Projet : Coloriage de graphe

Programmation fonctionnelle, ENSIIE

Semestre 4, 2019–20

1 Informations pratiques

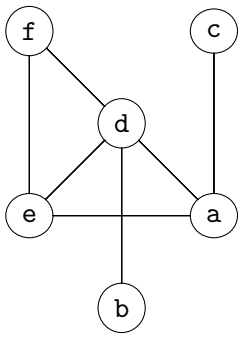
Le code rendu comportera un Makefile, et devra pouvoir être compilé avec la commande `make`. **Tout projet ne compilant pas se verra attribuer un 0** : mieux vaut rendre un code incomplet mais qui compile, qu'un code ne compilant pas. Votre code devra être **abondamment commenté et documenté**.

Votre projet est à déposer sur `http://exam.ensiie.fr` dans le dépôt `ipf_fisa_2020` sur forme d'une archive `tar.gz` **avant le 4 mai à 23h59**. **Tout projet rendu en retard se verra attribuer la note 0.** Vous n'oublierez pas d'inclure dans votre dépôt un rapport (PDF) précisant vos choix, les problèmes techniques qui se posent et les solutions trouvées.

2 Sujet

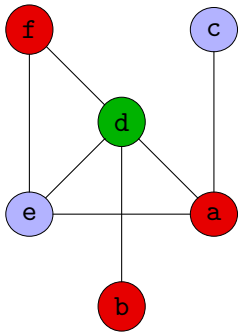
Étant donné un graphe non-orienté, un k-coloriage de ce graphe est une fonction qui associe à chaque sommet du graphe un entier entre 1 et k (une couleur) tel que deux voisins ont des couleurs différentes.

Par exemple, au graphe



on peut associer le 3-coloriage $\{a \mapsto 1; b \mapsto 1; c \mapsto 2; d \mapsto 3; e \mapsto 2; f \mapsto 1\}$ ce qui

correspond au graphe colorié suivant :



Le problème du k-coloriage de graphe est, pour un k fixé, de fournir un tel coloriage ou d'indiquer que le coloriage n'est pas possible. Ce problème a plusieurs applications en informatique, par exemple l'allocation de registres dans un compilateur, ou l'ordonnancement de tâches. C'est un problème NP complet pour tout $k \geq 3$. Il existe des algorithmes pour approximer ce problème en temps polynômial. Pour ce projet, nous allons néanmoins recherche un solution exacte.

L'algorithme que nous allons utiliser est le suivant : à chaque sommet du graphe on va associer un ensemble de couleurs encore disponibles, qui au départ contiendra toutes les couleurs (ensemble des entiers de 1 à k).

```
fonction colorier(graphe G, couleurs_dispo C)
si G non vide
  choisir un sommet s dans G
  pour chaque couleur c encore disponible pour s dans C
    C' := C dans lequel on a enlevé la couleur c aux couleurs
                        disponibles des voisins de s
    G' := G privé de s
    si colorier(G, C') réussit et retourne col
    alors retourner col + { s -> c }
    sinon passer à la couleur suivante
  retourner Échec
sinon retourner le coloriage vide
```

3 Graphes non orientés

On va utiliser la même structure de données que vue en cours pour implémenter les graphes. La différence avec les graphes orientés est que s'il existe une arête de u à v, alors v sera dans l'ensemble des sommets associés à u, et u sera également dans l'ensemble des sommets associés à v.

1. Définir le type `graph` comme vu en cours, avec des nœuds contenant des `string`.

2. Écrire une fonction `add_edge : string -> string -> graph -> graph` tel que `add_edge u v g` retourne le graphe `g` dans lequel on a ajouté une arête **non-orientée** entre `u` et `v`.
3. Écrire une fonction `remove_vertex : string -> graph -> graph` tel que `remove_vertex u g` retourne le graphe `g` dans lequel on a retiré le sommet `u` ; pour cela, il faut retirer l'association à `u` dans `g`, mais il faut aussi retirer `u` des ensembles associés aux voisins de `u`.

4 Coloriages disponibles

4. Définir un module `IntSet` pour les ensembles d'entiers.
5. Définir une fonction `color_set : int -> IntSet.t` tel que `color_set n` retourne l'ensemble des entiers de 1 à `n`.
6. Définir le type `disp_color` des coloriages disponibles. C'est un type qui associe à chaque sommet un ensemble de couleurs (ensemble d'entier).
7. Définir une fonction `init_colors : graph -> int -> disp_color` tel que `init_colors g k` retourne un map qui associe à chaque sommet de `g` l'ensemble des entiers de 1 à `k`.
   On pourra utiliser `StringMap.map`.
8. Définir une fonction `remove_color : int -> string -> disp_color` telle que `remove_color i v c` retourne `c` dans laquelle on a supprimé `i` des couleurs disponibles pour le sommet `v`.

5 Coloriage

9. Définir une exception `Failed` qui indiquera qu'aucun coloriage n'a pu être trouvé.
10. Définir une fonction `try_first : (int -> 'a) -> IntSet.t -> 'a` telle que `try_first f s` retourne le premier résultat de l'application de `f` à un élément de `s`qui ne lève pas l'exception `Failed`.
    Par exemple, si `s` est $\{1, 2, 3\}$, que `f 1` lève l'exception `Failed` et que `f 2` retourne `42`, alors `try_first f s` retournera `42`.
    Pour cela, on peut utiliser un algorithme récursif :

```
fonction try_first(f, s)
  si s est vide,
    alors lever Failed
  sinon
    choisir i dans s
    essayer
      retourner f i
    avec Failed ->
      try_first(f, s privé de i)
```

    On pourra par exemple utiliser la fonction `IntSet.choose`.

11. Définir un type `coloring` pour les coloriage ; c'est une association qui a chaque sommet associe un entier (sa couleur).
12. Définir une fonction (récursive) `color : graph -> disp_color -> coloring` telle que `color g c` retourne un coloriage pour `g` compatible avec les couleurs disponibles de `c`, ou lève `Failed` si ce n'est pas possible, en utilisant l'algorithme si dessus.

6 Extensions

13. Ajouter une sortie au format `dot` (`https://fr.wikipedia.org/wiki/DOT_(langage)`) : écrire une fonction qui prend un graphe et un coloriage et qui affiche sur la sortie standard un graphe au format `dot` avec les couleurs.
14. Ajouter une entrée au format `dot` : écrire une fonction qui parse un fichier au format `dot` et qui retourne le graphe associé. Il sera opportun d'utiliser les outils Ocamllex/Ocamlyacc (`http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`).