TP1 Recherche opérationnelle : Métaheuristiques de voisinage

Dans ce TP, nous allons implémenter deux métaheuristiques afin de résoudre le problème de coloration de graphe. La première méthode sera une méthode de descente, et la seconde consistera en l'un des premiers algorithmes vraiment efficaces pour ce problème (Hertz et De Werra, 1987), portant le nom de TABUCOL, et basée, comme son nom l'indique, sur une recherche Tabou.

1. Environnement de développement

Dans ce TP, vous allez coder en Java SE en utilisant la librairie Graphstream. Cette librairie permet de visualiser et de travailler sur des graphes.

Pour commencer le TP, nous vous donnons sur la page du cours 3 .jar (Librairies.zip) à ajouter à votre projet pour qu'il compile avec cette librairie. Par ailleurs, nous vous fournissons 5 fichiers .java (Sources.zip) correspondant à l'interface graphique Swing, à la génération de graphe et la visualisation d'une coloration. Le code fourni compile et affiche l'interface graphique. Sous votre IDE favori, créez donc un projet (Java standard, pas FX) avec ces sources, ajoutez les librairies graphstream et faites tourner le programme. Vous pouvez aussi télécharger le fichier TP_Tabu pour Eclipse pour avoir tout directement. Le bouton « aleatoire » est déjà connecté et doit afficher une coloration aléatoire des sommets du graphe. Dans la suite du TP, nous vous demanderons de compléter ce code en implémentant les algorithmes de coloration, dans la classe *FenetrePrincipale.java.* **Seul ce fichier .java sera à rendre, ne modifiez rien ailleurs. Par ailleurs, on vous demandera de rendre également un fichier exécutable .jar qui contient l'ensemble de votre programme.**

La documentation en ligne de Graphstream est ici :
http://graphstream-project.org/doc/Tutorials/Getting-Started/

Cependant, vous avez le code et les infos suffisantes pour ne pas à avoir à trop vous plonger là dedans.

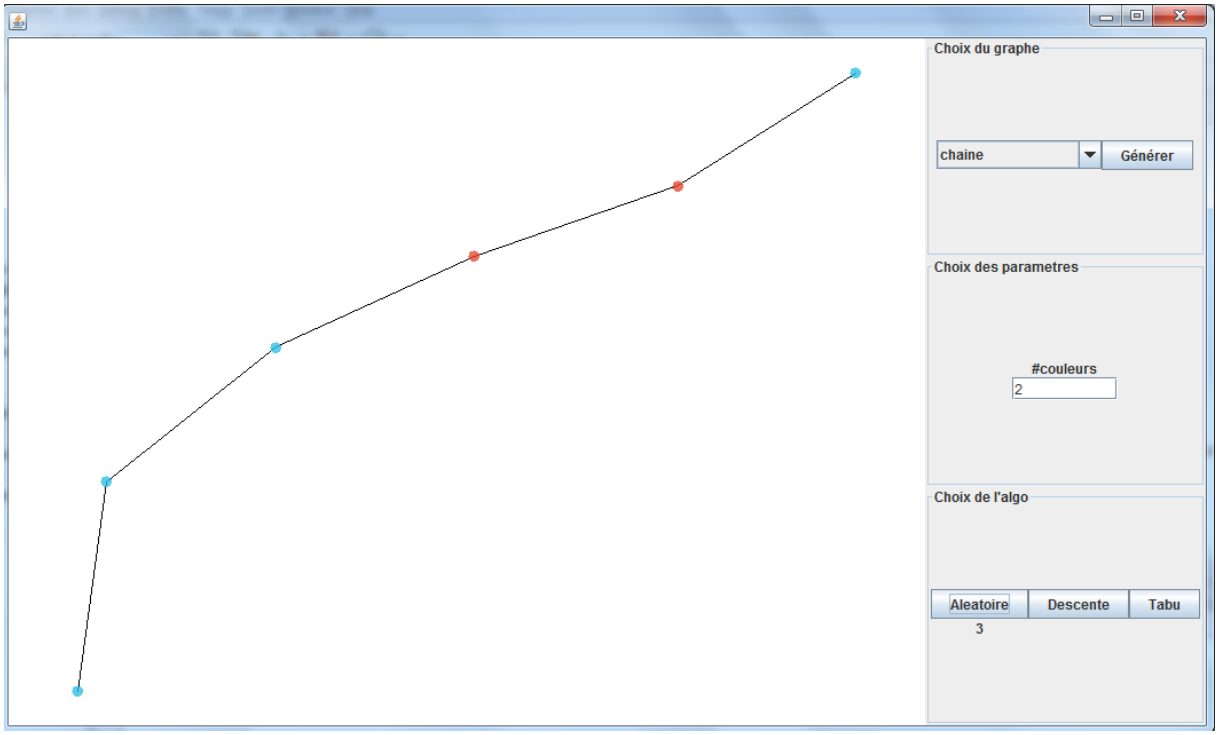
Si le lien ci-dessus vous permet de comprendre rapidement la façon dont les graphes sont implémentés sous Graphstream (classes Graph, Node et Edge avec des itérateurs ou des accès directs aux nœuds via getNode(index)), la documentation ci-dessous :

http://graphstream-project.org/doc/Tutorials/Storing-retrieving-and-displaying-data-in-graphs/

est utile pour la notion d'attribut dynamique. En effet, la couleur d'un sommet pourra être stockée comme un attribut de la classe Node. On peut créer autant d'attributs que l'on veut pour une classe. Dans *Fenetre_principale.java*, lisez le code de la classe anonyme correspondant au bouton *alea_button*, il est facile à comprendre et servira de base pour les fonctions que vous allez coder.

2. Méthodes de voisinage pour la coloration de graphes

Dans le cadre de la coloration de graphe, les métaheuristiques se basent sur le problème d'optimisation qui consiste à minimiser le nombre de conflits du graphe. Etant donné k couleurs et une coloration impropre des sommets du graphe avec ces couleurs, deux sommets sont dit **en conflit** s'ils sont voisins et qu'ils ont la même couleur. Une coloration propre du graphe avec k couleurs est donc une coloration sans aucun conflit.



Exemple pour k=2 pour une chaine de longueur 6, la solution ci-dessus possède 3 conflits.

Toute coloration du graphe avec k couleurs est appelée une solution. Deux solutions sont dites voisines si elles ont **exactement un sommet** qui n'a pas la même couleur. Le principe des algorithmes de voisinage consiste à partir d'une solution aléatoire (avec potentiellement beaucoup de conflits), et à explorer des solutions voisines jusqu'à tomber si possible sur une solution avec 0 conflit (ce qui correspondra à une k-coloration propre).

Exercice 1. Dans le fichier *FenetrePrincipale.java*, codez les deux fonctions *getNbConflits* qui, pour un attribut de coloration passé en paramètre, retournent respectivement le nombre de conflits d'un seul sommet et de tout le graphe. Testez-les en cliquant sur le bouton « aleatoire », qui doit afficher à l'écran le nombre de conflits (cf. screenshot ci-dessus).

Méthode de descente

On va maintenant coder l'algorithme de la plus grande pente vu en CM. Il sera appelé au moment où le bouton « descent_button » est cliqué. Codez-le à cet endroit dans la fonction *descent()*.

Pour rappel, l'algorithme est le suivant :

- On génère une solution de départ aléatoire sur *nbcouleurs* couleurs (cf fonction alea deja codée). Il s'agira de la **solution courante**.

- On explore toutes les solutions voisines, càd pour chaque sommet **ayant au moins un confit**, on change sa couleur en testant les (nbcouleurs-1) autres couleurs disponibles, et pour chacune, on compte le nombre de conflits du graphe. Si l'une d'entre elles est meilleure que la solution courante, **on prend comme nouvelle solution courante celle qui a le moins de conflits.** Si toutes les solutions voisines ont au moins autant de conflits que la solution courante ou si on a trouvé une solution sans conflit, l'algorithme s'arrête.

Exercice 2 Testez votre fonction via le JLabel sous le bouton *descente* indiquant le nombre de conflits à la fin de l'algorithme. Pour plusieurs graphes, vérifiez le gain obtenu comparé au nombre de conflits de la coloration aléatoire sans descente.

Recherche Tabou

Testez votre algorithme précédent sur des grandes chaines (100 sommets) ou des grilles, avec 2 couleurs. Normalement ces graphes sont coloriables avec deux couleurs mais la méthode de descente stoppe sur un minimum local avec un grand nombre de conflits. La recherche tabou devrait répondre à ce problème en autorisant l'algorithme à sortir des minima locaux.

Exercice 3 On vous demande maintenant de coder un algorithme de recherche tabou pour le problème de coloration, dans la fonction *tabu()*. Les différentes recherches sur ce domaine ont montré qu'il était préférable de stocker dans la liste tabou des couples (sommet, couleur) plutôt que des colorations entières. De cette façon, on explore plus largement l'espace puisqu'on interdit à l'algorithme de visiter toutes les solutions ayant le sommet v colorié avec la couleur k si (v,k) est dans la liste.

Pour rappel, la base de cet algorithme sera la méthode de descente codée précédemment, avec les variations suivantes :

- Une liste tabou qui contient des couples (sommet, couleur). Lors de la descente, on s'interdit de visiter les solutions qui sont dans la liste. La taille de la liste sera fixe (10 par exemple). Si tous les voisins ont un nombre de conflits qui n'est pas inférieur à celui de la solution courante, on choisit le « moins pire » d'entre eux en terme de nombre de conflits. Si toutes les solutions voisines sont tabou, alors on choisit un couple (sommet, couleur) au hasard. N'oubliez pas que la liste tabou fonctionne comme une file de type FIFO. Vous pourrez utiliser une LinkedList Java par exemple.

- Un critère dit *d'aspiration* est ajouté qui dit, dans sa version simplifiée, que si on tombe sur un voisin avec moins de conflits que la solution courante, bien que ce voisin soit dans la liste tabou, alors on relaxe la contrainte tabou et on l'accepte dans le voisinage autorisé.

- L'algorithme s'arrête, soit quand on tombe sur une solution sans conflit, soit quand le nombre maximum d'itérations consécutives sans améliorer la meilleure solution a été atteint (on pourra le fixer à 300 par exemple).

Testez votre algorithme et comparez-le à votre algorithme de descente simple, par exemple sur des chaines de longueur 100 avec 2 couleurs, ou des arbres tertiaires de hauteur 5 avec 3 couleurs.

Tabucol

Exercice 4 L'algorithme de Hertz et De Werra possède plusieurs optimisations qui rendent l'algorithme plus efficace. Nous allons en implémenter deux d'entre elles (vous pourrez écrire une seconde fonction (tabu_opt par ex)) :

- La taille de la liste : plutôt que de choisir une taille fixe, on va opter sur une taille dynamique qui permet d'explorer l'espace plus loin quand la solution courante n'est pas bonne, et de rester proche de la solution courante quand elle est presque optimale. Les paramètres suggérés par Galinier et Hao (1999) sont les suivants : à chaque itération, la taille de la liste est recalculée comme étant : 0.6*(nombre de conflits de la solution courante)+R où R est un nombre aléatoire entre 0 et 9.

- La rapidité des deux algorithmes ci-dessus est mise à mal à cause du temps mis pour tester toutes les solutions voisines (il y en a N*(k-1)) et recalculer le nombre de conflits à chaque fois. On peut complètement optimiser ce calcul vu que lors d'une recoloration, seules deux couleurs sont mises en jeu. Le principe est le suivant : on construit au début de l'algorithme une matrice C de taille *nbsommets x nbcouleurs*. Pour la solution courante, une case C[v][i] doit contenir le nombre de voisins de v ayant la couleur i. Initialisez cette matrice avec les bonnes valeurs à partir de la coloration aléatoire choisie au début. Puis, pour chaque solution voisine de la solution courante, consistant à recolorier un sommet v de la couleur i en j, on connait son nombre de conflits comme étant le nombre de conflits de la solution courante + C[v][j]- C[v][i] . Un seul parcours de la matrice suffit donc à trouver le meilleur voisin. Une fois ce voisin v_opt choisi et recolorié de i en j, on met à jour la matrice C pour l'itération suivante en faisant :

  Pour tout sommet u voisin de v_opt, C[u][i] = C[u][i] – 1 et C[u][j] = C[u][j] +1.

Testez cet algorithme sur des graphes à plus de 100 sommets pour constater la différence.