

# Mini-projet numéro 2

## Algorithmique et complexité

**Alves Dos Reis Emmanuel**

**Baudet Romain**

**Khayati Hiba**

**Bouzidi Héba**

## Table des matières

Question 1	2
Question 2	6
Question 4	7
Question 5	8
Question 6	8
Question 7	9

Note : pour les questions de ce projet où un code était demandé, ce code a été écrit en java. Vous pouvez trouver tout le code sur notre [repository GitHub](#), que nous avons passé en public au moment de rendre le projet. Tout le code contenu a été écrit par notre groupe.

## Question 1

Pour pouvoir donner les séquences rouges, nous avons donné des noms aux sommets du graphe :

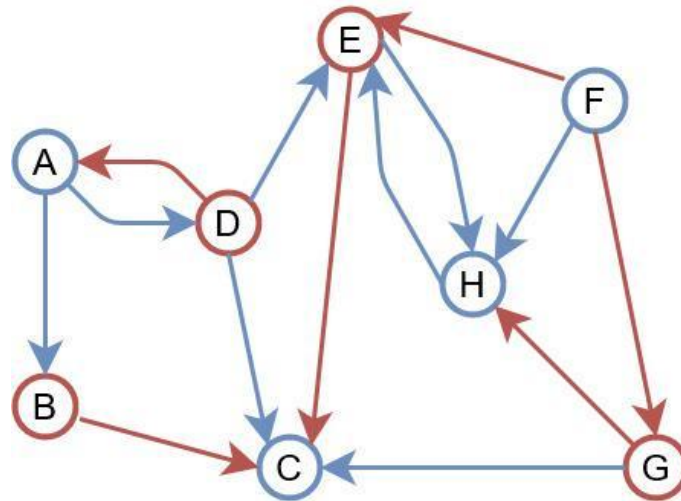


Figure 1 : graphe G avec des noms sur les nœuds pour les identifier.

Une séquence rouge pour  $k = 5$  est : (D, B, G, H, A)

Une séquence rouge pour  $k = 6$  est : (E, C, G, H, D, A)

Une séquence rouge pour  $k = 7$  est : (E, C, G, H, D, B, A)

Ci-dessous les états successifs du graphe lorsqu'on applique les séquences rouges proposées.

- **Pour  $k = 5$  :**

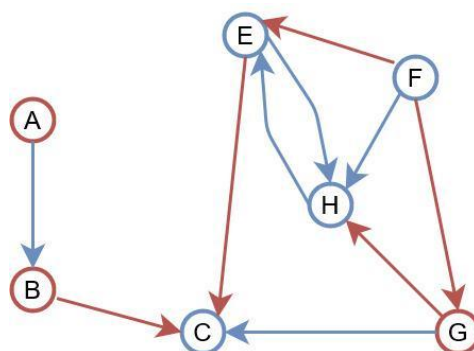


Figure 2 : graphe G après suppression du nœud D

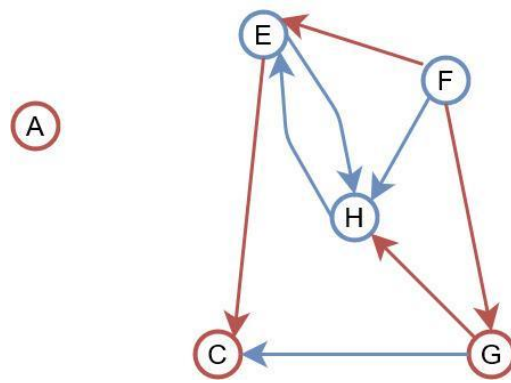


Figure 3 : graphe G après suppression des nœuds D et B

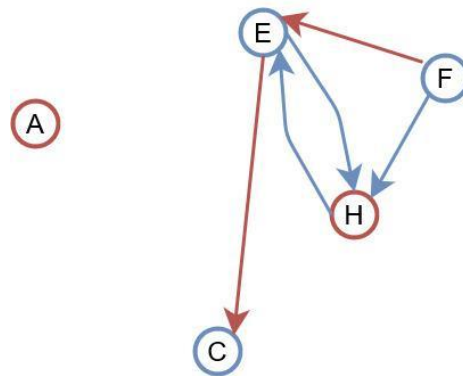


Figure 4 : graphe G après suppression des nœuds D, B et G

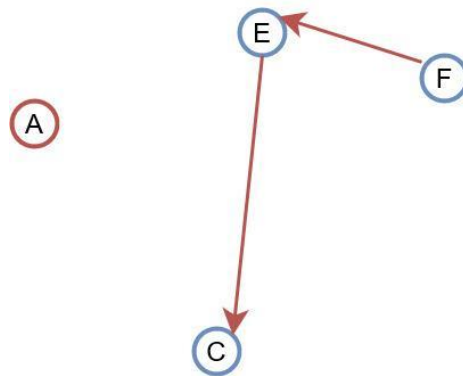


Figure 5 : graphe G après suppression des nœuds D, B, G et H

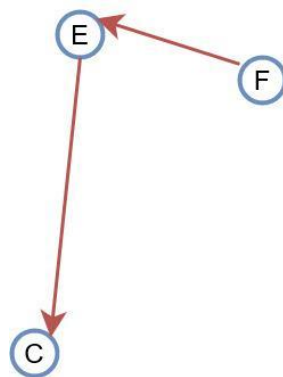


Figure 6 : graphe G après suppression des nœuds D, B, G, H et A

- Pour  $k = 6$  :

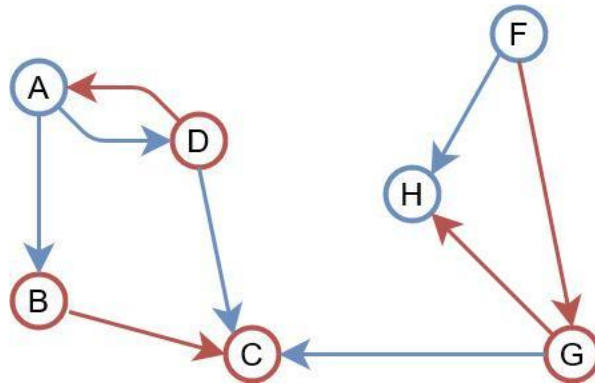


Figure 7 : graphe G après suppression du nœud E

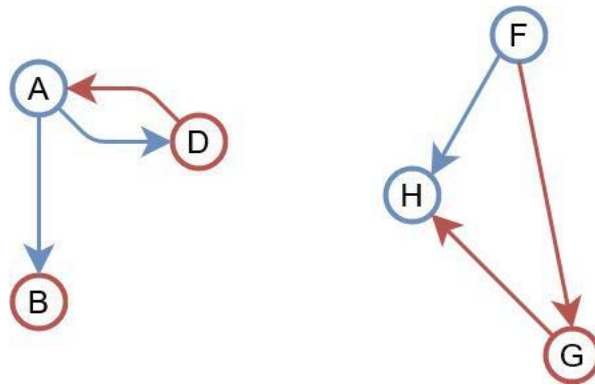


Figure 8 : graphe G après suppression des nœuds E et C

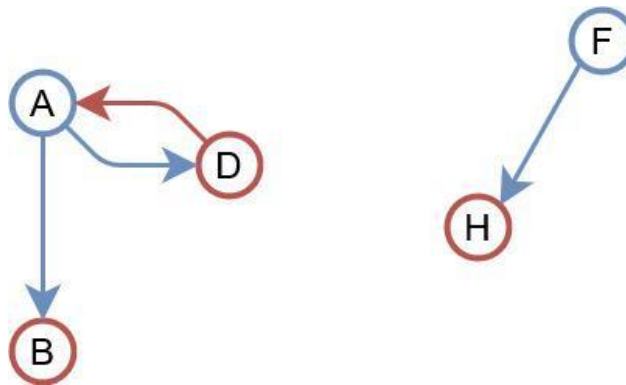


Figure 9 : graphe G après suppression des nœuds E, C et G

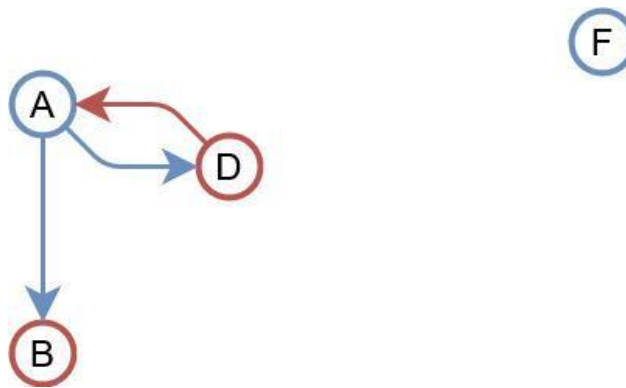


Figure 10 : graphe G après suppression des nœuds E, C, G et H



Figure 11 : graphe G après suppression des nœuds E, C, G, H et D



Figure 12 : graphe G après suppression des nœuds E, C, G, H, D et A

- **Pour  $k = 7$ ,** la première partie est la même que pour  $k = 6$ , mais on supprime B avant de supprimer A.



Figure 13 : graphe G après suppression des nœuds E, C, G, H, D et B



Figure 14 : graphe G après suppression des nœuds E, C, G, H, D, B et A.

## Question 2

Pour montrer que le problème rouge-bleu est NP complet, il faut montrer :

- Que le problème rouge-bleu est dans NP
- Que tout problème de NP peut être réduit polynomialement au problème rouge-bleu

### **Partie 1 : montrons que le problème rouge-bleu est dans NP.**

NP est la classe des problèmes qu'on sait résoudre en temps polynomial sur une machine de Turing non-déterministe.

Étant donné une instance de problème rouge-bleu, on parcourt les sommets en choisissant de manière non-déterministe les sommets à supprimer. Pour la vérification, on vérifie que les sommets restants sont tous bleus, et ensuite on compte le nombre de sommets supprimés (différence entre le nombre de sommets au départ et le nombre de sommets à la fin) pour savoir si ce dernier est supérieur ou égal à  $k$ .

### **Partie 2 : montrons que tout problème de NP peut être réduit polynomialement au problème rouge-bleu.**

On utilise pour cela le problème STABLE, qui est le suivant : étant donné un graphe fini  $G$  et un entier positif  $J$ , inférieur au nombre de nœuds du graphe, le graphe  $G$  admet-il un stable (sous-graphe vide) de cardinalité au moins  $J$  ?

La réduction d'un problème STABLE à un problème rouge-bleu est la suivante : on garde le graphe du problème STABLE, à ceci près qu'on colorie tous les sommets en rouge, et on transforme toutes les arêtes entre deux sommets en deux arcs bleus entre ces deux sommets, chacun dans un sens (en effet, on passe d'un problème où les graphes sont non orientés à un problème où les graphes sont orientés).

Cette transformation est polynomiale puisque la transformation de tous les sommets en sommets rouges se fait en  $O(n)$ , et la construction des chemins se fait en maximum  $O(n^2)$  (quand le graphe est complet). Ainsi la transformation a une complexité de  $O(n^2)$

#### Stable $\Rightarrow$ Rouge-Bleu :

Soit un graphe  $G$ . Supposons que  $G$  possède un stable de taille  $J$ . Cela signifie que chaque sommet de ce graphe ne sont pas reliés entre eux, ils ne sont donc pas voisins. Après transformation tous les sommets deviennent rouges et toutes les arêtes deviennent bleus et bidirectionnelles. Ainsi, supprimer un sommet rouge est équivalent à supprimer ses voisins en bleu (et donc on ne peut plus les supprimer). Donc si  $G$  possède un stable de taille  $J$  cela signifie que au moins  $J$  sommets ne sont pas voisins entre eux et donc que l'on peut supprimer au moins  $J$  sommets rouges et donc qu'il existe une séquence rouge de taille au moins  $J$ .

#### Rouge-Bleu $\Rightarrow$ Stable :



Soit une instance du problème rouge-bleu décrite précédemment (tous les sommets sont rouges et toutes les arêtes sont bleus et bidirectionnelle). Dans ce graphe, supprimer un sommet rouge revient à transformer ses voisins en bleus. Supposons qu'il existe une séquence rouge de taille  $J$ . Comme décrit précédemment, supprimer un sommet revient à transformer ses voisins en bleu donc s'il existe une séquence rouge de taille  $J$  alors cela signifie que  $J$  sommets ne sont pas voisins entre eux. Après transformation les arêtes du graphe ne sont pas modifiées.

Montrons par l'absurde qu'il existe un stable de taille au moins  $J$  après transformation. S'il n'existe pas de stable de taille au moins  $J$  alors il existe strictement moins de  $J$  sommets qui ne sont pas voisins entre eux. Comme il y avait une séquence rouge de taille  $J$  alors cela signifie qu'une séquence rouge de taille  $J$  doit supprimer deux sommets voisins. Or lorsque qu'un sommet est supprimé ses voisins deviennent bleus et donc il est impossible de supprimer deux sommets qui sont voisins et donc la séquence rouge ne peut pas être de taille au moins  $J$ . Donc par l'absurde s'il existe une séquence rouge de taille  $J$  alors après transformation il existe un stable de taille au moins  $J$ .

Donc comme Stable est NP-difficile et que Stable se réduit en Rouge-Bleu et que Rouge-Bleu est dans NP alors Rouge-Bleu est NP-complet.

## Question 4

- **Algorithme numéro 1 :**

On parcourt les sommets rouges du graphe, et on calcule le « score » de chaque sommet rouge de la manière suivante : un sommet a initialement un score de 0 ; pour chaque arête sortante de ce sommet : si l'arête est bleue et que le sommet d'arrivée est rouge, alors on décrémente le score de 1. Sinon, si l'arête est rouge et que le sommet d'arrivée est bleu, alors on incrémente le score de 1. Dans les autres cas, le score reste tel quel. Une fois qu'on a calculé le score de chaque sommet, on prend celui dont le score est le plus élevé, et on l'ajoute à notre séquence rouge. On continue à retirer des sommets rouges ainsi, jusqu'à ce qu'il n'en reste plus.

Dans un graphe possédant initialement  $N$  nœuds, à chaque itération, on parcourt dans le pire des cas  $N$  nœuds (dans le cas où tous les nœuds sont rouges). On répète cette étape au maximum  $N$  fois (si on réussit à enlever tous les nœuds). La complexité est donc  $n*n$  : cet algorithme est quadratique.

- **Algorithme numéro 2 :**

On parcourt les nœuds rouges du graphe, et à chaque nœud rouge on regarde si ce dernier possède au moins une arête sortante rouge pointant vers un nœud bleu. Si on en trouve un, alors on l'ajoute à la séquence rouge et on applique l'algorithme sur le graphe ainsi obtenu. Si on n'en trouve pas, alors on parcourt de nouveau les sommets rouges, en cherchant cette fois-ci un sommet n'ayant pas d'arête sortante bleue qui pointe vers un nœud rouge. Si on en trouve un, alors on l'ajoute à la séquence rouge, et on applique l'algorithme sur le graphe ainsi obtenu. Si on n'en trouve pas, alors on prend le premier nœud rouge que l'on trouve, on l'ajoute à la séquence rouge, et on applique l'algorithme sur le graphe ainsi obtenu.

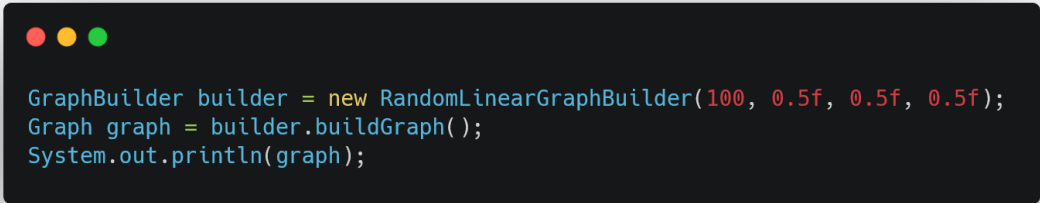
Dans un graphe possédant initialement  $n$  nœuds, à chaque itération, on parcourt dans le pire des cas  $3*n$  nœuds (puisque'il y a 3 « niveaux » de recherche de nœud candidat), et on répète cette étape au maximum  $n$  fois. La complexité est donc  $n*n$  : cet algorithme est quadratique.

Ces algorithmes devraient être relativement efficaces, car ils permettent de supprimer en priorité les sommets rouges produisant d'autres sommets rouges, et ainsi de propager la couleur rouge. Nous avons écrit un programme pour évaluer leur efficacité sur des graphes de petites taille (10 nœuds) : dans environ 96% des cas, le premier algorithme trouve la séquence rouge maximale (on le sait en comparant la taille de la séquence rouge trouvée à celle de la taille de la séquence rouge trouvée par un algorithme naïf, qui teste toutes les possibilités, dont la complexité est exponentielle mais qui va plutôt vite sur des petits graphes), et pour le second algorithme, c'est autour de 89%.

## Question 5

La fonction construisant un graphe selon les paramètres  $p$ ,  $q$  et  $r$  donnés se trouve dans la classe `src/fr/polytech/graph_builder/RandomLinearGraphBuilder.java` que vous pouvez consulter dans le repository [GitHub](#) ou bien trouver dans le dossier *code* qui se trouve dans l'archive de notre rendu.

Pour créer un graphe, il faut d'abord créer un objet `RandomLinearGraphBuilder` : son constructeur prend comme paramètres le nombre de nœuds (fixé à 100 pour cette question),  $p$ ,  $q$  et  $r$ . Ensuite, on peut appeler la méthode `buildGraph()` de cet objet pour récupérer un graphe construit aléatoirement selon les paramètres donnés précédemment. Exemple ci-dessous :



```
GraphBuilder builder = new RandomLinearGraphBuilder(100, 0.5f, 0.5f, 0.5f);
Graph graph = builder.buildGraph();
System.out.println(graph);
```

## Question 6

Les algorithmes décrits dans la question 4 sont disponibles dans les classes suivantes :

- `src/fr/polytech/solver/RedBlueMaximizationScoreBasedSolver.java` pour le premier algorithme
- `src/fr/polytech/solver/RedBlueMaximizationBasicSolver.java` pour le second algorithme

Ils sont consultables dans notre repository [GitHub](#) ou bien dans le dossier *code* qui se trouve dans l'archive de notre rendu.

Ils sont encapsulés dans des objets de type `RedBlueMaximizationSolver` : ainsi, pour appliquer l'un de ces algorithmes sur un graphe, il suffit d'instancier la classe dans laquelle il se trouve, puis d'appeler la méthode `solve()` du graphe en lui passant en paramètre l'instance de l'algorithme créée précédemment. Exemple ci-dessous :

```
GraphBuilder builder = new RandomLinearGraphBuilder(100, 0.5f, 0.5f, 0.5f);
Graph graph = builder.buildGraph();

RedBlueMaximizationSolver solver = new RedBlueMaximizationScoreBasedSolver();
int tailleSequenceRouge = graph.solve(solver);
System.out.println(tailleSequenceRouge);
```

## Question 7

- **Algorithme 1 :**

P Q	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	0	0	0	0	0	0	0	0	0	0	0
0.1	10	10.91	11.91	13.02	14.22	15.57	17.12	18.84	20.68	22.84	25.32
0.2	19.98	21.57	23.29	25.15	27.20	29.40	31.84	34.49	37.39	40.57	44.07
0.3	29.98	32.00	34.18	36.53	39.04	41.69	44.62	47.71	50.97	54.43	58.19
0.4	39.97	42.24	44.63	47.17	49.87	52.71	55.73	58.86	62.16	65.49	69.06
0.5	50.01	52.23	54.64	57.19	59.82	62.57	65.47	68.41	71.42	74.42	77.48
0.6	59.99	62.10	64.30	66.61	69.03	71.49	74.01	76.60	79.11	81.68	84.16
0.7	69.99	71.76	73.63	75.58	77.55	79.61	81.61	83.65	85.65	87.58	89.46
0.8	79.99	81.31	82.66	84.08	85.53	86.99	88.42	89.81	91.19	92.50	93.70
0.9	89.98	90.70	91.43	92.19	92.97	93.75	94.49	95.22	95.92	96.58	97.17
1	100	100	100	100	100	100	100	100	100	100	100

- **Algorithme 2 :**

P Q	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
0	0	0	0	0	0	0	0	0	0	0	0
0.1	10	10.89	11.87	12.96	14.14	15.5	17.03	18.77	20.62	22.81	25.33
0.2	19.98	21.51	23.15	24.95	26.96	29.14	31.56	34.24	37.19	40.45	44.07
0.3	29.98	31.85	33.92	36.16	38.59	41.21	44.13	47.27	50.63	54.23	58.18
0.4	39.97	42.02	44.22	46.62	49.22	52.03	55.05	58.26	61.69	65.24	69.06
0.5	50.01	51.95	54.13	56.51	59.03	61.74	64.66	67.71	70.86	74.12	77.47
0.6	59.99	61.77	63.73	65.85	68.16	70.60	73.15	75.86	78.56	81.36	84.16
0.7	69.99	71.44	73.05	74.84	76.72	78.75	80.81	82.96	85.14	87.30	89.45
0.8	79.99	81.03	82.17	83.44	84.83	86.28	87.77	89.26	90.80	92.29	93.70
0.9	89.98	90.53	91.13	91.82	92.54	93.33	94.12	94.91	85.70	96.47	97.17
1	100	100	100	100	100	100	100	100	100	100	100

On voit que les moyennes obtenues avec l'algorithme 1 sont toujours meilleures que celles de l'algorithme 2 ; cependant, lorsque Q vaut 1, les moyennes sont quasiment identiques pour les deux algorithmes.

Ces résultats devraient être assez précis, car nous avons pour les obtenir réalisé un grand nombre de simulations. La manière de faire était la suivante : l'utilisateur spécifie un nombre de simulations N par case au moment de lancer le programme, et pour chaque case du tableau N graphes sont générés puis résolus par nos deux algorithmes. Les résultats sont écrits dans un fichier dont le chemin d'accès est également passé en paramètre, au format suivant : une ligne représente une simulation, et on y écrit p, q et la taille de la séquence rouge obtenue de la manière suivante : p|q|taille\_sequence . Une fois tout

le tableau fait, le programme recommence, sauf si l'utilisateur lui a demandé de s'arrêter. Nous avons donc laissé ce programme tourner assez longtemps sur un serveur pour effectuer un maximum de simulations, afin que les moyennes obtenues se rapprochent de la moyenne théorique, et ainsi avoir des estimations plus précises. [Ce programme est disponible sur notre repository en tant que JAR exécutable](#), et [le programme source est également disponible](#), et des exemples de commandes d'exécution avec les bons paramètres sont présents. Une fois les résultats obtenus, nous utilisons un autre programme pour lire ces fichiers de résultats (1 par algorithme), calculer la moyenne pour chaque case du tableau, et produire un fichier final au format CSV correspondant au tableau. De nouveau, [ce programme est sur notre repository](#), et [le programme source également](#).