

Rapport TD3 - Table de transposition

Par Stanislaw PLUSZKIEWICZ et Ellmo RIVETTI

Objectif du TD:

L'objectif du TD est de modifier le code de TSCP afin de l'optimiser et de l'améliorer. Nous allons utiliser les techniques de hachage afin d'y parvenir.

Réalisation du TD:

Déroulement:

Nous avons commencé le développement du TD lors de la séance prévue. Durant cette séance, nous avons commencé par travailler sur la seconde partie du TD qui consiste à implémenter une table de transposition.

Lors de notre travail personnel, nous avons décidé de commencer par la partie 1 du TD.

Voici le temps passer sur la réalisation de ce TD (à multiplier par deux pour avoir le temps total du binôme):

- Partie 1:
 - 6h
- Partie 2:
 - 5h perso + TD
- Rapport:
 - 2h

Chaque optimisation a été développée sur une branche spécifique de notre [GitHub](#).

Branches:

- [Partie 1 - Optimisation des fonctions de hashing](#)
- [Partie 2 - Implémentation d'une table de transposition](#)

Partie 1: Optimisation des fonctions de hashing

Objectif

Lors de cette partie du TD, nous devons optimiser l'utilisation de la méthode `setHash()` afin que celle-ci ne soit plus appelée par la méthode `makemove()`. Nous calculons un hash pour sauvegarder le score d'un état de la partie. Ainsi lors de mouvements répétitifs, TSCP ne va pas recalculer les scores mais se servir de scores existants, calculés lors de mouvements précédents. Dans TSCP, la méthode `setHash()` va recalculer (à chaque fois que celle-ci est appelée) l'intégralité du hash en itérant sur chaque case de l'échiquier.

Afin de remédier à ce problème, nous avons dû modifier le hash directement dans `makemove()` grâce à l'opérateur XOR.

Réalisation

Premièrement, nous avons supprimé l'appel à `setHash()` dans `makemove()`.

Ensuite, nous avons implémenté le code nécessaire à la mise à jour du hash en direct.

Nous avons donc récupéré le contenu du cour afin d'implémenter les lignes de codes nécessaires:

```
hash ^= hash_piece[color[(int)m.to]][piece[(int)m.to]][(int)m.from];  
hash ^= hash_piece[color[(int)m.to]][piece[(int)m.to]][(int)m.to];
```

Nous devons indexer les tableaux `color` et `piece` via la variable `m.to` car les valeurs de ces tableaux à l'index `m.from` ont été "vidées" au préalable. En effet, lors du déplacement d'une pièce, afin de mettre à jour l'échiquier, les valeurs des tableaux à l'index de `m.from()` vont désormais se trouver à l'index `m.to()` (c'est logique).

La première ligne de code nous permet de supprimer la pièce sur la case d'origine. La valeur du hash va donc se changer à 0 pour cette case.

La seconde ligne de code va alors ajouter au hash l'information sur la position de la pièce à cette nouvelle case.

Une fois que nous avons écrit ces lignes, nous avons donc implémenté une méthode de test appelée `set_hash_test()` afin de vérifier si notre nouvelle hash est la même que celle calculée par la méthode `set_hash()`.

Cette méthode va donc comparer la variable `hash` avec une variable calculée via `set_hash()` et ainsi, grâce à des `ASSERT()`, nous pouvons voir s'il y a une différence entre nos hashes.

Remarque:

Nous n'avons pas pu utiliser la méthode `set_hash()` existante dans notre méthode de test car celle-ci modifie "hash" qui est une variable globale. De plus, nous n'avons pas pu modifier le contenu de `set_hash()` car cette méthode est utilisée à d'autres endroits dans le programme (ex: au début lors de l'initialisation).

Grâce à cette méthode, nous avons pu voir que le code que nous avons ajouté n'était pas suffisant pour que le programme fonctionne. Nous avons donc dû ajouter du code pour tous les cas spéciaux de `makemove()` (prise en passant, promotion, ...). Nous avons aussi récupéré les cas spéciaux qui étaient présent dans la méthode `set_hash()`:

```
if (side == DARK)
    hash ^= hash_side;
if (ep != -1)
    hash ^= hash_ep[ep];
```

Nous avons donc ajoutés ceux-ci dans le `makemove()`

Malgré ces développements, les hash n'étaient toujours pas les mêmes. Nous avons donc décidé d'améliorer nos méthodes de test. Nous avons créé la méthode `debug_hash()` qui nous a permis de visualiser nos hash et ainsi d'effectuer divers tests (XOR avec certaines variables, etc..) afin de voir les résultats que nous obtenions.

Grâce à cette méthode, nous avons donc pu visionner les valeurs des hash et nous avons pu, à la main (via ce site web: <http://xor.pw/>), effectuer des XOR sur ces hash et donc de voir le résultat obtenu. Nous avons donc pu voir qu'il nous manquait une étape lors du "en passant" et nous l'avons donc rajoutée.

Remarque:

Nous avons remarqué qu'il n'était pas nécessaire de modifier la méthode `takeback()` car celle-ci ne faisait pas appel au `set_hash()`. De plus, au début de la méthode, le hash précédent est récupéré depuis l'historique:

```
hash = hist_dat[hply].hash;
```

Remarque:

Lors de nos développements de ce week-end, nous nous sommes rendu compte que le code de TSCP ne fonctionnait plus à un certain horaire (environ 20h30). Nous avons donc cru, au début, que nous avons cassé notre programme en écrivant un code erroné. C'est alors que plusieurs autres groupes ont rapporté le même problème au même moment. Nous nous sommes donc rendu compte qu'en changeant l'heure de nos PCs pour les mettre dans l'après-midi, tout fonctionnait. Nous supposons que ce problème est dû au fonction de création de nombres aléatoires qui se basent sur l'heure système pour créer les valeurs. En conclusion, ce problème nous a fait perdre pas mal de temps car nous ne comprenions pas pourquoi plus rien ne fonctionnait. (:

Résultats

Résultat initial de TSCP

Afin de tester notre code, nous avons utilisé la méthode `bench()` de TSCP. Lors de ces benches, nous avons pris une profondeur de 7.

Tous les tests ont été effectués sur le même PC afin qu'il n'y ai pas de changement dû aux différentes puissances de calcul des appareils. De plus, chaque bench est effectué en mode *Release* et en x64.

Voici les résultats obtenus via le code original de TSCP:

```
tscp> bench

8 . r b . . r k .
7 p . . . . p p p
6 . p . q p . n .
5 . . . n . . N .
4 . . p P . . . .
3 . . P . . . P .
2 P P Q . . P B P
1 R . B . R . K .

  a b c d e f g h

ply      nodes  score  pv
1         130    20    c1e3
2        3441     5   g5e4 d6c7
3        8911    30   g5e4 d6c7 c1e3
4       141367   10   g5e4 d6c7 c1e3 c8d7
5       550778   26   c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598   16   g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562   27   g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 20608 ms
ply      nodes  score  pv
1         130    20    c1e3
2        3441     5   g5e4 d6c7
3        8911    30   g5e4 d6c7 c1e3
4       141367   10   g5e4 d6c7 c1e3 c8d7
5       550778   26   c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598   16   g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562   27   g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 20952 ms
ply      nodes  score  pv
1         130    20    c1e3
2        3441     5   g5e4 d6c7
3        8911    30   g5e4 d6c7 c1e3
4       141367   10   g5e4 d6c7 c1e3 c8d7
5       550778   26   c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598   16   g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562   27   g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 21207 ms

Nodes: 28757562
Best time: 20608 ms
Nodes per second: 1395456 (Score: 5.739)
Opening book missing.
```

Résultat de notre programme

Grâce à cette optimisation, nous avons pu obtenir les résultats suivants:

Cela nous a permis de gagner **2.405 secondes** sur les valeurs originales de TSCP. De 20608 ms à 18203 ms.

TSCP parcourt 1395456 nodes par secondes et cette optimisation , 1579825 nodes par secondes.

Nous obtenons donc un score de **6.497** soit environ **+0.758** par rapport à TSCP.

```
tscp> bench

8 . r b . . r k .
7 p . . . . p p p
6 . p . q p . n .
5 . . . n . . N .
4 . . p P . . . .
3 . . P . . . P .
2 P P Q . . P B P
1 R . B . R . K .

a b c d e f g h

ply      nodes    score  pv
1         130       20  c1e3
2        3441        5  g5e4 d6c7
3        8911       30  g5e4 d6c7 c1e3
4       141367      10  g5e4 d6c7 c1e3 c8d7
5       550778      26  c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598      16  g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562      27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 18245 ms
ply      nodes    score  pv
1         130       20  c1e3
2        3441        5  g5e4 d6c7
3        8911       30  g5e4 d6c7 c1e3
4       141367      10  g5e4 d6c7 c1e3 c8d7
5       550778      26  c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598      16  g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562      27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 18203 ms
ply      nodes    score  pv
1         130       20  c1e3
2        3441        5  g5e4 d6c7
3        8911       30  g5e4 d6c7 c1e3
4       141367      10  g5e4 d6c7 c1e3 c8d7
5       550778      26  c2a4 d6c7 g2d5 e6d5 c1e3
6      5919598      16  g2d5 d6d5 c1f4 b8a8 f4e5 c8d7
7     28757562      27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 18274 ms

Nodes: 28757562
Best time: 18203 ms
Nodes per second: 1579825 (Score: 6.497)
Opening book missing.
```

Partie 2 - Implémentation d'une table de transposition

Objectif

Cette partie vise à implémenter une table de transposition. Celle-ci vise à reconnaître des positions d'échiquier et passer l'étape d'évaluation. Nous stockons les informations d'évaluation pour tous les états de partie rencontrés. Ainsi les évaluations sont directement en $O(1)$ dans la méthode quiesce qui applique l'algorithme alpha-beta.

Réalisation

Nous avons commencé à développer cette partie lors de la session de livecode du TD. A la fin de celle-ci, le code ne compilant et ne fonctionnant pas, nous avons passé plusieurs heures afin de résoudre les problèmes existants. Néanmoins, celles-ci ont été dépensées en vain car même après un visionnage des sessions de TD de blackboard des deux groupes (2h30 par session (:) , il nous était impossible de corriger les/l'erreur présente dans notre base de code.

Conclusion

Nous pouvons donc voir que grâce aux techniques de hachage, nous avons pu améliorer la vitesse d'exécution de notre programme mais aussi ses performances de recherche.

Lors de la première optimisation, nous avons remanié du code de TSCP afin de le rendre plus rapide (un peu à la manière du TD précédent). Cela permet donc au programme de parcourir de plus en plus de nœuds et donc d'obtenir de meilleurs résultats.

La seconde optimisation aurait permis (si elle avait été développée) d'améliorer les méthodes de recherches de TSCP. Nous allons donc stocker les évaluations des configurations d'échiquiers rencontrées lors de l'exécution. Ainsi on ne recalcule pas une configuration déjà rencontrée mais on se contente de ressortir les évaluations existantes.