Flood it!

Rapport Lot C

May 15, 2017

Groupe : Les Prez (*Sylia Mehenni, Rémi Jamous, Simon Chollet, Elyne Amsellem*)

Sommaire

Objectif: Réalisation d'un solveur de jeu avec affichage en mode texte dans le terminal des différentes solutions trouvées.

1 Introduction

Remarque : ce rapport traite uniquement du travail réalisé pour le Lot C, en tenant compte des fonctions codées pour les Lots A et B. Un rapport d'ensemble sera réalisé lors de la dernière étape du projet.

Dans cette troisème étape, le but est de réaliser un solveur naïf (avant une version optimisée) qui trouve toutes les solutions possibles d'une partie en parcourant chaque branche de l'arbre. Lorsque le solveur termine, il affiche chaque solution (couleurs jouées dans l'ordre jusqu'à la victoire) une par une.

Les objectifs de ce troisième lot sont d'utiliser les structures de données et les fonctions de manipulation de ces structures codées dans les lots A et B pour réaliser le solveur.

2 Organisation

2.1 Répartition des taches

Référent Sprint 3 : Rémi Jamous

Simon et Elyne ont codé le programme principal du solveur naïf, ainsi que les fonctions auxilliares utiles. Sylia s'est occupée des commentaires, du fichier Doxygen et des tests Valgrind. Rémi et Simon a supervisé la répartition des taches et le débuggage d'une partie du code. Rémi et Simon ont également travaillé sur le solveur optimisé.

2.2 Mise en commun

Le code est mis en commun sur *https://github.com* afin que chacun puisse apporter des modifications si nécessaire et ajouter des commentaires. On prend soin de commenter efficacement le code avec les paramètres que prend la fonction et ce qu'elle retourne.

On utilise ensuite le générateur de document *Doxygen* pour récupérer un fichier .html contenant l'ensemble du code et des commentaires. Il a fallu pour cela commenter le code selon les conventions reconnues par *Doxygen* et télécharger l'application qui permet de réaliser le fichier .html.

2.3 Tests

On réalise des tests en compilant le programme dans un terminal, avec le nouveau *Makefile* que l'on a codé. Ces tests permettent d'éliminer les fautes de codes qui peuvent créer des erreurs et de vérifier que le code fait ce qu'on lui demande.
Chaque fonction contient également des tests intrinsèques au code pour vérifier où se trouvent les bugs (par des *printf* par exemple), cf Annexes.

2.4 Outils

On ajoute à notre programme un *readme.txt* afin de permettre au correcteur/utilisateur de comprendre le fonctionnement du jeu avec le solveur intégré. Cela ne change pas grand-chose, si ce n'est que le nombre de coups autorisés n'est pas laissé au choix du joueur mais calculé par le solveur (cf *Main*).

3 Fonctions

On code les fonctions suivantes, en plus des fonctions du Lot A et du lot B:

- On code une structure de pile nommée *pile_ couleur* avec les fonctions usuelles de manipulation de piles (*new_ pile_ couleur()*, *est_ vide()*, *supprime_ pile()*, *depile()*, *empile()*).
- *trouve*, fonction qui prend en argument une pile_couleur. La fonction est de type *void*, elle se contente de stocker une solution trouvée dans une pile temporaire et de l'afficher.
- *comparaison_ plateau*, fonction qui prend en arguments 2 plateaux (type structure définie dans le Lot A) et de comparer leurs taches respectives pour déterminer si ce sont ou non les mêmes. Le but est de tester si il y a eu un changement, on appellera cette fontion après un appel à *colorie_ tache*. Cette fonction renvoie *True* si les 2 taches sont identiques (ie un nb différent de 0) et *False* sinon (0).
- *solveur_ beta*, une fonction principale de ce lot, pour coder le fonctionnement du solveur. Elle prend en arguments 2 plateaux, un du jeu et un pour la tache initiale. On ajoute un pointeur d'entiers pour le nombre de coups maximum autorisé, et une *pile_ couleur* pour stocker le chemin parcouru dans l'arbre. Cette fonction crée une copie du plateau de jeu et de la tache. Ensuite pour chaque couleur du jeu (Jaune, Bleu, Rouge, Vert, Marron, Gris), elle change la couleur de la tache et calcule le nombre de coups césessaire à la victoire. Elle affiche ensuite le résultat. On fait un test de comparaison entre la tache initiale et la nouvelle tache, si elles sont différentes on teste la condition de victoire, sinon on fait un appel récursif au solveur. Cet algorithme est considéré comme notre solveur naïf, cependant il est déja un peu optimisé car il stocke la taille de la solution trouvée et ne cherche pas de solution qui nécessite un plus grand nombre de coups. Cf Annexes.
- *somme*, fonction qui retourne la somme (entier) des éléments de la matrice qui représente une tache.
- *tri*, une fonction de type *void*, qui prend en argument 2 tableaux d'entiers; le deuxième contiendra les indices du premier correspondant aux éléments du premiers dans l'ordre decroissant (perme d'accéder facilement aux indices du premier tableau en commençant par l'indice qui correspond au plus grand élément).
- *solveur_ opti*, une autre version du *solveur_ alpha* optimisée. En effet, il va regarder la taille de la tache pour chacune des couleurs et faire les appels récursif en commençant par la couleur qui augmente le plus la taille de la tache. Cela permet de trouver rapidement (Remarque : Rapidement dans le sens que la pile d'appel sera rapidement bornée pour un nombre pas trop grand) une solution qui est proche de la meilleure solution et donc d'optimiser les calculs.

4 Difficultés rencontrées, remarques sur le code

Dans cette seconde partie, nous avons rencontrés des difficultés vis-à-vis de l'optimisation du solveur. Coder le solveur *naïf* n'était pas trop compliqué car l'agorithme était plus ou moins fourni (mis à part des petits problèmes comme le fait qu'il ne terminait pas) mais l'intérêt était de réussir à trouver un solveur plus performant. Nous avons donc rajouteé à la première version (cf fichier solveur_alpha.c et solveur_omega.c) quelques fonctions (*tri, somme*) afin de réduire le nombre de calculs effectués par l'ordinateur lors de l'éxécution du programme.

5 Annexes

Pour ce lot, on a donc 2 fichiers **Main** selon si on utilise le solveur naïf ou optimisé. Ci-dessous, les codes utilisant chacun des algorithmes.

```
int main() {
  int n = 4;
  plateau jeu = aleatoire(n, 6);
  plateau tache = new(n);
  pile_couleur pc;
  tache.contenu[0][0] = 1;
  pc = new_pile_couleur(n*n);
  int nb_coups=-1;
  aff(jeu);
  solveur_beta(jeu, &nb_coups, &pc, tache);
  supprime(&jeu);
  supprime(&tache);
  supprime_pile(&pc);
  return(0);
}
```

```
int main() {
  int n = 4;
  plateau jeu = aleatoire(n, 6);
  plateau tache = new(n);
  pile_couleur pc;
  tache.contenu[0][0] = 1;
  pc = new_pile_couleur(n*n);
  int nb_coups=1000;
  aff(jeu);
  solveur_opti(jeu, &nb_coups, &pc, tache);
  supprime(&jeu);
  supprime(&tache);
  supprime_pile(&pc);
  return(0);
}
```

On réalise les tests avec *Valgrind* pour tester les fuites mémoires dans un fichier en annexe au dossier, qu'il faut également compiler. Ci-joints les résultats.