` pour la détection de cycle. Elle devra modifier les valeurs des objets pointés par `x` et `y` de façon à ce que `a[*x]` et `a[*y]` soient égales.

   Vous pouvez vous contenter d'effectuer *une* recherche de cycle et de « renvoyer » une collision triviale (avec `*x` et `*y` égaux) dans le cas (improbable pour des valeurs de `n` même modéré) où le point de départ choisi se trouverait déjà sur un cycle. Dans le cas contraire, la collision renvoyée devra être non triviale.
6. Testez.
7. Constatez une conséquence du « paradoxe des anniversaires » : pour une fonction F uniformément aléatoire de domaine $[\![n]\!]$, le nombre espéré d'évaluations de F pour trouver une collision est environ $\sqrt{n}$.