Rapport du projet de programmation fonctionnelle : Mappagani

Kostia CHARDONNET Sambo Boris ENG
L3 Informatique (Groupe 2)

1 Présentation du projet

L'objectif du projet est de concevoir et d'implémenter un **jeu de puzzle** où le but est de colorier une carte avec quatre uniques couleurs sans que deux couleurs côte-à-côte soit de même couleur. Le programme fournit une interface graphique permettant une interaction entre un joueur et le jeu mais introduit aussi des possibilités annexes comme la résolution automatique du puzzle qui équivaut à un abandon.

Notre programme se découpe en 9 fichiers (donc modules) dont les relations de dépendances sont décrites ci-dessous :

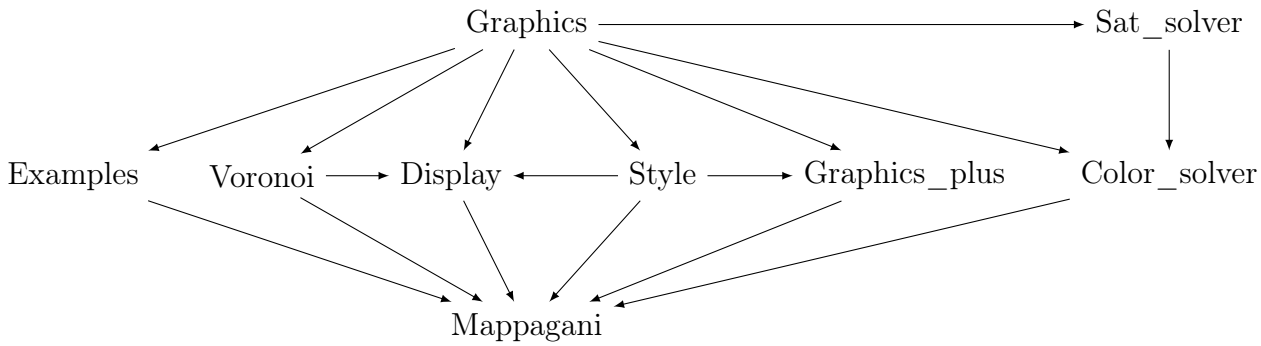

FIGURE 1 – Graphe des dépendances du projet. On considère que $A \to B$ si et seulement si B est dépendant de A.

Dans ce document nous visons la description de nos modules et notre méthodologie (en prodiguant des détails sur notre prise en main du problème et l'implémentation du jeu).

2 Description des modules essentiels

On omet la présentation du module *Graphics* d'OCaml, *Sat_solver* et *Examples* (tous deux donnés par les enseignants), considérés comme déjà connus par le lecteur.

Globalement notre projet se divise en deux parties **Manipulation graphique des voronoi** (voronoi) et **Résolution logique du jeu** (color_solver), une dichotomie qui a aussi pu définir nos rôles respectifs dans le projet.

En ce qui concerne le reste (hors extensions), nous avons séparé fonctions d'affichages (display) et boucle d'interaction (mappagani).

Voronoi

Ce module permet de réaliser des opérations sur le type *voronoi*. Il y a principalement deux fonctions intéressantes : `regions_and_pixelList` et `adjacences_voronoi`.

La fonction `regions_and_pixelList` permet de calculer la matrice des régions et liste qui contient pour chaque indice de seed la liste des pixels de la région correspondante. Cette liste de pixels nous permet de colorier uniquement certaines régions plutôt que de re-colorier entièrement la carte.

Pour la matrice des régions on calcule la distance chaque de pixel à chaque seed dans une liste et on récupère l'indice du seed le plus proche. Cet indice va identifier la région et ainsi la couleur que devra prendre le pixel.

La fonction `adjacences_voronoi` permet de calculer la matrice d'adjacence : une matrice booléenne qui nous permet de connaître les relations d'adjacence entre les régions. Elle est calculée depuis la matrice des régions : on détermine les relations d'adjacences par extraction des régions les plus proches aux "points de frontières" entre les régions. Par définition, un point de frontière est équidistant à au moins deux seeds. Cette matrice est particulièrement utile pour la résolution du jeu (voir *color_solver*).

Nous avons aussi d'autres fonctions secondaire comme `adjacents_to` qui renvoie la liste des régions adjacentes à une région ainsi que `is_complete_voronoi` qui nous permet de vérifier si toutes les régions du voronoi ont été coloriées.

Color_solver

Ce module permet de générer des formules représentant les contraintes du jeu afin de les passer à un *SAT Solver* qui pourra résoudre les contraintes et par conséquent donner une solution au jeu. Il contient aussi une fonction (assez triviale) permettant de vérifier si la carte est bien coloriée (pour vérifier l'existence d'une éventuelle victoire du joueur).

Il s'est avéré que chaque contraintes pouvaient s'exprimer assez intuitivement à l'aide de la fonction `List.fold_left` à condition de les appliquer sur les bons objets. Soit S une séquence ordonnée d'identifiants de seeds et C l'ensemble des couleurs.

Contrainte d'existence. Formellement nous avons utilisé la notation suivante pour les variables : $[i, c]$ pour *"Le seed d'indice i est coloriée avec la couleur c"*. La contrainte d'existence se présente formellement ainsi : $\bigwedge_{i \in S} \bigvee_{c \in C} [i, c]$.

Au niveau de l'implémentation on a une fonction `(i -> map (c -> (true, [i, c])) C)` qu'on va appliquer à chaque $i \in S$ et accumuler successivement les clauses produites dans une liste grâce à `List.fold_left` pour générer la contrainte voulue.

Contrainte d'unicité. Pour définir l'unicité des couleurs pour chaque seeds, nous avons retenu la tentative suivante : définir la contrainte sous forme purement clausale (sans faire appel à une quelconque implémentation de la logique propositionnelle). Nous formalisons la contrainte ainsi : $\bigwedge_{i \in S} \bigwedge_{(c_1,c_2) \in C^2} \neg[i, c_1] \vee \neg[i, c_2]$. Ainsi pour tout seed et on exclut au moins une couleur pour chaque choix possible.

Dans l'implémentation la fonction de génération des choix de couleur n'est pas un pur produit cartésien et retient uniquement les couples significatifs. La contrainte se génère finalement à l'aide de deux `List.fold_left` imbriqués.

Contrainte d'adjacence. La définition de cette contrainte rejoint un peu la même philosophie d'exclusion que la précédente mais défini informellement par *"Prenons un i quelconque. Pour chacun de ses voisins j, et toute couleur c, l'un des deux n'a pas cette couleur"*. Pour nous aider nous avons une fonction `adjacent_to i adj` qui effectue une simple lecture de la matrice d'adjacence `adj` pour le seed indicté par i.

Contrainte de présence. La contrainte de présence ne doit pas être oubliée. Elle créer simplement une clause unitaire pour chaque couleur déjà présente sur la carte afin de prendre en compte la configuration courante du jeu.

La conjonction des contraintes est ensuite simplement passée à un *SAT Solver* nous on extrait le résultat pour l'exploiter en générant une liste d'associations (identifiant de seed, couleur).

Display

Ce module contient toutes les fonctions d'affichage principales : affichage de la carte, affichage d'une région en particulier (permet de colorier une seule région), affichage de la carte en noir (pour signifier une fin de partie).

Les fonctions étaient au départ définies à l'aide de boucles FOR (car plus intuitives). Dans une volonté de produire un code purement fonctionnel, comme les boucles ne s'appliquaient que sur des tableaux (`Array`) nous avons utilisé la correspondance (FOR sur `Array` ≡ `Array.iteri`) illustrée ci-dessous :

```
for i = 0 to Array.length m do
  for j = 0 to Array.length m.(0) do
      ...
  done;
done;;
```

```
Array.iteri (fun i line ->
  Array.iteri (fun j _ ->
    ...
  ) line
) m;;
```

Mappagani

Le module *Mappagani* représente le module principal, celui contenant le moteur du jeu, la boucle d'interactions.

La configuration d'une partie est marquée par deux concepts :

— L'**état** du programme : PLAY, QUIT, END (fin de partie, post-affichage de solution), WIN (victoire d'une partie), NEWMAP, RESET, GAMEOVER (fin de jeu). Les transitions d'états sont représentées par le graphe ci-dessous qu'on peut aussi voir comme un graphe des interactions.
Les transitions avec des flèches en pointillés représentent des transitions vers des états temporaires et instables dits *"de requêtes"* capturés par le programme pour revenir sur un état stable.

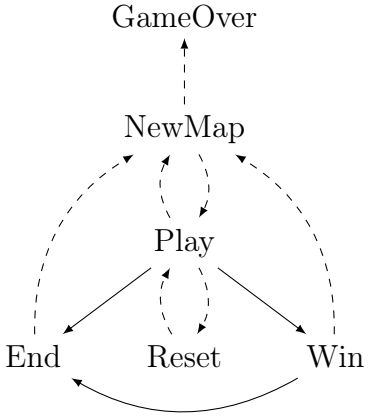

FIGURE 2 – Graphe des états : cela a été omit pour plus de lisibilité mais tous les états peuvent aller vers QUIT

— Les **flags** qui représentent des configurations mutables (activables/désactivables) au cours de l'exécution du programme (affichage d'un marquage pour les seeds originellement coloriés par exemple).

Architecture. La définition du jeu est ensuite assez linéaire : le programme compose son interface graphique grâce à notre module *Graphics_plus* (menu, boutons) et *Graphics* d'OCaml (affichages, fenêtre) et on retrouve dans la boucle principale (fonction `game`) une grande condition permettant de gérer l'interprétation des états du programme.

Transition (partie → partie'). Un point important est que nous travaillons toujours sur une copie du voronoi courant car l'original peut servir à diverses tâches comme recommencer la partie ou vérifier les régions déjà coloriées au départ. La transition d'une partie à une autre se fait par transfert *d'environnement* : on passe en paramètre de nombreuses informations comme le menu, l'état du programme, les matrices de région et d'adjacences, les flags etc.

3 Extensions réalisées

Style

Le module *Style* est un simple masquage de la notion de couleur dans le module *Graphics* d'OCaml. Il redéfinit simplement les couleurs par d'autres valeurs et définit des couleurs

pour l'interface graphique que nous utilisons.

La façon correcte de voir ce module est en fait en tant qu'instance parmi d'autres d'une notion plus générale de "style". C'est un moyen assez naïf mais direct de moduler les styles sans passer par d'autres méthodes (feuille de style comme dans le langage *CSS* par exemple).

Graphics_plus

Graphics_plus est un super-ensemble du module *Graphics* d'OCaml paramétré par un "style" au sens défini précédemment. Le module combine simplement des fonctions primitives de *Graphics* afin d'apporter des fonctionnalités plus avancées et modulables. La mise en place d'une interface plus travaillée permet d'augmenter l'ergonomie du jeu et rendre le déroulement d'une partie plus intuitive et agréable.

Nous avons introduit les fonctionnalités suivantes :

— **Boutons** : un bouton est défini par un type *record* contenant une position, une taille, un label, un état (activé/désactivé) mais surtout une action représentée par une fonction de type `unit -> unit`. Nous avons développé diverses fonctions utilitaires permettant de les utiliser : vérifier si le curseur de la souris est dans un bouton, afficher un bouton, changer son état etc.

— **Menu de boutons** : s'ajoute à la notion de bouton, une notion de menu définie comme étant une liste de boutons. On retrouve ainsi des fonctions généralisées comme la désactivation ou désactivation totale ou partielle d'un menu.

— **Images** : des fonctions d'affichage d'images très primitives sont fournies dans le module *Graphics*. Nous les avons exploité afin de pouvoir afficher des images au format *bitmap* (.bmp) aisément manipulables pour afficher des éléments graphiques comme le logo du jeu.