Devoir surveillé n°1

Corrigé

⋆ ⋆ ⋆

Exercice

1. ARRÊT est indécidable et semi-décidable (voir cours).
2. Montrons que coARRÊT $\leqslant_m$ $\text{ARRÊT}_{\texttt{fini}}$. Cela montrera que $\text{ARRÊT}_{\texttt{fini}}$ n'est pas semi-décidable (et donc pas décidable non plus).

   Pour ce faire, supposons que `arret_fini` résout le problème $\text{ARRÊT}_{\texttt{fini}}$. Alors la fonction suivante résout coARRÊT :

   ```
   let coarret <f, x> =
       let g _ = universel <f, x> in
       arret_fini <g>
   ```

   – si (f, x) est une instance positive de coARRÊT, alors $f(x)$ ne termine pas. On en déduit qu'il n'existe aucun `y` tel que `g y` termine, donc `g` est une instance positive de $\text{ARRÊT}_{\texttt{fini}}$;
   – si (f, x) est une instance négative de coARRÊT, alors $f(x)$ termine. On en déduit qu'il existe une infinité de `y` tels que `g y` termine, donc `g` est une instance négative de $\text{ARRÊT}_{\texttt{fini}}$.
3. Montrons que ARRÊT $\leqslant_m$ $\text{ÉQUIV}_{\exists}$. Supposons que `equiv_existe` résout le problème $\text{ÉQUIV}_{\exists}$. Alors la fonction suivant résout ARRÊT :

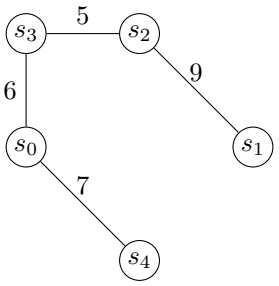
   ```
   let arret <f, x> =
       let g _ = () in
       let h _ = ignore (universel <f, x>) in
       equiv_existe <g, h>
   ```

   On en déduit que $\text{ÉQUIV}_{\exists}$ est indécidable. Ce problème est toutefois semi-décidable. L'algorithme suivant permet de résoudre les instances positives, en considérant une énumération $(x_i)_{i\in\mathbb{N}}$ des arguments :
   – $n = 1$
   – Tant qu'on n'a pas trouvé une entrée x_i telle que $f(x_i)$ et $g(x_i)$ terminent et renvoient la même valeur :
     * simuler une étape de calcul supplémentaire de f et de g pour chacun des n premiers arguments
     * $n = n + 1$

Problème : goulot maximal

1 Généralités

Question 1 On obtient :



Question 2 Par récurrence :

- pour $i = 0$, G_0 n'a pas d'arêtes, donc $n = n - 0$ composantes connexes ;
- si on suppose le résultat vrai pour i, alors ajouter l'arête a_{i+1} peut diminuer le nombre de composantes connexes d'au plus 1.

On conclut par récurrence.

Dès lors, G possède au moins $n - k$ composantes connexes. Si G est connexe, on en déduit que $n - k \leqslant 1$, soit $|S| - 1 \leqslant |A|$.

Question 3 Dans la récurrence précédente, si à l'étape d'hérédité le nombre de composantes connexes n'a pas augmenté de 1, c'est que l'arête ajoutée était entre deux sommets d'une même composante connexe, donc a créé un cycle.

Dès lors, si G est sans cycle, alors il possède au plus $n - k$ composantes connexes, soit $n - k \geqslant 1$, donc $|S| - 1 \geqslant |A|$.

Question 4

(a) $\Rightarrow$ (b) G étant connexe et sans cycle, par la question précédente, on a $|A| \geqslant |S| - 1$ et $|A| \leqslant |S| - 1$, soit $|A| = |S| - 1$.

(b) $\Rightarrow$ (c) Supposons que G contient un cycle. Alors on peut supprimer une arête de ce cycle sans perdre la connexité. Le nouveau graphe $G' = (S, A')$ ainsi obtenu est connexe et vérifie $|A'| = |A| - 1 = |S| - 2 < |S| - 1$, ce qui est absurde d'après la question précédente.

(c) $\Rightarrow$ (a) Notons $G_i = (S_i, A_i)$ les composantes connexes de G, pour $i \in [\![1, m]\!]$. Chaque composante connexe étant connexe et sans cycle, on a par la question précédente $|A_i| = |S_i| - 1$. En sommant, on obtient :

$$|A| = \sum_{i=1}^{m} |A_i| = \left(\sum_{i=1}^{m} |S_i| \right) - m = |S| - m = |S| - 1$$

On en déduit $m = 1$, c'est-à-dire que le graphe est connexe.

Question 5 Supposons que G possède deux arbres couvrants de poids minimal, $T_1 = (S, B_1)$ et $T_2 = (S, B_2)$, avec $B_1 \neq B_2$. La différence symétrique $B_1 \Delta B_2$ est donc non vide. Soit a l'arête de poids minimale de $B_1 \Delta B_2$. Sans perte de généralité, supposons que $a \in B_1$.

Le graphe $H = (S, B_2 \cup \{a\})$ contient un cycle. Si on suppose que toutes les arêtes de ce cycle sont dans B_1, alors T_1 contiendrait un cycle, ce qui est absurde. Il existe donc une arête $a' \in B_2 \setminus B_1$. Par minimalité du poids de a dans $B_1 \Delta B_2$ et injectivité, $f(a) < f(a')$.

Le graphe $T = (S, B_2 \cup \{a\} \setminus \{a'\})$ est donc un graphe connexe (car H était connexe et on a supprimé une arête d'un cycle), et possède $|S| - 1$ arêtes. C'est donc un arbre. De plus, $f(T) = f(T_2) + f(a) - f(a') < f(T_2)$. Cela contredit la minimalité de T_2. On conclut par l'absurde.

Question 6 Si $a = st \in B^*$, alors $(S, B^* \setminus \{a\})$ est constitué de deux composantes connexes C_1 et C_2 (l'une contenant s et l'autre t). Le chemin $(s_0 = s, s_1, \ldots, s_k = t)$ constituant le reste du cycle contient une arête $a' = s_i s_{i+1}$ reliant un sommet de C_1 et un sommet de C_2. Nécessairement $a' \notin B^*$ (sinon T^* contiendrait un cycle).

Comme précédemment, $T = (S, B^* \cup \{a'\} \setminus \{a\})$ est un arbre couvrant. De plus, $f(T) = f(T^*) + f(a') - f(a) < f(T^*)$, ce qui est absurde. On en déduit que $a \notin B^*$.

Question 7 On sépare selon les éléments d'indices pairs et ceux d'indices impairs, ce qui permet de ne faire qu'un seul parcours de la liste.

```
let rec separation lst = match lst with
    | [] | [_] -> lst, []
    | t1 :: t2 :: q ->
        let lst1, lst2 = separation q in
        t1 :: lst1, t2 :: lst2
```

Question 8 On distingue selon la tête de chaque liste.

```
let rec fusion lst1 lst2 = match lst1, lst2 with
    | [], lst | lst, [] -> lst
    | t1 :: q1, t2 :: q2 ->
        if t1 <= t2 then t1 :: fusion q1 lst2
        else t2 :: fusion lst1 q2
```

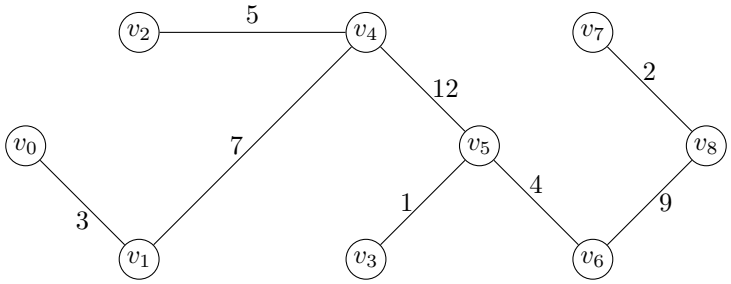
Question 9 On sépare, on trie récursivement, on fusionne.

```
let rec tri_fusion lst = match lst with
    | [] | [_] -> lst
    | _ ->
        let lst1, lst2 = separation lst in
        fusion (tri_fusion lst1) (tri_fusion lst2)
```

2 Algorithme de Kruskal inversé

Question 10 L'algorithme de Kruskal consiste à parcourir les arêtes par ordre de poids croissant, et à conserver celles qui ne créent pas de cycle. Sa complexité contient celle du tri des arêtes par poids ($\mathcal{O}(|A| \log |A|) = \mathcal{O}(|A| \log |S|)$) et le reste dépend de l'implémentation de la structure Union-Find. Avec une structure avancée, par exemple par tableau de parenté en utilisant l'union par taille, la vérification et l'union se font en $\mathcal{O}(\log |S|)$, soit une complexité totale en $\mathcal{O}(|A| \log |S|)$.

Question 11 Techniquement, les poids étant tous distincts, rien n'impose d'appliquer l'algorithme de Kruskal inversé pour obtenir le bon arbre... L'arbre obtenu est le suivant.



Question 12 On tire parti du fait que les comparaisons se font par ordre lexicographique (d'où le poids en premier). Il suffit de créer la liste des triplets et d'appeler `tri_fusion`. Le test `s < t` garantit l'absence de doublons. On pense à renvoyer la liste renversée pour avoir les poids décroissants.

```
let tri_aretes g =
    let n = Array.length g in
    let aretes = ref [] in
    for s = 0 to n - 1 do
        let traiter (t, p) =
            if s < t then aretes := (p, s, t) :: !aretes
        in
        List.iter traiter g.(s)
    done;
    List.rev (tri_fusion !aretes)
```

Question 13 On écrit un parcours de graphe classique, avec uniquement la contrainte du poids des arêtes.

```
let connectes g s t pmax =
    let n = Array.length g in
    let vus = Array.make n false in
    let rec dfs u =
        if not vus.(u) then begin
            vus.(u) <- true;
            let traiter (v, p) =
                if p < pmax then dfs v
            in
            List.iter traiter g.(u)
        end
    in
    dfs s;
    vus.(t)
```

Grâce au tableau de booléens (créé en temps $\mathcal{O}(|S|)$), la fonction `dfs` n'est appelée qu'au plus une fois par sommet. Cette fonction fait un parcours de la liste d'adjacence de ce sommet. La complexité totale des appels est donc en $\mathcal{O}\left(\sum_{s\in S}(\deg(s)+1)\right) = \mathcal{O}(|S|+|A|)$.

Question 14 On suit le principe de l'algorithme, en s'adaptant à la structure de données. Ici, on part plutôt d'un graphe sans arête, et on ajoute celles qu'on conserve, c'est-à-dire celles qui briseraient la connexité si on ne les gardait pas. On peut utiliser la fonction précédente, car lorsqu'on essaie de supprimer une arête $a = \{s, t\}$, s'il existe un chemin de s à t dans $(S, B \setminus \{a\})$, un tel chemin ne passe que par des arêtes de poids $< f(a)$ (car sinon, on aurait pu supprimer une arête de poids supérieur en laissant a).

```
let kruskal_inverse g =
    let n = Array.length g in
    let couv = Array.make n [] in
    let traiter (p, s, t) =
        if not (connectes g s t p) then begin
            couv.(s) <- (t, p) :: couv.(s);
            couv.(t) <- (s, p) :: couv.(t)
        end;
    in
    List.iter traiter (tri_aretes g);
    couv
```

Question 15 On remarque que par principe de l'algorithme, « (S, B) est connexe » est un invariant de la boucle **Pour** de l'algorithme. Si on suppose, après être sorti de la boucle, que (S, B) contient un cycle, alors l'arête de poids maximal de ce cycle aurait dû être supprimée lors qu'elle a été parcourue. On en déduit que (S, B) est connexe sans cycle, c'est-à-dire un arbre. Comme il a le même ensemble de sommets que G, c'est

bien un arbre couvrant.

Question 16 On commence par le cas injectif. Soit $T^* = (S, B^*)$ l'unique arbre couvrant minimal. Montrons que « $(A \setminus B) \cap B^* = \emptyset$ » est un invariant de boucle :

- initialement, $A = B$, donc $A \setminus B = \emptyset$, d'où l'égalité ;
- si on suppose le résultat vrai avant une itération, alors à l'itération suivante, deux cas sont possibles ;
  - B n'a pas été modifié, auquel cas il vérifie toujours la propriété ;
  - B a été modifié en $B \setminus \{a\}$. Dans ce cas, l'arête a est dans un cycle (car le graphe $(S, B \setminus \{a\})$ reste connexe) et de poids maximal dans ce cycle (car sinon, on aurait supprimé une autre arête du cycle avant d'atteindre a). Par la question 5, $a \notin B^*$. On a alors $(A \setminus B) \cap B^* = \{a\} \cap B^* = \emptyset$.

Finalement, on en déduit que $B^* \setminus B = \emptyset$. Comme B et B^* sont de même cardinaux, ils sont égaux.

Pour le cas non injectif, on peut forcer l'injectivité de la fonction de pondération (par exemple en posant $f'(st) = (f(st), s, t)$), appliquer le résultat précédent, et montrer qu'avec la fonction de pondération initiale, on a bien un poids minimal (par exemple en projetant les poids selon la première composante).

Question 17 Le tri des arêtes est en $\mathcal{O}(|A| \log |S|)$. On applique $|A|$ fois l'appel à `connecte`, qui est en $\mathcal{O}(|A| + |S|)$. Le graphe étant connexe, on a $|S| = \mathcal{O}(|A|)$ (sauf pour le graphe à un seul sommet, inintéressant ici), donc la complexité totale est en $\mathcal{O}(|A|^2)$.

C'est moins bien que l'algorithme de Kruskal (un facteur $|A|$ au lieu de $\log |S|$).

3 Calcul du goulot maximal

Question 18 Le chemin $(s_0, s_2, s_1, s_4, s_7, s_6, s_8)$ est un goulot maximal, de capacité 7.

3.1 Première approche

Question 19 Deux possibilités :

- soit on calcule un arbre couvrant minimal de $(S, A, -f)$;
- soit on adapte l'algorithme de Kruskal (ou Kruskal inversé) en parcourant les arêtes dans l'ordre inverse.

En effet, T est est un arbre couvrant maximal de (S, A, f) si et seulement si T est un arbre couvrant minimal de $(S, A, -f)$: il y a bijection entre les arbres couvrant de (S, A, f) et ceux de $(S, A, -f)$, et la comparaison des poids est inversée.

Question 20 Notons $\sigma = (s = s_0, s_1, \ldots, s_k = t)$ un chemin de s à t dans T. Supposons par l'absurde que σ n'est pas un goulot maximal de s à t dans G et soit σ^* un goulot maximal de s à t dans G. Par hypothèse, $c(\sigma) < c(\sigma^*)$. Soit a l'arête de poids minimal de σ. Alors $(S, B \setminus \{a\})$ contient deux composantes connexes. Il existe une arête a' de $\sigma*$ qui relie ces deux composantes connexes (car σ^* est un chemin qui relient ces deux composantes). De plus, $f(a') > f(a)$ (car $f(a') \geqslant c(\sigma^*) > c(\sigma) = f(a)$), donc $a' \neq a$. On en déduit, comme dans la première partie, que $T' = (S, B \cup \{a'\} \setminus \{a\})$ est un arbre couvrant, de poids $f(T') = f(T) + f(a') - f(a) > f(T)$, ce qui contredit la maximalité de T.

Question 21 Il faut bien penser à libérer tous les tableaux.

```
void liberer_graphe(graphe G){
    for (int s=0; s<G.n; s++){
        free(G.voisins[s]);
        free(G.poids[s]);
    }
    free(G.degres);
    free(G.voisins);
    free(G.poids);
}
```

Question 22 On commence par écrire une fonction récursive qui fait un parcours en profondeur, en ayant en argument le sommet courant et son prédécesseur. Par convention, on suppose qu'un sommet non vu aura un prédécesseur égal à -2.

```
void DFS(graphe T, int* pred, int t, int pt){
    if (pred[t] == -2){
        pred[t] = pt;
        for (int i=0; i<T.degres[t]; i++){
            DFS(T, pred, T.voisins[t][i], t);
        }
    }
}
```

Dès lors, on peut écrire la fonction demandée :

```
int* predecesseurs(graphe T, int s){
    int* pred = malloc(T.n * sizeof(int));
    for (int t=0; t<T.n; t++){
        pred[t] = -2;
    }
    DFS(T, pred, s, -1);
    return pred;
}
```

C'est un simple parcours de graphe, donc linéaire en le nombre d'arêtes et de sommets. Comme on l'applique à un arbre, le nombre d'arêtes est égal au nombre de sommets moins 1, donc la complexité est bien linéaire en n.

Question 23 L'idée est de parcourir le tableau de prédécesseurs depuis t jusqu'à tomber sur le sommet s. On fait un premier parcours pour déterminer la taille du chemin, afin de pouvoir allouer correctement la mémoire, puis un deuxième parcours du tableau pour remplir le chemin.

```
int* goulot_max(graphe G, int s, int t, int* lc){
    graphe T = arbre_max(G);
    int* pred = predecesseurs(T, s);
    int m = 0;
    int cur = t;
    while (pred[cur] != -1){
        m++;
        cur = pred[cur];
    }
    *lc = m + 1;
    int* sigma = malloc((m + 1) * sizeof(int));
    cur = t;
    while (pred[cur] != -1){
        sigma[m] = cur;
        m--;
        cur = pred[cur];
    }
    sigma[0] = s;
    free(pred);
    liberer_graphe(T);
    return sigma;
}
```

3.2 Un algorithme efficace

Question 24 Remarquons que le nombre d'arêtes est environ divisé par deux à chaque itération :

- si la suppression des arêtes laisse un chemin de s à t, on a supprimé $\left\lfloor \frac{|A|}{2} \right\rfloor$ arêtes (celles de poids inférieur à la médiane) ;
- sinon, on n'a gardé que ces arêtes là, et supprimé les autres, soit supprimé $\left\lceil \frac{|A|}{2} \right\rceil$ arêtes.

Le nombre de passages dans la boucle est donc de l'ordre de $\log_2 |A|$ au maximum, et la complexité totale est en $\mathcal{O}\left(\sum_{k=0}^{+\infty} \frac{|A|}{2^k}\right) = \mathcal{O}(|A|)$.

Question 25 Montrons qu'il existe un goulot maximal de s à t dont l'arête de poids minimal n'est jamais supprimée :

- si la suppression des arêtes laisse un chemin de s à t, alors les arêtes supprimées ne font pas partie d'un goulot maximal (car la capacité des chemins restants est strictement plus grande que le poids des arêtes supprimées) ;
- sinon, le goulot maximal passe nécessairement par l'une de ces arêtes, et l'une d'entre elles est de poids minimal dans le goulot maximal. Le fait de fusionner les composantes ne change pas la suite du problème.

Question 26 Il suffit de supprimer toutes les arêtes du graphe dont le poids est strictement inférieur à la capacité du goulot et de lancer un simple parcours de graphe depuis s.

⋆ ⋆ ⋆