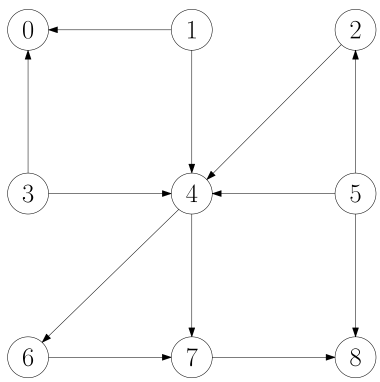
I Sujet d'oral CCINP

Dans cet exercice, tous les graphes sont orientés. On représente un graphe orienté $G = (S, A)$, avec $S = \{0, \dots, n-1\}$, en C par la structure suivante :

```c
typedef struct {
    int n; // nombre de sommets
    int degre[100]; // degre[s] est le degré sortant de s
    int voisins[100][10]; // voisins[s] contient les successeurs de s
} graphe;
```




```c
graphe g_exemple = {
      .n = 9,
      .degre = {0,2,1,2,2,3,1,1,0},
      .voisins = {
      /* 0 */ {-1}, // Degré 0 : ignoré
      /* 1 */ {0, 4},
      /* 2 */ {4},
      /* 3 */ {0, 4},
      /* 4 */ {6, 7},
      /* 5 */ {2, 4, 8},
      /* 6 */ {7},
      /* 7 */ {8},
      /* 8 */ {-1} // Degré 0 : ignoré
      }
};
```

Pour $s \in S$ on note $A(s)$ l'ensemble des sommets accessibles à partir de s. Pour $s \in S$, le maximum des degrés des sommets accessibles depuis s est noté $d^*(s)$. Par exemple, pour le graphe ci-dessus, $A(2) = \{2, 4, 6, 7, 8\}$ et $d^*(2) = 2$ car le degré sortant de 4 est $d^+(4) = 2$. Dans cet exercice, on cherche à calculer $d^*(s)$ pour chaque sommet $s \in S$.

On représente un sous-ensemble de sommets $S' \subseteq S$ par un tableau de booléens de taille n, contenant `true` à la case d'indice s' si $s' \in S'$ et `false` sinon.

1. Écrire une fonction `int degre_max(graphe* g, bool* partie)` qui calcule le degré maximum d'un sommet $s' \in S'$ dans un graphe $G = (S, A)$ pour une partie $S' \subseteq S$ représentée par `partie`, c'est-à-dire qui calcule $\max\{d^+(s') \mid s' \in S'\}$.

> Solution : On renvoie 0 si la partie est vide (l'énoncé ne spécifie rien).
>
> ```c
> int degre_max(graphe *g, bool *partie){
>       int dmax = 0;
>       for (int i = 0; i < g->n; i++) {
>             if (partie[i] && g->degre[i] > dmax)
>                   dmax = g->degre[i];
>       }
>       return dmax;
> }
> ```

2. Écrire une fonction `bool* accessibles(graphe* g, int s)` qui prend en paramètre un graphe et un sommet s et qui renvoie un tableau de booléens de taille n représentant $A(s)$.

> Solution :

```c
void dfs(graphe *g, int s, bool *vus){
    vus[s] = true;
    for (int i = 0; i < g->degre[s]; i++){
        int voisin = g->voisins[s][i];
        if (!vus[voisin]) dfs(g, voisin, vus);
    }
}
bool* accessibles(graphe *g, int s){
      bool *vus = malloc(g->n * sizeof(bool));
      for (int i = 0; i < g->n; i++) vus[i] = false;
      dfs(g, s, vus);
      return vus;
}
```

3. Écrire une fonction `int degre_etoile(graphe* g, int s)` qui calcule $d^*(s)$ pour un graphe et un sommet passés en paramètre. Quelle est la complexité de cette fonction ?

> Solution :
>
> ```c
> int degre_etoile(graphe *g, int s){
>       bool *acc = accessibles(g, s);
>       int resultat = degre_max(g, acc);
>       free(acc);
>       return resultat;
> }
> ```
>
> La complexité est en $\mathrm{O}(n+p)$.

4. Donner un tri topologique du graphe ci-dessus.

> Solution : L'ordre $1 < 5 < 2 < 3 < 0 < 4 < 6 < 7 < 8$ convient.

5. Dans cette question, on suppose que le graphe $G = (S, A)$ est acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en $\mathrm{O}(|S| + |A|)$.

> Solution : On parcourt les sommets en ordre topologique inverse (on commence par 8 dans l'exemple ci-dessus), et on remplie un tableau t par programmation dynamique pour contenir les valeurs de $d^*(i)$. On a $d^*(i) = \max(d^+(i), \max(d^*(j) \mid (i,j) \in A))$, or les arcs ij vérifient tous $i < j$, et les valeurs de $d^*(j)$ ont donc déjà été calculées.

6. On ne suppose plus le graphe acyclique. Décrire un algorithme permettant de calculer tous les $d^*(s)$ pour $s \in S$ en temps $\mathrm{O}(|S| + |A|)$.

> Solution : À l'intérieur d'une composante fortement connexe C, on a simplement $d^*(s) = \max(d^+(s') \mid s' \in C)$ qui se calcule facilement en temps linéaire en la taille de la composante. L'idée est donc la suivante :
>
> - on calcule les composantes fortement connexes à l'aide de l'algorithme de Kosaraju, en temps linéaire en la taille du graphe
> - ces composantes fortement connexes forment un graphe acyclique $G' = (S', A')$ (et Kosaraju les fournit dans l'ordre topologique)
> - on calcule $d(C) = \max d^+(s)$ à l'intérieur de chaque composante fortement connexe $C \in S'$
> - on applique ensuite l'algorithme de la question précédente avec la formule $d^*(C) = \max(d(C), \max(d^*(C') \mid \text{il existe une arête entre } C \text{ et } C'))$.
>
> On vérifie facilement que cet algorithme est en temps linéaire en la taille du graphe initial.

II Graphes semi-connexes

Soit $G = (S, A)$ un graphe orienté. On dit que G est semi-connexe si pour tous $(s,t) \in S^2$, il existe un chemin de s à t ou un chemin de t à s.

1. Donner un algorithme très simple de complexité $\mathrm{O}(n(n+p))$ (avec $n = |S|$ et $p = |A|$) permettant de déterminer si un graphe orienté est semi-connexe.

> Solution : Pour chaque sommet s : on fait un parcours depuis s dans G, un parcours depuis s dans G^T (le graphe transposé) et on vérifie qu'on visite tous les sommets.

2. Soit $G = (S, A)$ un graphe acyclique et $(s_0, \dots, s_{n-1})$ un ordre topologique de G. Montrer que G est semi-connexe si et seulement si pour tout $i \in [\![0, n-2]\!]$, $(s_i, s_{i+1}) \in A$.

> Solution :
>
> $\Rightarrow$ par contraposée, supposons i tel que $(s_i, s_{i+1}) \notin A$. Alors il n'existe pas de chemin de s_i à s_{i+1} (un tel chemin devrait passer par un s_j, $j > i+1$, ce qui est absurde car $s_{i+1} \prec s_j$). Il n'existe pas non plus de chemin de s_{i+1} à s_i (car $s_i \prec s_{i+1}$). Donc G n'est pas semi-connexe.
>
> $\Leftarrow$ si pour tout $i \in [\![0, n-2]\!]$, $(s_i, s_{i+1}) \in A$, alors deux sommets s_i et s_j sont reliés par le chemin $s_i, s_{i+1}, \dots, s_j$ ou $s_j, s_{j+1}, \dots, s_i$: le graphe est semi-connexe.

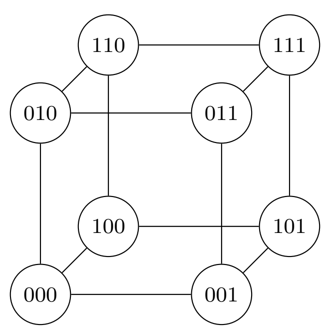
3. En déduire un algorithme plus efficace qui décide si un graphe orienté quelconque est semi-connexe et préciser sa complexité.

> Solution : G est semi-connexe si et seulement si le graphe G' des composantes fortement connexes de G est semi-connexe. On peut donc :
>
> - calculer G' en $\mathrm{O}(n+p)$ avec l'algorithme de Kosaraju ;
> - calculer un tri topologique de G' en $\mathrm{O}(n+p)$;
> - vérifier que pour tout $i \in [\![0, n-2]\!]$, $(s_i, s_{i+1}) \in A$ dans G' en $\mathrm{O}(n+p)$, en parcourant toutes les listes d'adjacence.
>
> La complexité totale est donc en $\mathrm{O}(n+p)$.

III Hypercube

Un hypercube Q_n a pour sommets les mots binaires de taille n, 2 sommets étant reliés s'ils différent d'un bit.

1. Dessiner Q_3.

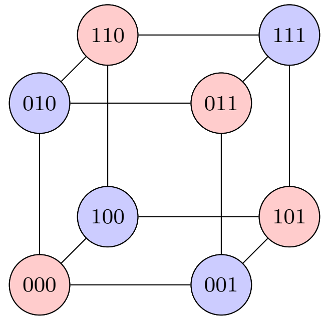
> Solution :
>
> 
>

2. Quel est le nombre de sommets et d'arêtes de Q_n?

> Solution : Il a 2^n sommets chacun de degré n donc, d'après la formule des degrés, $|E| = \dfrac{\sum \deg(v)}{2} = n2^{n-1}$.

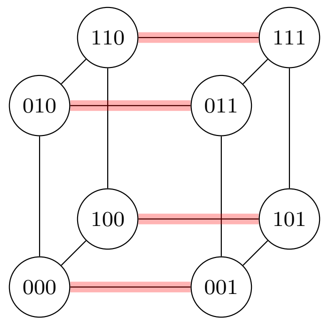
3. Montrer que Q_n est biparti.

> Solution : Soit A l'ensemble des sommets de Q_n ayant un nombre pair de 1, et B l'ensemble des sommets de Q_n ayant un nombre impair de 1. Alors, comme il n'y a pas d'arêtes entre deux sommets de A ni entre deux sommets de B, Q_n est biparti.



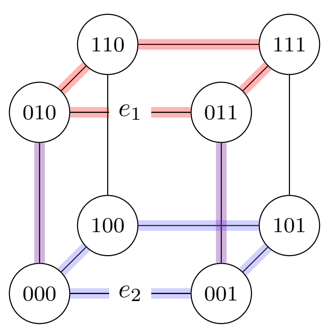
4. Montrer que Q_n possède un couplage parfait.

> Solution : Soit $M = \{\{u0, u1\} \mid u$ est un mot de taille $n-1\}$. En d'autres termes, M contient toutes les arêtes d'un sommet v vers le sommet obtenu en échangeant le dernier bit de v. Chaque sommet est adjacent à une unique arête de M, par définition. Donc M est un couplage parfait.



5. Soit $n \geq 2$. Montrer que Q_n est hamiltonien : il existe un cycle qui visite tous les sommets exactement une fois. Dessiner un tel cycle de Q_3.

> Solution : On commence par remarquer qu'on peut obtenir Q_n à partir de deux copies de Q_{n-1} en reliant chaque sommet à sa « copie ».
> On peut alors démontrer que Q_n est hamiltonien, par récurrence sur n. C'est évident pour $n = 1$. Supposons que Q_n soit Hamiltonien. Q_{n+1} peut être construit à partir de deux Q_n qui possèdent des cycles hamiltoniens C_1 et C_2, qui sont des copies l'un de l'autre. Soient $e_1 = \{u_1, v_1\}$ et $e_2 = \{u_2, v_2\}$ deux arêtes copies l'une de l'autre, dans C_1 et C_2. Alors on peut remplacer, dans $C_1 \cup C_2$, e_1 et e_2 par (u_1, u_2) et (v_1, v_2) afin d'obtenir un cycle Hamiltonien de Q_{n+1}, ce qui achève la récurrence.



IV Questions sur les couplages

1. Soit G un graphe. Montrer que si G a un couplage parfait alors G possède un nombre pair de sommets. La réciproque est-elle vraie ?

2. Soit M_1 et M_2 deux couplages d'un graphe G, avec M_2 maximal (c'est-à-dire qu'on ne peut pas ajouter d'arête à M_2 en conservant un couplage). Montrer que $|M_1| \leq 2|M_2|$, puis donner un cas d'égalité.

> Solution : Soit V_2 l'ensemble des sommets couverts par M_2. Chaque arête couvre deux sommets différents donc $|V_2| = 2|M_2|$.

> Pour toute arête $e = \{u, v\}$ de M_1, au moins l'un des sommets u ou v est couvert par M_2 (sinon, on pourrait ajouter e à M_2).
> Donc $\boxed{|M_1| \leq |V_2| = 2|M_2|}$.

3. Soit $G = (V, E)$ un graphe. Une couverture par sommets (*vertex cover*) de G est un ensemble C de sommets tels que chaque arête de G est adjacente à au moins un sommet de C. L'objectif est de trouver une couverture par sommets C^* de cardinal minimum.
   On propose l'algorithme suivant :

   2-approximation de vertex cover

   $M \longleftarrow$ couplage maximal de G
   $C \longleftarrow$ ensemble des sommets couverts par M

   Montrer que C est bien une couverture par sommet et que $|C| \leq 2|C^*|$.

4. Soit M un couplage d'un graphe G. Rappeler le fonctionnement de l'algorithme de recherche de chemin M-augmentant dans G. Quelle condition G doit-il vérifier ? Donner un contre-exemple si ce n'est pas le cas.

5. Soit $M = (m_{i,j})$ une matrice de taille $n \times n$. On définit le permanent de M :

   $$\text{per}(M) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} m_{i,\sigma(i)}$$

   où S_n est l'ensemble des permutations de $\{1, \ldots, n\}$.
   Définir un graphe G dont le nombre de couplages parfaits est égal à $\text{per}(M)$.
   Remarque : il n'existe pas d'algorithme efficace pour calculer le permanent d'une matrice, contrairement au déterminant qui peut être calculé en $O(n^3)$ avec l'algorithme du pivot de Gauss.