# DS n° 5 — Partie I — corrigé

*En bleu figurent les commentaires du rapport de jury de l'épreuve.*

## 1 Complexité de Kolmogoroff

### 1.1 Un premier exemple

☐ **1** – On a $K(y_0) \leq 10^{10} + 1$. En effet, en choisissant $x = 1\underbrace{000\ldots}_{10^{10}} = x_0$, on a :

- `eval x` $= y_0$
- $|x| = 10^{10} + 1$

*Oubli régulier du chiffre 1. Plusieurs candidats n'ont pas pris le temps de la réflexion et ont répondu $10^{10^{10}}$ ou n'ont pas compris la définition de la complexité de Kolmogoroff.*

☐ **2** – Soit $n \in \mathbb{N}$, soit $n' = \lfloor \frac{n}{2} \rfloor$. On a donc :

- si $n$ est pair, alors $n = 2n'$, et donc $10^n = \left(10^{n'}\right)^2$ ;
- si $n$ est impair, alors $n = 2n' = 1$, et donc $10^n = 10 \cdot \left(10^{n'}\right)^2$.

On en déduit directement le code OCaml suivant :

```
let exp10 n =
  (* Calcul 10^n *)
  let rec exp10 n =
    if n = 0 then 1
    else
      let y = exp10 (n / 2) in    (* 10^(n') *)
      if n mod 2 = 0 then y * y   (* (10^(n'))^2 *)
      else 10 * y * y             (* 10 * (10^(n'))^2 *)
  in exp10 n
in string_of_int (exp10 (exp10 10))
```

Remarquons que la fonction auxiliaire est ici inutile (mais on aurait pu en profiter pour réaliser une fonction auxilaire récursive terminale). Elle est rendue nécessaire par le fait que la fonction `exp10` proposée ne comporte pas le mot clé `rec`. Le programme précédent est une chaîne de caractères de longueur 286 qui est une description de $y_0$. On en déduit donc que $K(y_0) \leq 286$.

*De très nombreux candidats écrivent l'algorithme récursif de l'exponentiation rapide en rappelant deux fois le calcul sur $10^{n'}$ plutôt que de stocker le résultat. Certains oublient le cas d'initialisation, où n'en font qu'un seul en 0 alors que leur écriture nécessite un cas d'initialisation aussi en 1.*

☐ **3** –

- débordement d'entiers ;
- nombre d'appels récursifs imbriqués de l'ordre de $\log_2\left(10^{10^{10}}\right) \geq \log_2(2^{3\times 10^{10}}) = 3 \times 10^{10}$.

Chaque appel récursif induit au minimum un empilement de 1 octet sur la pile, et donc finalement on a un coût mémoire minoré par 30Go, soit certainement un "Stack Overflow".

*Il était attendu du candidat qu'il évoque les problèmes de dépassement d'entier et, éventuellement en fonction de leur programme, de dépassement de la capacité de la pile.*

### 1.2 Quelques propriétés

☐ **4** – Considérons l'encodage suivant :

- 0 désigne la chaîne de caractères vide ;
- $n \in \mathbb{N}^\star$ désigne une chaîne de caractère non vide :
  - de premier symbole le caractère de code ASCII $(n-1) \mod 256$ ;
  - dont l'encodage (récursif) du reste de la chaîne a pour valeur $\lfloor \frac{n-1}{256} \rfloor$.

Ce qui se traduit par la fonction $\varphi : \mathbb{N} \to \Sigma^\star$ définie par :

$$\forall n \in \mathbb{N},\ \varphi(n) = \begin{cases} \varepsilon & \text{si } n = 0 \\ ((n-1) \mod 256) \cdot \varphi(\lfloor \frac{n-1}{256} \rfloor) & \text{sinon} \end{cases}$$

On obtient alors le code OCaml suivant :

```
let rec phi n =
  if n = 0 then ""
  else (String.make 1 (Char.chr ((n-1) mod 256))) ^ (phi ((n-1) / 256))
```

*De nombreux candidats ont proposé une bijection basée sur l'écriture en base 256. Les détails précis de cette bijection n'ont que rarement été trouvés, la justification non plus. Affirmer que cela découle de l'écriture en base 256 est une affirmation hâtive et erronée.*

☐ **5** – Il s'agit simplement d'itérer sur les entiers en partant de 0 :

```
let psi m =
  let rec recherche_premier n =
    if kolmogoroff (phi n) >= m then n
    else recherche_premier (n + 1)
  in recherche_premier 0
```

*On peut noter que plusieurs candidats n'ont pas bien lu l'énoncé et ont compris $\psi(m) = \{K(\varphi(n)) \geq m, n \in \mathbb{N}\}$.*

☐ **6** – Soit $m \in \mathbb{N}$ :

- par définition de $p = \psi(m)$, on a $K(\varphi(p)) \geq m$, soit $K(\varphi(\psi(m))) \geq m$ ;
- de plus, soit $c_m$ la chaîne de caractères représentant l'entier $m$. Soit $\ell$ la chaîne de caractères suivante :

```
let phi n =
   ...
in
let kolmogoroff s =
   ...
in
let psi m =
   ...
in
```

où les `...` sont le code des définitions de chaque fonction.

Soit la chaîne $x =$`"("` $\cdot\ \ell\ \cdot$ `" phi (psi"` $\cdot\ c_m\ \cdot$ `"))"`. On a alors :

- `(eval x)` $= \varphi(\psi(m))$ ;
- $|x| = C + |c_m|$, où $C$ ne dépend pas de l'entier $m$, et $|c_m| = \lceil \log_{10}(m+1) \rceil$.

Ainsi, $K(\varphi(\psi(m)) \leq C + |c_m| = \lceil \log_{10}(m+1) \rceil$. Ceci étant vrai pour tout $m \in \mathbb{N}$, on en déduit bien que $K(\varphi(\psi(m))) = \mathcal{O}(\log(m))$.

On a donc à une contradiction : une telle fonction `kolmogoroff` n'existe pas. Autrement dit, la fonction $K$ n'est pas calculable.

*La première et la dernière partie de la question ont été en général bien traitées. Seuls les meilleures compositions ont traité la domination asymptotique logarithmique qui découlait du calcul du nombre de chiffres en numération de position.*

☐ **7** – Un compilateur.

*Trop de candidats ont inventé leur propre terme pour cette question. Globalement les candidats ont répondu compilateur, transpileur ou interpréteur. L'expression compilateur était celle attendue.*

☐ **8** – On fait l'hypothèse supplémentaire, omise par le sujet, que le décompresseur est surjectif. On peut aussi, si le décompresseur n'est pas surjectif, considérer que s'il n'existe pas de description d'une chaîne $y$ par rapport à $\mathcal{D}$ alors $K_{\mathcal{D}}(y) = +\infty$ et dans ce cas l'inégalité est triviale.

Soit $\mathcal{D}$ un décompresseur, soit $c$ la chaîne de caractères `"let d = ..."` représentant le code OCaml de ce décompresseur. Soit alors une chaîne de caractères $y$, et soit l'ensemble $Z = \{|z| \mid z \in \Sigma^\star \text{ et } \mathcal{D}(z) = y\}$. Par surjectivité, $Z \neq \emptyset$ et $Z \subseteq \mathbb{N}$, soit donc $z_0$ tel que $|z_0| = \min(Z)$. Ainsi, $z_0$ décrit $y$ par $\mathcal{D}$ et $K_{\mathcal{D}}(y) = |z_0|$. Considérons finalement la chaîne de caractères $x =$ `"("` $\cdot\ c\ \cdot$ `" in d "` $\cdot\ z_0\ \cdot$ `")"`. On a :

- `(eval x)` $= \mathcal{D}(z_0) = y$ ;
- $|x| = |z_0| + |c| + 7 = K_{\mathcal{D}}(y) + Cte$

## 2 Estimation de la complexité grâce au compresseur de Huffman

☐ **9** – `int`

☐ **10** – $10^{10^{10}}$

☐ **11** –

- utilisation de noms de variables non descriptifs ;
- manque d'indentation.

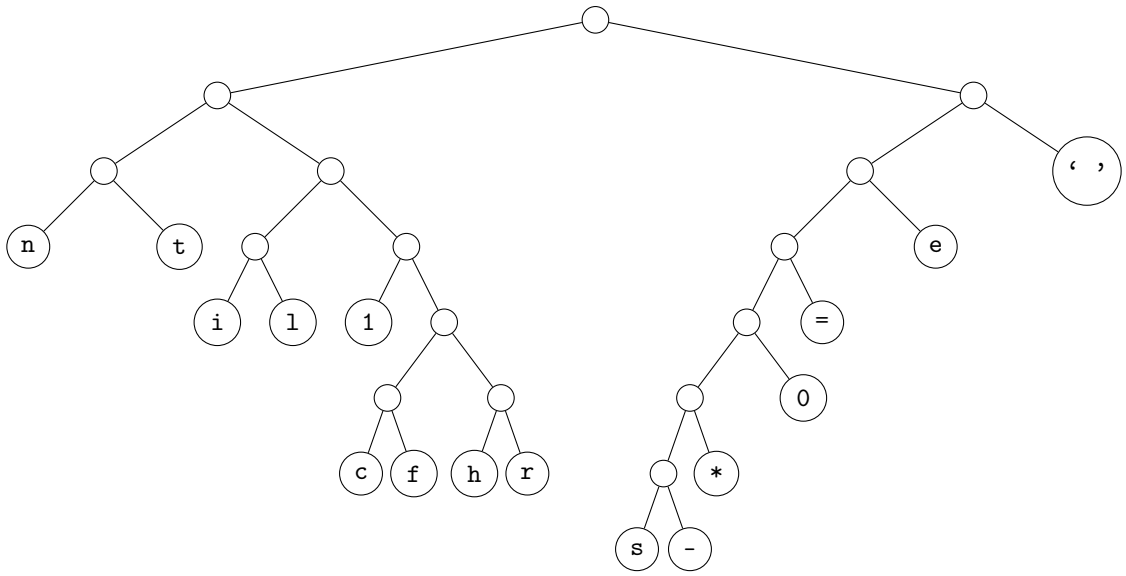
*La majorité des candidats ont évoqué les indentations et retours à la ligne. Beaucoup d'entre eux ont aussi évoqué le nom des variables et fonctions. Il est à noter que, contrairement à ce qu'affirme certaines compositions, le caractère récursif d'une fonction n'est pas censé rendre la lecture d'un code «impénétrable», en tout cas pas pour un candidat de MPI.*

☐ **12** – On itère sur les caractères : s'il est déjà présent, on ajoute une nouvelle association avec la valeur précédente +1 ; s'il n'était pas déjà présent on ajoute une nouvelle association avec la valeur 1.

```
let count s =
  let h = Hashtbl.create 256 in
  for i = 0 to String.length s - 1 do
    match Hashtbl.find_opt h s.[i] with
    | None -> Hashtbl.add h s.[i] 1
    | Some y -> Hashtbl.add h s.[i] (y + 1)
  done;
  h
```

*De nombreux candidats ont fait un parcours de chaîne pour le décompte de chaque caractère de la chaîne, voire pour les 256 caractères ASCII, voire plusieurs fois par caractère (une fois pour chaque caractère de la chaîne). Il est attendu que les candidats soient capables de faire ce décompte en un seul parcours, et, par ailleurs, sans utiliser de tableau auxiliaire.*

☐ **13** – On obtient l'arbre suivant :



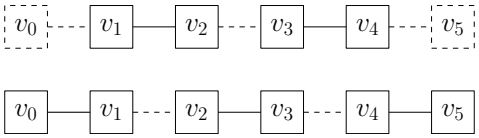

☐ **14** – Dans l'encodage de Huffman, la taille (en bits) utilisée pour représenter un caractère est sa profondeur dans l'arbre de Huffman. Ainsi, ici on a un coût de :

$$8\times3+8\times3+4\times4+4\times4+4\times4+1\times6+1\times6+1\times6+1\times6+1\times7+1\times7+1\times6+3\times5+4\times4+10\times3+19\times2=239$$

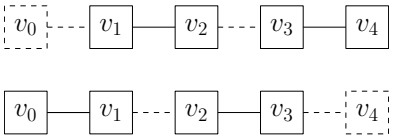
# Solutions

❑ 15

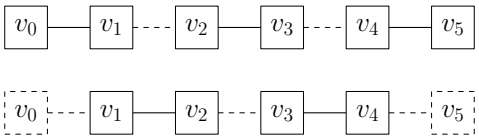
**a.** $v_0$ était libre et se retrouve apparié à $v_1$, $v_l$ était libre et se retrouve apparié à $v_{l-1}$. Tous les autres sommets du chemin ne font que changer de partenaire : $M \oplus P$ est un couplage, et $|M \oplus P| = 1 + |M|$.




**b.** $v_0$ était libre et se retrouve apparié à $v_1$, $v_l$ était apparié à $v_{l-1}$ et se retrouve libre, les autres sommets changent de partenaire : $M \oplus P$ est un couplage et $|M \oplus P| = |M|$.




**c.** $v_0$ et $v_l$ étaient couverts et deviennent libres, les autres sommets changent de partenaire : $M \oplus P$ est un couplage et $|M \oplus P| = |M| - 1$.




❑ 16 Considérons le graphe $H = (X \sqcup Y, M \oplus M')$. Chaque sommet est incident à au plus une arête de $M$ et au plus une arête de $M'$, donc de degré au plus 2. On en déduit que les composantes connexes de $H$ sont des sommets isolés, des cycles et des chemins. Les cycles sont nécessairement de longueur paire (les arêtes doivent alterner), et contiennent donc autant d'arêtes de $M$ que de $M'$. Comme $|M'| > |M|$, il y a donc au moins une composante connexes chemin qui contient une arête de plus de $M'$ que de $M$ : cette composante est un chemin augmentant pour $M$, inclus dans $M \oplus M'$.

❑ 17 À savoir faire !

```c
double **make_double_matrix(int n, double x) {
    double **M = malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
        M[i] = malloc(n * sizeof(double));
        for (int j = 0; j < n; j++) {
            M[i][j] = x;
        }
    }
    return M;
}
```

❏ 18 On a fait $n+1$ `malloc`, il doit y avoir $n+1$ `free`.

```c
void free_double_matrix(double **M, int n) {
    for (int i = 0; i < n; i++) {
        free(M[i]);
    }
    free(M);
}
```

❏ 19 C'est du cours, mais il faut prendre garde à gérer correctement `from` :

- au départ, les seuls chemins autorisés sont ceux réduits à une seule arête, donc `from[i][j]` vaut `i` si l'arête est présente, $-1$ sinon ;
- ensuite, si le plus court chemin trouvé de `i` à `j` passe par `k`, alors son dernier arc est celui du plus court chemin de `k` à `j`.

```c
int **floyd_warshall(double **M, int n) {
    int **from = make_int_matrix(n, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (M[i][j] < INFINITY) { from[i][j] = i; }
        }
    }
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                double with_k = M[i][k] + M[k][j];
                double without_k = M[i][j];
                if (with_k < without_k) {
                    M[i][j] = with_k;
                    from[i][j] = from[k][j];
                }
            }
        }
    }
    return from;
}
```

❏ 20 L'initialisation de `from` est en $O(n^2)$, et on a ensuite trois boucles imbriquées de longueur $n$, avec une boucle interne en $O(1)$ : la complexité totale est en $O(n^3)$.

❏ 21 On a :

$$\begin{aligned} c(M \oplus P) &= \sum_{e \in M \oplus P} c(e) \\ &= \sum_{e \in M \setminus P} c(e) + \sum_{e \in P \setminus M} c(e) \qquad\qquad M \oplus P = (M \setminus P) \sqcup (P \setminus M) \\ &= \sum_{e \in M} c(e) - \sum_{e \in M \cap P} c(e) + \sum_{e \in P \cap \overline{M}} c(e) \\ &= c(M) + c_M(P) \end{aligned}$$

❑ 22 On sait d'après 16 que $M \oplus M'$ contient au moins un chemin $P$ augmentant pour $M$. Ce chemin est « diminuant » pour $M'$ (alternant, avec les deux arêtes extrêmes dans $M'$). On pose alors $N = M' \oplus P$ : d'après 15, $N$ est un couplage de cardinal $|M'| - 1 = |M|$. De plus, on a bien $N \oplus P = M' \oplus P \oplus P = M'$.

❑ 23 Remarquons d'abord que $|M_0| = 0$ et qu'on a $|M_{k+1}| = |M_k| + 1$, donc $|M_k|$ est bien égal à $k$. Ensuite, l'invariant est évidemment vérifié pour $k = 0$, on suppose donc qu'il est vérifié pour un certain $k < n$. Soit $M'$ un couplage de cardinal $k+1$ : d'après la question 22 appliquée à $M'$ et $M_k$, il existe un couplage $N$ de cardinal $k$ et un chemin $P$ augmentant pour $M_k$ (et pour $N$) tels que $M' = N \oplus P$. On a donc $c(M') = c(N) + c_N(P) = c(N) + c_{M_k}(P)$. Or :

- $c(M_{k+1}) = c(M_k \oplus P') = c(M_k) + c_{M_k}(P')$ avec $P'$ un plus court chemin augmentant pour $M_k$ ;
- on a $c_{M_k}(P) \geqslant c_{M_k}(P')$ puisque $P$ est augmentant pour $M_k$, et $c(N) \geqslant c(M_k)$ d'après l'invariant au rang $k$ ;
- on en déduit $c(M_{k+1}) \leqslant c(N) + c_{M_k}(P) = c(M')$.

L'invariant est donc conservé ; en fin de boucle, on a $M_k$ de poids minimal parmi tous les couplages de cardinal $k$. De plus, $M_k$ est alors de cardinal maximum puisqu'il n'admet pas de chemin augmentant : il est donc optimal.

❑ 24 Par construction, les chemins augmentants pour $M$ sont exactement ceux de la forme $P = x_1, y_1, \ldots, x_k, y_k$ avec $P' = s \to x_1 \to y_1 \to \cdots \to x_k \to y_k \to t$ un chemin de $G_M$. Par définition de $\lambda$, on a alors $\lambda(P') = c_M(P)$ : un chemin augmentant de coût minimal pour $M$ est exactement un chemin de poids minimal de $s$ à $t$ dans $G_M$, privé de ses deux arcs extrêmes.

❑ 25 Un cycle de $G_{M_k}$ correspond à un cycle alternant $C$ de $M_k$. On a alors $M_k \oplus C$ un couplage de même cardinal que $M_k$, et de poids $c(M_k) + \lambda(C)$. On a montré que $M_k$ était de coût minimal parmi tous les couplages de cardinal $k$, donc nécessairement $\lambda(C) \leqslant 0$.

❑ 26 Attention aux indices. Le programme est proposé à la page suivante.

❑ 27 Après différence symétrique, tous les sommets du chemin sont couverts.

```c
void augment(int *matching, int *path, int len) {
    for (int i = 0; 2 * i < len; i++) {
        int x = path[2 * i];
        int y = path[2 * i + 1];
        matching[x] = y;
        matching[y] = x;
    }
}
```

❑ 28 Les 5 erreurs sont :

- La taille du graphe est $N + 2$ et non $N$ dans l'appel à l'algorithme de Floyd-Warshall : `floyd_warshall(G_orient, N + 2);;`
- Un chemin est un tableau d'entiers pas un entier donc il faut écrire : `int *path = NULL;` et non `int path = NULL;;`
- Il faut libérer la matrice `from`, donc ajouter juste avant la fin : `free_int_matrix(from, N + 2);;`
- La variable `i` n'est jamais incrémentée et la boucle `while` ne termine donc pas. Il faut ajouter en fin de boucle `i++`.
- Il manque une accolade ouvrante après `while (v != s)`.

Voici le programme corrigé :

```c
int *shortest_augmenting_path(bipartite_t *g, int *matching, int *len) {
    int N = g->n + g->p;
    double **G_orient = build_oriented(g, matching);
    int **from = floyd_warshall(G_orient, N + 2);
    int s = N;
    int t = N + 1;
    *len = 0;
    int *path = NULL;
    if (from[s][t] != -1) {
        path = malloc(N * sizeof(int));
        int i = 0;
        int v = from[s][t];
        while (v != s) {
            path[i] = v;
            v = from[s][v];
            i++;
        }
        *len = i - 1; // Nombre de sommets moins un
    }
    free_double_matrix(G_orient, N + 2);
    free_int_matrix(from, N + 2);
    return path;
}
```

```c
double **build_oriented(bipartite_t *g, int *matching) {
    int N = g->n + g->p;
    double **G_orient = make_double_matrix(N + 2, INFINITY);
    for (int i = 0; i < g->n; i++) {
        for (int j = g->n; j < N; j++) {
            if (matching[i] == j) {
                G_orient[j][i] = -g->adj[i][j];
            } else {
                // OK même si l'arête ij n'existe pas
                G_orient[i][j] = g->adj[i][j];
            }
        }
    }
    int s = N;
    int t = N + 1;
    for (int i = 0; i < g->n; i++) {
        if (matching[i] == -1) { G_orient[s][i] = 0.; }
    }
    for (int j = g->n; j < N; j++) {
        if (matching[j] == -1) { G_orient[j][t] = 0.; }
    }
    return G_orient;
}
```

❏ 29 Il ne reste plus grand chose à faire :

```c
int *compute_optimal_matching(bipartite_t *g) {
    int N = g->n + g->p;
    int *matching = malloc(N * sizeof(int));
    for (int i = 0; i < N; i++) { matching[i] = -1; }
    while (true) {
        int len;
        int *path = shortest_augmenting_path(g, matching, &len);
        if (len == 0) { return matching; }
        augment(matching, path, len);
        free(path);
    }
    return matching;
}
```

❏ 30 À chaque passage dans la boucle **while**, le coût dominant est celui de l'appel à Floyd-Warshall, en $O(|V|^3)$, et l'on ajoute une arête au couplage. Comme le couplage contient au plus $|V|/2$ arêtes à la fin, le nombre de passages est en $O(|V|)$ et la complexité totale en $O(|V|^4)$.

❏ 31 On considère $L(w_1) = 1$ et $L(w_2) = 3, 2$ pour un coût total de $4 + (2 + 8) = 14$.

❏ 32 Considérons une liste $L(w) = x_1, \ldots, x_k$, et notons $c(w) = \sum_{i=1}^{k} F(x_i)$. Pour $i = 1$ à $k$, on a $F(x_i) = \sum_{l=1}^{i} p(x_i, w)$, donc :

$$\begin{aligned} c(w) &= \sum_{i=1}^{k} \sum_{l=1}^{i} p(x_l, w) \\ &= \sum_{l=1}^{k} \sum_{i=l}^{k} p(x_l, w) \\ &= \sum_{l=1}^{k} (k - l + 1) p(x_l, w) \\ &= \sum_{l=1}^{k} g(x_l) p(x_l, w) \\ &= \sum_{t \in L(w)} g(t) p(t, w(t)) \end{aligned}$$

Comme les $L(w)$ forment une partition des tâches, on en déduit immédiatement $C = \sum_{t \in T} g(t) p(t, w(t))$.

❏ 33 On considère le graphe biparti pondéré $G = (T \sqcup I, E, \lambda)$, où :

- $T$ est l'ensemble des tâches du problème ;
- $W$ contient $n$ sommets $w_{j,1}, \ldots, w_{j,n}$ pour chaque machine $w_j$ du problème ;
- $E$ contient toutes les arêtes $t_i w_{j,k}$ avec $t_i \in T$ et $w_{i,k} \in W$ (le graphe est biparti complet) ;
- $\lambda(t_i, w_{j,k}) = k p(t_i, w_j)$.

Le sommet $w_{j,k}$ représente la $k$-ème position en partant de la fin dans la liste des tâches programmées sur $w_j$ ; l'arête $t_i w_{j,k}$ indique que la tâche $i$ est programmée à cette position sur cette machine.

- À un ordonnancement, on peut immédiatement associer un couplage T-parfait de G (qui couvre tous les sommets de T), en utilisant l'interprétation donnée ci-dessus.
- Le coût de ce couplage est exactement celui de l'ordonnancement d'après la question précédente.
- À un couplage M de G, on peut associer un ordonnancement à condition que :
  — le couplage couvre tous les sommets de T (on parle de couplage T-parfait) ;
  — les $w_{j,k}$ soient utilisés « dans l'ordre », c'est-à-dire que $w_{j,k}$ couvert et $k' < k$ implique $w_{j,k'}$ couvert (s'il y a une troisième tâche en partant de la fin sur une machine, il doit y avoir une deuxième et une première tâche en partant de la fin).
- Il existe clairement des couplages T-parfaits (par exemple $\{t_1w_{1,1}, t_2w_{1,2}, \dots, t_nw_{1,n}\}$), donc tout couplage optimal est T-parfait.
- De plus, si un couplage contient une arête $t_iw_{j,k}$ et laisse un sommet $w_{j,k'}$ avec $k' < k$ libre, alors on peut remplacer l'arête $t_iw_{j,k}$ par l'arête $t_iw_{j,k'}$ pour obtenir un nouveau couplage, donc le coût est diminué de $(k - k')p(t_i, w_j) > 0$.
- Ainsi, tout couplage optimal de G fournit un ordonnancement valide, et comme le coût d'un couplage est exactement celui de l'ordonnancement associé, trouver un ordonnancement optimal revient exactement à trouver un couplage optimal de G.

❑ 34 Le graphe contient $n + nm = O(nm)$ sommets et $n \times nm = O(n^2m)$ arêtes. Il peut clairement être construit en temps $O(n^2m)$. Si on applique ensuite l'algorithme 1, on obtient du $O((n^2m)^4) = O(n^8m^4)$