Projet de Programmation Fonctionnelle 2016-2017 : Coloriage de cartes

Table des matières

1 Introduction

Le théorème des quatre couleurs est l'un des résultats les plus célèbres des mathématiques combinatoires. Il affirme qu'il est possible, en utilisant au plus quatre couleurs différentes, de colorier n'importe quelle carte découpée en régions connexes de sorte que deux régions adjacentes – c'est-à-dire ayant toute une frontière en commun (et non simplement un point) – reçoivent toujours deux couleurs distinctes [1]. La preuve du théorème donne un algorithme de coloration en temps quadratique, mais particulièrement complexe. La preuve de correction de cet algorithme nécessite l'usage d'un ordinateur pour résoudre 1478 cas critiques. Historiquement, le théorème des quatre couleurs est l'un des premiers résultats illustrant l'importance de l'utilisation des assistants de preuve en mathématiques [2].

Le but de ce projet est d'implémenter en OCaml un jeu de puzzle sur le thème du théorème des quatre couleurs. Etant donnée une carte en partie pré-coloriée (voir Figure 1), le jeu consistera à compléter ce coloriage à l'aide d'un ensemble donné de couleurs, en respectant une unique règle : deux régions adjacentes devront toujours être coloriées de deux couleurs distinctes. [3]

1. Pour plus de détails voir : `https://fr.wikipedia.org/wiki/Théorème_des_quatre_couleurs`.
2. Voir : `https://fr.wikipedia.org/wiki/Assistant_de_preuve`.
3. A titre d'exemple, vous trouverez à l'adresse suivante une implémentation de ce jeu par Simon Tatham : `http://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/map.html`

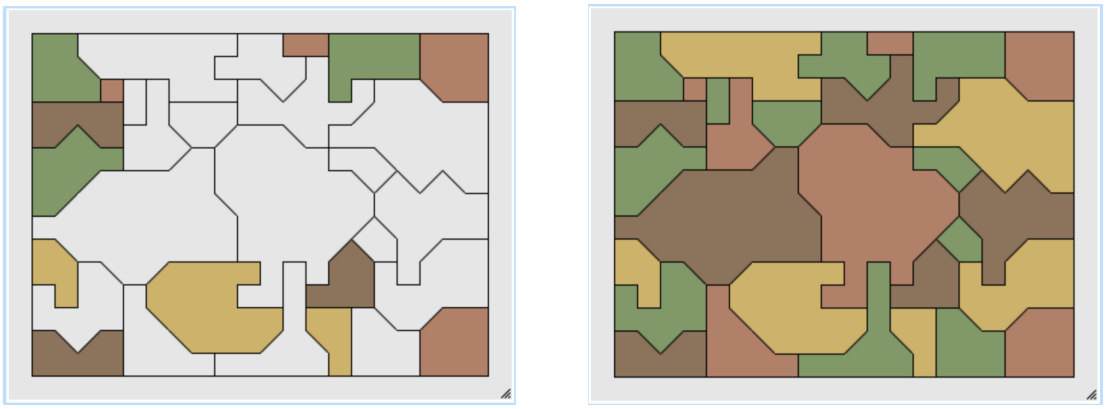


FIGURE 1 – Exemple d'une configuration initiale (à gauche) et finale (à droite) du jeu

Le travail minimal que nous vous demandons d'effectuer est le suivant :

> **Vous devrez écrire et nous présenter un programme OCaml permettant, étant donnée une carte en partie pre-coloriée :**
>
> 1. **d'afficher cette carte à l'écran ;**
> 2. **de jouer manuellement, l'utilisateur choisissant les couleurs des régions vides parmi celles déjà présentes dans le diagramme, le programme lui annonçant la fin du jeu dès que le coloriage est complet et correct ;**
> 3. **de résoudre le puzzle automatiquement, en trouvant un coloriage complet et correct à l'aide d'un résolveur SAT fourni sur Moodle [4].**

Cette liste de fonctionnalités n'est évidemment pas exhaustive : toute extension approuvée par votre chargé de TD sera la bienvenue. Votre programme devra pouvoir gérer de manière efficace un nombre raisonnable de régions (30 dans l'exemple ci-dessus).

2 Les diagrammes de Voronoï

La modélisation d'une carte en OCaml doit être raisonnablement concise (sans informations inutiles) ; elle doit en outre permettre de reconnaître facilement les relations de voisinage entre regions (nécessaires au déroulement du jeu), et d'afficher facilement une carte à l'écran. La représentation des cartes que nous avons choisie, facilement modélisable par un ensemble des types OCaml, est celle des *diagrammes de Voronoï*.

4. Attention, la preuve du théorème de quatre couleurs donne un algorithme de coloriage en temps quadratique à partir d'une carte sans aucun pre-coloriage. Par contre, la coloriage de cartes en partie pre-coloriées est un problème connu être difficile (NP-complet), équivalent au problème SAT de la satisfiabilité d'une formule propositionnelle. C'est pour cette raison qu'il est commode d'utiliser un solveur SAT en profitant de ses heuristiques.

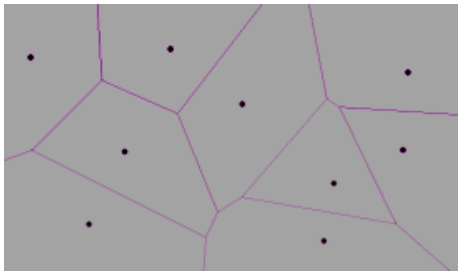


FIGURE 2 – Exemple d'un diagramme de Voronoï avec 9 germes. La region associée à un germe est l'ensemble des points plus proche de ce germe que de tout autre germe.

Un diagramme de Voronoï permet de découper le plan en régions à partir d'un ensemble discret de points, appelés *germes*. Chaque région ne renferme qu'un seul germe, et est formée de l'ensemble des points du plan plus proches de ce germe que de tout autre germe. La région représente en quelque sorte la "zone d'influence" du germe (Figure 2).

Pour mesurer la distance entre deux points, plusieurs notions de distance sont envisageables. Deux possibilités classiques sont la *distance euclidienne* usuelle et la *distance taxicab* :

$$\text{euclide}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$
$$\text{taxicab}((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|.$$

La distance euclidienne génère des cartes dont toutes les regions sont convexes, comme dans la Figure 2 et la carte gauche de la Figure 3. La distance taxicab donne des régions similaires à celles des cartes de la Figure 1, ainsi que la carte droite de la Figure 3. Entre ces deux notions, vous pouvez choisir celle que vous préférez, ou mieux encore, laisser ce choix à l'utilisateur.

3 Modélisation de diagrammes en OCaml

Vous devez utiliser les types `seed` et `voronoi` définis ci-dessous sans les modifier, afin de nous permettre de tester votre code avant et pendant la soutenance.

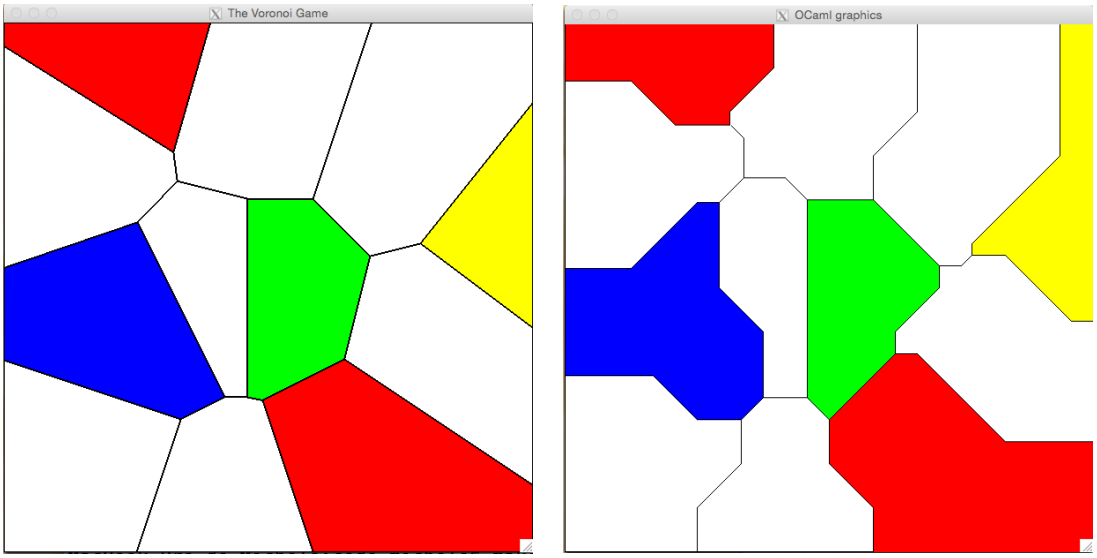
Les points considérés ici seront des coordonnées de pixels dans une fenêtre du module `Graphics`. Un germe (*seed* en anglais) du diagramme sera représenté par ses coordonnées entières (`x`, `y`) associées ou non encore associées à une couleur.

On utilisera le type `color option` pour représenter une association éventuelle, la valeur `None` correspondant à un germe dont la région n'a pas encore reçu de couleur. Le type `seed` sera donc défini comme l'enregistrement suivant :

```
type seed = {c : color option; x : int; y : int}
```

Un diagramme de Voronoï sera représenté par un tableau de germes associé aux dimensions de sa fenêtre graphique :

```
type voronoi = {dim : int * int; seeds : seed array}
```



(a) distance euclidienne (b) distance taxicab

FIGURE 3

Voici un exemple de valeur de type `voronoi` – son affichage, selon la notion de distance choisie, est représenté à la Figure 3 :

```
{dim = 600,600;
 seeds = [|
 {c=None; x=100; y=100}; {c=Some red; x=125; y=550};{c=Some green; x=300; y=300};
 {c=Some blue; x=150; y=250}; {c=None; x=250; y=300}; {c=None; x=300; y=500};
 {c=Some red; x=400; y=100}; {c=None; x=450; y=450}; {c=None; x=500; y=250};
 {c=Some yellow; x=575; y=350}; {c=None; x=75; y=470}; {c=None; x=250; y=50};
 |]
}
```

Vous trouverez sur `Moodle` un fichier `examples.ml` contenant d'autres valeurs de type `voronoi`. Noter que le contenu de ce fichier ne doit pas être intégré à votre code source (sauf, éventuellement, pour le tester durant son développement) : il doit rester un des fichiers parmi tous ceux nécessaires à la production de votre exécutable. Vous pouvez, bien sûr, compléter ce fichier par vos propres grilles, en variant la difficulté des puzzles correspondants.

4 Un SAT-solveur en OCaml

Le fichier `sat_solver.ml` déposé sur `Moodle` vous propose un outil permettant de résoudre automatiquement les puzzles de coloriage. La fonction principale de ce fichier est `solve`, prenant en argument une formule du calcul propositionnel sous forme normale conjonctive, et renvoyant : ou bien une affectation qui satisfait cette formule ; ou bien l'indication du fait qu'elle n'est pas satisfaisable [5].

4.1 Formes normales conjonctives

Un *littéral* est une variable propositionnelle ou la négation d'une variable propositionnelle. Une *clause disjonctive* est une disjonction de littéraux. Une *forme normale conjonctive* est une conjonction de clauses disjonctives. Par exemple, la formule suivante est une forme normale conjonctive (fnc), formée de trois clauses contenant chacune deux littéraux :

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_3)$$

Une fnc peut être représentée en OCaml par une liste de listes de couples, chaque liste de couples représentant une clause, chaque couple représentant un littéral. La partie gauche de chaque couple est un booléen indiquant, lorsque sa valeur est `false`, que la variable propositionnelle est précédée d'une négation dans la clause correspondante. Par exemple, la formule ci-dessus pourra être représentée par une valeur de la forme

```
[[(true, x1); (false, x2)]; [(false, x1); (false, x3)]; [(true, x2); (true, x3)]]
```

Appliquée à une telle liste, la fonction `solve` produira une valeur de la forme

```
Some [(false, x3); (true, x1); (true, x2)]
```

indiquant des valeurs pour `x1`, `x2` et `x3` satisfaisant la formule ci-dessus (`x3` à `false`, `x1` et `x2` à `true`). Appliquée à une liste représentant une formule insatisfaisable, `solve` renvoie `None`.

Les listes passées à la fonction `solve` seront toujours de cette forme, mais le choix pour les parties droites de couples du type le mieux adapté dépend, en général, du problème à résoudre. Si l'on souhaite par exemple résoudre une grille de de Sudoku [6], un choix naturel est celui des triplets d'entiers, une valeur de la forme `(i, j, k)` s'interprétant par : "la case de ligne `i` et de colonne `j` de la grille contient la valeur `k`". Afin d'éviter de lier la fonction `solve` à un choix de type particulier, le fichier `sat_solver.ml` fournit ce qu'on appelle un *foncteur*.

4.2 Utilisation du foncteur `Sat_solver.Make`

Les notions de *module* (un ensemble de déclarations de types et de définitions de fonctions regroupées sous un même nom, *eg.* `List` ou `Graphics`), de *foncteur* (un module paramétrique, intuitivement, l'équivalent d'une "fonction prenant en argument un module et renvoyant un module") et de *signature* (une spécification du type des éléments d'un module ou d'un paramètre de foncteur) font partie des notions les plus avancées de ce cours. Elles vous seront présentées en cours de semestre, et ne sont pas indispensables pour commencer la rédaction de ce projet.

5. Ce fichier est une version non optimisée du solveur SAT-MICRO de S. Conchon, J. Kanig et S. Lescuyer. Voir `https://hal.inria.fr/inria-00202831`. Pour plus de détails sur l'algorithme implémenté, *cf.* `https://fr.wikipedia.org/wiki/Algorithme_DPLL`.

6. Voir `https://fr.wikipedia.org/wiki/Sudoku`.

Le foncteur `Sat_solver.Make` peut prendre en argument tout module contenant la déclaration d'un certain type `t` ainsi que l'implémentation d'une certaine fonction `compare` de type `t -> t -> int`, *ie.* tout module dont la signature est :

```
module type VARIABLES = sig
  type t
  val compare : t -> t -> int
end
```

Le choix de `t` est libre. La fonction `compare` doit spécifier un certain ordre sur les valeurs de type `t`, en respectant la convention suivante : `compare de a b` doit renvoyer `0` si `a` et `b` sont égaux, une valeur positive si `a` est plus grand que `b`, une valeur négative sinon.

Supposons par exemple que, dans la modélisation d'un problème donné, le type idéal pour représenter les variables propositionnelles soit le type `int`. On définit tout d'abord un module `Variables` respectant la signature `VARIABLES` :

```
module Variables = struct
  type t = int
  let compare x y = if x > y then 1 else if x = y then 0 else -1
end;;
```

On définit ensuite un nouveau module `Sat`, en passant le module `Variables` au foncteur `Sat_solver.Make` :

```
module Sat = Sat_solver.Make(Variables);;
```

Le module construit implémente alors la signature :

```
sig
  type literal = bool * int
  val solve : literal list list -> literal list option
end
```

Autrement dit, la fonction `Sat.solve` accepte en argument des listes de type `((bool * int) list) list`, et renvoie une valeur de type `((bool * int) list) option`. Le type `int` à droite de chaque occurrence de `bool` est `Variables.t`, c'est-à-dire le type `t` choisi dans le module `Variables`. On peut par exemple écrire

```
Sat.solve
  [[(true, 1); (false, 2)]; [(false, 1);(false, 3)]; [(true, 2); (true, 2)]]
```

et obtenir le résultat suivant :

```
Some [(false, 3); (true, 1); (true, 2)]
```

Plus généralement, quel que soit le choix de `Variables.t`, la fonction `Sat.solve` accepte des valeurs de type `(bool * Variables.t) list list`, et renvoie des valeurs de type `(bool * Variables.t) list option`. Dans le cadre de ce projet, vous devez choisir pour `Variables.t` la définition qui vous semble la mieux adaptée au problème posé.

5 Fonctions principales du traitement

Les éléments qui suivent ne sont que des suggestions. Vous êtes libres d'en implémenter des variantes ou de proposer des méthodes alternatives, à condition qu'elles soient correctement présentées et justifiées dans votre rapport.

Deux fonctionnalités importantes à implémenter sont :

1. une fonction `draw_voronoi`, qui affiche un diagramme de Voronoï dans une fenêtre du module `Graphics` ;
2. une fonction `adjacences_voronoi` permettant de déterminer, pour chaque région, quelles sont les régions adjacentes.

Plusieurs algorithmes permettent d'implémenter ces deux fonctions de manière élégante et efficace, mais leur mise en œuvre est complexe [7]. Nous vous proposons donc ci-dessous une méthode naïve et peu efficace mais très simple, et largement suffisante pour traiter des diagrammes de Voronoï d'environ une quarantaine de germes.

Calcul de la matrice des régions. À partir d'une valeur `v` de type `voronoi`, on peut calculer une matrice `m` indiquant, pour chaque pixel, le germe dont il est le plus proche – si plusieurs germes sont les plus proches de ce pixel, on peut choisir l'un d'entre eux arbitrairement.

Plus précisément, on peut calculer une matrice `m` de type `int array array`, de dimension `v.dim`, et dont la case `m.(i).(j)` vaut l'indice dans `v.seeds` de l'un des germes les plus proches du pixel de coordonnées (`i`, `j`) par rapport à la distance choisie. Cette matrice est appelée *matrice des régions* de `v`. Elle peut être calculée par une fonction

```
regions_voronoi : (int * int -> int * int -> int) -> voronoi -> int array array
```

prenant en argument une fonction de distance (de type `int * int -> int * int -> int` [8]) et un diagramme de Voronoï (de type `voronoi`), et renvoyant la matrice des régions de ce diagramme (de type `int array array`).

Affichage. Une fois calculée, la matrice des régions peut être passée à une fonction d'affichage `draw_voronoi` qui examine son contenu en donnant à chaque pixel la couleur de son germe associé, si ce germe en a une. Les pixels situés à la frontière de deux régions seront affichés en noir : ce sont tous ceux de coordonnées (`i`, `j`) tels que `m.(i).(j)` soit différent d'au moins une valeur définie [9] parmi `m.(i-1).(j)`, `m.(i+1).(j)`, `m.(i).(j-1)` et `m.(i).(j+1)`.

Calcul de la matrice d'adjacence. Supposons `v.seed` de longueur égale à `n`. La matrice des régions `m` de `v` peut être utilisée pour calculer une matrice de booléens `b` de taille $n \times n$, remplie de la manière suivante : pour tous `h`, `k` distincts, `b.(h).(k)` vaut `true` si et seulement s'il existe (`i`, `j`) tel que `m.(i).(j)` soit égal à `h`, et tel que l'une au moins des valeurs définies parmi `m.(i-1).(j)`, `m.(i+1).(j)`, `m.(i).(j-1)`, `m.(i).(j+1)` soit égale à `k`.

La matrice `b` est appelée *matrice d'adjacence* de `v`. Etant données deux régions associées à deux germes distincts d'indices `h`, `k` dans `v.seed`, ces régions sont dites *adjacentes* si et seulement si `b.(h).(k)` vaut `true`. Cette matrice pourra être calculée par une fonction

```
adjacences_voronoi : voronoi -> int array array -> bool array array
```

7. Voir les triangulation de Delaunay : `https://en.wikipedia.org/wiki/Delaunay_triangulation`

8. Dans le cas la distance euclidienne, la seule chose qui importe est la *comparaison* des distances. On peut donc se servir, pour cet argument, d'une fonction calculant le *carré* de la distance euclidienne de deux points à coordonnées entières. Son type sera bien `int * int -> int * int -> int`.

9. Ces quatre valeurs ne seront pas toutes définies pour un pixel situé en bord de fenêtre.

prenant en argument un diagramme de Voronoï (de type `voronoi`), sa matrice des régions `m` (de type `int array array`), et renvoyant sa matrice d'adjacence (de type `bool array array`).

Construction d'une fnc. La construction d'une forme normale conjonctive satsifaisable si et seulement si le puzzle à une solution (et indiquant dans ce cas une solution via les valeurs trouvées pour ses variables) peut se faire à l'aide d'une fonction

```
produce_constraints :
  color option array -> bool array array -> (bool * Variables.t) list list
```

Le premier argument est un tableau indiquant, pour chaque numéro de germe, si sa région est pré-coloriée dans la configuration initiale (`Some c` si elle l'est, `None` sinon). Le deuxième est la matrice d'adjacence de la carte. La fnc produite doit exprimer les trois propriétés suivantes :

— *Existence :* chaque région reçoit au moins une couleur.

— *Unicité :* chaque région reçoit au plus une couleur.

— *Adjacence :* deux régions adjacentes ne reçoivent pas la même couleur.

Extraction d'un coloriage à partir d'une affectation. Une fois une affectation calculée par `solve` pour la fnc produite, il devrait être facile d'associer une couleur à chaque région : si la carte a effectivement une solution et si votre fnc est correctement construite, la fonction `solve` devrait vous proposer une affectation spécifiant la couleur de chaque région.

6 Implémentation de la boucle d'interaction

Les éléments qui suivent sont imposés : votre programme doit implémenter au minimum les fonctionnalités décrites.

Sans tenir compte de ses extensions éventuelles, votre programme devra effectuer au moins les opérations suivantes :

0. Choisir aléatoirement une valeur parmi ceux définis dans le fichier `example.ml` [10], puis afficher dans une fenêtre graphique le diagramme correspondant.

1. Attendre une action de l'utilisateur, en mettant à jour l'affichage après chaque action traitée, en lui permettant :

— de colorier une région non encore coloriée à l'aide d'une couleur déjà présente ; si la configuration est gagnante, le féliciter et aller en **(2)**, sinon, revenir en **(1)** ;

— de supprimer ou modifier la couleur d'une région coloriée, à condition qu'elle n'ait pas été coloriée dans la configuration initiale ; revenir en **(1)** ;

— d'afficher la solution de la grille (voir Section 4) ; aller en **(2)**.

2. Proposer à l'utilisateur de quitter le jeu ou de revenir en **(0)** en jouant, s'il en reste, sur une grille de `example.ml` non encore jouée.

10. Pendant la soutenance, nous testerons votre programme en le recompilant avec notre propre fichier `examples.ml`. Il est donc essentiel que vous respectiez le choix des types décrit à la section 3.

7 Modalités de rendu

Le projet est à réaliser par groupes de 2 personnes au plus – les projets réalisés à plus de 3 personnes ne seront pas acceptés. Votre rendu consistera en :

- un code-source écrit en OCaml, compilable et utilisable tel quel sous Linux et/ou sur les ordinateurs de l'UFR ;
- un fichier texte nommé README contenant vos noms et indiquant brièvement comment compiler votre code-source et, si nécessaire, comment utiliser l'exécutable produit ;
- un rapport de quelques pages (au plus 5) au format PDF décrivant votre projet ainsi que les extensions réalisées, expliquant et justifiant les choix de conception ou d'implémentation que nous pourrions ne pas comprendre du premier coup ;
- tout autre fichier nécessaire à la compilation et à l'exécution : fichiers d'exemples, `Makefile`, script permettant de compiler votre code-source, etc.

Tous ces éléments seront placés dans une unique archive compressée en `.tar.gz`. L'archive devra *obligatoirement* s'appeler *`nom1-nom2`*`.tar.gz`, et s'extraire dans un répertoire *`nom1-nom2/`*, où *`nom1`* et *`nom2`* sont les noms des deux personnes constituant le groupe. Par exemple, si vous vous appelez Pierre Corneille et Jean Racine, votre archive devra s'appeler `corneille-racine.tar.gz` et s'extraire dans un répertoire `corneille-racine/`.

La date limite de soumission est le vendredi 6 janvier à 23h59.

La soumission se fera via la page Moodle du cours. Attention, il s'agit d'un projet, pas d'un TP — vous devez donc nous fournir un programme non seulement bien écrit, mais faisant au moins quelque chose. Il vaut mieux un projet incomplet mais bien écrit et dans lequel une partie des choses fonctionnent, plutôt que d'essayer de tout faire, et de nous rendre un programme mal rédigé et/ou dans lequel rien ne fonctionne.