Projet de Programmation Fonctionnelle 2016-2017 : Coloriage de cartes

Clara BRINGER et Pierre GERVAIS

Nous avons implémenté en OCaml un jeu sur le thème du théorème des quatre couleurs. Ainsi, lorsque le jeu est lancé, une carte s'affiche aléatoirement sur laquelle nous pouvons jouer manuellement, ou choisir d'afficher la solution.

Un fichier README.md expliquant comment compiler et exécuter le programme est fourni, ainsi qu'un fichier Makefile permettant la compilation.

Pour implémenter ce jeu, nous avons défini dans un fichier voronoi.ml les types *seed* et *voronoi*, et nous avons ouvert ce fichier dans le fichier principal, diagram.ml.

Dans diagram.ml, nous avons commencé par définir le type *game_state*, qui correspond au moment du jeu, ainsi que les modules *Variables_Voronoi*, dont le type est un couple d'un entier (représentant l'indice d'une région) et d'une couleur (correspondant à la région), et *Solver*, grâce au fichier sat_solver.ml fourni. Une variable de type *game_state* peut valoir : *Initialization* pour l'initialisation du jeu (choix de la fonction de distance, sélection aléatoire du diagramme parmi ceux proposés, ouverture du graphe) ; *In_game* pour le cours de la partie (tant que l'utilisateur n'a pas gagné ou n'a pas affiché la solution) ; *Between_games* pour proposer à l'utilisateur de rejouer ou de quitter s'il a gagné ou affiché la solution.

Nous avons ensuite défini une fonction de distance, permettant de calculer la distance entre deux points selon un nombre flottant passé en argument. En particulier, nous avons défini les distances taxicab et euclidienne selon les nombres 1 et 2 respectivement.

Puis nous avons implémenté la fonction *regions_voronoi* permettant de calculer les régions d'un diagramme. Elle commence par créer une matrice d'entiers de taille la dimension du diagramme, chaque case de cette matrice représentant ainsi un point de la carte. Elle parcourt ensuite la matrice

et la remplit avec l'indice de la région pour laquelle la distance entre son germe et le point courant est minimale.

Ensuite, nous avons créé la fonction *adjacences_voronoi* permettant de savoir si deux régions sont adjacentes ou non. C'est une matrice de booléens respectant les conditions décrites dans le sujet.

A partir de la fonction *regions_voronoi*, nous avons pu définir la fonction *draw_voronoi* permettant d'afficher un graphe dans une fenêtre de la hauteur du graphe mais plus large que lui, afin d'afficher sur le côté des boutons permettant le déroulement du jeu, c'est-à-dire des boutons de sélection de couleur, un bouton pour afficher la solution, un pour quitter le jeu et un pour rejouer. Nous avons également laissé un espace vide afin de pouvoir afficher d'éventuels messages, grâce à la fonction *draw_string_message*.

Pour afficher ces boutons, nous avons créé deux fonctions auxiliaires, *draw_color_squares* permettant d'afficher les boutons de sélection de couleur, *draw_buttons* permettant d'afficher les trois autres.

Pour afficher la carte, nous avons utilisé la matrice des régions et coloré chaque point selon la couleur du germe indiqué dans la case correspondante de cette matrice. Pour délimiter les régions, nous avons coloré en noir les points respectant les conditions décrites dans le sujet.

Puis nous avons défini une fonction qui renvoie un tableau des couleurs actuelles du jeu pour chaque région, qui servira à calculer les fonctions *exists*, *unique* et *adjacent* explicitées ci-dessous.

Pour résoudre un diagramme, nous avons créé la fonction *produce_constraints* permettant de produire la FNC nécessaire à l'utilisation du solveur fourni. Pour cela, nous avions besoin des fonctions *exists*, *unique* et *adjacent*, qui vérifient respectivement que pour chaque région, il existe une couleur ; que chaque région n'a au plus qu'une couleur ; et que deux régions adjacentes n'ont pas la même couleur. Si l'une de ces conditions n'est pas respectée, alors le diagramme n'a pas de solution. Chacune produit une liste de ces contraintes calculées, et *produce_constraints* produit la liste de toutes ces contraintes.

Nous avons ensuite pu définir une fonction *solve_voronoi* permettant de résoudre le diagramme grâce à la liste de contraintes produites et à la fonction *solve* fournie.

Puis nous avons commencé à coder le déroulement du jeu. Nous avions besoin des fonctions *color_region* (qui colorie une région vide) et *change_region* (qui change ou supprime la couleur d'une région colorée initialement vide).

Ensuite, nous avons défini une fonction *win*, déterminant si la coloration actuelle est gagnante. Une coloration n'est pas gagnante si toutes les régions ne sont pas colorées, ou si deux régions adjacentes ont la même couleur.

Grâce à ces fonctions, nous avons pu commencer le déroulement du jeu.

Quand la variable *game_state* (de type *game_state*) vaut *In_game*, le programme attend une action de l'utilisateur grâce à la fonction *first_click*, qui permet au joueur de cliquer sur la fenêtre autant de fois que nécessaire jusqu'à ce qu'il ait sélectionné une couleur ou le bouton de solution. Puis la fonction *second_click* lui permet de cliquer sur la fenêtre autant de fois que nécessaire jusqu'à ce qu'il ait sélectionné une région, le bouton de solution ou une nouvelle couleur. Dans ce dernier cas, la nouvelle couleur est prise en compte, et le jeu attend une nouvelle action de l'utilisateur. Puis, si le bouton de solution n'a pas été sélectionné, l'affichage se met à jour avec la nouvelle couleur pour la région sélectionnée. Si la nouvelle configuration n'est pas gagnante, on recommence.

Si le bouton de solution est sélectionné, soit il existe une solution pour la coloration actuelle, et elle s'affiche, et game_state prend la valeur Between_games ; soit il n'existe pas de solution pour la coloration actuelle, et la fenêtre se ferme.

Quand *game_state* vaut *Between_games*, l'affichage se met à jour, et un message de félicitation s'affiche si le joueur a gagné. Une fonction *third_click* permet à l'utilisateur de sélectionner un bouton : celui pour rejouer, ou celui pour quitter. S'il veut rejouer, le programme reprend au début de la fonction principale, avec en argument la liste des diagrammes restants (c'est-à-dire la même liste que précédemment, mais en ayant enlevé le diagramme actuel grâce à la fonction *delete_diagram*). S'il veut quitter, la fenêtre est fermée.

Afin de choisir aléatoirement une carte, nous utilisons la liste de diagrammes *list* définie dans examples.ml contenant les différents diagrammes décrits dans ce même fichier. Pour ajouter un diagramme, il suffit de rajouter son code dans le fichier, et de rajouter une case à la liste *list* le contenant.