Rapport Projet Informatique
Lot C Tâche 2

Adib Habbou

13 mai 2022

Table des matières

1. Modification Makefile

L'option `-pg` de gcc sert à générer du code supplémentaire pour écrire des informations de profilage adaptées au programme d'analyse `gprof`, j'ai ajouté l'option dans le Makefile au niveau de la variable `CC`.

2. Logiciel gprof

Le logiciel `gprof` est un logiciel de *GNU Binary Utilities* qui permet de faire du profilage de code pour contrôler, lors de l'exécution du programme, la liste des fonctions appelées et le temps passé dans chacune d'elles.

3. Options gprof

Les options utilisées sont les deux suivantes :

- `-b` indique à gprof de ne pas afficher les commentaires dans l'affichage généré
- `-p` force gprof à afficher le *Flat Profile*

4. Utilisation

- Il faut compiler et exécuter le programme d'abord :

```
~$ make clean
~$ make
~$ ./bin/oriieflamme
```

- Ensuite on lance `gprof` et on stocke le résultat dans un fichier texte `profile.txt` :

```
~$ gprof -b -p /bin/oriieflamme > profile.txt
```

- Enfin si l'on souhaite lire le résultat directement sur le terminal :

```
~$ cat profile.txt
```

5. Flat Profile

Le *Flat Profile* indique le temps total passé par le programme à exécuter chaque fonction, les fonctions sans temps apparent passé dessus, et sans appels apparents, ne sont pas mentionnées. Les fonctions sont triées d'abord par temps d'exécution décroissant, puis par nombre décroissant d'appels, puis par ordre alphabétique de nom. Juste avant les noms de colonne, une déclaration indique combien de temps chaque échantillon a pris dans notre cas 0.01 secondes.

Les colonnes du *Flat Profile* sont les suivantes :

- la colonne `% time` indique le pourcentage du temps d'exécution total que votre programme a passé dans cette fonction. La somme de toutes les valeurs de la colonne devraient être égale à 100 %. Cette colonne permet de rapidement identifier les `hot-spots`
- la colonne `cumulative seconds` indique le total cumulé de secondes que l'ordinateur a passées à exécuter ces fonctions plus le temps passé dans toutes les fonctions au-dessus de celle-ci dans ce tableau
- la colonne `self seconds` indique le nombre de secondes que l'ordinateur a passées à exécuter cette fonction seule
- la colonne `calls` indique le nombre total de fois où la fonction a été appelée. Si la fonction n'a jamais été appelée ou si le nombre de fois qu'elle a été appelée ne peut pas être déterminé parce que la fonction n'a pas été compilée avec le profilage activé alors le champ est vide
- la colonne `self ms/calls` représente le nombre moyen de millisecondes passées dans cette fonction par appel, si cette fonction est profilée. Sinon, ce champ est vide
- la colonne `total ms/calls` représente le nombre moyen de millisecondes passées dans cette fonction et ses descendants par appel, si cette fonction est profilée. Sinon, ce champ est vide pour cette fonction
- la colonne `name` indique le nom de la fonction

6. Résultats

Le *Flat Profile* de notre programme pour une game donnée est présent dans le fichier texte `profile.txt`. Bien évidemment ce fichier texte est présent à titre indicatif, si on relance une nouvelle partie (et qu'on arrive à pas tomber sur un `segmentation fault` idem que pas tomber sur la carte `Eric Lejeune`) alors le *Flat Profile* qu'on obtiendra ne sera pas forcément exactement le même quoique sensiblement similaire.

On peut tirer du *Flat Profile* obtenu concernant les fonctions les plus appelées les données suivantes (cf `profile.txt` pour plus de détails) :

- `get_plateau_carte` : 83 260 appels
- `get_case_etat` : 63 244 appels
- `get_case_carte` : 20 016 appels

On remarque très clairement que les fonctions les plus appelés sont les trois getters mentionnées ci-dessus. On retrouve principalement parmi les fonctions les plus appelées les getters et les setters ainsi que les fonctions qui agissent sur la structure pile.

7. Interprétation

On se rend donc compte que ce qui prend le plus de temps lors de l'exécution de notre programme sont les getters qui permettent de récupérer la carte dans un plateau, l'état d'une case ou encore la carte posée sur une case.

Les optimiser en améliorant la manière dont ils sont codés pour les getters complexes ou encore en diminuant le nombre de fois dont on les appelle pourrait considérablement baisser le temps d'exécution de notre programme et par conséquent améliorer son efficacité.

On peut donc imaginer qu'une bonne manière d'optimiser notre code est d'essayer d'appeller beaucoup moins de fois ces 3 getters en évitant de faire des boucles où on les appellent plusieurs fois par itération et concevoir plutôt de manière alternative notre code.

Une autre optimisation mais beaucoup plus minime serait de modifier notre structure (en l'occurrence dans notre cas les piles) pour essayer de stocker nos données (par exemple les cartes de la main) de manière à ce qu'on accède à l'information avec un coût plus faible et qu'on puisse également ajouter et supprimer des informations (entre autres des cartes) avec un coût plus faible également.

8. Difficulté

La première difficulté rencontrée lors de cette tâche est la compréhension de l'outil `gprof` qui a nécessité la lecture de documentation et de tutoriels afin d'appréhender son utilité et la manière de l'utiliser en l'adaptant au cas spécifique de notre projet.

La deuxième difficulté a été d'identifier l'élément à étudier. J'ai choisi le *Flat Profile* parce qu'il synthétise de manière assez claire la "complexité" du programme en question. De plus, son analyse est beaucoup plus claire et limpide que celle du *Call Graph* par exemple.

9. Remerciements

Merci d'avoir encadré ce projet durant ce deuxième semestre et surtout pour les nombreux conseils qui seront sûrement bien utile dans le futur au-delà de ce projet.

FIN