DM17

Un groupe de k gendarmes tente d'attraper un voleur qui cherche à s'enfuir. Des hélicoptères leur permettent de se rendre n'importe où. Le voleur, à pied, ne peut se rendre qu'à des endroits connectés à sa position mais de façon très rapide. En outre, comme il a piraté la radio de la gendarmerie, il sait où les gendarmes peuvent se rendre et ainsi profiter du moment où ses poursuivants sont dans les airs pour changer de place, tant qu'il ne passe pas par une position où se trouve un policier.

Plus formellement,cette situation peut être décrite par un graphe associé à un jeu à deux joueurs. Le joueur gendarme P dispose de k jetons gendarmes et le joueur voleur V possède un jeton voleur. Initialement, P place les jetons gendarmes sur le graphe .Puis le voleur place son jeton sur un des sommets du graphe et la partie commence. Un tour se joue de la façon suivante.

1. P choisit un ensemble de jetons qu'il enlève du graphe, choisit où il va les replacer et l'annonce à V.
2. V déplace alors son jeton le long d'un chemin partant de sa position courante, ne passant par aucun sommet occupé par un jeton gendarme, ou choisit de rester sur place.
3. P pose ses jetons là où il l'a annoncé.
4. Si un jeton de P se trouve sur le sommet où se trouve le jeton voleur, P gagne la partie.

Si le voleur a une stratégie pour ne jamais se faire attraper, V gagne la partie. Notons que les deux joueurs savent en permanence où se trouvent tous les jetons. Il n'y a donc pas d'informations cachées.

Implémentation du jeu

Cette partie est à traiter avec le langage C. Outre l'en-tête habituel, les bibliothèques `pthread.h` et `semaphore.h` sont chargées. La fin du sujet rappelle la syntaxe des fonctions de concurrence et de synchronisation.

On considère une implémentation concurrente utilisant $k+2$ fils d'exécution. Un fil correspond au joueur P et aux décisions qu'il prend, un fil correspond au joueur V, les k derniers fils représentent chaque gendarme.

Question 1. Les sommets du graphe sont représentés par des `int`. Ces fils sont appelés en début du programme et ne sont arrêtés qu'en cas de victoire du joueur P. Une variable globale `int* affectation` stocke l'ensemble des sommets où vont être déplacés les gendarmes, ainsi qu'une variable globale `k` stockant le nombre de gendarmes. Une variable `bool * changement` désigne si les affectations sont des déplacements ou des gendarmes restant sur place. Ainsi, `changement[i]` vaut `true` si un gendarme est aéroporté sur le sommet `affectation[i]` ; `false` si un jeton gendarme est déjà sur le sommet `affectation[i]` et reste sur sa position. Autrement dit, un déplacement du voleur est tel qu'il n'existe pas de `i` avec `changement[i] = true` et tel que le voleur passe par `affectation[i]`. Une variable globale `int voleur` représente la position du voleur. Une variable globale `bool victoireP`, initialisée à `false`, vaut `true` si un gendarme attrape le voleur et que la partie se termine.
Écrire une fonction `void initAffect()` qui initialise le vecteur `affectation`.

On suppose disposer de fonctions `void initP()` et `void initV()` qui initialisent les placements initiaux des joueurs P et V. Une fonction `void coupP(int v)` prend en entrée la position du voleur et place dans `affectation` les sommets sur lesquels doivent se déplacer les gendarmes et met à jour `changement`.
Une fonction `int coupV(int v)` prend en entrée la position du voleur et à partir des éléments d'`affectation` et de `changement` décide sur quel sommet se rendre sachant qu'il ne peut passer par un sommet occupé par un gendarme.

On souhaite écrire les fonctions `void tourP()` et `void tourV()` qui respectivement mettent d'abord à jour le tableau `affectation` grâce à `coupP` puis mettent à jour `voleur`. Ainsi, `tourP` correspond à l'étape 1 d'un tour de jeu, et `tourV` à l'étape 2. Pour ce faire, on suppose avoir déclaré globalement le sémaphore `sem_t semDecision`.

Question 2. Écrire la ligne d'initialisation de `semDecision` placée au début de `main`. Justifier votre choix.

Question 3. Écrire les fonctions `tourP` et `tourV`.

Question 4. Pourquoi n'est-il pas nécessaire de protéger l'accès à voleur par mutex ?

Question 5. Pour l'étape 3 d'un tour, chaque fil d'exécution de gendarme va appeler une fonction `int deplacement(int position)` qui prend en entrée la position courante du gendarme, vérifie s'il doit bouger et, si tel est le cas, choisit `i` tel que `changement[i] = true`, passe `changement[i]` à `false` et renvoie `affectation[i]`.
Écrire la fonction `deplacement`.

Question 6. Quels problèmes peuvent survenir si chaque fil d'exécution de gendarmes appelle la fonction `deplacement` ?

Question 7. Comment résoudre ce problème ?

On veut que le premier appel à `deplacement` s'effectue après `tourV`, puis une fois que les `k` appels à `deplacement` ont été effectués, on fait de nouveau appel à `tourP` si le voleur n'a pas été attrapé et on recommence. Pour s'assurer

que les appels à deplacement sont bien effectués après `tourV`, on propose d'utiliser un sémaphore incrémenté dans le fil d'exécution du voleur après l'appel à `tourV`.

Question 8. Justifier que ce n'est pas suffisant pour s'assurer que chaque fil d'exécution de gendarme effectue un appel à `deplacement`.

Question 9. Proposer une solution qui assure qu'un fil d'exécution de gendarme n'effectue qu'un déplacement par tour.

Question 10. Comment s'assurer que `tourP` ne s'effectue qu'une fois que chaque fil gendarme a effectué un appel à `deplacement` ?

Question 11. Justifier pourquoi on ne peut utilier `pthread_join`.

Question 12. Les fils d'exécution du joueur P, du voleur et des gendarmes font respectivement appel aux fonctions `joueurP`, `joueurV` et `gendarme`. Déclarer l'ensemble des variables globales à utiliser.

Question 13. Écrire une fonction `main` qui, notament, initialise les sémaphores et les autres variables, crée les fils d'exécution et les appelle.

Question 14. Donner les fonctions `joueurP`, `joueurV` et `gendarme`.

Nombre de gendarmes

On considère un graphe $G = (S, A)$ sur lequel va se jouer une partie de gendarmes aéroportés et du voleur. On s'intéresse au nombre de gendarmes nécessaire pour que le joueur P ait une stratégie gagnante.

Question 15. Montrer que si P a une stratégie gagnante avec k gendarmes alors P en a une avec $k' > k$ gendarmes.

Question 16. On note $k(G)$ le nombre minimum de gendarmes nécessaire pour que P ait une stratégie gagnante pour toute position de voleur initiale sur le graphe G. Borner $k(G)$.

Question 17. Prouver que si G est un arbre avec au moins deux sommets, $k(G) = 2$.

Question 18. Calculer $k(G)$ lorsque G est un graphe complet.

Question 19. Calculer $k(G)$ lorsque G est un cycle avec au moins 3 sommets.

Question 20. Une grille $\mathcal{G}_{n,m} = (S, A)$ est un graphe tel que $S = \{1, \dots, n\} \times \{1, \dots, m\}$ et :

$$A = \{((a,b),(a+1,b)) \mid 1 \leqslant a < n, 1 \leqslant b \leqslant m\} \bigcup \{((a,b),(a,b+1)) \mid 1 \leqslant a \leqslant n, 1 \leqslant b < m\}$$

On appelle lignes les sous-graphes induits par $\{i\} \times \{1, \dots, m\}$ et colonnes les sous-graphes induits par $\{1, \dots, n\} \times \{j\}$. Dessiner le graphe $\mathcal{G}_{4,4}$.

Question 21. On considére $\mathcal{G}_{n,n}$ avec $n \geqslant 2$. Montrer que si on place $n-1$ gendarmes, il existe toujours une ligne et une colonne sans gendarme.

Question 22. En déduire que $k\left(\mathcal{G}_{n,n}\right) \geqslant n$.

Question 23. Justifier pourquoi $k\left(\mathcal{G}_{n,n}\right) \neq n$.

Question 24. Prouver que P a une stratégie gagnante sur $\mathcal{G}_{n,n}$ avec $n+1$ gendarmes.
Indication : on peut commencer par placer un gendarme sur chaque case de la première colonne.

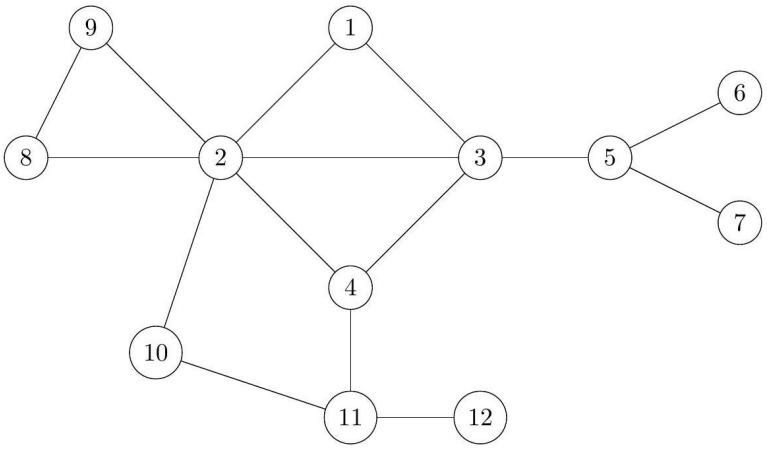
Largeur arborescente

Cette partie est à traiter avec le langage OCaml. Dans toute la suite, on considère uniquement des graphes non orientés ayant au moins un sommet.

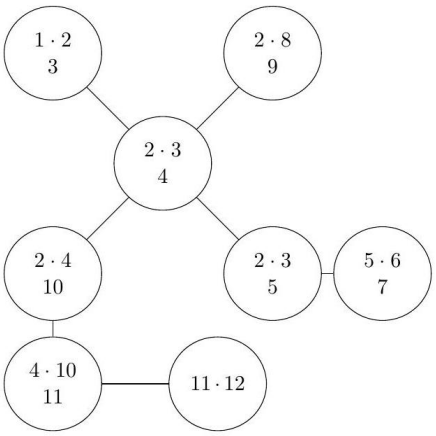
Soit $G = (S, A)$ un graphe. La *décomposition arborescente* de G est un couple (T, e) avec $T = (X, R)$ un arbre et e une fonction de X dans $\mathcal{P}(S)$ où $\mathcal{P}$ désigne l'ensemble des parties d'un ensemble. Autrement dit, e étiquette chaque sommet de T par un ou plusieurs sommets de G. En outre, (T, e) satisfait les conditions suivantes.

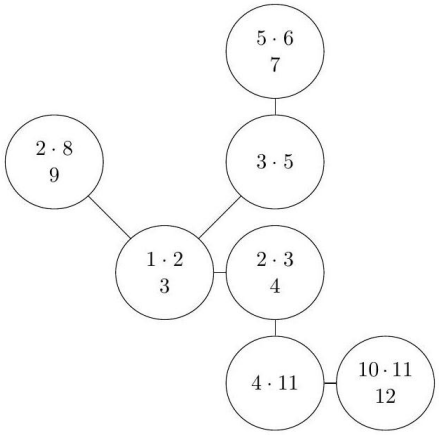
- **(i)** Pour tout sommet s de G, il existe un sommet t de T tel que $s \in e(t)$.
- **(ii)** Pour toute arête $(s,t) \in A$, il existe $u \in X$ tel que $s \in e(u)$ et $t \in e(u)$
- **(iii)** Pour tout sommet $s \in S$, l'ensemble $\{u \in X \mid s \in e(u)\}$ est connexe dans T. Autrement dit, l'ensemble des sommets de T tels que s fasse partie de leurs étiquettes forme un sous-arbre de T.

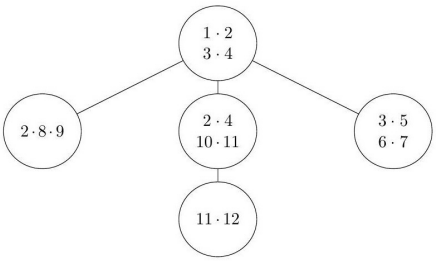
Question 25. On considère le graphe G_1 suivant.

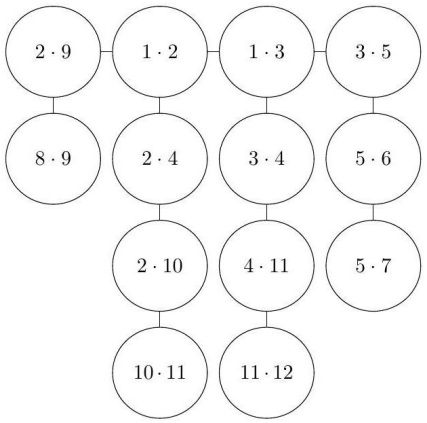


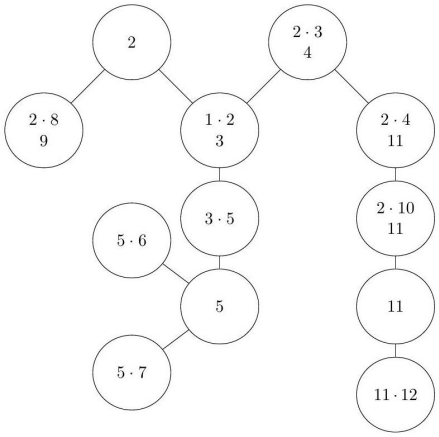
Pour chacun des arbres suivants, lesquels sont des décompositions arborescentes de G_1 ?











Question 26. Montrer que tout graphe possède au moins une décomposition arborescente.

Question 27. Pour définir les graphes, on adopte le type `graph` ci-dessous. Les sommets sont représentés par des entiers de 0 à $n-1$ et le type `graph` est une représentation par liste d'adjacence à l'aide d'un `array` de dimension n. Les décompositions arborescentes sont représentées par le type `da` où `g` représente le graphe dont l'élément est une décomposition, `arbre` la structure d'arbre, et `e` la fonction des sommets de l'arbre dans les parties des sommets de `g`.

```
type graph = int list array
type da = { g : graph; arbre : graph; e : int list array; }
```

□ **27.1.** Écrire une fonction `isArbre : da -> bool` qui vérifie si une décomposition arborescente représente un arbre.
□ **27.2.** Écrire une fonction `verif1 : da -> bool` qui vérifie que les éléments de `e` sont des sommets de `g` et que chaque sommet de `g` est présent au moins une fois.
□ **27.3.** Écrire une fonction `verif2 : da -> bool` qui vérifie la condition **(ii)**.
□ **27.4.** Écrire une fonction `verif3 : da -> bool` qui vérifie la condition **(iii)**.
□ **27.5.** En déduire une fonction `isda : da -> bool` qui vérifie qu'une décomposition est arborescente.

Question 28. La largeur l d'une décomposition arborescente d'un graphe est définie par :

$$l(T,e) = \max_{u\in X} \text{Card}(e(u)) - 1$$

La *largeur arborescente* d'un graphe G, notée $la(G)$, est la plus petite largeur de toute ses décompositions arborescentes.
□ **28.1.** Donner la largeur des décompositions arborescentes précédentes.
□ **28.2.** Quelle est la largeur arborescente d'un arbre ?
□ **28.3.** Prouver que la largeur arborescente d'un cycle de taille au moins 3 est 2.
□ **28.4.** Soit G un graphe, (T,e) une décomposition arborescente de G et (x,y) une arête de T. Comme T est un arbre, si on retire (x,y), alors on obtient deux composantes connexes disjointes T_x et T_y avec x un sommet de T_x et y un sommet de T_y. Posons :

$$S_x = \bigcup_{s\in T_x} e(s)\setminus e(y) \qquad S_y = \bigcup_{s\in T_y} e(s)\setminus e(x)$$

Montrer qu'il n'existe pas d'arête de S_x vers S_y dans G.
□ **28.5.** En déduire que tout graphe G est $la(G)+1$ coloriable.

Question 29. On pourrait montrer que $la(G) = k(G)-1$. On se contente d'établir que $la(G) \geqslant k(G)-1$. Pour ce faire, on montre que si le joueur P possède $la(G)+1$ policiers, il possède une stratégie gagnante. Considérons une décomposition arborescente (T,e) de G de largeur $la(G)$. Choisissons x un sommet de T. On commence à placer les policiers en $e(x)$. Soit s le sommet initial du voleur.
□ **29.1.** En considérant les descendants de x comme des sous-arbres, montrer que les sommets t de T tels que $s\in e(t)$ sont tous dans le même sous-arbre.
□ **29.2.** Décrire la stratégie gagnante pour P.

Annexe C

- `pthread_t` est le type représentant les fils d'exécutions.
- `pthread_create(pthread_t *thread, NULL, void *(*f), NULL)` est crée le fil `thread` sur la fonction `f`.
- `pthread_join(pthread_t th, void **retval)` permet d'attendre la fin d'exécution du fil `th`.
- `pthread_mutex_t` type représentant les mutex.
- `pthread_mutex_init(pthread_mutex_t *m, NULL)` fonction permettant d'initialiser un mutex.
- `pthread_mutex_lock(pthread_mutex_t * m )` verrouille le mutex `m`.
- `pthread_mutex_unlock (pthread_mutex_t * m )` déverouille le mutex `m`.
- `pthread_mutex_destroy(pthread_mutex_t * m )` détruit le mutex `m`.
- `sem_t` est le type des sémaphores.
- `sem_init(sem_t * sem, 0, unsigned int value)` permet d'initialiser `sem` à `value`.
- `sem_wait (sem_t *sem)` décrémente le sémaphore `sem`.
- `sem_post (sem_t *sem)` incrémente le sémaphore.
- `sem_destroy (sem_t *sem)` libère le sémaphore `sem`.