Tests de couverture, analyse statique de code et profilage

Le but de ce TD est d'utiliser des outils de développement complémentaires au débogueur. Il s'agit essentiellement d'« utilitaires binaires GNU » (ou binutils) qui représentent des outils de développement logiciel.

1 Test de couverture

Vous allez utiliser l'utilitaire nommé *gcov* qui permet d'effectuer des tests de couverture de vos programmes. Utilisez-le en association avec *GCC* pour analyser vos programmes et vous aider à les rendre plus efficaces, plus rapides à l'exécution et pour rechercher des parties non testées de vos programmes. Vous pouvez utiliser *gcov* comme un outil de « *profilage* » afin de vous aider à déterminer quelles sont les parties de votre code qui méritent un effort d'optimisation. Vous pouvez également utiliser *gcov* avec d'autres outils de « *profilage* », comme par exemple *callgrind* pour évaluer quelles parties de votre code consomment le plus de temps de calcul.

Exercice 1 – Récupérer le fichier *fichiers-05.tar.gz*, puis désarchiver le.

Exercice 2 – Consultez l'introduction à gcov disponible à cette adresse https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro.

Exercice 3 – Compilez le fichier *foo.c* avec les options nécessaires à *gcov* : *-ftest-coverage -fprofile-arcs*.

Exercice 4 – Exécutez le programme ainsi généré en passant différentes valeurs en paramètres.

Exercice 5 – Executez la commande *gcov foo.c*. Cette commande analyse les différentes exécutions de votre programme. En particulier elle compte le nombre de fois que chaque ligne de code a été exécuté.

Exercice 6 – Le fichier généré, *foo.c.gcov*, indique pour chaque ligne le nombre de passages. Les lignes marquées ##### n'ont pas été exécutées. Certaines lignes n'ont pas été exécutées parce que vous n'avez pas assez "testé" le programme, d'autres parce qu'elles ne peuvent jamais être exécutées. Essayez de repérer ces dernières.

Exercice 7 – La boucle for (initialisation ; condition ; itération) est composée de 3 blocs (le bloc d'initialisation, le bloc de condition et le bloc d'itération). Utilisez *gcov* pour connaître le nombre de fois que votre programme est passé dans chaque bloc de la boucle for (option `-a`).

Exercice 8 – Expliquez le résultat produit par la commande : *gcov -b foo.c*

Exercice 9 – Interprétez le contenu du fichier *foo.c.gcov*

2 gcov et cmake

Exercice 10 – Créez un fichier `CMakeLists.txt` permettant de générer l'exécutable `foo`.

Exercice 11 – Ajoutez 2 tests permettant de tester foo avec comme paramètre 2 puis 5.

Exercice 12 – Compilez puis vérifiez que tout se passe correctement.

Exercice 13 – Ajoutez avec votre fichier `CMakeLists.txt` ce bout de code :

```
option(ENABLE_DEBUG "debug options" ON)
if(ENABLE_DEBUG)
  set(CMAKE_C_FLAGS "-std=c99 -g -fprofile-arcs -ftest-coverage -Wall")
  set(CMAKE_LD_FLAGS "-fprofile-arcs -ftest-coverage")
else()
  set(CMAKE_C_FLAGS "-std=c99 -O3 -Wall")
endif()
```

Ce bout de code vous permet de compiler votre code selon deux modes (mode debug ou pas). La commande `ccmake` vous permet d'éditer vos options de compilation. Dans le mode debug vous observez que les options de compilation permettent d'utiliser `gcov`. En particulier, si on exécute

```
make test
make ExperimentalCoverage
```

on peut consulter dans le sous-répertoire Testing/CoverageInfo les informations générées par `gcov`.

Exercice 14 – En étoffant un jeu de tests (i.e. en invoquant *`foo`* avec un jeu de valeurs représentatives), tentez de repérer s'il existe une partie de votre code qui est probablement mort, c'est-à-dire qui n'est jamais exécutée.

Exercice 15 – Modifiez le `CMakeLists.txt` de votre projet pour y inclure l'utilisation de gcov.

Exercice 16 – Repérez les parties de `game.c`, de `game_io.c` et de votre solveur qui ne sont pas couvertes par vos tests.

3 Analyse statique de code

`cppcheck` est un outil d'analyse statique de code. Il permet de repèrérer un certain nombre d'erreurs classiques dans du code. Son utilisation est relativement simple :

Exercice 17 – Placez-vous dans votre répertoire source et exécutez les commandes suivantes

```
$ cppcheck --enable=all --xml-version=2 -I ../include *.c  2> report.xml
$ cppcheck-htmlreport --source-dir=. --report-dir=report --file=report.xml
```

Exercice 18 – Ouvrez dans votre navigateur le fichier `./report/index.html`. Corrigez les erreurs les plus critiques.

4 Optimisation et profilage

L'optimisation est une tâche complexe, coûteuse en temps, et qui introduit souvent des bugs et une complexité supplémentaire dans le code. On dit souvent qu'un programme passe 80% de son temps d'exécution dans 20% du code. Les outils de profilage permettent d'identifier ces parties du code dont le temps d'exécution est prépondérant, pour guider le développeur à effectuer les optimisations seulement là où cela s'avère strictement nécessaire. L'outil *`callgrind`* est le profileur qui vous permet d'identifier ces parties les plus gourmandes.

Exercice 19 – Lancez la commande suivante

```
valgrind --tool=callgrind ./undead_solve NB_SOL ...
```

Exercice 20 – Remarquez qu'un fichier `callgrind.out.<pid>` a été généré. Ouvrez ce fichier avec le programme `kcachegrind`.

Exercice 21 – Identifiez les parties de votre code qui mériteraient d'être optimisées en priorité.