Devoir Non Surveillé 2

À rendre le 6 janvier 2025

INFORMATIQUE MPI/MPI*

Tournez la page S.V.P.

Vue d'ensemble du sujet

Dans ce sujet, on se propose d'étudier le problème du voyageur de commerce (TSP pour Travelling Salesman Problem)

Chemins hamiltoniens, et problème du voyageur de commerce (TSP)

Soit $G = (S, A)$ un graphe non orienté. Un **cycle hamiltonien** de G est un cycle passant une fois et une seule par chaque sommet.
On définit le problème d'optimisation du voyageur de commerce (TSP) :

Instance : $G = (S, A, w)$ un graphe pondéré non orienté, avec $|S| = n$.

Solution : Un cycle hamiltonien $\mathcal{C} = (s_0, s_1, \ldots, s_{n-1}, s_n = s_0)$ minimisant la fonction de cout :

$$w(\mathcal{C}) = \sum_{i=0}^{n-1} w(s_i, s_{i+1})$$

Ce sujet est composé de 5 parties indépendantes, utilisant les langages de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I étudie une résolution du problème par recherche exhaustive naïve.
- La partie II s'intéresse à l'algorithme de programmation dynamique proposé par Held et Karp (et aussi indépendamment par Bellman).
- La partie III s'intéresse à une résolution gloutonne, et en montre les limites.
- La partie IV traite la difficulté du problème de seuil associé.
- La partie V montre la difficulté d'approximer une solution à un facteur constant près.
- Enfin, la partie VI montre que dans certains cas, il est possible de fournir une 2-approximation de la solution.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des 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 a été fourni en guise de réponse.

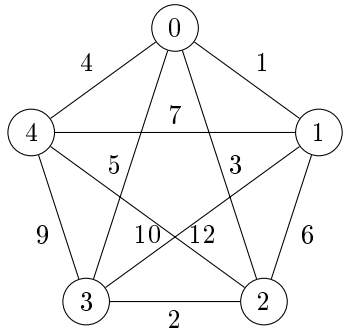
Partie I. Approche naïve (C) – 12 pts

☐ 1 – On suppose que G est un graphe complet d'ordre n. Déterminer le nombre de cycles hamiltoniens différents dans G. On supposera que deux cycles hamiltoniens sont différents s'ils ne sont pas constitués des mêmes arêtes.

Réponse 1. Intéressons nous aux permutations de $[\![0, n-1]\!]$. Chacune des $n!$ permutations définie bien un cycle hamiltonien dans un graphe complet. Chaque cycle apparaît $2n$ fois, car il y a n possibilités pour le choix du sommet de départ, et 2 possibilités pour le sens de parcours. Le nombre de cycles hamiltoniens est donc

$$\frac{(n-1)!}{2}.$$

Pour la suite de cette partie, on suppose qu'un graphe $G = (S, A, w)$ pondéré, non orienté et complet est représenté par sa matrice d'adjacence implémentée en C dans un tableau unidimensionnel d'entiers, les lignes de la matrice étant consécutives dans le tableau. Le graphe étant supposé complet, la valeur ∞ n'apparaitra pas dans la matrice. Par exemple, le graphe G_0 de la figure pourra être représenté par le code suivant :



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

C

```c
int G0[25] =
{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};
```

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

☐ 2 – Écrire une fonction `int w(int *G, int n, int s, int t)` qui prend en argument un tableau correspondant à un graphe $G = (S, A, w)$, un entier $n = |S|$, et deux entiers $(s, t) \in S^2$ et renvoie la valeur $w(s, t)$ correspondant au poids de l'arête (s, t).

Réponse 2.

poids d'une arête — C

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

On choisit de représenter un cycle hamiltonien $\mathcal{C} = (s_0, s_1, \ldots, s_{n-1}, s_n = s_0)$ d'un graphe G d'ordre n par un tableau de taille n contenant les sommets du cycle.

☐ 3 – Écrire une fonction `int poids_cycle(int *G, int* c, int n)` qui prend en argument un graphe $G = (S, A, w)$, un cycle $\mathcal{C}$ de G et un entier $n = |S|$ et renvoie $w(\mathcal{C})$ tel que défini précédemment. Par exemple, si `int *c = {0, 2, 4, 3, 1};` , alors `poids_cycle(G0, c, 5)` renverrai `35`.

Réponse 3.

poids d'un cycle C

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

On remarque que toute permutation de $[\![0, n-1]\!]$ forme un cycle hamiltonien de G. On propose de parcourir toutes les permutations par ordre lexicographique pour conserver celle de poids minimal. Par exemple, la permutation qui suit $(0, 2, 4, 3, 1)$ dans l'ordre lexicographique est $(0, 3, 1, 2, 4)$.
Si $p = (p_0, p_1, \ldots, p_{n-1})$ est une permutation de $[\![0, n-1]\!]$, on définit les indices suivants :
- $j = \max\{i \in [\![0, n-2]\!] \mid p_i < p_{i+1}\}$ et $j = -1$ si cet ensemble est vide.
- $k = \max\{i \in [\![j+1, n-1]\!] \mid p_j < p_i\}$ et $k = n$ si $j = -1$.

☐ 4 – En utilisant les indices j et k, décrire en français ou en pseudo-code un algorithme permettant de modifier p pour obtenir la permutation suivante dans l'ordre lexicographique et renvoyer «faux» si p était la dernière permutation et «vrai» sinon. On supposera pour la suite qu'une telle fonction est implémentée par une fonction `bool permut_suivante(int *p, int n)`.

Réponse 4. Premièrement, on peut remarquer qu'une permutation d'un sous ensemble de $[\![0, n-1]\!]$ est maximal pour l'ordre lexicographique si elle est triée dans le sens inverse.
L'indice j est tel que le sous-tableau $(p_i)_{j<i<n}$ est maximal. Ce sous-tableau étant trié dans le sens inverse, l'indice k est tel que p_k est le plus petit élément plus grand que p_j dans ce sous-tableau.
Pour trouver la permutation suivante, il faut changer p_j car toutes les permutations commençant par $(p_0, \ldots, p_j)$ ont été testées, mais pas toutes celles commençant par $(p_0, \ldots, p_{j-1})$ (par maximalité de j). Comme $(p_0, \ldots, p_{j-1})$ doivent rester inchangés, on échange p_j avec p_k qui est le nombre suivant dans l'ordre. On peut remarquer que les $(p_i)_{j<i<n}$ restent triés dans le sens inverse, on a donc juste à les remettres dans le bon ordre en les inversant.

parcours des permutations dans l'ordre lexicographique C

```c
void swap(int *p, int j, int k) {
        p[j] = p[j] + p[k];
        p[k] = p[j] - p[k];
        p[j] = p[j] - p[k];
}

bool permut_suivante(int *p, int n) {
        int j = n-2;

        while (j >= 0 && p[j] >= p[j+1]) {
                j = j - 1;
        }
        if (j == -1) {
                return false;
        }
        int k = n-1;
        while (p[j] >= p[k]) {
                k = k - 1;
        }
        swap(p, j, k);
        int min = j+1,
                max = n-1;
        while (max > min) {
                swap(p, min, max);
                min = min + 1;
                max = max - 1;
        }
        return true;
}
```

☐ 5 – Écrire une fonction `int *PVC_naif(int *G, int n)` qui prend en argument un graphe $G = (S, A, w)$ et un entier $n = |S|$ et renvoie un pointeur vers un tableau contenant un cycle hamiltonien de G de poids minimal.

Réponse 5.

Résolution de PVC par recherche exhaustive C

```c
int *PVC_naif(const int *G, int n) {
        int *p = malloc (n * sizeof (int));
        int *best = malloc (n * sizeof(int));

        for (int i = 0; i < n; i = i + 1) { // boucle 1
                p[i] = i; best[i] = i;
        }

        int cout_p = poids_cycle(G, p, n);
        int cout_best = cout_p;

        while (permut_suivante(p, n)) { // boucle 2
                cout_p = poids_cycle(G, p, n); // op 1
                if (cout_p < cout_best) {
                        cout_best = cout_p;
                        for (int i = 0; i < n; i = i + 1) { // boucle 3
                                best[i] = p[i];
                        }
                }
        }
        free(p);
        return best;
}
```

☐ 6 – Déterminer la complexité temporelle de `PVC_naif` en fonction de $n = |S|$. On admettra que `permut_suivante` a une complexité amortie en $\mathcal{O}(1)$.

Réponse 6. L'opération de la ligne `op 1` a pour complexité $O(n)$, ainsi que la totalité de `boucle 1` et `boucle 3`. La `boucle 2` effectue $n!$ itérations, une par permutation de $[\![0, n-1]\!]$, dont chacune a pour complexité $O(n)$. Donc la complexité globale de notre fonction est $O(n \times n!)$, soit

$$O((n+1)!)$$

Remarque : pour éviter d'effectuer la `boucle 3` plusieurs fois, on peut ne garder en mémoire que le poids du meilleur cycle, et faire un deuxième parcours jusqu'à retrouver la permutation qui atteint ce poids, cela évite aussi de créer 2 tableaux.

Partie II. Algorithme de Held-Karp (OCaml) – 8 pts

L'approche naïve précédente effectue beaucoup de calculs superflus. En effet, on peut remarquer que si $(0, s_1, s_2, \ldots, s_k)$, $k < n-1$, est le début d'un cycle hamiltonien de poids minimal, alors la manière de le compléter ne dépend pas de l'ordre des sommets $s_1, s_2, \ldots, s_{k-1}$ mais uniquement de s_k et de $S \setminus \{0, s_1, \ldots, s_k\}$.
L'algorithme de Held-Karp est un algorithme de programmation dynamique, qui utilise une table de hachage pour mémoïser les résultats.
On choisit de représenter un ensemble de sommets qui a déjà été ajouté au cycle par un tableau de booléens `vus`, valant `true` si le sommet appartient à l'ensemble, et `false` sinon.
Pour pouvoir stocker les résultats déjà calculés dans la table de hachage, il est nécéssaire de transformer les valeurs en des objets non mutables, comme par exemple des entiers ou des tuples d'objets non mutables.

☐ 7 – Écrire une fonction `clef : int -> bool array -> int * int` qui prend en argument un entier s_k et un tableau de booléens représentant un sous-ensemble $S' \subset S$, $(S = [\![0, n-1]\!])$ et renvoie un couple d'entiers représentant de manière unique s_k et S' et pouvant servir de clef dans une table de hachage.
On pourra assimiler le tableau de booléens à la décomposition binaire d'un entier compris entre 0 et 2^{n-1}.

Réponse 7. En utilisant `Array.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a` :

Calcule une valeur immuable représentant un entier et un tableau de booléens

```
let clef (sk : int) (tab : bool array) : int * int =
  (sk, Array.fold_left (fun x b -> 2*x + if b then 1 else 0) 0 tab);;
```

On choisit en OCaml de représenter un graphe par matrice d'adjacence. Ainsi, le graphe G_0 sera 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|] |];;
```

On rappelle que les tables de hachages peuvent être manipulées en OCaml avec les fonctions suivantes :
- `Hashtbl.create : int -> ('a, 'b) Hashtbl.t`
  prend en argument un entier m et crée une table vide occupant un espace mémoire de taille proportionnelle à m (on pourra choisir m = 1 en pratique) ;
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit`
  prend en argument une table, une clé et une valeur et rajoute une association ;
- `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 argument une table et une clé et renvoie la valeur associée.

On suppose créées en variables globales :
- une matrice d'entiers correspondants à un graphe $G = (S, A, w)$;
- une table de hachage par la commande `let tabh = Hashtbl.create 1;;`.

☐ 8 – Écrire une fonction

`completer_cycle : int array -> bool array -> int -> unit`

qui prend en argument un tableau d'entiers `c`, un tableau de booléens `vus` et un entier k tels que :
- `c` et `vus` sont deux tableaux de même taille $n = |S| > k$;
- les éléments `c.(i)` tels que $i \leq k$ sont tous distincts et compris entre 0 et $n-1$, et `c.(0) = 0` ;
- les éléments s tels que `vus.(s)` vaut `true` sont exactement `c.(0)`,..., `c.(k)`

La fonction doit modifier `c` en un cycle hamiltonien de G sans modifier les éléments d'indices inférieurs ou égaux à k, en minimisant le poids du cycle parmi ceux qui commencent par `c.(0)`,..., `c.(k)`.

Réponse 8. On se donne une fonction qui calcule le poids d'un cycle :

Poids d'un cycle

```
let poids (c : int array) =
  let n = Array.length c in
  let rec boucle acc i =
    if i = n-1 then
      acc
    else
      boucle (acc + g0.(c.(i)).(c.(i+1))) (i+1)
  in boucle (g0.(c.(0)).(c.(n-1))) 0;;
```

On regarde en priorité si le résultat est déjà présent dans la table de hachage, sinon on le calcule et on le stocke pour plus tard.

Programmation dynamique et récursive

```
let rec completer_cycle (c : int array) (vus : bool array) (k : int) : unit =
  let n = Array.length c in
  if k = n-1 then
    ()
  else
    try let best = Hashtbl.find tabh (clef c.(k) vus) in
        for i = k+1 to n-1 do
          c.(i) <- best.(i)
        done
    with Not_found ->
      let working_copy = Array.copy c in
      let current_min = ref max_int in
      for i = 0 to n-1 do
        if not vus.(i) then
          begin
            working_copy.(k+1) <- i; vus.(i) <- true;
            completer_cycle working_copy vus (k+1); vus.(i) <- false;
            let curr_w = poids working_copy in
            if curr_w < !current_min then
              begin
                current_min := curr_w;
                Hashtbl.replace tabh (clef c.(k) vus) (Array.copy working_copy)
              end
          end
      done;
      completer_cycle c vus k;;
```

☐ 9 – En déduire une fonction `PVC_dynamique : unit -> int array` qui renvoie un tableau correspondant à un cycle hamiltonien de poids minimal du graphe G, calculé selon l'algorithme de Held-Karp.

Réponse 9. Il s'agit d'un simple appel à la fonction précédente, après avoir créer et initialiser les tableaux :

```
let pvc_dynamique () : int array =
  let n = Array.length g0 in
  let vus = Array.make n false in
  let c = Array.init n (fun x -> x) in
  vus.(0) <- true; completer_cycle c vus 0;
  c;;
```

☐ 10 – Déterminer la complexité de `PVC_dynamique` en fonction de $n = |S|$.

Réponse 10. La boucle de taille n, comportant un appel de fonction de complexité $O(n)$ est effectuée au plus une fois par couple $(s_k, \texttt{vus})$, dont il en existe $n2^n$ différents. Donc la complexité de notre algorithme est au plus

$$O(n^3 2^n).$$

Partie III. Heuristique du plus proche voisin (C) – 8 pts

Pour éviter la recherche exhaustive, on propose un algorithme glouton qui ajoute les sommets un par un en choisissant toujours celui qui coûte le moins cher, c'est-à-dire le plus proche voisin.
Pour l'implémenter, on utilise un tableau `c` correspondant au cycle en cours de construction et un tableau `vus` de booléens, permettant de savoir quels sommets ont déjà été vus et rajoutés au cycle.

☐ 11 – Écrire une fonction

```
int plus_proche(int *G, bool *vus, int n, int s)
```

qui prend en argument un graphe $G = (S, A, w)$, un tableau `vus` de booléens, un entier $n = |S|$ et un sommet $s \in S$ et renvoie un sommet t vérifiant :

— `vus[t]` vaut `false` ;
— $w(s,t)$ est minimal parmi les sommets t non vus. Par exemple, si `vus` est défini par

```
bool vus[5] = {true, false, true, true, false};
```

alors l'appel à `plus_proche(G0, vus, 5, 2)` renverra `1` car les seuls sommets non vus sont 1 et 4, et $w(2,1) = 6 < w(2,4) = 10$.

Réponse 11.

Plus proche voisin — C

```c
int plus_proche(const int *G, bool *vus, int n, int s) {
        int min, curr, v, i;
        i = 0;

        while ((i < n) && vus[i]) {
                i = i + 1;
        }
        if (i == n) {
                printf("Tous les sommets sont vus\n");
                return -1;
        }
        v = i; min = w(G, n, s, i);
        while (i < n) {
                curr = w(G, n, s, i);
                if (!vus[i] && curr < min && s != i) {
                        v = i; min = curr;
                }
                i = i + 1;
        }
        return v;
}
```

Remarque : Pour l'utilisation faite de cette fonction, on a toujours `vus[s]` qui est vérifiée donc le test `s != i` est inutile si on se restreint au problème visé.

☐ 12 – En déduire une fonction

```
int *PVC_glouton(int *G, int n)
```

qui prend en argument un graphe $G = (S, A, w)$ et un entier $n = |S|$ et renvoie un pointeur vers un tableau contenant un cycle hamiltonien de G construit selon l'heuristique du plus proche voisin. On commencera par le sommet 0 comme sommet initial.

Réponse 12.

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

Remarque : On oublie pas de libérer la mémoire utilisée par `vus` pour éviter les fuites.

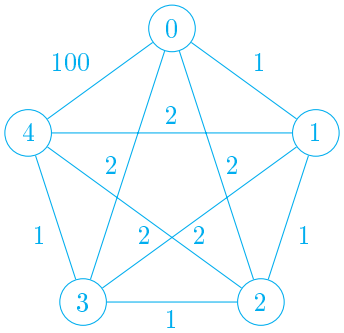
☐ 13 – Déterminer la complexité temporelle de `PVC_glouton` en fonction de $n = |S|$.

Réponse 13. Chaque appel à `plus_proche` se termine en $O(n)$, il y a $n-1$ tels appels donc le complexité de notre fonction est

$$O(n^2).$$

☐ 14 – Déterminer et représenter graphiquement un graphe pondéré complet d'ordre 5 tel que l'heuristique du plus proche voisin ne renvoie jamais de cycle hamiltonien de poids minimal en partant du sommet 0, quelles que soient les manières de traiter les cas d'égalité.

Réponse 14.



L'algorithme glouton trouve le cycle de poids 104 alors qu'il en existe un de poids 10 en prenant les arêtes du centre (l'optimal étant 7).

Partie IV. Difficulté du problème de seuil – 12 pts

Dans cette partie, on se propose de montrer que le problème de seuil associé au problème du voyageur de commerce est NP-complet. Pour se faire, on va montrer une suite de réductions depuis le problème 3-SAT dont on sait qu'il est NP-complet vers le problème de seuil associé au problème du voyageur de commerce, en passant par 3 intermédiaires : l'existences de chemins ou de cycles hamiltoniens dans des graphes orientés ou non-orientés.

Différents problèmes de décision

Problème du voyageur de commerce (TSP) - seuil

Instance : $G = (S, A, w)$ un graphe pondéré non orienté, avec $|S| = n$, et un seuil k.
Solution : V si il existe un cycle hamiltonien $\mathcal{C} = (s_0, s_1, \ldots, s_{n-1}, s_n = s_0)$ tel que :

$$w(\mathcal{C}) \leq k$$

F sinon.

uHAM-CYCLE

Instance : $G = (S, A)$ un graphe non orienté, avec $|S| = n$.
Solution : V si il existe un cycle hamiltonien $\mathcal{C} = (s_0, s_1, \ldots, s_{n-1}, s_n = s_0)$, F sinon.

uHAM-PATH

Instance : $G = (S, A)$ un graphe non orienté, avec $|S| = n$, s et t deux sommets de S.
Solution : V si il existe un chemin hamiltonien $\mathcal{C} = (s = s_0, s_1, \ldots, s_{n-1} = t)$, F sinon.

dHAM-PATH

Instance : $G = (S, A)$ un graphe orienté, avec $|S| = n$, s et t deux sommets de S.
Solution : V si il existe un chemin hamiltonien $\mathcal{C} = (s = s_0, s_1, \ldots, s_{n-1} = t)$, F sinon.

3-SAT

Instance : $\varphi = \wedge_{j=1}^{m} C_j$ avec $C_j = \ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}$ où $\ell_{j,k} \in \cup_{i=1}^{n} \{x_i, \bar{x}_i\}$, une formule booléenne de m clauses sur n variables dont chaque clause contient 3 littéraux.
Solution : V si il existe une valuation des x_i qui satisfait φ, F sinon.

☐ 15 – Donner une réduction polynomiale de uHAM-CYCLE à la version seuil du problème du voyageur du commerce : à partir d'un graphe non orienté $G = (S, A)$, construire un graphe pondéré non orienté $G' = (S', A', w)$ et un seuil k tel que G' admet un cycle hamiltonien de poids inférieur à k si et seulement si G admet un cycle hamiltonien.

Réponse 15. On pose w la fonction constante égale à 1, $G' = (S, A, w)$, et $k = |S|$.
La construction de G' se fait en temps polynomial en la taille de G.
Tout cycle hamiltonien dans G' contient $|S|$ arêtes, et est donc de poids $|S|$, donc il en existe un si et seulement s'il en existe un de poids au moins $|S|$.
$G \mapsto G'$ est donc une réduction polynomiale de uHAM-CYCLE à la version seuil du problème du voyageur du commerce.

☐ 16 – Donner une réduction polynomiale de uHAM-PATH à uHAM-CYCLE.
On pourra ajouter un sommet pour clore le cycle.

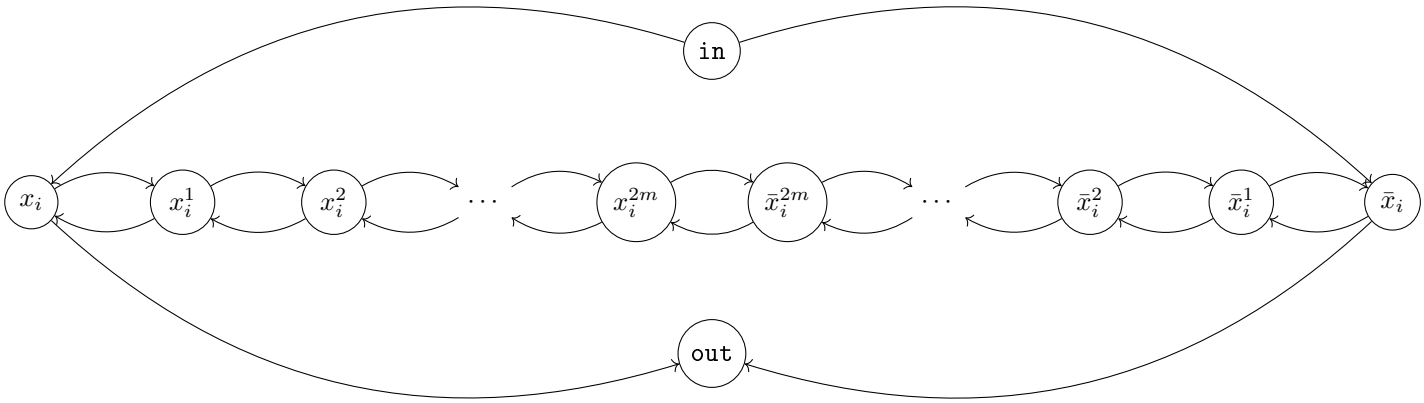
Réponse 16. Soit $G = (S, A)$ un graphe non orienté, s et t deux sommets de S. On pose $G' = (S', A')$ avec $S' = S \cup \{x\}$ et $A' = A \cup \{\{s, x\}, \{x, t\}\}$.
La construction de G' se fait en temps polynomial en la taille de G.

S'il existe un chemin hamiltonien $(s = s_0, s_1 \ldots, s_{n-1} = t)$ dans G, alors $(x, s, s_1, \ldots, t, x)$ est un cycle hamiltonien dans G'.
Réciproquement, s'il existe un cycle hamiltonien $(x, s_0, \ldots, s_{n-1}, x)$ dans G', alors nécessairement on a $s_0 = s$ et $s_{n-1} = t$ (quitte à réordonnre le cycle, comme G est non-orienté). Donc $(s_0, \ldots, s_{n-1})$ est un chemin hamiltonien de s à t dans G.
$(G, s, t) \mapsto G'$ est une réduction polynomiale de uHAM-PATH à uHAM-CYCLE.

☐ 17 – Donner une réduction polynomiale de dHAM-PATH à uHAM-PATH.
On pourra par exemple découper chaque sommet en 3.

Réponse 17. Soit $G = (S, A)$ un graphe non orienté, s et t deux sommets de S. On pose $S' = \cup_{x \in S}\{x_i, x, x_o\}$, et $A' = \cup_{(x,y)\in A}\{\{x_o, y_i\}\} \cup_{x \in S} \{\{x_i, x\}, \{x, x_o\}\}$, $G' = (S', A')$, $s' = s_i$ et $t = t_o$.
Cette construction se fait en temps polynomial en la taille de G.
Si $(s, ldots, t)$ est un chemin hamiltonien de s à t dans G, alors $(s_i, s, s_o, \ldots, t_i, t, t_o)$ est un chemin hamiltonien de s_i à t_o dans G' : en effet, si v suit u dans G, alors on a l'arête $(u, v) \in A$ et $\{u_o, v_i\} \in A'$.
Réciproquement, si $(s_i = s_0, s_1, \ldots, s_{3n-1} = t_o)$ est un chemin hamiltonien de s_i à t_o dans G', alors le sommet suivant s_i ne peut être que s : si $s \neq t$, dans le chemin $(s_i, x, \ldots, s, \ldots, t_o)$, s doit avoir deux voisins distincts de s_i, ce qui n'existe pas. De plus, on montre par récurrence sur k que les sommets $s_{3k}, s_{3k+1}, s_{3k+2}$ sont de la forme x_i, x, x_o pour un même $x \in S$: si $k = 0$ on a montré que $s_1 = s$, et le seul autre voisin de s est s_o. Si s_{3k+2} est de la forme x_o et que $s_{3k+1} = x$, alors les seuls autres voisins de x_o sont des y_i, pour $(x, y) \in A$, donc s_{3k+3} est de la forme y_i. De même que pour s, le sommet y ne peut alors être visité qu'à cette étape (s_{3k+4}) pour ne pas aboutir à une contradiction plus tard, et la seule possibilité pour s_{3k+5} est y_o.
$(G, s, t) \mapsto (G', s_i, t_o)$ est une réduction polynomiale de dHAM-PATH à uHAM-PATH.

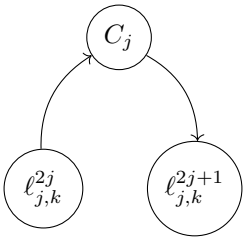
Pour chaque variable x_i, on considère le gadget G_i suivant, que l'on va ensuite mettre bout à bout :



☐ 18 – Combien y-a-t'il de chemins hamiltoniens de `in` à `out` dans ce graphe ?

Réponse 18. Il y a deux chemins hamiltoniens de `in` à `out` : celui commençant par x_i et celui commençant par $\bar{x}_i$

Pour chaque littéral $\ell_{j,k}$ dans la clause C_j, on ajoute aussi la construction suivante :



☐ 19 – Montrer qu'un chemin hamiltonien dans G_i peut alors être modifié pour passer par tous les sommets C_j dans lesquels x_i figure mais aucun sommet C_j où $\bar{x}_i$ ne figure, ou vice-versa par tous les sommets C_j dans lesquels $\bar{x}_i$ figure sans passer par les sommets C_j dans lesquels x_i figure.

Réponse 19. Dans le chemin $(\texttt{in}, x_i^1, x_i^2, \dots, x_i^{2m}, \bar{x}_i^{2m}, \dots, \texttt{out})$, toutes les arêtes x_i^{2j}, x_i^{2j+1} peuvent être remplacés par $x_i^{2j}, C_j, x_i^{2j+1}$, pour tout j tels que x_i apparaît dans C_j.
De même pour l'autre chemin, avec les j tels que $\bar{x}_i$ apparaît dans C_j.

☐ 20 – Donner une réduction polynomiale de 3-SAT à dHAM-PATH, et conclure quand à la difficulté du problème de seuil associé au problème du voyageur du commerce.

Réponse 20. À partir d'une formule φ en 3-CNF, le graphe composé de tous les G_i pour chaque variable x_i, avec les arêtes de $\texttt{out}_i$ vers $\texttt{in}_{i+1}$ et de tous les sommets C_j, avec leurs arêtes depuis et vers les sommets des littéraux y apparaîssant se construit en temps polynomial.
Si φ est satisfiable par un modèle v, le chemin hamiltonien passant par les chemins des G_i commençant par x_i^1 si $v(x_i) = V$ et $\bar{x}_i^1$ si $v(x_i) = F$ peut être modifié pour passer par tous les C_j car il existe un littéral dont la valeur est V pour chaque clause.
D'autre part, s'il existe un chemin hamiltonien, on peut retrouver un modèle v de φ en posant $v(x_i) = V$ si le chemin hamiltonien dans G_i commence par le sommet x_i^1 et F sinon. En effet, pour chaque clause C_j, un des chemins hamiltonien dans les G_i a été modifié pour y passer, donc le littéral correspondant à sa valeur de vérité à V et la clause est satisfaite.

On a : 3-SAT $\leq_P$ dHAM-PATH $\leq_P$ uHAM-PATH $\leq_P$ uHAM-CYCLE $\leq_P$ TSP - seuil
Or on sait d'après le théorème de Cook-Levin que 3-SAT est NP-Complet, donc TSP - seuil est NP difficile.

De plus, on a codé une fonction qui calcule le poids d'un cycle en temps polynomial, donc un cycle de poids supérieur au seuil constitue un certificat de taille polynomial en la taille du graphe, et le problème TSP - seuil est dans NP.
C'est donc un problème NP-Complet.

Partie V. Difficulté d'approximer le problème d'optimisation – 4 pts

On peut aussi montrer qu'il est difficile d'approximer la solution avec un facteur constant d'approximation garanti.

On suppose dans cette partie qu'il existe un algorithme de complexité polynomiale qui, en prenant un graphe $G = (S, A, w)$ pondéré, non orienté et complet, renvoie un cycle hamiltonien $\mathcal{C}$ de G tel que si $\mathcal{C}^*$ est un cycle hamiltonien de poids minimal, alors $w(\mathcal{C}) \leq \alpha w(\mathcal{C}^*)$, avec $\alpha \geq 1$ est une constante fixée.
Pour $G = (S, A)$ un graphe non orienté, on considère le graphe pondéré non orienté complet $K_G = (S, S^2, w)$ défini par :

- pour $a \in A$, $w(a) = 1$;
- pour $a \notin A$, $w(a) = \alpha|S|$.

☐ 21 – 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 $\alpha|S|$.

Réponse 21. Si G possède un cycle hamiltonien, ce cycle a pour poids $|S|$ dans K_G. Si G ne possède pas de cycle hamiltonien, tout cycle hamiltonien dans K_G possède au moins une arête non dans A et une autre arête donc est de poids supérieur à $\alpha|S|$.

☐ 22 – En déduire qu'on peut résoudre uHAM-PATH en temps polynomial, et conclure sur la difficulté d'approximer la solution du problème du voyageur de commerce.

Réponse 22. On peut construire le graphe K_g en temps polynomial, et on applique l'algorithme qu'on suppose exister sur l'entrée $K_g, \alpha|S|$. Si le résultat est un cycle hamiltonien de poids inférieur ou égal à $\alpha|S|$, on a la garanti que G possède un cycle hamiltonien. Sinon, on sait que K_G ne possède pas de cycle hamiltonien de poids $|S|$. Or tout cycle hamiltonient de poids $|S|$ dans K_G est un cycle hamiltonien dans G.
Comme uHAM-PATH est NP-Difficile (voir section précédente), il est impossible d'approximer la solution du problème du voyageur de commerce dans le cas général, à moins que $P = NP$.

Partie VI. Approximation dans le cas métrique en utilisant un arbre couvrant minimal (OCaml) – 6 pts

Le problème du voyageur de commerce est difficile à approximer dans le cas général, mais sous certaines hypothèse, on peut trouver des algorithmes efficaces qui s'approchent du résultat. On s'intéresse ici à un graphe pondéré complet $G = (S, A, w)$ dont la fonction de poids w vérifie l'inégalité triangulaire :

$$\forall s, t, u \in S^3, w(s, u) \leq w(s, t) + w(t, u).$$

C'est le cas par exemple si on prend pour sommet des points du plan et pour poids la distance euclidienne entre ces points.
Pour la suite, on note $\mathcal{C}^*$ un cycle hamiltonien de poids minimal de G et T un arbre couvrant minimal. On étend de manière naturelle la fonction w aux sous-graphes et aux ensembles d'arêtes de G comme la somme des poids des arêtes qui les composent.

☐ 23 – Montrer que $w(T) \leq w(\mathcal{C}^*)$

Réponse 23. Si on enlève une arête quelconque à $\mathcal{C}^*$, on obtient un arbre couvrant, qui a donc un poids supérieur ou égal à $w(T)$.

☐ 24 – En considérant un parcours en profondeur préfixe de T, en déduire l'existence d'un cycle hamiltonien $\mathcal{C}_T$, calculable en temps polynomial en la taille de G, tel que $w(\mathcal{C}_T) \leq 2w(\mathcal{C}^*)$

Réponse 24. On le montre par induction sur la structure d'arbre de T : le parcours préfixe des nœuds de T suivi du retour à la racine donne un cycle de poids inférieur au double du poids de T.
Si T est vide ou réduit à un nœud, la propriété est triviale.
Soit r la racine de T et $T_1, \ldots, T_k$ ses sous-arbres. Chacun de ces sous-arbres T_i possède un cycle $t_{i,0}, \ldots, t_{i,k_i}, t_{i,0}$ de poids inférieur au double du poids de T_i.

Le cycle $r, t_{0,0}, \dots, t_{0,k_0}, t_{1,0}, \dots, r$ possède un poids plus petit que le même cycle où on rajoute les arêtes $t_{i,k_i}, t_{i,0}, r, t_{i+1,0}$.
Ce nouveau cycle contient les cycles de chaque fils T_i, et deux fois chaque arête $r, t_{i,0}$, donc son poids est inférieur au double du poids de T, donc au double du poids de $\mathcal{C}^*$.

☐ 25 – Écrire une fonction `PVC_approx : int array array -> int array` qui prend en argument un graphe $G = (S, A, w)$ représenté par une matrice d'adjacence et renvoie un tableau correspondant à un cycle hamiltonien de G dont le poids est au plus le double du poids minimal d'un cycle hamiltonien de G.

On pourra suppose l'existence d'une fonction `kruskal` qui implémente l'algorithme de Kruskal et dont vous préciserez le type.

Réponse 25. Pour faciliter notre tâche au dépens de celui qui implémente l'algorithme de Kruskal, nous décidons de représenter l'arbre de manière inductive pour faciliter le parcours préfixe :

```
type 'a arb = Node of 'a * 'a arb list

let kruskal : int array array -> int arb = fun x -> assert false

let pvc_approx g =
  let a = kruskal g in
  let n = Array.length g in
  let cycle = Array.make n 0 in
  let rec prefix i (Node (x, l)) =
    cycle.(i) <- x; List.fold_left prefix i l in
  assert (prefix 0 a = n); cycle
```

Fin de l'épreuve