Rapport Méthodes et Outils pour la Conception Avancée

Castel Antonin, Reboul Paul, Vandendorpe Thomas

March 19, 2018

1 Introduction

Ce rapport présente les modifications apportées au programme de puissance 4 au fil des séances. Nous nous sommes dans un premier temps familiarisé avec le code donné, et l'avons rendu plus modulaire, puis nous avons étudié des principes de tests, et enfin nous avons fini par utiliser des outils permettant de trouver des erreurs.

2 Modularité, maintenabilité, réutilisabilité

Les 3 premières séances avaient pour but de découvrir le code fournis, de corriger les erreurs les plus flagrantes, puis de séparer le code en plusieurs fichiers afin d'avoir un code plus modulaire. Nous avons également créé un git afin de faciliter le travail en groupe et de garder un historique des modifications effectuées.

2.1 Séance 1

Nous avons corrigé les premières erreurs détectées lors de la compilation avec les warning: variables déclarées mais non utilisées, fonction sans valeur de retour. Puis à l'aide de gdb nous avons repéré l'endroit où se trouvait une erreur de segmentation venant d'une mauvaise allocation de la mémoire. En continuant à utiliser l'application, nous avons observé et corrigé d'autres erreurs notamment lors de l'ouverture de fichier ou de l'affichage.

Résumé des modifications

- fonction XMLformating: Suppression variables inutilisées. Passage en argument des noms de fichiers. Gestion des erreurs en cas de fichier inexistant.
- fonction Medium: Ajout valeur de retour.
- fonction Main: Allocation du board.

- fonction Print: Modification de l'affichage. Ajout de define pour les éléments des cases du tableau(case vide).
- fonction saveLoad: gestion fichier inexistant.
- fonction highScore: gestion fichier inexistant.

2.2 Séance 2 - Séance 3

Lors de ces séances nous avons séparé notre programme en plusieurs fichiers. Nous avons ainsi plusieurs .c dans un dossier src/ et plusieurs .h dans un dossier include/. L'utilisation de variables globales par le fichier de base pose problème pour la séparation en plusieurs fichiers, deux choix sont possibles: utiliser des variables extern ou créer un structure (qu'on passera donc en arguments aux fonction en ayant besoin). Nous avons choisi cette seconde option qui bien que nécessitant de modifier beaucoup le code nous semble plus facile à être utilisé par la suite. Nous avons profité de ce découpage pour essayer de comprendre ce que sont censées faire certaines fonctions.
La découpe du code est faites en 5 fichiers .c (et leur .h):

- appli.c : le main
- board.c/board.h : les fonctions manipulant la structure board(le plateau de puissance 4)
- action.c/action.h : les actions(les règles) du jeu
- ia.c/ia.h : les IAs
- score.c/score.h: les informations de scores

Ce découpage est amené à changer au fure et à mesure de notre compréhension globale du programme.

Une fois le programme découpé, nous avons créer un Makefile permettant la compilation de fichiers séparés. Nous avons pris soin de faire un Makefile qui soit générique et qui permet de facilement modifier les options de compilations.

Résumé des modifications

- Structuration du code en plusieurs fichiers
- Création du Makefile
- Spécification de quelques fonctions

3 Qualité des tests, analyse de couverture

3.1 Séance 4

Cette séance était dédiée aux tests. Dans un premier temps nous nous sommes occupés de la mesure de la couverture de tests du programme grâce à l'outil gcov. Cet outil nécessite de compiler avec des flags spécifiques, nous avons donc modifié notre Makefile en conséquence. Nous avons ensuite exécuté des tests et avons pu observer la couverture grâce à gcov. Afin d'avoir un aperçu visuel de la couverture nous avons utilisé lcov qui génère des rapports de couverture en HTML. Notons que la couverture ne donne d'information que sur les parties du programme qui ont été exploré, et pas sur la qualité des tests. C'est pourquoi nous avons ensuite travaillé sur l'automatisation de tests à l'aide de CuTests. Ces tests unitaires sont ajouté à notre code dans un fichier .c. Cependant nous avons rencontré plusieurs problèmes:

1. L'exécution des tests se font à partir d'une autre fonction main, avec deux main notre Makefile ne fonctionnait plus!
2. Il restait encore des fonctions dont on ignorait ce qu'elles faisaient, et qu'on ne pouvait donc pas tester.
3. Certaine fonction ne se prêtaient pas aux tests unitaires (pas de valeur de retour ou valeurs de retour mal choisies)
4. Impossibilité de tester la fonction main du puissance 4(qui était assez conséquente en taille)

Nous avons donc répondu à ces problèmes en:

1. Améliorant encore notre Makefile pour qu'il fonctionne malgrè les deux fonctions main.
2. Comprenant mieux les fonctions indéterminés, quitte à les récoder lorsqu'elles sont trop compliquées(les fonctions du sytème d'undo/redo par exemple)
3. Mettant des valeurs de retour plus adaptés lorsque c'est possible
4. Découpant la fonction main en plusieurs petites fonctions

Résumé des modifications

- Modification du Makefile pour prendre en compte la couverture et plusieurs main
- Tests pour observer la couverture
- Tests unitaires
- Découpage du main
- Réécriture et redéfinition de certaines fonctions

4 Détection de défauts et erreurs

4.1 Séance 5 - Séance 6

Une première partie consistait à utiliser Klee, qui est un outils simulant des exécutions du programme. Suite au difficulté d'adapter notre Makefile et notre code pour Klee, nous avons effectué cette partie sur le code initial du puissance 4 (comme le prévoyait l'énoncé). L'utilisation de Klee a nécessité de remplacer tous les scanfs par des fonctions de klee et à masquer les printf. Klee a pu détecter 3,4 erreurs d'accès mémoire invalides. La seconde partie consistait à utiliser valgrind sur notre programme. Nous avons encore trouver des erreurs d'accès mémoires, venant principalement des accès au tableau représentant le puissance 4.

Résumé des modifications

- Correction de bugs

5 Analyse de Performances

5.1 Séance 7

Lors de cette séance, nous avons utilisé gprof pour analyser notre programme, et ainsi déterminer quelles fonctions étaient les plus coûteuses. Cet outil nécessite que notre programme tourne un certain temps afin d'avoir des résultats exploitables. En utilisant une trop petite grille comme configuration pour notre puissance 4, on ne peut rien en tirer. Nous avons donc réalisé le profiling avec pour un puissance 4 de hauteur 100 et de largeur 100. De plus, nous avons généré des entrées à l'aide d'un script afin de ne pas avoir à jouer de partie "à la main". Voici un aperçu du résultat obtenu:

```
Flat profile:

Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls  us/call  us/call  name
56.46      9.76     9.76     9907   985.34   985.34  board_print
11.97     11.83     2.07    10000   207.04   207.04  diagonal
11.16     13.76     1.93    10000   193.03   193.03  horizontalScore
10.47     15.57     1.81    10000   181.03   181.03  verticalScore
 9.83     17.27     1.70    10000   170.03   170.03  diagonal1
 0.06     17.28     0.01    14906     0.67     0.67  checkfull
 0.06     17.29     0.01                             main
 0.00     17.29     0.00    24906     0.00     0.00  board_currentPlayerSymbole
 0.00     17.29     0.00    10000     0.00     0.00  board_put
 0.00     17.29     0.00     9906     0.00     0.00  board_colIsFull
 0.00     17.29     0.00     9906     0.00     0.00  checknum
                              ...
```

On voit que les fonctions coûteuses sont la fonction d'affichage, et les fonctions de calcul du score. On pourra difficilement optimiser la fonction d'affichage qui nécessitera toujours de parcourir le tableau. Par contre, les fonctions de calculs des scores peuvent être facilement améliorées. Au lieux de parcourir tout le

tableau du puissance 4 pour recalculer tous les scores, on peut simplement mettre à jour les scores en prenant en compte seulement les changements affectés par le coup joué précédent, on fait ainsi moins de calculs, mais on stocke plus de chose en mémoire.

Nouveau profiling avec les modifications:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 99.37      6.10     6.10     9907   615.83   615.83  board_print
  0.16      6.11     0.01    40000     0.25     0.25  board_lastPlayRow
  0.16      6.12     0.01    14906     0.67     0.67  checkfull
  0.16      6.13     0.01    10000     1.00     1.25  diagonal1
  0.16      6.14     0.01    10000     1.00     1.25  horizontalScore
  0.00      6.14     0.00    80000     0.00     0.00  board_lastPlayCol
  0.00      6.14     0.00    24906     0.00     0.00  board_currentPlayerSymbole
  0.00      6.14     0.00    10000     0.00     0.00  board_put
  0.00      6.14     0.00    10000     0.00     0.25  diagonal
  0.00      6.14     0.00    10000     0.00     0.25  verticalScore
  0.00      6.14     0.00     9906     0.00     0.00  board_colIsFull
  0.00      6.14     0.00     9906     0.00     0.00  checknum
                                ...
```

Nous avons donc grandement amélioré les perfomances du programme, qui passe mainenant presque tout son temp à l'affichage. A noter que notre solution n'est pas du tout la plus optimale, mais les changements on été relativement simple et rapide à faire, ce qui est un bon compromis entre le temps passé à recoder les fonctions et le gain de perfomance.

Résumé des modifications

- Mise à jour Makefile pour profiling
- Profiling
- Ajout de commentaires pour doxygen

5.2 Séance 8

Lors de cette séance, nous avons utilisé Valgrind (perf n'étant pas disponible) pour analyser notre programme, afin de voir le pourcentage de "miss". Nous avons diminué de moitié la taux de cache misses en faisant des modifications dans notre code source, notamment au niveau des boucles et des conditions de branchement. Résultats après une partie 'joueur contre joueur' :

```
==31722== I    refs:      5,061,558
==31722== I1   misses:        1,584
==31722== LLi misses:        1,484
==31722== I1   miss rate:     0.03%
==31722== LLi miss rate:      0.03%
==31722==
==31722== D    refs:      2,113,672  (1,280,321 rd   + 833,351 wr)
==31722== D1   misses:        5,044  (     3,035 rd   +   2,009 wr)
==31722== LLd misses:        4,204  (     2,287 rd   +   1,917 wr)
==31722== D1   miss rate:       0.2% (       0.2%     +     0.2%  )
==31722== LLd miss rate:        0.2% (       0.2%     +     0.2%  )
==31722==
```

```
==31722== LL refs:            6,628  (     4,619 rd   +    2,009 wr)
==31722== LL misses:          5,688  (     3,771 rd   +    1,917 wr)
==31722== LL miss rate:         0.1% (       0.1%     +      0.2%  )
```

Le taux de miss que nous avons réussit à atteindre est correct, bien que encore améliorable, mais ça nécessiterait de modifier lourdement le code source, ce qui prendrait un temps significatif pour n'améliorer que très peu les performances.

Résumé des modifications

- Optimisation du code pour minimiser le taux de cache et branch misses

```
EXEC=appli AllTests
appli:
AllTests:

COUVERTURE=y
PROFILING=y

SRC=src
HEADERS=Include

CC=gcc
CFLAGS= -I./$(HEADERS) -g
LDFLAGS=

#############################################################################
ifeq ($(COUVERTURE),y)
        CFLAGS+= -fprofile-arcs -ftest-coverage
        LDFLAGS+= -fprofile-arcs -ftest-coverage
endif
ifeq ($(PROFILING),y)
        CFLAGS+= -pg
        LDFLAGS+= -pg
endif
#############################################################################

all: $(EXEC)

%.o:$(SRC)/%.c
        $(CC) $(CFLAGS) -c $< -o $@

%: %.o $(patsubst $(SRC)/%.c,%.o,$(shell grep -L "int main" $(SRC)/*.c))
        $(CC) -o $@ $^ $(LDFLAGS)

profiling: appli
        ./appli $(args)
        gprof ./appli > profiling.txt

clean:
        rm *.o *.gcda *.gcno $(EXEC)
```