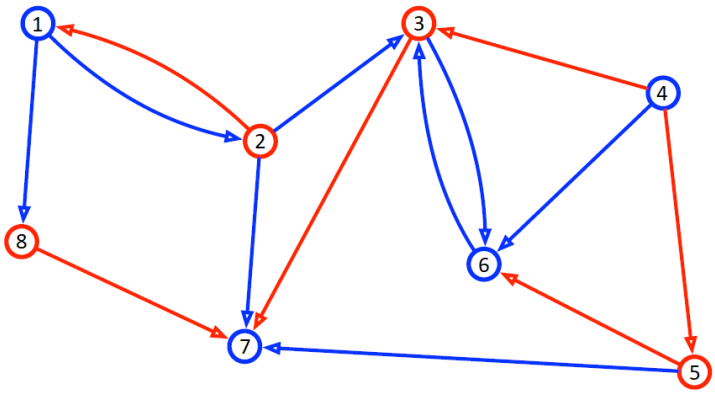
Mini-projet numéro 2 – Algorithmes et Complexité.

Algorithmes pour le Jeu Rouge Bleu

Marcel Marsais–Lacoste, Tom Niget, Thomas Prévost, Emmeline Vouriot

1. Solutions pour valeurs particulières de k



Le problème ***ROUGE-BLEU*** pour le graphe ci-dessus admet des solutions pour :

- $k = 5$: $(2, 8, 1, 5, 6)$
- $k = 6$: $(2, 8, 1, 7, 5, 6)$
- $k = 7$: $(3, 2, 5, 6, 8, 1, 7)$

2. NP-complétude du problème

Appartenance à NP

Il existe pour ce problème un certificat polynomial : il suffit en effet de donner la suite des nœuds à supprimer. Pour vérifier que ce certificat est valide, il ne reste plus qu'à supprimer les nœuds de la liste un par un. Si aucun nœud bleu n'a été supprimé, c'est que le certificat est bien valide.

NP-difficulté

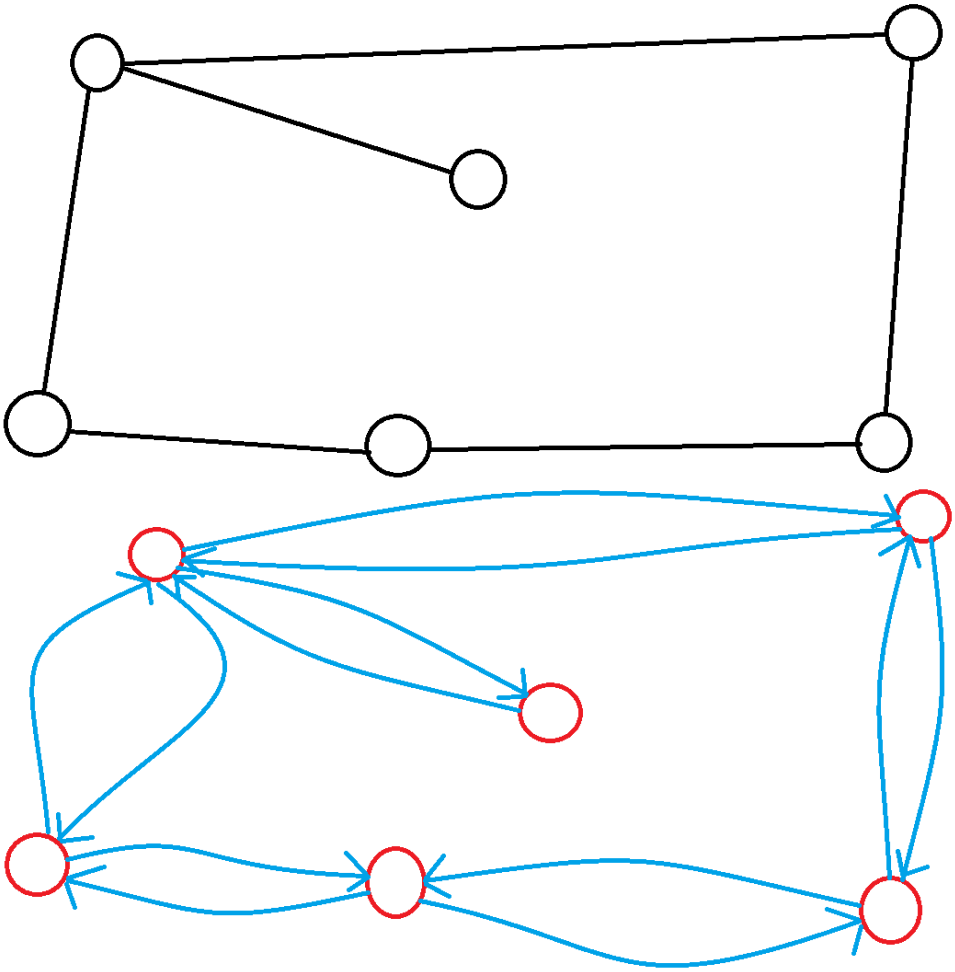
STABLE (NP-complet)	***ROUGE-BLEU***
un graphe fini non orienté $G(V, E)$	un graphe fini orienté bicolore $G'(V, A, c)$
un entier positif $J \leq \|V\|$	un entier positif k

Soit $s(G)$ une instance de ***STABLE***. On réduit ***STABLE*** au problème ***ROUGE-BLEU***.

La transformation en un graphe G' fini orienté bicolore consiste à changer chaque arête en 2 arcs bleus de sens opposés, puis à colorer tous les nœuds en rouge. Lorsqu'on supprime un nœud rouge de G, les nœuds de $s(G)$ restent rouges, et les nœuds voisins des nœuds de $s(G)$ (qui ne se trouvent pas dans $s(G)$) deviennent bleus, et donc ne pourront pas être utilisés par la suite. Ainsi le nœud rouge sera ajouté au graphe stable (graphe avec des nœuds qui n'ont aucune liaison entre eux) afin d'obtenir à la fin un graphe stable avec J nœuds ou plus. On se retrouve donc bien à la fin avec $s(G)$ contenant seulement des nœuds rouges, avec une séquence rouge résolvant le problème ***ROUGE-BLEU*** en G', et avec $k = J$.

Par construction, toute instance positive de ***STABLE*** sera une instance positive de ***ROUGE-BLEU***, et, réciproquement, étant donnés G et G', si G contient un graphe G'_k résolvant ***ROUGE-BLEU***, alors il contient un stable de J nœuds avec $k = J$.

Le caractère polynomial de la transformation est ainsi immédiat.
Le problème ***STABLE*** étant NP-complet, et pouvant être réduit au problème ***ROUGE-BLEU***, alors on peut dire que le problème ***ROUGE-BLEU*** est NP-difficile. ■



NP-complétude

Finalement, le problème ***ROUGE-BLEU*** appartient à NP, et est NP-difficile.
Donc le problème ***ROUGE-BLEU*** est NP-complet. ■

4. Algorithmes polynomiaux pour la maximisation

Algorithme 1 – `FlatGraph::getSequenceMax`

Le premier algorithme fait un seul parcours, du nœud ayant l'id le plus petit au nœud ayant l'id le plus grand.

Lors de son parcours, chaque nœud rouge sera traité en fonction de 3 facteurs : la couleur des arcs autour de celui-ci, l'orientation de ces arcs, et la couleur de ses nœuds voisins en fonction de l'orientation des arcs (arc orienté vers un de ces nœuds).

Il y a donc 4 conditions, entraînant 2 types de suppressions de nœuds rouges (un type de suppression avec parcours à droite, et un sans parcours, en fonction du cas) :

- Si les 2 arcs autour du nœud rouge courant sont sortants et rouges, et que les 2 nœuds voisins sont bleus, alors le nœud courant rouge peut être supprimé car cela entraînera le changement de couleur des 2 nœuds bleus, voisins du nœud rouge courant, en rouge. Le prochain nœud courant sera le nœud d'id inférieur, car ce nœud a changé de couleur, il est passé de bleu à rouge. Il faut donc le traiter à son tour ;

Ci-dessous, un exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud rouge 3. Le nœud 3 sera donc supprimé. Les nœuds 2 et 4 deviennent ensuite rouges, donc le prochain nœud à traiter sera le nœud d'id inférieur, le nœud 2.

- Si un arc est sortant du nœud rouge courant, qu'il est rouge, qu'il pointe le nœud voisin d'id supérieur, et que le nœud de destination de cet arc est bleu, et si, en même temps, il n'y a pas d'arc rouge sortant du nœud rouge courant pointant un nœud bleu d'id inférieur, alors le nœud courant rouge peut être supprimé, car il n'affectera pas la partie gauche du graphe (du fait de cette condition). Le prochain nœud courant sera le nœud d'id supérieur ;

Ci-dessous, un exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud rouge 3. Le nœud 3 sera donc supprimé. Le nœud 4 devient ensuite rouge. Le prochain nœud à traiter sera le nœud d'id supérieur, le nœud 4.

- Si un arc A est sortant du nœud rouge courant, qu'il est rouge, qu'il pointe le nœud voisin d'id inférieur, et que le nœud de destination de cet arc est bleu, et si, en même temps, il n'y a pas d'arc B rouge sortant du nœud rouge courant pointant un nœud bleu d'id supérieur, alors il faut faire un parcours dans le sens opposé de l'arc A rouge sortant (schématiquement un parcours à droite, c'est-à-dire vers les nœuds ayant des ids plus grands). Tant que le parcours passe par un arc *bien orienté* (c'est-à-dire dans le sens du parcours) et bleu, et que le nœud suivant est rouge, on continue le parcours. Dès lors que l'arc suivant n'est pas *bien orienté*, on arrête le parcours et on remonte tous les nœuds rouges en les supprimant dans l'ordre de la remontée, jusqu'à la suppression du premier nœud rouge qui est considéré comme le nœud « courant » de la condition. Le prochain nœud courant sera le nœud d'id inférieur, car ce nœud a changé de couleur, il est passé de bleu à rouge. Il faut donc le traiter à son tour ;

Ci-dessous, un exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud rouge 2. Il y a parcours à droite, et donc suppression ensuite, dans l'ordre, des nœuds 3 puis 2. Le nœud 1 devient ensuite rouge, donc il sera le prochain nœud à traiter (le nœud 1 étant le nœud d'id inférieur au nœud 2 rouge courant supprimé).

- Si les 2 arcs autour du nœud rouge courant ne sont pas simultanément sortants et rouges, et que, en même temps, les 2 nœuds voisins ne sont pas simultanément bleus, il suffit de faire le même parcours que précédemment (avec à la fin le nœud rouge courant supprimé, car il n'affectera pas la partie gauche du graphe qui a déjà été traitée aux itérations précédentes). Le prochain nœud courant sera le nœud d'id supérieur ;

Ci-dessous, un exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud rouge 3. Le nœud 3 sera donc supprimé, sans parcours à droite. Le prochain nœud à traiter sera le nœud d'id supérieur, le nœud 4.

Ci-dessous, un autre exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud rouge 2. Il y a parcours à droite, et donc suppression ensuite, dans l'ordre, des nœuds 3 puis 2. Le prochain nœud à traiter sera le nœud d'id supérieur, le nœud 3. Ce dernier n'existant plus, le prochain nœud à traiter sera donc le nœud 4.

- Sinon, pour tous les autres cas (nœud courant n'existant plus ou bleu), on passe au traitement du prochain nœud d'id supérieur.

Ci-dessous, un exemple de sous-graphe illustrant ce cas, avec comme nœud courant le nœud bleu 3. Le prochain nœud à traiter sera le nœud d'id supérieur, le nœud 4.

Cet algorithme est donc trivialement de complexité $O(n)$ (pire cas : $3n$, meilleur cas : n).

Algorithme 2 - `FlatGraph::getSequenceMaxBis`

Le deuxième algorithme se base sur le tri par insertion. En effet, il consiste à comparer tous les nœuds entre eux afin de déterminer ceux qui doivent être supprimés en priorité.

Admettons que l'on cherche à supprimer les nœuds rouges. Un nœud A doit être supprimé avant un autre nœud B si :

- La suppression du nœud A change la couleur du nœud B de bleu en rouge (car un arc rouge relie B à A)
- La suppression du nœud B change la couleur du nœud A de rouge en bleu. Auquel cas, le nœud A doit être supprimé avant que le nœud B ne change la couleur de A.

À la première itération, on part de la liste des nœuds dans l'ordre original. On prend le dernier nœud N que l'on fait « remonter » tout au début de la liste. On fait la même chose pour tous les nœuds suivants, qui s'arrêtent de monter dès lors qu'ils rencontrent un nœud devant être supprimé avant eux.

Exemple. Les nœuds en vert correspondent à la partie déjà triée. Le nœud B doit être supprimé avant le nœud G, il n'y a aucune contrainte entre le nœud G et les nœuds A, B et D. On commence par faire remonter le nœud G jusqu'à la fin de la liste triée (A, B, C et D).

A	B	C	D	E	F	G
A	B	C	D	G	E	F

On fait ensuite remonter G grâce à un tri par insertion, il remonte donc jusqu'à B.

A	B	C	G	D	E	F
A	B	G	C	D	E	F

On voit bien ici qu'on ne peut pas utiliser un algorithme de tri plus efficace de complexité $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$. En effet les relations de comparaison ne sont absolument pas transitives. Il est donc nécessaire de comparer tous les nœuds entre eux.

En outre, cet algorithme ne peut fonctionner que sur des graphes chemins (c.-à-d. où chaque nœud est de degré au plus 2), comme l'exemple fourni dans la question 2. Il est, entre autres, absolument incapable de gérer les cycles.

À la fin des itérations, il demeure néanmoins possible d'avoir à la fin des nœuds « insupprimables », c'est-à-dire qu'ils resteront bleus quoi qu'il arrive. C'est la raison pour laquelle, après le tri, l'algorithme effectue une copie du graphe et essaie de supprimer les nœuds dans l'ordre précédemment indiqué. S'il s'aperçoit qu'un nœud reste bleu malgré tout, il est simplement retiré de la liste.

5. Génération aléatoire de graphes

Voir code en annexe.

6. Implémentation des algorithmes

Voir code en annexe.

7. Mesures d'efficacité des algorithmes

Nos deux algorithmes fournissent systématiquement des chaînes de même longueur mais dont l'ordre peut différer.

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	11	12	13	14	15	17	18	21	22	26
0,2	19	21	23	24	27	30	31	34	37	40	44
0,3	31	32	33	38	40	42	46	49	51	54	59
0,4	40	42	45	47	51	53	55	59	63	65	69
0,5	50	53	55	57	62	63	64	68	71	74	78
0,6	59	62	65	67	70	72	75	77	80	83	84
0,7	70	72	75	76	78	82	82	84	86	88	89
0,8	79	81	83	85	86	88	89	90	91	93	94
0,9	90	91	92	93	93	94	95	95	96	97	97
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	11	12	13	14	15	17	18	21	22	26
0,2	19	21	23	24	27	30	31	34	37	40	44
0,3	31	32	33	38	40	42	46	49	51	54	59
0,4	40	42	45	47	51	53	55	59	63	65	69
0,5	50	53	55	57	62	63	64	68	71	74	78
0,6	59	62	65	67	70	72	75	77	80	83	84
0,7	70	72	75	76	78	82	82	84	86	88	89
0,8	79	81	83	85	86	88	89	90	91	93	94
0,9	90	91	92	93	93	94	95	95	96	97	97
1	100	100	100	100	100	100	100	100	100	100	100

8. Algorithme polynomial exact (bonus)

Le deuxième algorithme fonctionne sur le même principe que le tri par insertion. On commence par une liste contenant les numéros des nœuds. Plus un numéro est au début de la liste, plus il sera supprimé tôt. Pour chaque nœud, et en partant de la fin, on le fait remonter dans la liste en le comparant aux autres nœuds déjà triés. On arrête de monter dès lors que l'on trouve un nœud qui doit être supprimé avant (si sa suppression permet au nœud courant de prendre la bonne couleur ou si la suppression du nœud courant donne la mauvaise couleur au nœud comparé).

La complexité du tri par insertion est quadratique donc polynomiale.

Démonstration. Le premier nœud à être placé dans la liste triée ne sera comparé avec aucun autre. Le $2^{\text{ème}}$ sera comparé avec 1 nœud. Le $3^{\text{ème}}$ sera comparé avec les 2 précédents nœuds etc... Il y a donc $1 + 2 + 3 + \cdots + (n-1)$ ou $\sum_{i=1}^{n-1} i$ comparaisons, avec n le nombre de nœuds dans le graphe. Le fait de déplacer et de permuter l'itérateur dans la liste ayant toujours la même complexité ne rentre pas en compte dans notre analyse.

Selon la formule de la somme de Gauss, nous avons donc $n \cdot (n-1) \div 2$ comparaisons entre les nœuds, ce qui équivaut à $(n^2 - n) \div 2 = O(n^2)$ comparaisons.

L'algorithme est donc de complexité quadratique ; en outre, il est polynomial. ■