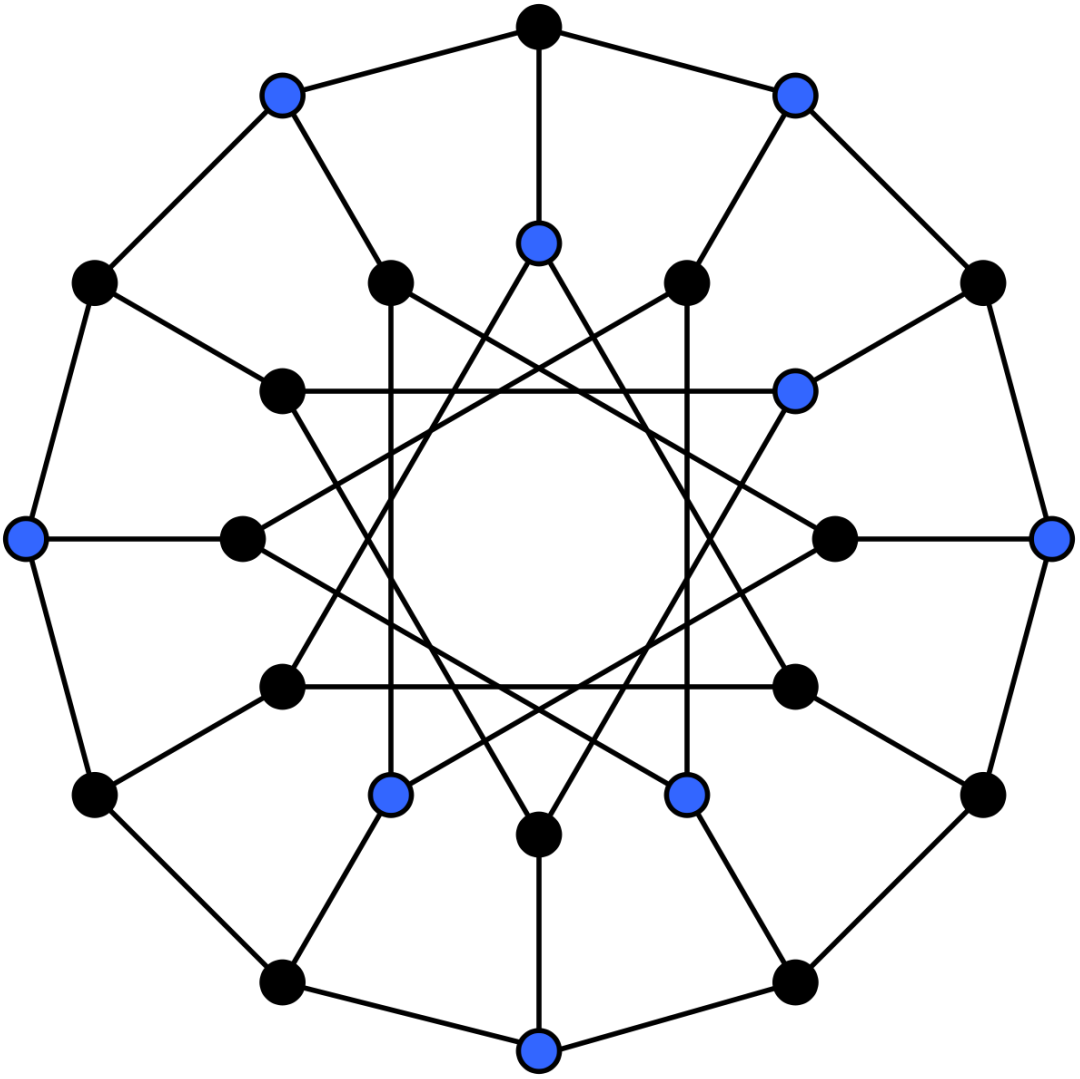
Rapport

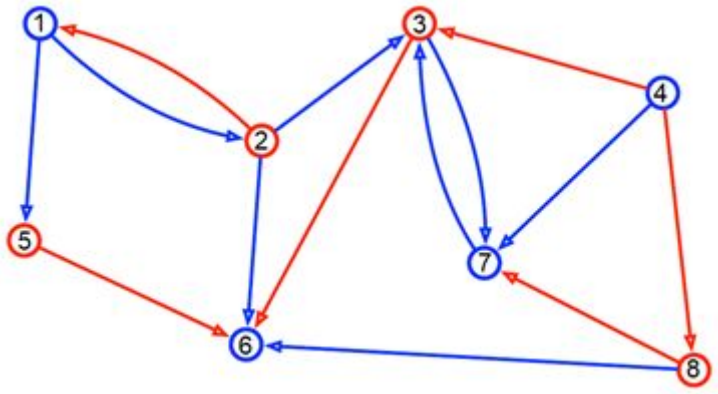
Mini-Projet 2 : Probleme Rouge et Bleu



Membre de l'équipe :

Julien **Kaplan** -- Rachid **El adlani** -- Abdelouhab **Belkhiri** -- Léo **Marache**

Question 1



Pour k = 5 la séquence rouge est {3,2,1,8,7}
Pour k = 6 la séquence rouge est {2,5,1,6,8,7}
Pour k = 7 la séquence rouge est {3,2,5,1,6,8,7}

Question 2

On peut affirmer que notre problème peut être considéré dans NP, c'est un problème solvable en temps polynomial par une machine non-déterministe, du fait que si l'on souhaite trouver une solution à notre problème avec un nombre de sommet très grand dans le graphe, cela sera impossible à résoudre en temps polynomial par une machine déterministe. Cependant, on peut vérifier en temps polynomial que le résultat renvoyé est juste.

Montrons que le problème rouge-bleu est NP-Complet

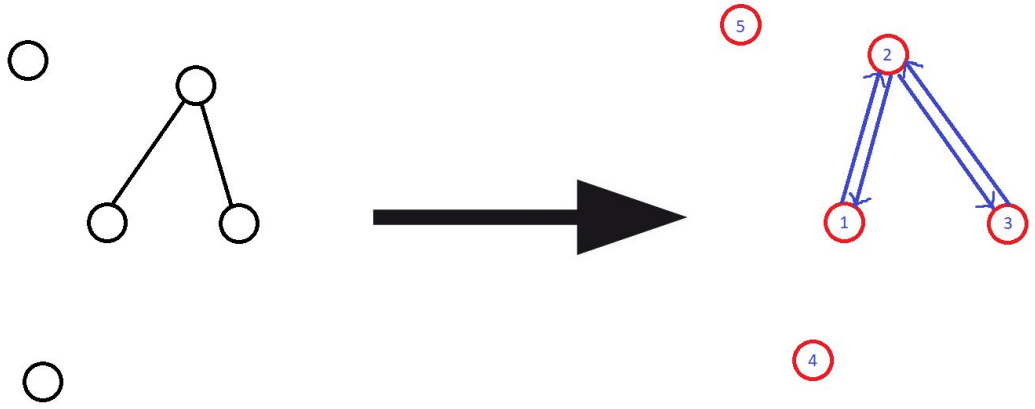
Maintenant montrons que le problème rouge-bleu est NP-complet. On a choisi de réduire notre problème au problème **Stable**:
Soit G un graphe non-orienté dont on cherche un stable de taille au moins K.
On construit G' à partir de G en dédoublant chaque arête en une paire d'arc (car nous voulons un graphe orienté pour rouge-bleu) on colore :

- tous les nœuds en rouge
- tous les arcs en bleu.

Cette transformation est polynomiale puisqu'elle effectue un nombre d'actions égal au nombre d'arêtes x2 plus le nombre de nœuds du graphe d'origine.

S'il existe une séquence rouge dans G', alors les nœuds de cette séquence forment un stable dans G (et réciproquement). Il y a équivalence entre séquence rouge et stable car dès que l'on prend un sommet rouge, tous ses voisins deviennent bleus et ce sommet est isolé, il formera donc un stable. Réciproquement si dans le graphe de départ un nœud est sélectionné pour faire partie du stable, alors aucun de ses voisins n'est sélectionné, il doit donc faire partie de la séquence rouge car aucun de ses voisins n'aura été pris et ne l'aura colorié en bleu.

exemple de transformée d'une instance de stable vers une instance d'un rouge-bleu



Question 3

Pseudo code de l'algorithme :

Règle : le noeud ne change pas un voisin de rouge vers bleu

Variables: ListeNoeudsRouges = la liste des noeuds rouges du graphe

Entrée : un entier *k* , un graphe

Début :

On choisit un noeud rouge à supprimer dans ListeNoeudsRouges, il faut que ce noeud suive la **règle** (décrite au-dessus)

Tant que il existe des noeuds rouges dans ListeNoeudsRouges

on supprime le noeud sélectionné

on met à jour ListeNoeudsRouges

on met à jour le noeud à supprimer qui sera le prochain noeud rouge dans ListeNoeudsRouges qui suit la règle

Si la variable k est inférieure ou égale au nombre de sommets rouges supprimés
On renvoie vrai
Sinon
On renvoie faux

Sortie : true ou false

Preuve de l'algorithme

Pour prouver l'algorithme il suffit de prouver qu'il est possible de supprimer tous les noeuds rouges d'un graphe avec la règle suivante: "retirer le noeud seulement s'il ne change pas un voisin de rouge vers bleu"

Preuve que la règle traite tous les cas:

Si un noeud rouge ne suit pas la règle, on regarde le voisin qui provoque le blocage, s'il ne suit toujours pas la règle on regarde encore son voisin, et vu qu'on a un graphe "linéaire" et que le graphe n'est pas infini, il y aura forcément un noeud rouge qui n'aura soit pas de voisin soit un voisin bleu, donc on pourra supprimer ce noeud. Cette suppression aura pour conséquence que le noeud précédent suivra désormais la règle et donc pourra être supprimé lui aussi.

Preuve qui si l'on suit la règle, la séquence rouge est maximale:

L'algorithme ne retire pas les nœuds qui changent un voisin de rouge vers bleu ce qui veut dire que tous les nœuds bleus pouvant être transformés en rouges subiront cette transformation et qu'ils seront ajoutés dans la séquence des rouges, on aura donc le nombre maximal de nœuds rouges possible.

Complexité de l'algorithme:

Chaque parcours du graphe va supprimer un et un seul nœud, il y a donc au plus N parcours. Et un parcours correspond à au plus N opérations de temps constant. Donc la complexité est de $N + N-1 + N-2 + \dots + 2 + 1$ soit $O(N^2)$. Cet algorithme est bien polynomial.

Question 4

Heuristique 1

L'idée est de tester la suppression de chacun des nœuds, et d'ensuite supprimer le nœud qui change le plus de voisins Bleu à Rouge. On effectue cette étape tant qu'il y a des nœuds rouges dans le graphe.

Cet algorithme est efficace puisqu'il n'est pas d'une grande complexité (voir la preuve plus bas dans le rapport) et qu'il est basé sur une règle de bon sens: on supprime le noeud qui fait apparaître le plus de nouveaux noeuds rouges, ce qui est une bonne chose quand on sait que l'on souhaite avoir la plus longue séquence rouge à la fin.

Voici le pseudo-code de l'heuristique 1 :

Entrée : Un graphe complet symétrique
Début :
On initialise un noeud à supprimer ***NoeudASuppr***

Tant que il y a des noeuds rouges dans le graphe
On initialise un noeud actuel ***NoeudCourant***
On initialise un compteur ***BleuARouge*** comptant le maximum de changement de sommet bleu vers rouge
Pour tous les noeuds rouges Nr du graphe
Faire
On initialise un nombre nb au nombre de sommets bleus transformés en rouges si on supprime Nr
Si *BleuARouge* est inférieur à nb alors
NoeudCourant ← ***NoeudASuppr***
BleuARouge ← **nb**
Sinon
On continue
Fin Pour
On supprime ***NoeudASuppr***
Fin Tant Que
On renvoie le nombre de noeuds supprimés
Fin

Heuristique 2

L'idée est de tester la suppression de chacun des nœuds, et d'ensuite supprimer le nœud qui change le moins de voisins de Rouge à Bleu. On effectue cette étape tant qu'il y a des nœuds rouges dans le graphe.

Cet algorithme est efficace puisqu'il n'est pas d'une grande complexité (voir la preuve plus bas dans le rapport) et qu'il est basé sur une règle de bon sens: on supprime le noeud qui fait disparaître le moins de noeuds rouges, ce qui est une bonne chose quand on sait que l'on souhaite avoir la plus longue séquence rouge à la fin.

Voici le pseudo-code de l'heuristique 2 :

Entrée : Un graphe complet symétrique
Début :
On initialise un noeud à supprimer ***NoeudASuppr***

Tant que il y a des noeuds rouges dans le graphe
On initialise un noeud actuel ***NoeudCourant***
On initialise un compteur ***RougeABleu*** comptant le maximum de changement de sommet rouge vers bleu
Pour tous les noeuds rouges Nr du graphe
Faire
On initialise un nombre *nb* au nombre de sommets rouges transformés en bleus pour le noeud Nr
Si *RougeABleu* est supérieur à nb alors
NoeudCourant ← ***NoeudASuppr***
RougeABleu ← **nb**
Sinon
On continue
Fin Pour
On supprime ***NoeudASuppr***
Fin Tant Que
On renvoie le nombre de noeuds supprimés
Fin

Heuristique 3 (bonus)

L'idée est de tester la suppression de chacun des nœuds, et d'ensuite supprimer le nœud qui change le moins de voisins de rouge à bleu et qui change le plus de voisins bleus à rouge. On effectue cette étape tant qu'il y a des nœuds rouges dans le graphe.

Cet algorithme est efficace puisqu'il n'est pas d'une grande complexité (voir la preuve plus bas dans le rapport). Il suit une règle de bon sens: supprimer le nœud qui donnera le graphe avec le plus de vertex rouges possible (c'est-à-dire qui change le moins de voisins de Rouge à Bleu et qui change le plus de voisins bleus). C'est en quelque sorte une combinaison des deux heuristiques précédentes.

Voici le pseudo-code de l'heuristique 3 :

Entrée : Un graphe complet symétrique
Début :
On initialise un noeud à supprimer ***NoeudASuppr***

Tant que il y a des noeuds rouges dans le graphe
On initialise un noeud actuel ***NoeudCourant***
On initialise un compteur **Moins*RougeABleuEtPlusBleuARouge*** comptant le maximum de changement de sommet rouge vers bleu
Pour tous les noeuds rouges Nr du graphe
Faire
On initialise un nombre *nb* qui fait le compte avec une différence entre le nombre de sommets rouges transformés en bleus et le nombre de transformé de bleu à rouge pour le noeud Nr
Si Moins*RougeABleuEtPlusBleuARouge* est supérieur à nb alors
NoeudCourant ← ***NoeudASuppr***
RougeABleu ← **nb**
Sinon
On continue
Fin Pour
On supprime ***NoeudASuppr***
Fin Tant Que
On renvoie le nombre de noeuds supprimés
Fin

La Complexité de nos 3 heuristiques:

Ces trois heuristiques ont la même complexité:
Chaque parcours du graphe va supprimer un et un seul nœud, il y a donc au plus N parcours. Pour chacun de ces parcours on va regarder tous les noeuds (au plus N noeuds) et pour chacun de ces noeuds on va regarder tous ses voisins (au plus N-1 voisins), donc la complexité est de N x (N-1) + (N-1) x (N-2) + (N-2) x (N-3) + … + 3x2 + 2x1 soit **O(N^3)**.

Questions 5 et 6

Voir le code sur le repo Github dans la branche main:
https://github.com/JulienK-hub/algo-miniprojet2/tree/main
ou bien le fichier .zip dans le dossier du rapport.

Question 7

Tableau de **l'heuristique 1** en fonction des valeurs de p et q

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	100	99	99	98	98	97	96	93	90	77	1
0.2	100	99	98	98	97	97	96	93	90	78	1
0.3	100	99	99	98	97	97	95	93	89	77	1
0.4	100	99	99	98	98	96	96	93	89	78	1
0.5	100	99	99	98	98	97	95	93	90	77	1
0.6	100	99	99	98	98	97	95	97	90	78	1
0.7	100	99	99	98	97	97	96	93	89	77	1
0.8	100	99	99	98	98	96	95	93	89	78	1
0.9	100	99	99	98	98	97	95	94	89	78	1
1	100	99	99	98	98	97	96	93	90	77	1

Tableau de **l'heuristique 2** en fonction des valeurs de p et q

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	100	99	99	99	98	97	96	95	89	74	1
0.2	100	99	99	98	98	97	95	93	89	75	1
0.3	100	99	99	99	98	97	96	94	88	74	1
0.4	100	99	99	99	98	97	96	94	90	76	1
0.5	100	99	99	98	98	97	96	93	90	73	1
0.6	100	99	99	98	98	97	96	94	89	75	1
0.7	100	99	99	99	98	97	96	94	89	74	1
0.8	100	99	99	98	98	98	96	93	89	75	1
0.9	100	99	99	99	98	97	96	93	88	74	1
1	100	99	99	99	98	97	96	94	89	74	1

Tableau de **l'heuristique 3** en fonction des valeurs de p et q (bonus)

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	100	100	100	100	99	99	99	99	97	84	1
0.2	100	100	100	99	99	99	99	99	97	84	1
0.3	100	100	100	100	99	99	99	98	96	83	1
0.4	100	100	100	99	99	99	99	98	97	85	1
0.5	100	100	100	100	99	99	99	99	97	83	1
0.6	100	100	100	100	99	99	99	98	96	84	1
0.7	100	100	100	100	100	99	99	98	97	83	1
0.8	100	100	100	100	99	99	99	98	97	83	1
0.9	100	100	100	100	99	99	99	99	96	85	1
1	100	100	100	100	99	99	99	98	96	83	1

Nos heuristiques sont de même complexité. Elles prennent toutes en moyenne 20 secondes pour compléter le tableau. Pour les comparer nous nous intéresserons à leurs fiabilités qui se résument par les tableaux.

Entre les heuristiques 1 et 2, on remarque qu'elles sont assez similaires mis-à-part pour q = 0.9, où l'heuristique 1 semble légèrement meilleure en termes de fiabilité.

Quant à l'heuristique 3, dans chaque cas, elle est identique ou meilleure dans ses résultats que les deux heuristiques 1 et 2. L'heuristique 3 est la combinaison de ces deux dernières, ce n'est donc pas surprenant.