Lycée Franklin Roosvelt

MPI - Informatique
Année 2022 – 2023

DS 6 : 03 Avril

4H

Les questions de la section 2 *Implémentation du jeu* sont à effectuer en premier et à rendre au bout de 2h.

1 Jeu des gendarmes aéroporté et du voleur

On va ici présenter le jeu des gendarmes aéroportés et du voleur. Un groupe de k gendarmes tentent d'attraper un voleur qui cherche à s'enfuir. Pour ce faire, ils disposent d'hélicoptères qui leur permettent de se rendre n'importe où. Le voleur est à pied et ne peut se rendre qu'à des endroits connectés à sa position, mais de façon très rapide. Cependant, il a piraté la radio de la gendarmerie et sait où les gendarmes vont se rendre. Il peut donc 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.

On donne maintenant une définition plus formelle du jeu. Il s'agit d'un jeu à deux joueurs qui se joue sur un graphe. 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, puis la partie commence. Un tour se joue de la façon suivante :

1. P choisi un ensemble de jetons qu'il enlève du graphe, choisie 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 actuelle, ne passant par aucun sommet occupé par un jeton gendarme, ou choisi 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 pas d'informations cachées.

2 Implémentation du jeu

Cette partie est à effectuer en C. Un rappel sur les fonctions de concurrence et synchronisation et effectué à la fin de ce sujet. En plus de l'en-tête habituel, on suppose que les bibliothèques suivantes sont incluses : `pthread.h` et `semaphore.h`.

Avant de nous intéresser aux stratégies de ce jeu, on va s'intéresser à son implémentation. Pour ce faire, on considère une implémentation concurrente utilisant $k + 2$ fils d'exécution. Un fil correspond au joueur P et aux décisions qu'il prend, un fils correspond au joueur V, et les k derniers fils représentent chaque gendarme. Les sommets du graphe sont représentés par des `int`. Ces fils sont appelés en début du programme et ne sont arrêté qu'en cas de victoire du joueur P.

On dispose d'une variable globale `int* affectation` qui stocke l'ensemble des sommets où vont être déplacé les gendarmes, ainsi qu'une variable globale `k` stockant le nombre de gendarmes.

La 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]` et vaut `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]`. De plus, on dispose d'une variable globale `int voleur` représentant la position du voleur.

Considérons également une variable globale `bool victoireP` initialisé à `false`. Cette variable doit valoir `true` si un gendarme attrape le voleur et que la partie se termine.

1. Écrire une fonction `void initAffect()` qui initialise le vecteur `affectation`.

On suppose disposer d'une fonction `void initP()` et `void initV()` qui initialise les placements initiaux des joueurs P et V, ainsi que d'une fonction `void coupP(int v)` qui 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`.

De même, on suppose disposer d'une fonction `int coupV(int v)` qui prend en entrée la position du voleur et à partir des éléments d'`affectation` et de `changement` décide dans 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 peut supposer avoir déclaré globalement le sémaphore `sem_t semDecision`.

2. Écrire la ligne d'initialisation de `semDecision` qu'on placera au début du `main`. Justifier votre choix de valeur initial.
3. Écrire les fonctions `tourP` et `tourV`.
4. Justifier pourquoi il n'est pas nécessaire de protéger l'accès à `voleur` par mutex.

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

5. Écrire la fonction `deplacement`.
6. Quels problèmes peuvent survenir si chaque fils d'exécution de gendarmes appellent la fonction `deplacement` ?
7. Expliquer comment résoudre ce problème.

On veut que le premier appel à `deplacement` s'effectue après `tourV`, puis une fois que les k appels a `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`.

8. Justifier que ce n'est pas suffisant pour s'assurer que chaque fil d'exécution de gendarme effectue un appel à `deplacement`.
9. Proposer une façon pour s'assurer qu'un fil d'exécution de gendarme n'effectue qu'un déplacement par tour.
10. Comment s'assurer que `tourP` ne s'effectue qu'une fois que chaque fil gendarme a effectué un appel à `deplacement` ?
11. Justifier pourquoi on ne peut utilier `pthread_join`.

Les fils d'exécution du joueur P, du voleur et des gendarmes font respectivement appel aux fonctions `joueurP`, `joueurV` et `gendarme`.

12. Déclarer l'ensemble des variables globales que vous utilisez.
13. Écrire votre main qui notament initialise vos sémaphores et autres variables, crée les fils d'exécution et les appelle.
14. Donner les fonctions `joueurP`, `joueurV` et `gendarme`.

3 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.

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

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.

16. Borner $k(G)$.
17. Prouver que si G est un arbre avec au moins deux sommets, $k(G) = 2$.
18. Calculer $k(G)$ lorsque G est un graphe complet.
19. Calculer $k(G)$ lorsque G est un cycle avec au moins 3 sommets.

On s'intéresse maintenant au cas de la grille. Une grille $\mathcal{G}_{n,m} = (S, A)$ est un graphe tel que $S = \{1; \ldots; n\} \times \{1; \ldots; m\}$ et

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

On appelle lignes les sous-graphes induit par $\{i\} \times \{1; \ldots; m\}$ et colonnes les sous-graphe induit par $\{1; \ldots; n\} \times \{j\}$.

20. Dessiner le graphe $\mathcal{G}_{4,4}$.

Considérons $\mathcal{G}_{n,n}$ avec $n \geq 2$.

21. Montrer que si on place $n-1$ gendarmes, il existe toujours une ligne et une colonne sans gendarme.
22. En déduire que $k(\mathcal{G}_{n,n}) \geq n$.
23. * Justifier pourquoi $k(\mathcal{G}_{n,n}) \neq n$.
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.*

4 Largeur arborescente

Cette partie est à effectuer en OCaml.

On va maintenant s'intéresser à un paramètre relatif au graphe appelé la **largeur arborescente**. Notre but va être d'étudier cette valeur mise en relation avec $k(G)$.

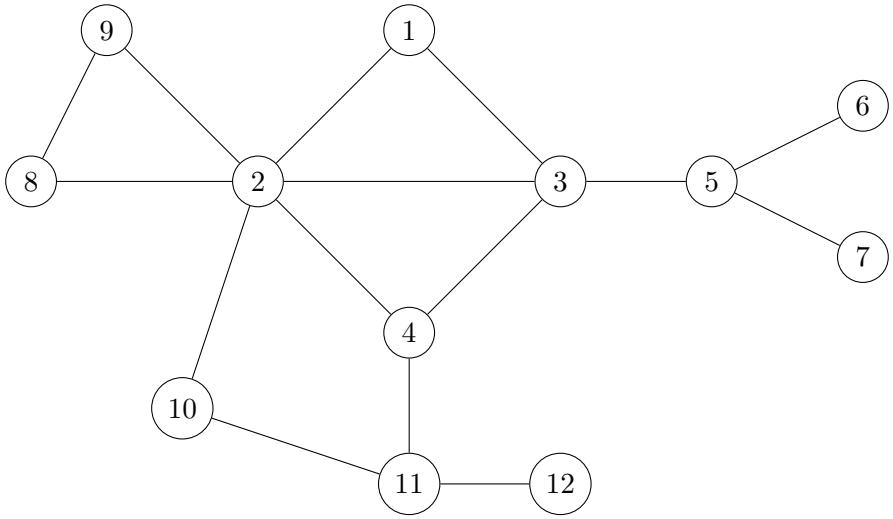
Dans toute la suite, on considère uniquement des graphes non dirigés à au moins un sommet. Soit $G = (S, A)$ un graphe. La décomposition arborescente d'un graphe est un couple (T, e) avec $T = (X, R)$ un arbre et e une fonction de X dans $\mathcal{P}(S)$ où $\mathcal{P}$ désigne l'ensemble des parties d'un ensemble. Autrement dit, e étiquette chaque sommet de T par un ou plusieurs sommets de G. De plus (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 $t \in e(u)$

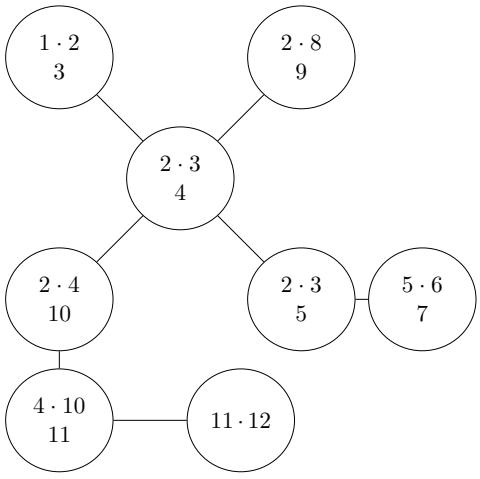
(iii) Pour tout sommet $s \in S$, l'ensemble $\{\ u \in X \mid s \in e(u)\ \}$ est connexe dans T. Autrement dit, l'ensemble des sommets de T tels que s fasse partie de leurs étiquettes forme un sous-arbre de T.

Considérons le graphe G_1 suivant :

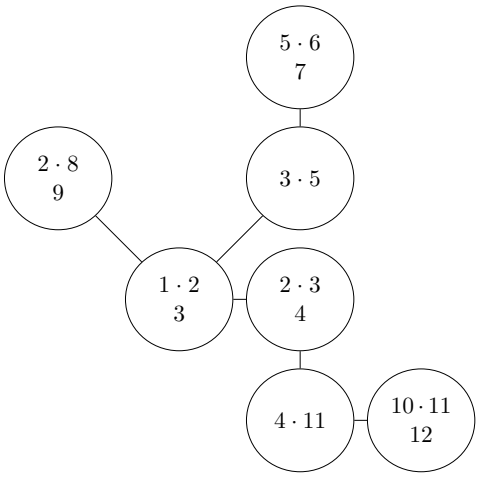


25. Pour chacun des arbres suivants, lesquels sont des décompositions arborescentes de G_1 ?

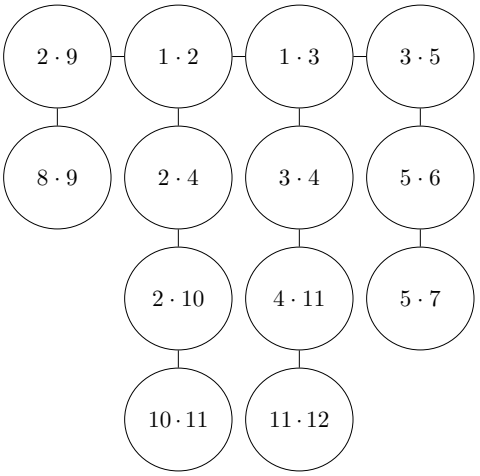
a) T_1 :



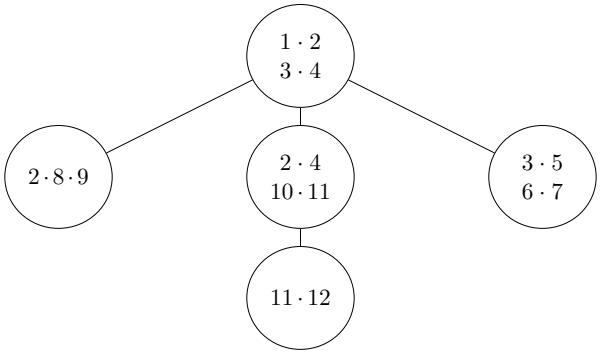
b) T_2 :



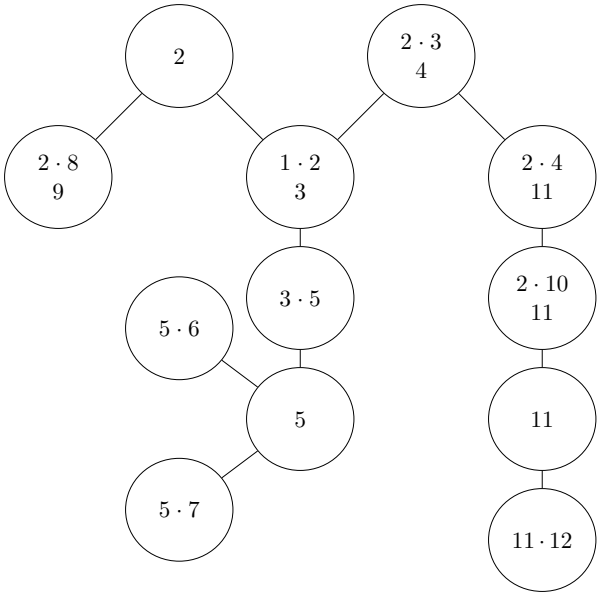
c) T_3 :



d) T_4 :



e) T_5 :



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

Pour définir les graphes en OCaml on utilise la structure de donnée suivante : `type graph = int list array`. 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.

On représente les décompositions arborescentes par le type `da` définie par :

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

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.

27. Écrire une fonction `isArbre : da -> bool` qui vérifie si une décomposition arborescente représente bien un arbre.
28. Écrire une fonction `verif1 : da -> bool` qui vérifie que les éléments de `e` sont bien des sommets de `g` et que chaque sommet de `g` est bien présent au moins une fois.
29. Écrire une fonction `verif2 : da -> bool` qui vérifie la condition (ii).
30. Écrire une fonction `verif3 : da -> bool` qui vérifie la condition (iii).

31. En déduire une fonction `isda : da -> bool` qui vérifie qu'une décomposition arborescente en est bien une.

On définit maintenant la largeur arborescente d'un graphe. La largeur l d'une décomposition arborescente d'un graphe est :

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

La largeur arborescente d'un graphe G est la plus petite largeur de toute ses décompositions arborescentes et on la note $la(G)$.

32. Donner la largeur des décompositions arborescentes précédentes.
33. Quelle est la largeur arborescente d'un arbre ?
34. Prouver que la largeur arborescente d'un cycle de taille au moins 3 est 2.

Soit G un graphe et (T, e) une décomposition arborescente de G. Soit (x, y) une arête de T. Comme T est un arbre, si on y 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. Soit $S_x = \bigcup_{s \in T_x} e(s) \setminus e(y)$ et $S_y = \bigcup_{s \in T_y} e(s) \setminus e(x)$.

35. Montrer qu'il n'existe pas d'arête de S_x vers S_y dans G.
36. * En déduire que tout graphe G est $la(G) + 1$ coloriable, où $la(G)$ désigne la largeur arborescente de G.

On veut maintenant montrer que $la(G) = k(G) - 1$. Ici, on se contentera de montrer que $la(G) \geq k(G) - 1$. Pour ce faire, il suffit de montrer 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 et on commence à placer les policiers en $e(x)$. Soit s le sommet initial du voleur.

37. En considérant les descendants de x comme des sous-arbres, montrer que les sommets t de T tel que $s \in e(t)$ sont tous dans le même sous-arbre.
38. * Décrire la stratégie gagnante pour P.

Rappel de C

On rappel les fonctions suivantes en C. On supposera les bibliothèques nécessaires chargées. Lorsque certains paramètres ne sont pas important, au lieu de les donner on écrit directement par quoi les remplacer (0 ou NULL).

— `pthread_t` type représentant les fils d'exécutions.

— `pthread_create(pthread_t *thread, NULL,void *(*f),NULL)` est la fonction créant 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` 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`.