Préliminaires

Présentation du sujet

L'épreuve est composée d'un problème unique comportant 27 questions. On y étudie le problème du voyageur de commerce. L'objectif est la conception et l'analyse d'algorithmes exacts ou approchés permettant de le résoudre.

Le problème est divisé en deux sections. Dans la première section (page 2) on établit que le problème du voyageur de commerce est un problème NP-complet et on en conçoit deux algorithmes exponentiels de résolution exacte. Dans la seconde (page 4), on construit un algorithme d'approximation pour le problème du voyageur de commerce sous certaines hypothèses et on constate puis montre que ces hypothèses sont a priori nécessaires pour espérer concevoir une approximation pour ce problème.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désignera la même entité, mais du point de vue mathématique avec la police en italique (par exemple, n ou n') et du point de vue informatique avec celle en romain à espacement fixe (par exemple `n` ou `nprime`).

Travail attendu

Pour répondre à une question, il est permis d'utiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. Des rappels de programmation sont fait en annexe et peuvent être utilisés directement.

Selon les questions, il faudra coder les fonctions à l'aide du langage de programmation C ou OCaml en reprenant le prototype des fonctions fourni par l'énoncé, ou en pseudo-code (c'est-à-dire dans une syntaxe souple mais conforme aux possibilités offertes par le langage C). Quand l'énoncé demande de coder une fonction, sauf demande explicite, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que les pré-conditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles et de structurer le code en blocs ou à l'aide de fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C aurait été fourni en guise de réponse.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

1 Algorithmes exacts pour `PVC`

Si G est un graphe non orienté, un *cycle hamiltonien* de G est un cycle passant une et une seule fois par chaque sommet de G. Un cycle hamiltonien dans un graphe à n sommets sera noté $(s_0, ..., s_{n-1})$ plutôt que $(s_0, ..., s_{n-1}, s_0)$ (sans répétition du premier sommet).

Le sujet étudie le problème du voyageur de commerce, noté `PVC` défini comme suit :

$$\begin{cases} \textbf{Entrée :} \text{ Un graphe } G = (S, A) \text{ à } n \text{ sommets non orienté, complet et pondéré par } f. \\ \textbf{Solution :} \text{ Un cycle hamiltonien } c = (s_0, ..., s_{n-1}) \text{ dans } G. \\ \textbf{Optimisation :} \text{ Minimiser le poids } f(c) \text{ de } c, \text{ défini par } f(c) = f(s_0, s_{n-1}) + \sum_{i=0}^{n-2} f(s_i, s_{i+1}). \end{cases}$$

Dans l'ensemble du sujet, si $G = (S, A)$ est un graphe, on identifiera l'ensemble S à $\{0, 1, ..., |S| - 1\}$. Sauf mention contraire, G désigne toujours un graphe non orienté, pondéré par f et complet.

1.1 NP-complétude de `PVC`

Dans cette partie, on admet que le problème `CYCLE HAM` ci-dessous est NP-complet :

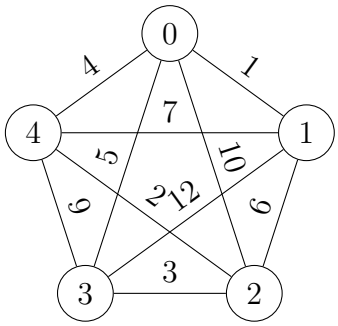
$$\begin{cases} \textbf{Entrée :} \text{ Un graphe } G = (S, A) \text{ non orienté.} \\ \textbf{Question :} \text{ Existe-t-il un cycle hamiltonien dans } G \text{ ?} \end{cases}$$

1. Donner la version décisionnelle `PVC-deci` du problème de décision `PVC`.
2. Montrer que `PVC-deci` est un problème appartenant à la classe NP.
3. Montrer que `PVC-deci` est un problème NP-complet.

1.2 Un algorithme naïf

Les questions de programmation de cette partie doivent être écrites en utilisant la syntaxe du langage C. Les en-têtes `#include <stdlib.h>` et `#include <stdbool.h>` ont été déclarées.

Dans cette partie et la partie 2.1, un graphe non orienté, pondéré et complet est représenté par sa matrice d'adjacence, laquelle est implémentée en C par un tableau unidimensionnel d'entiers, les lignes de la matrice étant consécutives dans le tableau. Par exemple, le graphe G_0 en figure 1 est représenté par le tableau suivant :



$$M_0 = \begin{pmatrix} 0 & 1 & 10 & 5 & 4 \\ 1 & 0 & 6 & 12 & 7 \\ 10 & 6 & 0 & 3 & 2 \\ 5 & 12 & 3 & 0 & 9 \\ 4 & 7 & 2 & 9 & 0 \end{pmatrix}$$

```
int G0[25] =
  {0, 1, 10, 5, 4,
   1, 0, 6, 12, 7,
   10, 6, 0, 3, 2,
   5, 12, 3, 0, 9,
   4, 7, 2, 9, 0};
```

FIGURE 1 - Le graphe G_0, sa matrice d'adjacence et sa représentation en C.

Par ailleurs, un cycle hamiltonien $c = (s_0, ..., s_{n-1})$ dans un graphe G d'ordre n est représenté par un tableau contenant les sommets du cycle.

4. Si G est un graphe complet à n sommets, combien y a-t-il de cycles hamiltoniens différents (c'est-à-dire non constitués des mêmes arêtes) dans G ?

5. Écrire une fonction `int f(int* G, int n, int s, int t)` de spécification :
   Pré-conditions : G est un graphe pondéré à n sommets et s et t sont des sommets de G.
   Valeur de retour : le poids de l'arête (s, t).

6. Écrire une fonction `int poids_cycle(int* G, int*c, int n)` dont la spécification suit :
   Pré-conditions : G est un graphe pondéré à n sommets et c en est un cycle.
   Valeur de retour : Un entier correspondant au poids du cycle c dans G.

   Par exemple, si c est le cycle $(0, 2, 4, 3, 1)$, `poids_cycle(G0,c,5)` renvoie 34.

On remarque que toute permutation de $[\![0, n-1]\!]$ forme un cycle hamiltonien dans G. Pour déterminer une solution à `PVC` on propose naïvement de parcourir toutes les permutations dans l'ordre lexicographique, de calculer le poids de chacune et de conserver l'une de celles de poids minimal. On rappelle que la permutation qui suit $(0, 2, 4, 3, 1)$ dans l'ordre lexicographique est $(0, 3, 1, 2, 4)$.

Si $p = (p_0, ..., p_{n-1})$ est une permutation de $[\![0, n-1]\!]$, on définit les indices suivants :

- $j = \max\{i \in [\![0, n-2]\!] \,|\, p_i < p_{i+1}\}$ et $j = -1$ si cet ensemble est vide.
- $k = \max\{i \in [\![j+1, n-1]\!] \,|\, p_j < p_i\}$ et $k = n$ si $j = -1$.

7. En utilisant les indices j et k, décrire en pseudo-code un algorithme permettant de modifier une permutation p de $[\![0, n-1]\!]$ en place de sorte à obtenir la permutation suivant p dans l'ordre lexicographique. Si p est la dernière permutation selon l'ordre lexicographique, cet algorithme renvoie faux, sinon il renvoie vrai. On suppose dans la suite que cette fonction est implémentée par la fonction `bool permutation_suivante(int* p, int n)`.

8. Écrire une fonction `int* PVC_naif(int* G, int n)` de spécification :
   Pré-conditions : G est un graphe pondéré et n son nombre de sommets.
   Valeur de retour : Un tableau représentant un cycle de poids minimal dans G.

9. Déterminer la complexité temporelle de `PVC_naif` en fonction du nombre de sommets de G. On admet pour ce faire que `permutation_suivante` a une complexité amortie en $O(1)$.

1.3 Algorithme de Held-Karp

Les questions de programmation de cette partie doivent être écrites en utilisant la syntaxe du langage OCaml. L'utilisation des fonctions usuelles des modules `Array` et `List` est autorisée.

En Ocaml, les graphes complets sont représentés par matrice d'adjacence. Par exemple, le graphe G_0 décrit en figure 1 est représenté par :

```
let g0 = [|[|0; 1; 3; 5; 4|];
           [|1; 0; 6; 12; 7|];
           [|3; 6; 0; 2; 10|];
           [|5; 12; 2; 0; 9|];
           [|4; 7; 10; 9; 0|]|]
```

Par ailleurs, si S est l'ensemble de sommets d'un graphe, on représente un sous ensemble $S' \subset S$ par un tableau de booléens de taille $|S|$ contenant `true` en case i si le sommet i appartient à S'.

L'algorithme de Held-Karp est un algorithme de programmation dynamique permettant de résoudre `PVC`. Pour ce faire, il détermine des solutions aux sous-problèmes suivants : "quel est un chemin de poids minimal entre le sommet s_0 et le sommet s_k passant une et une seule fois par les sommets intermédiaires $s_1, ..., s_{k-1}$?". Le problème qui nous intéresse est celui déterminant un chemin de poids minimal du sommet 0 au sommet 0 et passant par les sommets intermédiaires de $\{1, ..., n-1\}$.

Si $s \in S$ et $S' \subset S \setminus \{0, s\}$ est un ensemble de sommets, on note $P_{\min}(s, S')$ le poids minimal d'un chemin de 0 à s passant une et une seule fois par chaque sommet de S'. On note également $\text{pred}(s, S')$ un sommet qui précède s sur un tel chemin. Par convention, $\text{pred}(0, \emptyset) = 0$.

10. Donner une formule de récurrence vérifiée par $P_{\min}(s, S')$. Expliquer comment calculer $\text{pred}(s, S')$.

Afin d'éviter d'effectuer plusieurs fois le même calcul d'un même $P_{\min}(s, S')$, on choisit de mémoïser les résultats déjà obtenus dans une table de hachage.

11. Expliquer pourquoi il ne faut pas utiliser d'objet mutable comme clé dans une table de hachage.

12. Écrire une fonction `encodage (s:int) (ensemble:bool array) :int*int` telle que :
Pré-conditions : $s \in S$ est un sommet et `ensemble` représente un ensemble $S' \subset S$.
Valeur de retour : un couple d'entiers représentant de manière unique le couple (s, S').

    On justifiera cette unicité. Un tel couple d'objets non mutables peut à présent servir de clé dans une table de hachage.

Des rappels quant à la manipulation de tables de hachage en OCaml sont fait en annexe. On suppose dans la fin de cette partie que les deux variables globales suivante sont créées :

- Une matrice d'entiers `g` correspondant à un graphe $G = (S, A, f)$.
- Une table de hachage, créée via `let tabh = Hashtbl.create 1`.

13. Écrire une fonction `chemin_min (s:int) (ensemble:bool array) :int*int` où :
Pré-conditions : $s \in S$ est un sommet de G et `ensemble` représente un ensemble $S' \subset S$.
Valeur de retour : le couple $(P_{\min}(s, S'), \text{pred}(s, S'))$.
Effets : les résultats intermédiaires du calcul sont stockés dans `tabh` au fil du calcul.

    On explicitera son fonctionnement.

14. En déduire une fonction `pvc_dynamique : unit -> int array` qui renvoie un cycle hamiltonien de poids minimal dans le graphe représenté par `g`, calculé selon l'algorithme de Held-Karp.

15. Déterminer les complexités spatiale et temporelle de `pvc_dynamique`.

2 Approximations pour PVC

2.1 Heuristique du plus proche voisin

Dans cette partie, on réutilise les modélisations introduites en partie 1.2. Afin d'obtenir un cycle hamiltonien de poids faible à défaut d'être minimal en un temps raisonnable, on utilise l'heuristique

du plus proche voisin. Cette dernière consiste à partir d'un sommet quelconque du graphe G, l'ajouter au cycle C en construction, puis de répéter les opérations suivantes :

- Regarder quel est le dernier sommet s ajouté à C.
- Déterminer l'ensemble V des voisins de s qui n'ont pas encore été ajoutés à C.
- Parmi les éléments de V, déterminer celui le plus proche de s et l'ajouter à C.

16. Déterminer le cycle renvoyé par l'heuristique du plus proche voisin appliquée au graphe G_0 décrit en figure 1 et en choisissant 0 comme sommet initial. Reprendre la question en partant du sommet 3. L'un de ces cycles est-il de poids minimal dans G_0 ?

17. Écrire une fonction `int plus_proche(int* G, bool* vus, int n, int s)` de spécification :
Pré-conditions : G est un graphe à n sommets, s est un sommet de G tel que `vus[s]` est vrai et `vus` est un tableau de booléens dont l'une des cases au moins contient `false`.
Valeur de retour : Un sommet t de G vérifiant :
    - `vus[t]` vaut `false`.
    - $f(s,t) = \min\{f(s,u) \mid \texttt{vus[u]} = \texttt{false}\}$.

    Ainsi, si `vus = {true, false, true, true, false}`, `plus_proche(G0,vus,5,2)` renvoie 4.

18. Écrire une fonction `int* PVC_glouton(int* G, int n)` de spécification :
Pré-conditions : G est un graphe pondéré à n sommets.
Valeur de retour : un cycle hamiltonien de G construit selon l'heuristique du plus proche voisin avec 0 comme sommet initial.

19. Déterminer la complexité de `PVC_glouton`.

20. Montrer que `PVC_glouton` n'est pas un algorithme d'approximation à facteur constant de `PVC`.

21. Dessiner un graphe pondéré complet à 5 sommets tel que l'heuristique du plus proche voisin ne renvoie jamais un cycle hamiltonien de poids minimal, quel que soit le sommet de départ.

2.2 Approximation dans le cadre métrique

Dans cette partie, $G = (S, A, f)$ est un graphe non orienté, pondéré, complet et tel que la fonction de poids f vérifié l'inégalité triangulaire ; c'est-à-dire :

$$\forall s,t,u \in S,\ f(s,u) \leq f(s,t) + f(t,u)$$

On étend de manière naturelle la fonction f aux sous-graphes et aux chemins de G ; par exemple, le poids d'un sous-graphe est la somme des poids des arêtes qui le composent. On note $c^\star$ un cycle hamiltonien de poids minimal de G et $T = (S, B)$ un arbre couvrant minimal de G.

22. Montrer que pour toute arête $a \in A$, on a $f(a) \geq 0$.

23. Montrer que $f(T) \leq f(c^\star)$.

24. En considérant un parcours en profondeur de T, en déduire l'existence d'un cycle hamiltonien c_T dans G, calculable en temps polynomial et tel que $f(c_T) \leq 2f(c^\star)$.

Pour la question suivante, on représente un graphe non orienté pondéré via le type :

```
type graphe = (int*int) array
```

c'est-à-dire par listes d'adjacence. Si (t, p) est un couple dans la liste d'adjacence d'un sommet s, t est un voisin de s et p le poids de l'arête non orientée (s, t). On suppose disposer d'une fonction `acm : graphe -> graphe` permettant de calculer un arbre couvrant minimal d'un graphe pondéré.

25. Écrire une fonction `pvc_approx : (g:graphe) :int array` de spécification :
Pré-conditions : G est un graphe pondéré par une fonction vérifiant l'inégalité triangulaire.
Valeur de retour : un tableau représentant un cycle hamiltonien de G de poids au plus deux fois le poids d'un cycle hamiltonien de poids minimal dans G.

    On exige une complexité polynomiale pour cette fonction.

2.3 Inapproximabilité dans le cas général

Dans cette partie, on cherche à montrer que l'hypothèse sur la fonction de poids faite à la partie précédente est critique pour concevoir un algorithme d'approximation de `PVC`.

Supposons qu'il existe une α-approximation (avec $\alpha > 1$) $\mathcal{A}$ de `PVC` ayant une complexité polynomiale en la taille de son entrée $G = (S, A, f)$. Si $G = (S, A)$ est un graphe non orienté avec $|S| > 2$, on considère le graphe non orienté pondéré et complet $K_G = (S, S^2, f)$ suivant :

- Pour $a \in A$, $f(a) = 1$.
- Pour $a \notin A$, $f(a) = \alpha|S|$.

26. Montrer que $G = (S, A)$ possède un cycle hamiltonien si et seulement si K_G possède un cycle hamiltonien de poids inférieur ou égal à $\alpha|S|$.

27. En déduire que sous ces hypothèses, on peut résoudre `CH` en temps polynomial et conclure.

Annexe : rappels de programmation

On rappelle que les tables de hachage peuvent être manipulées en Ocaml avec les fonctions suivantes :

- `Hashtbl.create : int -> ('a,'b) Hashtbl.t` prend en entrée un entier m et créé une table de hachage vide occupant un espace mémoire de taille proportionnelle à m. La taille de la table est automatiquement redimensionnée au besoin.
- `Hashtbl.add : ('a,'b) Hashtbl.t -> 'a -> 'b -> unit` prend en argument une table, une clé et une valeur et ajoute une association dans la table entre cette clé et cette valeur.
- `Hashtbl.mem : ('a,'b) Hashtbl.t -> 'a -> bool` teste si une table contient une clé donnée.
- `Hashtbl.find : ('a,'b) Hashtbl.t -> 'a -> 'b` prend en entrée une table et une clé et renvoie la valeur associée à la clé dans la table si elle existe ; l'exception `Not_found` est levée sinon.