Composition d'Informatique n°1

Corrigé

⋆ ⋆ ⋆

Problème 1 : théorème de Rice

Question 1 Supposons qu'il existe une fonction `appartient : string -> bool` qui résout le problème `Appartient`. Définissons la fonction `paradoxe` suivante :

```
let rec paradoxe <f> =
    if appartient <f, f> then paradoxe <f>
    else true
```

Considérons alors un appel `paradoxe <paradoxe>` :

- si cet appel termine et renvoie `true`, alors `appartient <paradoxe, paradoxe>` renvoie `true`, donc l'appel `paradoxe <paradoxe>` s'appelle récursivement indéfiniment et ne termine pas ;
- si cet appel ne termine pas ou renvoie `false`, alors `appartient <paradoxe, paradoxe>` renvoie `false`, donc `paradoxe <paradoxe>` termine et renvoie `true`.

Dans les deux cas on a une contradiction, ce qui met en défaut l'existence de la fonction `appartient`. Le problème `Appartient` est donc indécidable.

Par ailleurs, la fonction

```
let appartient <f, x> = universel <f, x>
```

résout partiellement le problème, qui est donc semi-décidable.

Question 2

a) Ce problème n'est pas associé à une propriété sémantique, car par exemple les fonctions :

```
let f1 x = true

let f2 x =
    for i = 0 to 2 do () done;
    true
```

ont le même langage de fonction, mais l'une est une instance négative et l'autre une instance positive du problème.

b) Ce problème est bien associé à une propriété sémantique, à savoir $P = \{E \subseteq \Sigma^* \mid |E| \leqslant 10\}$.

Question 3 Comme P est non triviale, il existe une fonction vérifiant la propriété et une fonction ne vérifiant pas la propriété. Pour $\langle g \rangle$ le code source de la fonction vérifiant la propriété, on a bien $L(g) \in P$.

Question 4 On montre l'équivalence de la réduction :

- supposons que $\langle f, x \rangle$ soit une instance positive de `Appartient`. Alors $f(x)$ termine et renvoie `true`. On en déduit que $h(y)$ termine et renvoie `true` si et seulement si $g(y)$ termine et renvoie `true`. Comme $L(g) \in P$, cela signifie que $L(h) \in P$, donc $\langle h \rangle$ est une instance positive de Π_P ;
- supposons que $\langle f, x \rangle$ soit une instance négative de `Appartient`. Alors $f(x)$ ne termine pas ou renvoie `false`. On en déduit que c'est également le cas pour $h(y)$, pour tout y, soit que $L(h) = \emptyset \notin P$, donc $\langle h \rangle$ est une instance négative de Π_P.

Il s'agit bien d'une réduction many-one. Le problème `Appartient` étant indécidable, on en déduit que Π_P est indécidable.

Question 5 Si $\emptyset \in P$, alors $\emptyset \notin \mathcal{P}(\Sigma^*) \setminus P$. Il suffit donc de faire le raisonnement précédent sur coΠ_P, pour montrer qu'il est indécidable. Or, si co$A$ est indécidable, alors A aussi, ce qui permet de conclure que Π_P est indécidable.

On a bien montré que dans tous les cas, si P est non triviale, alors Π_P est indécidable.

Question 6

a) Ce problème est décidable : il suffit de lire le code source de la fonction et de voir s'il contient une boucle `for` (éventuellement en faisant de l'analyse syntaxique pour vérifier que le mot-clé `for` est bien associé à une boucle) ;

b) ce problème est indécidable : il est associé à une propriété sémantique P non triviale. En effet, pour :

```
let f1 x = true

let f2 x = false
```

on a $L(f_1) \notin P$ et $L(f_2) \in P$. Par le théorème de Rice, le problème est bien indécidable.

Problème 2 : voyageur de commerce

Présentation du problème

1 Approche naïve

Question 7 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 8 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 9 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 10 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 11 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 12 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 13 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.

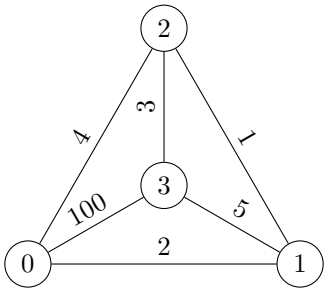
On obtient le cycle $(0, 1, 2, 4, 3)$, de poids $1 + 6 + 2 + 9 + 5 = 23$. Ce n'est pas le cycle de poids minimal, car $(0, 1, 4, 2, 3)$ est de poids $1 + 7 + 2 + 3 + 5 = 18 < 23$.

Question 14 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 15 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 16 On propose le graphe suivant :



En effet, voici les cycles renvoyés par l'algorithme glouton, selon le sommet de départ, tous de poids 106 :
– $(0, 1, 2, 3)$;
– $(1, 2, 3, 0)$;
– $(2, 1, 0, 3)$;
– $(3, 2, 1, 0)$.

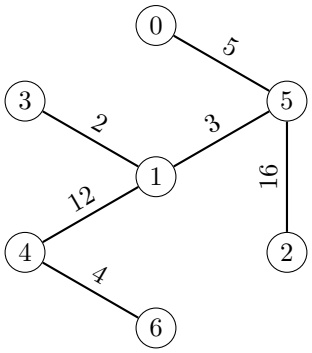
Or le cycle $(0, 1, 3, 2)$ est de poids 14.

3 Algorithme de Prim

Question 17 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 naïve, on effectue les opérations en temps $\mathcal{O}(|S|)$, soit une complexité totale en $\mathcal{O}(|A||S|)$.

Question 18 On obtient l'arbre couvrant :



Question 19 On manipule des graphes représentés par tableaux de listes d'adjacence et 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.

Ces implémentations permettent des opérations d'ajouts à B et R en temps constant. De plus, de par le principe de l'algorithme, 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 $\mathcal{O}(|S| + |A|\log|A|) = \mathcal{O}(|A|\log|S|)$, comme la complexité de l'algorithme de Kruskal.

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), qui est donc connexe.

Sachant que chaque ajout de sommet à R entraîne un ajout d'arête à B, $|B| = |S| - 1$. (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 Dans T^*, qui est connexe, il existe un (unique) chemin de s_i à t_i. Sachant que $s_i \in R_i$ et $t_i \notin R_i$, ce chemin contient bien une arête reliant un sommet de R_i à $S \setminus R_i$.

Question 22 On montre que P est un invariant de boucle :

- initialement, $B = \emptyset$, et comme G est connexe, il possède un arbre couvrant minimal, donc P est vérifiée ;
- si on suppose que P est vérifiée au début du i-ème passage dans la boucle Tant que, soit $T^* = (S, B^*)$ l'arbre couvrant minimal correspondant. On distingue :
  - si a_i n'est pas ajoutée à B_i, alors $B_{i+1} = B_i \subseteq B^*$;
  - si a_i est ajoutée à B_i, et $a_i \in B^*$, alors à nouveau, $B_{i+1} \subseteq B^*$;
  - sinon, notons a l'arête donnée par la question précédente. Posons $T' = (S, B^* \cup \{a_i\} \setminus \{a\})$. Alors T' est un arbre :

— il est connexe car on retire une arête d'un cycle dans $(S, B^* \cup \{a_i\})$, qui est connexe ;

— son nombre d'arêtes est $|B^*| = |S| - 1$.

De plus, par principe de l'algorithme de Prim, $f(a_i) \leqslant f(a)$, donc $f(T') \leqslant f(T^*)$, soit T' est un arbre couvrant minimal.

Enfin, $B_{i+1} \subseteq B^* \cup \{a_i\} \setminus \{a\}$, donc P est vérifiée.

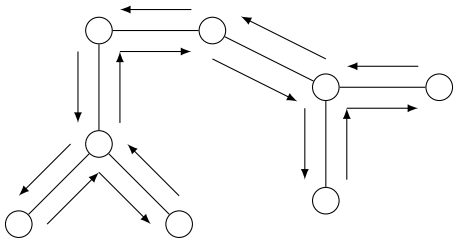
Finalement, à la fin de l'algorithme, on a $B \subseteq B^*$, et $|B| = |B^*|$, donc l'arbre renvoyé est bien un arbre couvrant minimal.

4 Approximation de PVC dans le cas métrique

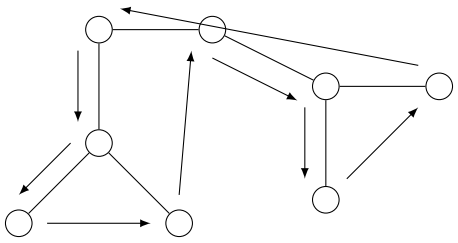
Question 23 Soient $s, t \in S^3$. Alors $f(s,t) \leqslant f(s,t) + f(t,t)$. On en déduit $f(t,t) \geqslant 0$. Dès lors, $0 \leqslant f(t,t) \leqslant f(t,s) + f(s,t) = 2f(s,t)$. On en déduit $f(s,t) \geqslant 0$.

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 $f(T) \leqslant f(c^*) - f(a) \leqslant f(c^*)$ car $f(a) \geqslant 0$ d'après la question précédente.

Question 25 Si on appelle **tour** d'un arbre un parcours des sommets dans l'ordre d'un parcours en profondeur, mais en ne passant que par des arêtes, alors le poids d'un tour correspond au double du poids des arêtes (voir le schéma suivant).



Dans le graphe complet, faire un cycle hamiltonien correspondant à l'ordre du parcours en profondeur permet de « prendre des raccourcis », d'après l'inégalité triangulaire. On en déduit que le poids du cycle hamiltonien est inférieur ou égal au double du poids de l'arbre, lui-même inférieur ou égal au poids d'un cycle hamiltonien optimal.



Question 26 On commence par écrire une fonction qui fait un parcours en profondeur. En plus du graphe et de sa taille, la fonction prendre en argument :

- le sommet courant ;
- un tableau des sommets déjà vus ;
- un tableau correspondant à l'ordre préfixe en cours de construction ;
- un pointeur vers l'indice de la prochaine case du tableau `ordre` à modifier.

Le principe correspond à l'écriture usuelle d'un DFS, en pensant à vérifier l'existence d'une arête avant de lancer un appel récursif.

```
void dfs(int* T, int n, int s, int* vus, int* ordre, int* idx){
    if (!vus[s]){
        vus[s] = true;
        ordre[*idx] = s;
        (*idx)++;
        for (int t=0; t<n; t++){
            if (T[n*s + t] == 1) dfs(T, n, t, vus, ordre, idx);
        }
    }
}
```

Dès lors, l'algorithme d'approximation consiste à calculer l'arbre couvrant minimal, et faire un parcours en profondeur dessus.

```
int* PVC_approx(int* G, int n){
    int* T = prim(G, n);
    int* vus = malloc(n * sizeof(*vus));
    int* ordre = malloc(n * sizeof(*ordre));
    int idx = 0;
    for (int i=0; i<n; i++) vus[i] = false;
    dfs(T, n, 0, vus, ordre, &idx);
    free(vus);
    return ordre;
}
```

⋆ ⋆ ⋆