Devoir maison n°2

À rendre le lundi 30 septembre

⋆ ⋆ ⋆

Castors affairés et algorithme de Prim

Les deux parties sont indépendantes.

1 Castors affairés

Dans cette partie, on suppose que l'exécution d'un programme OCaml se fait sur un ordinateur à mémoire infinie. En particulier, on suppose que la représentation des entiers est non bornée.

On dit qu'une fonction $f : \mathbb{N} \to \mathbb{N}$ est **calculable** s'il existe une fonction OCaml `f : int -> int`, dont l'exécution termine toujours, qui calcule les images par f.

Un castor affairé (*busy beaver* en anglais) est un programme dont l'exécution termine toujours et qui calcule une valeur la plus grande possible parmi tous les programmes de même taille. Formellement, on s'intéresse au problème d'optimisation suivant : `Castor affairé` :

- **Instance** : un entier naturel n.
- **Solution** : la valeur k renvoyée par un programme OCaml contenant n caractères (parmi les 128 caractères possibles de l'encodage ASCII), dont l'exécution termine toujours et renvoie un entier.
- **Optimisation** : maximiser k.

Par exemple, le programme suivant :

```
let k=99 in k*k*k
```

est un programme de taille 17, qui termine toujours et renvoie 970299. Ce n'est pas un castor affairé, en effet le programme suivant (qui n'est toujours pas un castor affairé) renvoie une valeur plus grande :

```
99999999999999999
```

Pour un entier n donné, on notera $C(n)$ la valeur maximale renvoyée par un programme OCaml de taille n qui termine et renvoie un entier. On pose $C(0) = 0$ par convention.

On cherche à montrer que le problème du castor affairé n'est pas calculable, c'est-à-dire que la fonction C n'est pas calculable.

Question 1 Combien existe-t-il de programme OCaml à n caractères, en supposant qu'on dispose de 128 caractères différents possibles ? Expliquer pourquoi le fait qu'il y en ait un nombre fini ne permet pas de conclure que le problème est calculable.

Question 2 Montrer que la fonction C est correctement définie.

Question 3 Montrer que C est une fonction croissante, puis que pour $n \in \mathbb{N}$, $C(n+4) > C(n)$.

Question 4 Que vaut $C(1)$? Le fait qu'on puisse déterminer cette valeur contredit-il la non-calculabilité du problème ?

On considère $f : \mathbb{N} \to \mathbb{N}$ une fonction calculable.

Question 5 Montrer qu'il existe un entier k tel que pour tout $n \in \mathbb{N}$, $f(n) \leqslant C(\lfloor \log_{10} n \rfloor + k)$.

Question 6 Montrer qu'il existe un entier $n_0 \in \mathbb{N}$ tel que $f(n_0) < C(n_0)$.

Question 7 Conclure.

On considère le problème `Arrêt` :

* **Instance** : le code source d'un programme OCaml.
* **Question** : est-ce que l'exécution de ce programme termine ?

Question 8 Montrer, en utilisant la non-calculabilité du problème du castor affairé, que le problème `Arrêt` est indécidable.

2 Algorithme de Prim

Dans cette partie, on s'intéresse à un algorithme de calcul d'arbre couvrant de poids minimal d'un graphe non orienté, pondéré, connexe, $G = (S, A, f)$.

2.1 Une file de priorité

L'algorithme de Prim utilise des files de priorité efficaces pour obtenir une bonne complexité. On se propose ici de faire une implémentation des tas binaires en C.

Comme la file de priorité contiendra des arêtes, on commence par leur définir un type, via :

```
struct Arete {
    int s;
    int t;
    int pds;
};

typedef struct Arete arete;
```

Une arête $a = \{s, t\}$ de poids $f(a) = p$ sera donc représentée par un objet `a` de type `arete` tel que `a.s` $= s$, `a.t` $= t$ et `a.pds` $= p$.

On représente alors un tas-min arborescent via un tableau par le type suivant :

```
struct Tasmin {
    int taille;
    int capa;
    arete* data;
};

typedef struct Tasmin tasmin;
```

tel que si `tas` est un objet de type `tasmin` représentant un tas d'arêtes $\mathcal{T}$, alors :

- `tas.taille` correspond à la tailel du tas, c'est-à-dire au nombre d'éléments de $\mathcal{T}$;
- `tas.capa` correspond à la capacité du tas, c'est-à-dire la taille maximale que peut atteindre $\mathcal{T}$;
- `tas.data` est un tableau de taille `tas.capa`, tel que les éléments de $\mathcal{T}$ sont stockés aux indices entre 0 et `tas.taille` $- 1$. Les éléments aux indices $\geqslant$ `tas.taille` sont des arêtes arbitraires ;
- la représentation arborescente de $\mathcal{T}$ est un tas-min, c'est-à-dire que c'est un arbre presque-parfait, et que tout nœud a une priorité (ici le poids de l'arête) plus petite que celles de ses enfants ;
- dans le tableau, les enfants gauche et droit de l'élément à l'indice i se trouvent aux indices $2i+1$ et $2i+2$ respectivement.

Comme la taille du tas a vocation à être modifiée, on utilisera parfois des objets de type `tasmin*`. Lorsqu'on parlera d'un tas, cela pourra désigner un objet de type `tasmin` ou de type `tasmin*` selon la situation. On se réfèrera au type des fonctions pour lever les ambiguïtés.

Question 9 Écrire une fonction `tasmin* creer_tas(int capa)` qui crée un tas de taille nulle et de capacité donnée en argument.

Question 10 Écrire une fonction `void liberer_tas(tasmin* tas)` qui libère l'espace mémoire utilisé par un tas.

Question 11 Écrire une fonction `void inserer(tasmin* tas, arete a)` qui insère une arête dans un tas. La fonction devra commencer par vérifier que le tas n'est pas déjà à capacité maximale. Après l'insertion, le nouveau tas devra à nouveau vérifier toutes les contraintes sur les tas.

Question 12 Écrire une fonction `int enfant_min(tasmin tas, int i)` qui prend en argument un tas et un indice i et renvoie l'indice de l'enfant du nœud d'indice i ayant la plus petite priorité. La fonction devra renvoyer l'indice i lui-même s'il n'a pas d'enfant. On s'assurera de ne pas lire de cas dans le tableau à un indice supérieur ou égal à la taille du tas.

Question 13 En déduire une fonction `arete extraire(tasmin* tas)` qui extrait l'arête de plus petite priorité dans un tas et la renvoie. La fonction devra commencer par vérifier que le tas n'est pas vide.

Question 14 Quelles sont les complexités temporelles des fonction `inserer` et `extraire` en fonction de la taille du tas ? Justifier brièvement.

2.2 Algorithme de Prim

L'algorithme de Prim est donné par le pseudo-code suivant :

Entrée : Graphe $G = (S, A, f)$ pondéré non orienté connexe
Début algorithme
  Poser $B = \emptyset$.
  Poser $R = \{s_0\}$ un sommet choisi arbitrairement.
  Tant que $R \neq S$ **Faire**
    Poser $\{s, t\} \in A$ l'arête de poids minimal telle que $s \in R$ et $t \notin R$.
    Ajouter t à R.
    Ajouter $\{s, t\}$ à B.
  Renvoyer (S, B)

Question 15 Déterminer le graphe renvoyé par l'algorithme de Prim appliqué au graphe G_1 représenté figure 1, en supposant $s_0 = 0$.

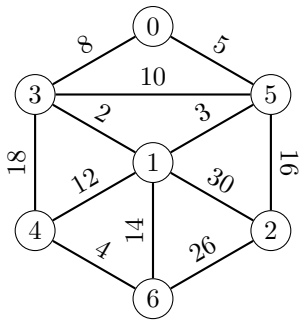

FIGURE 1 – Le graphe G_1

On représente un graphe pondéré en C comme un tableau de listes d'adjacence, implémentées par :

```c
struct Liste {
    int t;
    int pds;
    struct Liste* suiv;
};

typedef struct Liste liste;
```

Question 16 Écrire une fonction `liste* cons(liste* lst, int t, int pds)` qui prend en argument une liste et deux entiers `t` et `pds` et renvoie une liste ayant comme tête ces deux valeurs et comme queue la liste `lst`.

Dès lors, on peut implémenter un graphe par :

```c
struct Graphe {
    int n;
    liste** adj;
};

typedef struct Graphe graphe;
```

tel que si $G = (S, A, f)$ est représenté par l'objet `g` de type `graphe`, alors :

- `g.n` est égal à $|S|$ et $S = [\![0, n-1]\!]$;
- `g.adj` est un tableau de taille n tel que pour $s \in S$, `g.adj[s]` est une liste chaînée contenant les voisins de s. Ainsi, pour chaque maillon `lst` de la liste, `lst->t` correspond à un voisin t de s, et `lst->pds` correspond au poids de l'arête $\{s, t\}$.

Question 17 Écrire une fonction `int nombre_aretes(graphe g)` qui prend en argument un graphe $G = (S, A, f)$ et renvoie $|A|$.

Question 18 Écrire une fonction `void ajout_voisins(graphe g, tasmin* tas, int s)` qui prend en argument un graphe G, un tas-min $\mathcal{T}$ et un sommet $s \in S$ et ajoute au tas $\mathcal{T}$ toutes les arêtes $\{s, t\}$ pour t voisin de s.

Question 19 En déduire une fonction `graphe prim(graphe g)` qui prend en argument un graphe G et renvoie un arbre couvrant minimal de G calculé selon l'algorithme de Prim.

Question 20 Déterminer, en justifiant, la complexité temporelle de l'algorithme de Prim en fonction de $|S|$ et $|A|$.

Question 21 Montrer que l'algorithme de Prim renvoie un arbre couvrant de G.

On cherche à montrer que l'arbre $T = (S, B)$ renvoyé par l'algorithme de Prim est bien un arbre couvrant minimal de G. Pour ce faire, supposons qu'il ne le soit pas, et soit $T^* = (S, B^*)$ un arbre couvrant minimal. Notons $(a_1, a_2, \ldots, a_{n-1})$ les arêtes ajoutées à B au cours de l'algorithme de Prim, dans cet ordre. Par hypothèse, $T \neq T^*$, donc il existe $i \in [\![1, n-1]\!]$ minimal tel que $a_i \notin B^*$. Considérons T^* comme l'arbre couvrant minimal qui maximise cette valeur de i. On pose R l'ensemble des sommets juste avant l'ajout de a_i à B au cours de l'algorithme.

Question 22 On pose $a_i = \{s, t\}$. Montrer que dans un chemin de s à t dans T^*, il existe une arête $a^* \in B^*$, reliant un sommet de R et un sommet de $S \setminus R$.

Question 23 Montrer que $f(a_i) \leqslant f(a^*)$ et que cela entraîne une contradiction, puis conclure.