Correction DS 6

Xavier Badin de Montjoye

1 Jeu des gendarmes aéroportés et du voleur

Plusieurs remarques :
— Même si non précisé, le jeu se joue sur un graphe non orienté.
— Le voleur n'est pas limité aux sommets dans le voisinage direct de sa position. Son déplacement suit un chemin **arbitrairement long** ne passant par aucun jeton gendarme.
— On peut considérer que plusieurs policiers se trouvent sur le même sommet, mais ça n'a pas d'incidence sur la suite.

2 Implémentation du jeu

1.

```c
void initAffect(){
    affectation = (int*) malloc(k*sizeof(int));
}
```

q1.c

2. `sem_init(&semDecision, 0, 0);`
On initialise le sémaphore à 0. On veut que tant qu'il n'a pas été incrémenté, le joueur V ne puisse pas agir. `tourP()` incrémentera donc le sémaphore pour permettre à V de jouer.

3.

```c
void tourP(){
    coupP(voleur);
    sem_post(&semDecision);
}

void tourV(){
    sem_wait(&semDecision);
    voleur = coupV(voleur);
}
```

q3.c

4. Du fait de l'utilisation d'un sémaphore, `tourP` et `tourV` ne s'effectue pas en même temps. De ce fait, les lectures et écritures de voleurs ne sont pas concurrentes et n'ont donc pas besoin d'être protégées par mutex.

5.

```c
int deplacement(int position){
    int possible;
    for (int i = 0; i<k;i++){
        //On verifie si le policier doit rester au meme endroit.
        if(affectation[i]==position && changement[i]==false){
            return position;
        }
        //On enregistre un indice representant une position a occuper
        if(changement[i]==true){
```

```
            possible = i;
        }
    }
    changement[possible]=false;
    return affectation[possible];
}
```

q5.c

6. Plusieurs fils d'exécution peuvent choisir la même affectation. En effet, supposons qu'un fil trouve i tel que `changement[i]==true`, puis qu'un autre fil le trouve également, et qu'ensuite il passe tous les deux `changement[i]` à `false` et renvoie tous les deux `affectation[i]`.

7. Pour résoudre ce problème, on peut utiliser un mutex qui sera verrouillé avant chaque appel à `deplacement` et libérer ensuite.

8. Il est possible qu'un même fil d'exécution appelle plusieurs fois `deplacement`. Le sémaphore nous assure en effet que seul k appels sont effectués, mais pas qu'ils sont effectués par des fils différents.

9. Pour s'en assurer, plusieurs solutions sont envisageables. On en propose une qui consiste à implémenter une barrière de synchronisation. Pour ce faire, on a besoin d'initialiser, une variable globale supplémentaire, `int compteur`, un mutex `pthread_mutex_t mCompte` et un sémaphore `sem_t semCompte` initialisé à 0. Ainsi, après un appel à déplacement, un fil gendarme va accéder à `compteur` et l'incrémenter s'il est différent de $k-1$, puis il va décrémenter `semCompte`. Si `compteur` a pour valeur $k-1$, le fil va juste libérer $k-1$ fois `semCompte`.

10. On propose deux méthodes. La première consiste à ajouter un nouveau sémaphore incrémenté une fois après chaque appel à `deplacement`, et décrémenter k fois avant un appel à `tourP`.

La seconde utilise encore un sémaphore, mais profite de notre barrière implémentée à la question précédente. Il est décrémenté une fois avant `tourP` et incrémenté une fois par le fil qui vérifie que la variable `compteur` vaut $k-1$.

11. La fonction `pthread_join` attend la fin d'exécution d'un thread, ce qui n'est pas l'utilité qu'on souhaite en faire ici, puisque les fils continuent tant que la partie n'est pas terminée.

12.

```
//Valeurs de positions
int* affectation;
bool* changement;
int voleur;
int k;

//Semaphores
sem_t semDecision;
sem_t semGTurn; //Incremente par V et decremente par gendarme.
sem_t semCompte;
sem_t semPTurn; //Incremente par gendarme et decremente par P

//Mutex
pthread_mutex_t mDecision;
pthread_mutex_t mCompte;

//Compteur
int compteur;

//Fils d'executions
pthread_t thP;
pthread_t thV;
pthread_t* thG;
// On utilise un tableau pour stocker les fils d'executions des k gendarmes.
```

q12.c

13.

```c
int main(int argc, char *argv[]){
    //Commencons par initialiser les differentes variables.
    initAffect();
    changement = (bool*) malloc(k*sizeof(bool));
    thG = (pthread_t*) malloc(k*sizeof(pthread_t));
    initP();
    initV();
    k=K;// On suppose l'existence d'une constante K.
    bool victoireP = false;

    //On initialise les mutex et les semaphores
    pthread_mutex_init(&mDecision,NULL);
    pthread_mutex_init(&mCompte,NULL);
    sem_init(&semDecision,0,0);
    sem_init(&semCompte,0,0);
    sem_init(&semPTurn,0,0);
    sem_init(&semGTurn,0,k);
    //semGTurn commence a k car les fils gendarmes jouent apres l'initialisation et
        donc doivent avoir leur premier appel avant le premier appel a tourP.

    //Enfin, on instancie les fils d'executions.
    pthread_mutex_init(&thP, NULL, joueurP, NULL);
    pthread_mutex_init(&thV, NULL, joueurV, NULL);
    for (init i=0; i< k; i++){
        pthread_mutex_init(&thG[i], NULL, gendarme, NULL);
    }

    //Il faut attendre la fin de l'execution de tous les fils avant d'arreter le
        programme
    pthread_join(thP, NULL);
    pthread_join(thV, NULL);
    for (init i=0; i< k; i++){
        pthread_join(thG[i],NULL);
    }

    //Enfin, on libere l'espace.
    free(affectation);
    free(changement);
    free(thG);
    pthread_mutex_destroy(&mDecision);
    pthread_mutex_destroy(&mCompte);
    sem_destroy(&semDecision);
    sem_destroy(&semGTurn);
    sem_destroy(&semPTurn);
    sem_destroy(&semCompte);
}
```

q13.c

14.

```c
//La fonction joueurP appelle tourP tant que P n'a pas gagne
//On verifie au debut de la boucle le semaphore semPTurn car les gendarmes doivent
    jouer apres l'initialisation.
void *joueurP(void* arg){
    while(not victoireP){
        sem_wait(&semPTurn);
        tourP();
    }
    pthread_exit(NULL);
}

// joueurV appelle tourV qui verifie en interne qu'elle est execute apres tourP.
// Elle termine en permettant aux fils gendarmes d'agir.
void *joueurV(void* arg){
    while(not victoireP){
        tourV();
        for (int i =0; i<k; i++){
```

```
17              sem_post(&semGTurn);
18          }
19      }
20      pthread_exit(NULL);
21  }
22
23
24  void *gendarme(void* arg){
25      int position =0;
26      while(not victoireP){
27          //On verifie que voleur a joue
28          sem_wait(&semGTurn);
29          // On se rend a la nouvelle position
30          pthread_mutex_lock(&mDecision);
31          position = deplacement(position);
32          pthread_mutex_unlock(&mDecision);
33
34          //On verifie que le voleur n'est pas attrape
35          if (position==voleur){
36              victoireP = true;
37          }
38
39          // On installe une barriere
40          bool dernier = false;
41          pthread_mutex_lock(&mCompte);
42          if(compteur = k-1){
43              dernier = true;
44              compteur = 0;
45          }
46          else {compteur++;}
47          pthread_mutex_unlock(&mCompte);
48          if (!dernier){sem_wait(&semCompte);}
49          else{
50              sem_post(&semPTurn);
51              for (int i = 0; i<k-1; i++)
52                  {sem_post(&semCompte);}
53          }
54      }
55      pthread_exit(NULL);
56  }
```

q14.c

3 Nombre de gendarmes

15. Supposons que P a une stratégie gagnante avec k gendarmes. Dans le cas où P a $k' > k$ gendarmes, il suffit de considérer la stratégie suivante : les $k' - k$ premiers gendarmes sont positionnés sur les sommets 1 à $k' - k$ et ils ne bougeront jamais. Puis, le joueur P joue comme s'il ne disposait que de k gendarmes en suivant une stratégie optimale. Si sa stratégie consiste à se rendre dans les $k' - k$ sommets, il va dans un sommet quelconque libre, ce qui ne change rien à sa stratégie car le sommet est déjà occupé. Les déplacements du voleur pouvant être effectuées dans le cas à k gendarmes, on s'assure que la partie termine et donc que le joueur P gagne.

16. Au minimum, il est nécessaire de disposer de $d+1$ gendarmes avec d le degré minimal de G. En effet, sinon, à tout instant, un des sommets parmi ceux où se trouve le voleur et son voisinage ne reçoit pas de gendarme et, le voleur peut s'y rendre.

Au maximum, il faut au plus $|S|$ gendarmes, un par sommet du graphe. Cette situation survient pour les graphes complets, par le résultat précédent.

Donc : $d + 1 \leq k(G) \leq |S|$

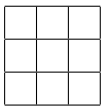
17. Soit G un arbre avec au moins deux sommets. Un arbre étant connexe, son degré minimal est 1 et $k(G) \geq 2$. Maintenant, considérons le cas où $k = 2$ et donnons une stratégie gagnante pour P.

On commence par enraciner l'arbre et on pose un gendarme sur sa racine et l'autre sur un sommet quelconque. Le voleur joue alors dans un des sous-arbres. On place le deuxième gendarme sur la racine du sous-arbre où se trouve le voleur. Comme le premier gendarme bloque la racine, le voleur ne peut se rendre dans un autre sous-arbre. Il doit donc se rendre dans un des sous-arbres définis par le second gendarme. On peut alors déplacer le premier gendarme à la racine de ce sous-arbre et recommencer jusqu'à atteindre le voleur.

À chaque étape, le voleur est bloqué dans un arbre de hauteur strictement inférieur. Cela implique que le voleur fini par se faire attraper.

18. Dans un graphe complet, on a $|S| - 1$ comme degré minimale, donc $k(G) \geq |S|$, donc $k(G) = |S|$.

19. Dans le cas d'un cycle G à au moins 3 sommets, le degré minimal est 2. On a donc, $k(G) \geq 3$. Or, si on pose un gendarme sur un sommet du graphe pour ne plus le bouger du reste de la partie, le graphe privé de ce sommet forme alors un arbre. Il existe une stratégie gagnante utilisant au plus 2 gendarmes sur cette arbre, de ce fait, il existe une stratégie gagnante utilisant 3 gendarmes sur G et $k(G) = 3$.

20. 

21. L'ensemble des lignes occupées par les gendarmes ainsi que l'ensemble des colonnes occupées par les gendarmes sont de cardinal borné par le nombre de gendarmes, donc par $n - 1$. Ainsi, il existe toujours une ligne et une colonne sans gendarme.

22. Considérons le cas où $k = n-1$. On va décrire une stratégie gagnante pour le joueur V. Tout d'abord, il se place sur le sommet (i, j) sans gendarme sur sa ligne et sa colonne. Puis à chaque tour, il se rend sur le prochain sommet (i', j') dont la ligne et la colonne sont inoccupées. Cela est possible, car il peut emprunter le chemin suivant : $(i, j) \to^* (i, j') \to^* (i', j')$. En effet, ni la ligne i, ni la colonne j' n'est occupé lors du déplacement du voleur.

23. Notons que le placement de n gendarmes entraîne l'existence d'une composante connexe comportant au moins $\frac{n(n-1)}{2}$ sommets. De plus, pour tout déplacement de gendarmes, une telle zone est accessible par cardinalité. Il en résulte que le voleur se trouve toujours dans une zone connexe non vide et est donc pas attrapé.

24. Considérons le cas $k = n + 1$ et montrons que P a une stratégie gagnante. On commence par placer les gendarmes sur les sommets de la première ligne et sur le sommet $(2, 1)$. Le voleur est donc nécessairement dans les cases suivantes. À chaque tour, on récupère le gendarme sur le plus petit sommet occupé dans l'ordre lexicographique, et on le place sur le premier sommet non occupé dans l'ordre lexicographique. À chaque étape, on note que le voleur ne peut se rendre dans les cases précédentes dans l'ordre lexicographique au

dernier gendarme joué. Ainsi, son nombre de place disponible décroit strictement, et les gendarmes finissent par le capturer.

4 Largeur arborescente

25. Oui b) Non : absence de l'arête $(2, 10)$ c) Non : pour 4 le graphe n'est pas connexe d) Oui e) Oui.

26. Il suffit de considérer l'arbre réduit à un unique sommet et étiquetés par tous les sommets du graphe. Les trois propriétés (i), (ii) et (iii) sont alors bien vérifiées.

$27 - 31$. Parcours de graphe pour 27 et 30. 28 et 29 direct (juste long). 31 facile.

32. a) 2 d) 3 e) 2

33. Tout d'abord, considérons le cas où l'arbre ne possède qu'un unique sommet. Dans ce cas, il est décomposition arborescente de lui-même et sa largeur arborescente est 0. De plus, si l'arbre a au moins deux sommets, alors il possède au moins une arête (u, v), et donc dans toutes ses décompositions arborescentes, il existe un sommet étiqueté par au moins u et v.

De ce fait, tout arbre possédant au moins deux sommets a une largeur arborescente supérieure ou égale à 1. Montrons que sa largeur arborescente est exactement 1.

Pour cela, commençons par enraciner notre arbre $\mathcal{T}$ à partir d'un sommet r. Pour tout sommet u autre que la racine, on note son parent $p(u)$. Considérons la fonction f qui à u associe $f(u) = \{u, p(u)\}$ et qui à r associe $f(r) = \{r\}$.

On note que pour tout sommet s de $\mathcal{T}$, $s \in f(s)$. De plus pour toute arête (s, t) de $\mathcal{T}$ on a soit s parent de t soit t parent de s. Supposons que $s = p(t)$, alors $s \in f(t)$ et $t \in f(t)$. Enfin, pour tous sommets s, t, si $s \in f(t)$ alors soit $t = s$, soit (s, t) est une arête de $\mathcal{T}$. Il en résulte que l'ensemble $\{\ t \mid s \in f(t)\ \}$ est connexe dans $\mathcal{T}$.

On en déduit donc que $(\mathcal{T}, f)$ est une décomposition arborescente de $\mathcal{T}$ de largeur 1. Donc, pour tout arbre de plus de deux sommets, sa largeur arborescente est égale à 1.

34. Soit G un cycle avec pour sommets $\{x_1, \ldots, x_k\}$ avec $k \geq 3$. Commençons par montrer qu'il n'existe pas de décomposition de largeur 1 de G. Si une telle décomposition existait, notée (T, e), on aurait les sommets $(s_i)_{1 \leq i < k}$ et s_k avec, pour $i < k$, $e(s_i) = \{x_i, x_{i+1}\}$ et $e(s_k) = \{x_1, x_k\}$. Considérons le sous-graphe de T induit par les sommets dont l'étiquette contient $x_3, x_4, \ldots, x_k$. Par la propriété (iii), il est connexe. On a donc un chemin de s_2 à s_k. Or, il existe un chemin de s_1 à s_2 et un chemin de s_1 à s_k. On a donc deux chemins distincts pour aller de s_1 à s_k donc T n'est pas un arbre, et on arrive à une contradiction. Donc $la(G) \geq 2$.

On propose maintenant la décomposition suivante : (T, e) avec $T = (S', A')$ où $S' = \{s_1, \ldots, s_{k-2}\}$, $A' = \{(s_i, s_{i+1} \mid 1 \leq i \leq k\}$ et $e(s_i) = \{s_i, s_{i+1}, s_k\}$. On note que la propriété (i) est trivialement respectée. Pour toute arête $a = (x_i, x_{i+1})$, on a $a \in e(s_i)$ et $(x_k, x_1) \in e(s_1)$ donc la propriété (ii) est vérifiée. Enfin, la connexité est également vérifiée. Ainsi, il s'agit bien d'une décomposition arborescente de G et $la(G) = 2$.

35. Par l'absurde, supposons avoir $(u, v) \in S_x \times S_y$ telle que (u, v) soit une arête de G. Il en résulte qu'il existe s, sommet de T, tel que $u \in e(s)$ et $v \in e(s)$. Par symétrie, on suppose que s soit dans T_x. On sait de plus qu'il existe $t \in T_y$ tel que $v \in e(t)$. Il en résulte qu'il existe un chemin de s à t dans T tel que pour tout sommet c de ce chemin, $v \in e(c)$. Or, on sait que sans l'arête (x, y), T_x et T_y ne sont pas connexes, donc x et y sont des éléments successifs de ce chemin.

Or, par définition de S_y on sait que $v \notin e(y)$. On trouve donc une contradiction. Ainsi, il n'existe pas d'arête entre S_x et S_y dans G.

36. Pour le montrer, on donne un algorithme de coloration glouton (Algorithme 1) qui prend en entrée un graphe G et une décomposition arborescente de largeur minimale de G (T, e) et qui renvoie une $la(G) + 1$ coloration de G. On représente les couleurs par des entiers compris entre 1 et $la(G) + 1$.

Il ne nous reste plus qu'à montrer la correction de cet algorithme. Tout d'abord, on note que chaque sommet n'est étiqueté que par au plus $la(G) + 1$ sommets et donc on peut toujours colorier les nouveaux sommets rencontrés. De plus, on définit l'invariant de boucle suivant : « Le graphe G restreint aux sommets coloriés satisfait que deux sommets reliés par une arête ne sont pas coloriés de la même couleur. »

Algorithme 1 : Coloration

Données : $G = (S, A)$ un graphe non dirigé et non vide, (T, e) une décomposition arborescente minimale de G

Résultat : Une $la(G) + 1$ coloration de G

```
1 début
2     C = Un tableau indexé par les sommets de G
3     On effectue un parcours en largeur de T
4     pour s sommet visité par T. faire
5         pour u dans e(s) à qui on n'a pas assigné de couleur. faire
6             C[u] ⟵ la première couleur non utilisée parmi les sommets de e(s) déjà coloriés
7     retourner C
```

Pour prouver que l'invariant est conservé à chaque boucle de l'algorithme, il nous suffit d'appliquer le résultat de la question précédente sur le sommet nouvellement colorié par la couleur k et les sommets déjà coloriés par k.

37. L'ensemble des sommets t tels que $s \in e(t)$ est connexe. Or s n'apparaît pas dans $e(x)$. Or tout chemin entre deux sommets dans des sous-arbres différents passe nécessairement par x par propriété d'un arbre. Donc, tous les sommets dont l'étiquette contient s sont tous dans le même sous-arbre.

38. On propose la stratégie suivante pour P. Tout d'abord, G place les gendarmes sur $e(x)$ et le voleur se place en s. On note f la racine du sous-arbre contenant des sommets de s. P déplace alors ses gendarmes sur les sommets de $e(f)$ en laissant sur place les gendarmes déjà sur un sommet de $e(f)$ et en déplaçant les autres.

On prétend maintenant que les sommets accessibles par V à partir de s sont contenus dans un sous-arbre de f ne contenant pas x, donc informellement que le voleur descend dans l'arbre. Cela découle directement de la question 35.

Ainsi, il en résulte que le voleur se retrouve coincer en bas de l'arbre et les gendarmes couvrent tous les sommets d'une feuille de T, et attrapent le voleur.