

Projet individuel d'algorithmique Programmation AP1  
Groupe 4.2  
Octobre 2010

## 1 Informations générales

### 1.1 Travail à rendre

Le projet est à réaliser en OCaml **individuellement**. Il sera accompagné d'un **dossier** contenant impérativement la description des choix faits, la description des types et des fonctions. Pour chaque fonction, on donnera impérativement l'interface complète (dans le code en commentaire et dans le rapport pour les fonctions présentées).

Même si le sujet est décomposé en questions, il est possible qu'une question se résolve par l'écriture d'une ou plusieurs fonctions intermédiaires. Celles-ci doivent comporter une interface également.

Le dossier fournira également des cas de tests accompagnés des résultats attendus et retournés.

Sur le site du cours figure un petit document sur ce que l'on attend dans un rapport. Consultez-le!

### 1.2 Calendrier et procédure de remise

Le projet (dossier + copie du listing de code) est à rendre au secrétariat **le 22 novembre à 16h au plus tard**. Le numéro du groupe, ainsi que le nom du chargé de TD, devront figurer en gros, et en rouge sur la page de garde.

Le fichier .ml contenant votre code devra être déposé électroniquement au plus tard **le 22 novembre à minuit**.

Les soutenances seront organisées dès la semaine suivante. Il vous faudra consulter les panneaux d'affichage et votre courrier électronique pour obtenir l'ordre de passage.

Enfin n'attendez pas pour vous mettre au travail! Un projet se travaille dès la remise du sujet afin d'avoir le temps de laisser murir la solution et de poser des questions au client (dans votre cas, votre chargé de TP).

### 1.3 Procédure de dépôt

Pour déposer le code du projet, consultez les informations sur les machines de l'école, tout figure.

Ne vous inquiétez pas, l'interface fabrique un nom de projet déposé à partir de votre login. Votre projet ne sera pas confondu avec celui d'un autre.

Indiquez quand même en commentaire dans votre fichier de code Ocaml votre nom et votre groupe. Ce sera plus facile pour le correcteur.

Vous pouvez déposer successivement plusieurs versions, le dernier dépôt écrase le précédent, seul le dernier dépôt est pris en compte

Enfin tout cela peut se faire de l'extérieur.

Le code à déposer est un fichier.ml commenté **avec les interfaces des fonctions**. **Le fichier doit pouvoir être compilé (sans aucune intervention humaine - lecture du buffer complet) sur**

les machines Yaka de l'école, n'oubliez pas de vérifier que tout fonctionne sur les machines de l'école avant de le dép oser - si vous l'avez développé sous un autre système d'exploitation.

## 2 Énoncé du projet : Un petit problème de coloriage

### 2.1 Introduction : un modèle très expressif

Nous allons aborder dans ce projet un modèle de théorie des graphes capable de représenter nombre d'applications réelles : le modèle de coloration de graphe. Le problème que nous allons chercher à résoudre est le suivant. Étant donné un graphe  $G = (V, E)$ , et  $K$  couleurs, colorer chaque sommet du graphe avec une des  $K$  couleurs de sorte à ne jamais affecter la même couleur aux deux extrémités d'une arête.

### 2.2 Première partie : les définitions

#### Question 1

Définir le type `vertex` des sommets d'un graphe. Un sommet sera identifiable par un entier. Définir ensuite le type `edge` des arêtes, puis le type `graph` des graphes. Attention à bien identifier vos besoins avant de décider de la représentation des graphes que vous allez adopter !

#### Question 2

Implanter les fonctions suivantes `add_vertex` de type `vertex → graph → graph` et `add_edge` de type `edge → graph → graph`.

#### Question 3

Écrire une fonction `make_graph` qui prend en paramètres une liste d'entiers représentant des sommets et une liste de couples d'entiers, représentant des arêtes, et construit le graphe contenant ces sommets et ces arêtes. Le type de cette fonction sera `int list → (int*int) list → graph`. La fonction échouera si l'une des arêtes a pour extrémité un sommet qui n'appartient pas au graphe.

#### Question 4

Définir le type `color` des couleurs ainsi que le type `coloring` des colorations. Comme pour les sommets, une couleur sera identifiable par un entier. Écrire ensuite une fonction `check_coloring` de type `coloring → graph → bool` qui vérifie qu'une coloration est valide pour un graphe.

### 2.3 Deuxième partie : un algorithme simple

Un sommet est dit trivialement colorable s'il a strictement moins de  $K$  voisins. Une heuristique simple consiste alors à supprimer un sommet trivialement colorable  $v$ , tenter de colorer le graphe restant, puis colorer  $v$  avec la plus petite couleur disponible (*i.e.* la couleur disponible dont l'identifiant est le plus petit). Si à un moment donné aucun sommet trivialement colorable ne reste et que le graphe n'est pas vide, alors l'algorithme échoue.

#### Question 5

Écrire une fonction `vertex_degree` de type `vertex → graph → int` qui calcule le degré d'un sommet d'un graphe, le degré d'un sommet étant son nombre de voisins.

#### Question 6

Écrire une fonction `remove_vertex` de type `vertex → graph → graph` qui supprime un sommet d'un graphe et renvoie le graphe résultant.

### Question 7

Écrire l'heuristique `greedy_coloring` décrite en début de cette partie. Son type est naturellement `graph`  
`→ int → coloring`.

## 2.4 Troisième partie : un algorithme dédié à une classe de graphes

Sous certaines conditions, l'algorithme présenté dans la partie précédente conduit à une coloration du graphe utilisant un nombre minimal de couleurs. C'est par exemple le cas si le sommet sélectionné à chaque itération est un sommet simplicial, *i.e.* dont les voisins sont tous reliés deux à deux (ou, autrement dit, forment une clique).

### Question 8

Modifier l'implantation de la question 7 afin de sélectionner non pas un sommet trivialement colorable mais un sommet simplicial à chaque étape. Si à un moment donné, aucun sommet simplicial n'existe et le graphe est non vide, alors l'algorithme échoue.

Quand il n'est pas possible de sélectionner un sommet simplicial à chaque étape, il est possible de modifier le graphe pour rétablir cette possibilité. Une heuristique simple pour ce faire consiste à choisir à chaque itération le sommet du graphe de plus bas degré, relier tous ses voisins deux à deux, puis supprimer ce sommet du graphe, jusqu'à ce que le graphe soit vide.

### Question 9

Implanter l'heuristique de modification du graphe.

## 2.5 Quatrième partie : un modèle plus précis

Nous considérons maintenant un graphe qui contient deux types d'arêtes. Les premières, appelées conflits, doivent avoir leurs extrémités ne colorées avec des couleurs différentes (comme précédemment). Les secondes, appelées affinités, doivent avoir leurs extrémités colorées si possible avec la même couleur. De plus, les affinités peuvent avoir un poids représentant le gain obtenu en satisfait cette affinité, *i.e.* en affectant à ses deux extrémités la même couleur. Notons que deux sommets ne pourront jamais être reliés à la fois par une affinité et un conflit (cela n'a d'ailleurs aucun sens).

Une heuristique simple pour satisfaire beaucoup d'affinités consiste à fusionner itérativement les extrémités d'une affinité, jusqu'à ce que le graphe ne contienne plus d'affinité, et à colorer le graphe résultant ensuite. La couleur affectée à un sommet est finalement la couleur affectée au sommet avec lequel il a été agrégé.

### Question 10

Définir le type `conflict` des conflits, le type `affinity` des affinités et le type `affinity_graph` des graphes possédant à la fois des conflits et des affinités.

### Question 11

Écrire une fonction `make_affinity_graph` qui prend en paramètres une liste d'entiers représentant des sommets, une liste de couples d'entiers, représentant les conflits, et une liste de triplets d'entiers représentant les affinités et leurs poids, et construit le graphe contenant ces sommets et ces arêtes. Le type de cette fonction sera `int list → (int*int) list → (int*int*int) list → graph`. La fonction échouera si l'une des arêtes a pour extrémité un sommet qui n'appartient pas au graphe ou si deux sommets sont liés à la fois par une affinité et un conflit.

**Question 12**

Écrire une fonction `coloring_value` qui vérifie qu'une coloration est valide et calcule sa valeur, *i.e.* la somme des poids des affinités satisfaites par cette coloration.

**Question 13**

Écrire la fonction `merge` de type `affinity → graph → graph` qui fusionne les extrémités de l'affinité donnée en paramètre.

**Question 14**

Implanter l'heuristique décrite ci-dessus pour un nombre de couleurs disponibles supposé infini.

**Question 15**

Proposer une amélioration de cette heuristique et l'implanter, pour un nombre de couleurs supposé infini, puis pour un nombre de couleurs limité à  $K$ .