COMPOSITION D'INFORMATIQUE n°1

Corrigé

⋆ ⋆ ⋆

Exercice

1. Supposons qu'il existe une fonction `arret : string -> string -> bool` qui résout le problème ARRÊT. Construisons alors la fonction suivante :

```
let rec paradoxe code =
    if arret code code then paradoxe code;;
```

Distinguons :

- si `paradoxe code_paradoxe` termine, alors `arret code_paradoxe code_paradoxe` renvoie `true`, donc chaque appel à `paradoxe code_paradoxe` relancera un appel récursif similaire, donc `paradoxe code_paradoxe` ne termine pas ;
- si `paradoxe code_paradoxe` ne termine pas, alors `arret code_paradoxe code_paradoxe` renvoie `false`, donc la fonction s'arrête au premier appel.

Dans les deux cas, on arrive à une contradiction, donc la fonction `paradoxe` ne peut pas exister, et donc la fonction `arret` non plus. On en déduit que le problème ARRÊT est indécidable.

Par ailleurs, l'algorithme suivant résout les instances positives de ARRÊT :

```
let arret f x =
    ignore (f x);
    true;;
```

Donc ARRÊT est semi-décidable.

2. Montrons que $\text{ARRÊT}_{\texttt{fini}}$ est indécidable est n'est pas semi-décidable par une réduction des instances positives du problème coARRÊT. Supposons qu'il existe une fonction `arret_fini` qui résout les instances positives du problème $\text{ARRÊT}_{\texttt{fini}}$. Considérons la fonction suivante :

```
let coarret f x =
    let rec g y = f x in
    arret_fini g;;
```

Distinguons :

- 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 $\text{ARRÊT}_{\texttt{fini}}$.

Finalement, la fonction `coarret` résout les instances positives de coARRÊT. Sachant que coARRÊT n'est pas semi-décidable, on en déduit que $\text{ARRÊT}_{\texttt{fini}}$ n'est pas semi-décidable (et donc pas décidable).

3. Montrons que $\text{ARRÊT}_{\geq 10}$ est indécidable par une réduction de ARRÊT. Supposons qu'il existe une fonction `arret_sup10` qui résout $\text{ARRÊT}_{\geq 10}$. Considérons la fonction suivante :

```
let arret f x =
    let g y = f x in
    arret_sup10 g;;
```

Alors `arret` résout bien le problème de l'arrêt (il y a soit une infinité, soit aucun argument `y` tel que `g y` termine, selon que (f, x) soit une instance positive ou négative de ARRÊT). On en déduit que $\text{ARRÊT}_{\geq 10}$ est indécidable.
Montrons par ailleurs que ce problème est semi-décidable en considérant l'algorithme suivant. En considérant les entrées possibles de f triées (par ordre lexicographique, par exemple) :

- $n = 1$
- Tant que le calcul ne s'est pas terminé pour au moins 10 arguments :
  * simuler une étape de calcul supplémentaire de f pour chacun des n premiers arguments
  * $n = n + 1$

Alors cet algorithme résout bien les instances positives de $\text{ARRÊT}_{\geq 10}$ qui est donc semi-décidable.

Problème : le voyageur de commerce

1 Approche naïve

Question 1 Le nombre de permutations des sommets est $n!$. Cependant, un cycle hamiltonien peut commencer par n'importe lequel des n sommets et le sens de parcours n'a pas d'importance (le graphe est non orienté). On en déduit que le nombre de cycles hamiltoniens dans un graphe complet est $\frac{(n-1)!}{2}$.

Question 2 On utilise ici la division euclidienne : s indique le numéro de la ligne de la matrice, et t indique le numéro de colonne.

```
int f(int* G, int n, int s, int t){
    return G[s*n + t];
}
```

Question 3 On initialise une variable `poids` par le poids de la dernière arête $\{s_{n-1}, s_0\}$. Ensuite, on itère pour i de 0 à $n-2$ pour ajouter le poids des arêtes restantes.

```
int poids_cycle(int* G, int* c, int n){
    int poids = f(G, n, c[0], c[n - 1]);
    for (int i=0; i<n-1; i++){
        poids += f(G, n, c[i], c[i+1]);
    }
    return poids;
}
```

Question 4 L'algorithme est le suivant :

- calculer j en partant de la fin de la permutation (on commence par $j = n-2$ et on décrémente tant que $j \geq 0$ et $p_j > p_{j+1}$) ;
- si $j = -1$, il s'agit de la dernière permutation et on renvoie « vrai » ;
- sinon :
  * calculer k en partant de la fin de la permutation (on commence par $k = n-1$ et on décrémente tant que $p_j > p_k$) ;
  * échanger les valeurs de p_j et p_k ;
  * renverser les valeurs entre les indices $j+1$ et $n-1$;
  * renvoyer « faux ».

Voici une implémentation possible (qui n'était pas demandée) de la fonction :

```
bool permut_suivante(int* p, int n){
    int j= n - 2;
    while (j>=0 && p[j] > p[j+1]){j--;}
    if (j == -1){
        return false;
    } else {
        int k = n - 1;
        while (p[j] > p[k]){k--;}
        int tmp = p[j];
        p[j] = p[k];
        p[k] = tmp;
        for (int i=0; i<(n-j-1)/2; i++){
            tmp = p[i+j+1];
            p[i+j+1] = p[n-i-1];
            p[n-i-1] = tmp;
        }
        return true;
    }
}
```

Question 5 On commence par créer deux tableaux : un pour la permutation qui va parcourir l'ensemble des permutations, et l'autre pour garder en mémoire celle qui correspond à un cycle hamiltonien de poids minimal. Tant qu'on n'a pas atteint la dernière permutation, on calcule le poids correspondant, et on met à jour le tableau `cmin` et le poids `poids_min` si on trouve un poids inférieur. On pense à libérer le tableau non renvoyé avant la fin de la fonction.

```
int* PVC_naif(int* G, int n){
    int* c = malloc(n * sizeof(*c));
    int* cmin = malloc(n * sizeof(*cmin));
    for (int i=0; i<n; i++){
        c[i] = i;
        cmin[i] = i;
    }
    int poids_min = poids_cycle(G, c, n);
    while (permut_suivante(c, n)){
        int poids = poids_cycle(G, c, n);
        if (poids < poids_min){
            for (int i=0; i<n; i++){
                cmin[i] = c[i];
            }
            poids_min = poids;
        }
    }
    free(c);
    return cmin;
}
```

Question 6 La boucle `while` est de taille $n!$ qui correspond au nombre de permutations. Dans cette boucle, on y fait les opérations suivantes :

- un appel à `permut_suivante` en $\mathcal{O}(1)$ amorti ;
- un appel à `poids_cycle` en $\Theta(n)$;
- une recopie éventuelle du tableau en $\Theta(n)$;

– d'autres opérations en temps constant.

La complexité totale est donc en $\Theta(n! \times n) = \Theta((n+1)!)$.

2 Heuristique du plus proche voisin

Question 7 On commence par trouver un sommet `tmin` non vu et différent de s. Ensuite, on parcourt les sommets restants et on garde en mémoire le sommet non vu qui minimise le poids de l'arête à s.

```
int plus_proche(int* G, bool* vus, int n, int s){
    int tmin = 0;
    while (tmin == s || vus[tmin]){tmin++;}
    for (int t=tmin+1; t<n; t++){
        if (!vus[t] && f(G, n, s, t) < f(G, n, s, tmin)){
            tmin = t;
        }
    }
    return tmin;
}
```

Question 8 On construit le cycle étape par étape :

– le sommet le plus proche de 0 est le sommet 1 ;

– le sommet le plus proche de 1 différent de 0 est le sommet 2 ;

– le sommet le plus proche de 2 différent de 0 et 1 est le sommet 4 ;

– le sommet le plus proche de 4 différent de 0, 1 et 2 est le sommet 3.

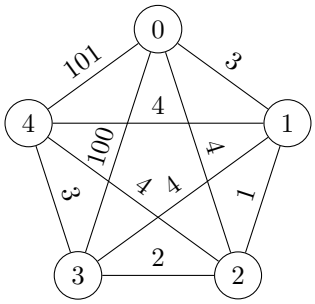
On obtient le cycle $(0, 1, 2, 4, 3)$, de poids $1 + 6 + 2 + 9 + 5 = 23$. Ce n'est pas le cycle de poids minimal, car $(0, 1, 4, 2, 3)$ est de poids $1 + 7 + 2 + 3 + 5 = 18 < 23$.

Question 9 On crée un tableau de booléens et un tableau correspondant au cycle renvoyé par l'algorithme. En commençant par 0, on cherche à chaque étape le plus proche voisin du dernier sommet ajouté au cycle, et on le note comme vu.

```
int* PVC_glouton(int* G, int n){
    int* c = malloc(n * sizeof(*c));
    bool* vus = malloc(n * sizeof(*vus));
    for (int i=0; i<n; i++){
        c[i] = 0;
        vus[i] = false;
    }
    c[0] = 0;
    vus[0] = true;
    for (int i=1; i<n; i++){
        c[i] = plus_proche(G, vus, n, c[i - 1]);
        vus[c[i]] = true;
    }
    free(vus);
    return c;
}
```

Question 10 La fonction `plus_proche` a clairement une complexité en $\Theta(n)$. Comme on y fait $n-1$ appels dans `PVC_glouton`, la complexité totale est en $\Theta(n^2)$.

Question 11 On propose le graphe suivant :



En effet, voici les cycles renvoyés par l'algorithme glouton, selon le sommet de départ :

- $(0, 1, 2, 3, 4)$, de poids 110 ;
- $(1, 2, 3, 4, 0)$, de poids 110 ;
- $(2, 1, 0, 3, 4)$, de poids 111 ;
- $(3, 2, 1, 0, 4)$, de poids 110 ;
- $(4, 3, 2, 1, 0)$, de poids 110.

Or le cycle $(0, 1, 3, 4, 2)$ est de poids 18.

3 Algorithme de Held-Karp

Question 12 On envisage chacun des prédécesseurs t possibles de s dans un chemin de 0 à s passant une et une seule fois par chaque sommet de X. Un chemin de poids minimal de 0 à s sera un chemin de poids minimal de 0 à t suivi de l'arête $\{t, s\}$. On a donc :

$$P_{\min}(s, X) = \min_{t \in X} \{P_{\min}(t, X \setminus \{t\}) + f(t, s)\}$$

Le sommet X qui permet d'atteindre ce minimum est égal à $pred(s, X)$.

Question 13 Supposons que `cle` est un objet mutable. Si on crée une association `(cle, valeur)` dans une table de hachage, puis qu'on modifie `cle`, alors le haché de `cle` a une grande probabilité d'avoir été modifié suite à la modification de la clé. Ainsi, après modification, l'association ne peut plus être trouvée dans la table de hachage avec la même variable comme clé (car on utilise le haché comme indice de tableau).

Question 14 On suit l'indication. On utilise l'évaluation de Horner pour éviter d'avoir à calculer des puissances de 2.

```
let encodage s tab_X =
    let code = ref 0 in
    for i = 0 to Array.length tab_X - 1 do
        code := 2 * !code;
        if tab_X.(i) then incr code;
    done;
    (s, !code);;
```

Question 15 On distingue les cas :

- si $X =$, $P_{\min}(s, X) = f(0, s)$ et $pred(s, X) = 0$ (y compris si $s = 0$). À noter, on teste si $X =$ en vérifiant si le code de X vaut 0 ;
- si (s, X) est déjà une clé de la table de hachage, on renvoie directement les valeurs associées ;

– sinon, on commence par trouver un premier sommet $s_{\min}$ dans X (c'est la boucle `while`), et on calcule $P_{\min}(s_{\min}, X \setminus \{s_{\min}\})$ (on pense à mettre à `false`, puis remettre à `true` la valeur dans le tableau). Ensuite, pour chaque sommet $t \in X$, on calcule $P_{\min}(t, X \setminus \{t\})$ et on met à jour $s_{\min}$ et le poids minimal le cas échéant. On ajoute finalement le bon couple de valeurs dans la table de hachage avant de renvoyer ce qu'il faut.

```
let rec chemin_min s tab_X =
    let (s, code) = encodage s tab_X in
    if code = 0 then (g.(0).(s), 0)
    else begin if not (Hashtbl.mem tabh (s, code)) then begin
        let smin = ref 1 in
        while not tab_X.(!smin) do incr smin done;
        tab_X.(!smin) <- false;
        let (pds, _) = chemin_min !smin tab_X in
        let poids_min = ref (pds + g.(!smin).(s)) in
        tab_X.(!smin) <- true;
        for t = !smin + 1 to Array.length tab_X - 1 do
            if tab_X.(t) then begin
               tab_X.(t) <- false;
               let poids = fst (chemin_min t tab_X) + g.(t).(s) in
               tab_X.(t) <- true;
               if poids < !poids_min then
                  (poids_min := poids; smin := t)
            end
        done;
        Hashtbl.add tabh (s, code) (!poids_min, !smin);
    end;
    Hashtbl.find tabh (s, code) end;;
```

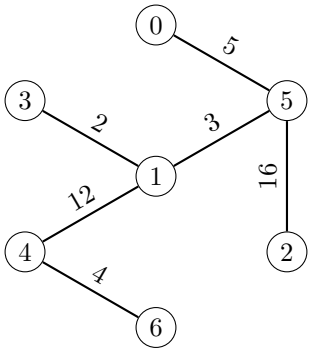
Question 16 On commence par créer un tableau de booléens correspondant à un ensemble $X = S \setminus \{0\}$, ainsi qu'un tableau correspondant au chemin à calculer. On calcule le prédécesseur de 0 dans un chemin de 0 à 0 passant par les sommets de X, puis on reconstruit le chemin à l'envers, en enlevant le dernier prédécesseur et en cherchant un chemin de 0 à ce sommet.

```
let pvc_dynamique () =
    let n = Array.length g in
    let tab_X = Array.init n (fun i -> i <> 0) in
    let cmin = Array.make n 0 in
    let pred = ref (snd (chemin_min 0 tab_X)) in
    for i = n - 1 downto 1 do
        cmin.(i) <- !pred;
        tab_X.(!pred) <- false;
        pred := snd (chemin_min !pred tab_X)
    done;
    cmin;;
```

Question 17 L'algorithme est fait de telle sorte que chaque couple (s, X) sera étudié, pour tout $s \in [\![0, n-1]\!]$ et tout $X \subset S \setminus \{0\}$. Il existe $n2^{n-1}$ tels couples. Par ailleurs, pour chacun des couples (s, X), tels que $X \neq \emptyset$, le premier calcul de $(P_{\min}(s, X), pred(s, X))$ fait un parcours linéaire d'un tableau de taille n, en plus des appels récursifs à `chemin_min`. Les autres calculs sont en $\mathcal{O}(1)$ grâce à la mémoïsation. La complexité temporelle totale est donc en $\Theta(n^2 2^n)$. La complexité spatiale est en $\Theta(n2^n)$ (le stockage dans la table de hachage).

4 Algorithme de Prim

Question 18 On obtient l'arbre couvrant :



Question 19 On fait les choix de structures suivants :

- on propose de ne pas représenter B explicitement, mais de créer directement un graphe par tableau de listes d'adjacence. Rajouter une arête $\{s,t\}$ à B correspondra à ajouter s à la liste d'adjacence de t et réciproquement ;
- on représente R par un tableau de booléens ;
- une structure qui n'est pas décrite intervient pour un calcul efficace : une file de priorité contenant des arêtes dont la priorité est leur poids. On peut initialiser cette file de priorité avec toutes les arêtes incidentes à s_0. Dans la boucle Tant que, on extrait l'arête $\{s,t\}$ de priorité minimale de la file de priorité, et on vérifie si $s \in R$ et $t \notin R$ (ou l'inverse) :
  - si ce n'est pas le cas, on ne fait rien et on passe à l'itération suivante ;
  - sinon, on ajoute t à R, on ajoute les arêtes dans le graphe en construction, puis on ajoute toutes les arêtes $\{t,u\}$ pour u voisin de t.

L'algorithme est fait de tel sorte que chaque arête du graphe passera au plus deux fois par la file de priorité. Le calcul de la complexité est alors comme suit :

- la création du graphe et de R sont en $\Theta(|S|)$;
- on fait au total au plus $2|A|$ ajout et extraction à une file de priorité.

La complexité totale est donc en $\mathcal{O}(|S| + |A|\log|A|) = \mathcal{O}(|A|\log|S|)$.

Question 20 Au cours de la boucle Tant que de l'algorithme, la propriété « (R,B) est connexe » est un invariant de boucle. C'est clair, car à chaque passage dans la boucle, on ajoute une arête reliant un sommet de R à un sommet de $\overline{R}$ qu'on ajoute à R. Finalement, à la fin de l'algorithme, $R = S$ et on renvoie (S,B).

L'algorithme termine bien, car le graphe G étant connexe, on pourra toujours trouver une arête entre un sommet de R et de $S \setminus R$, tant que $R \neq S$.

Sachant qu'il y a eu $|S|-1$ passage dans la boucle, $|B| = |S|-1$. (S,B) est donc un graphe connexe à $|S|-1$ arêtes. Il s'agit donc d'un arbre. Comme son ensemble de sommets est S, c'est un arbre couvrant.

Question 21 Notons $a_i = \{s,t\}$. Il existe dans T^* un chemin de s à t. Sachant que $s \in R$ et $t \notin R$, il existe une arête de ce chemin reliant un sommet de R à un sommet en dehors de R. En effet, si ce chemin est $(s_0, s_1, ..., s_k)$, alors il existe un indice $i \in [\![1,k]\!]$ minimal tel que $s_i \in S \setminus R$ (car $s_k \in S \setminus R$). Par minimalité, l'arête $a^* = \{s_{i-1}, s_i\}$ est bien l'arête cherchée.

Question 22 On a par ailleurs $f(a_i) \leq f(a^*)$. En effet, c'est l'arête a_i qui a été choisie lors de l'algorithme de Prim.

Dès lors, considérons $T' = (S, B^* \cup \{a_i\} \setminus \{a^*\})$ et remarquons :

- T' est connexe : en effet, il existe toujours un chemin entre u et v, de la forme $u \rightsquigarrow s \xrightarrow{a_i} t \rightsquigarrow v$;
- T' possède $|S|-1$ arêtes car $a_i \notin B^*$ et $a^* \in B^*$. On en déduit que T' est un arbre couvrant ;

– $f(a^*) \leq f(a_i)$. En effet, $f(T') = f(T^*) + f(a_i) - f(a^*) \leq f(T^*)$ car T^* est un arbre couvrant minimal.

On en déduit finalement que $f(T') = f(T^*)$, donc T' est un arbre couvrant minimal de G. Par ailleurs, la première arête choisie par l'algorithme de Prim qui n'apparaît pas dans T' est une arête a_j avec $j > i$. C'est contradictoire avec la construction de T^*. On en déduit que T est bien un arbre couvrant minimal.

5 Approximation de PVC dans le cas métrique

Question 23 Soient $s, t \in S^3$. Alors $f(s,t) \leq f(s,t) + f(t,t)$. On en déduit $f(t,t) \geq 0$. Dès lors, $0 \leq f(t,t) \leq f(t,s) + f(s,t) = 2f(s,t)$. On en déduit $f(s,t) \geq 0$.

Question 24 c^* est un cycle hamiltonien. Si on supprime une arête a de c^*, on obtient donc un arbre couvrant de G. Comme T est un arbre couvrant minimal, on a donc $f(T) \leq f(c^*) - f(a) \leq f(c^*)$ car $f(a) \geq 0$ d'après la question précédente.

Question 25 Soit $r \in S$ et (T, r) l'arbre T enraciné en r. Définissons par induction, pour $s \in S$, le **tour depuis** s, $tour(s)$:

– si s est une feuille dans (T, r), alors $tour(s) = s$;

– sinon, soient s_1, s_2, ..., s_k les fils de s. Alors $tour(s) = s, tour(s_1), s, tour(s_2), s, ..., s, tour(s_k), s$.

Montrons par induction que pour $s \in S$, $f(tour(s))$ est égal au double du poids des arêtes qui composent le sous-arbre de racine s :

– si s est une feuille, $f(tour(s)) = 0$ est bien le double des arêtes qui composent le sous-arbre de racine s ;

– sinon, si s_1, s_2, ..., s_k sont les fils de s et que le résultat est établi pour ces fils, alors $f(tour(s)) = f(s, s_1) + f(tour(s_1)) + f(s_1, s) + ... + f(s, s_k) + f(tour(s_k)) + f(s_k, s) = \sum_{i=1}^{k} 2f(s, s_i) + f(tour(s_i))$ est bien égal au double du poids des arêtes qui composent le sous-arbre de racine s.

On conclut par induction, et on a en particulier $f(tour(r)) = 2f(T)$.

Soit alors $s_0, s_1, ..., s_{n-1}$ les sommets de S parcours dans l'ordre d'un parcours en profondeur préfixe de T en partant du sommet $s_0 = r$. Considérons c_T le cycle hamiltonien $(s_0, s_1, ..., s_{n-1})$. Montrons que $f(c_T) \leq f(tour(r))$. En effet, les sommets $s_0, s_1, ..., s_{n-1}$ apparaissent dans $tour(r)$ pour la première fois dans le même ordre que dans c_T (par définition du parcours préfixe). Par l'inégalité triangulaire, pour $i \in [\![0, n-2]\!]$, le poids $f(s_i, s_{i+1})$ est inférieur ou égal au poids du chemin de s_i à s_{i+1} dans $tour(r)$. Comme $tour(r)$ termine par $r = s_0$, on a bien $f(c_t) \leq f(tour(r))$.

Finalement, $f(c_T) \leq f(tour(r)) = 2f(T) \leq 2f(c^*)$. Sachant que l'algorithme de Prim et un algorithme de parcours ont une complexité polynomiale, on a bien répondu à la question.

6 Approximer PVC est difficile dans le cas général

Question 26 Sens direct : si G possède un cycle hamiltonien, alors K_G possède ce même cycle hamiltonien, passant par des arêtes de poids 1. Ce cycle étant de taille $|S|$, le poids du cycle est $|S| \leq \alpha|S|$ (car $\alpha \geq 1$).

Sens réciproque : si K_G possède un cycle hamiltonien de poids $\leq \alpha|S|$, alors ce cycle ne peut passer que par des arêtes de poids 1 (car sinon le poids serait $> \alpha|S|$)). Comme ces arêtes existent dans G, on en déduit l'existence d'un cycle hamiltonien dans G.

Question 27 La construction de K_G se fait en temps polynomial en la taille de G. Dès lors, on peut appliquer l'algorithme d'approximation à K_G et trouver un cycle hamiltonien c tel que $f(c) \leq \alpha f(c^*)$. Distinguons :

– si $f(c) \leq \alpha|S|$, alors il existe un cycle hamiltonien dans G d'après la question précédente ;
– sinon, $\alpha|S| < f(c) \leq \alpha f(c^*)$, donc $|S| < f(c^*)$, on en déduit qu'il n'existe pas dans K_G de cycle hamiltonien ne passant que par des arêtes de poids 1, donc pas de cycle hamiltonien dans G.

On peut alors résoudre le problème `Cycle hamiltonien` en temps polynomial.

⋆ ⋆ ⋆