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:
  - 3h
- Partie 2:
  - xh
- Rapport:
  - 2h

Il nous a fallu environ 1h30 afin de réaliser l'optimisation 2 plus 15 minutes à la fin lors du merge des optimisations afin de résoudre un bug que nous n'avions pas vu.

Nous avons mis beaucoup plus de temps pour réaliser l'optimisation 1. Soit:

- 45 minutes lors du premier TD
- 2h lors du second TD (ou nous avons principalement fait la méthode sync_board())
- 6h en travail personnel chacun

Chaque optimisation a été développée sur une branche spécifique de notre GitHub. Une fois que les deux furent finies, nous avons merge ces branches sur *main* (la branche principale du projet Git).

Branches:

- Optimisation 1
- Optimisation 2
- Merge des deux optimisations

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 avions 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 hashs.

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 avions 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];
```

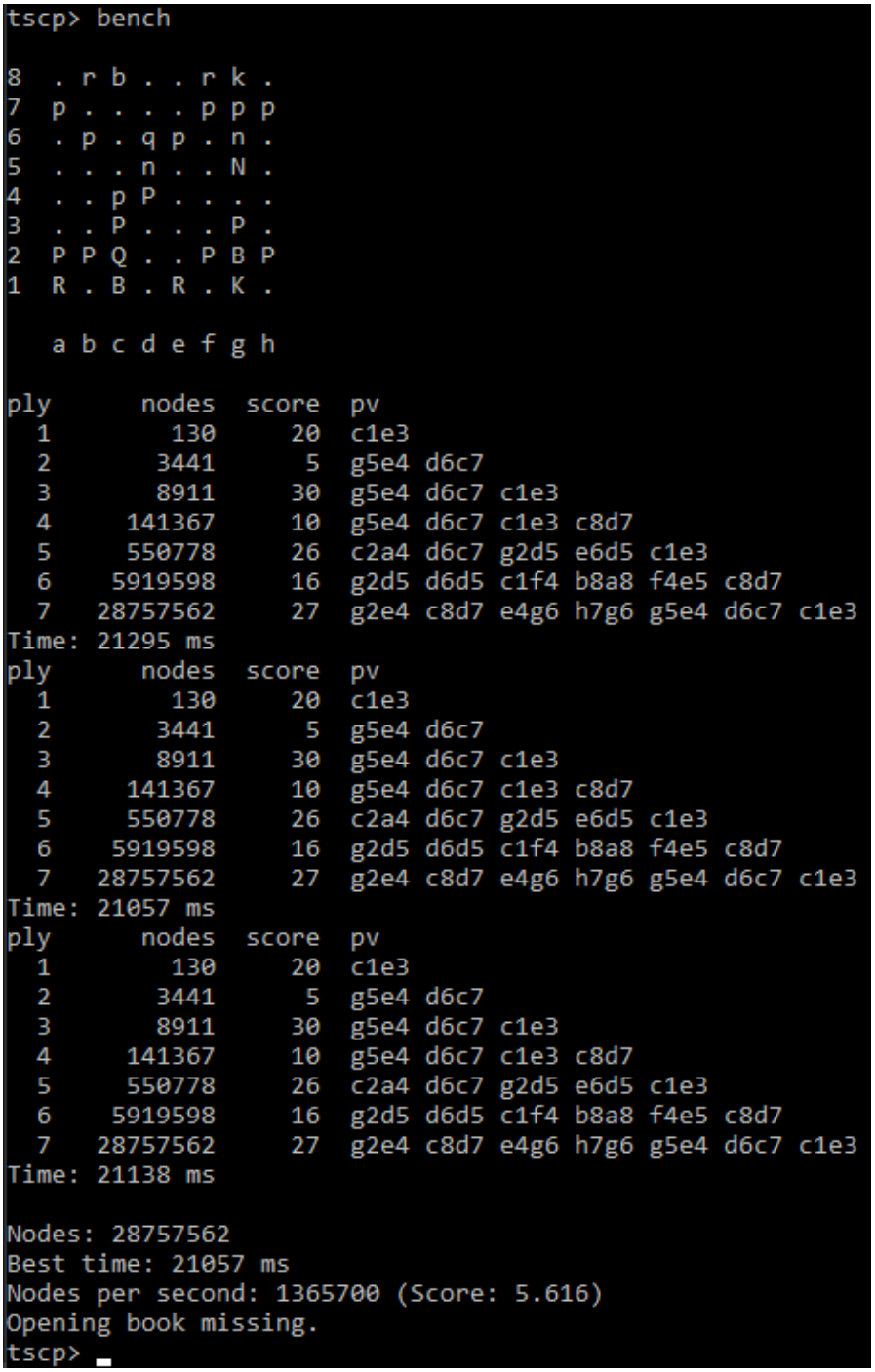
Nous avons donc ajoutés ceux-ci dans le *makemove()*

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().

Résultats

Résultat initial de TSCP

Afin de tester notre code, nous avons utilisé la méthode *bench()* de TSCP. Lors de ces benchs, 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:



Résultat de notre programme

Grâce à cette optimisation, nous avons pu obtenir les résultats suivants:
Cela nous a permis de gagner **5.292 secondes** sur les valeurs originales de TSCP. De 21057 ms à 15765 ms.
TSCP parcours 1365700 nodes par secondes et cette optimisation , 2094784 nodes par secondes.
Nous obtenons donc un score de **8.615** soit environ **+2.999** 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        131     20  c1e3
  2       4717      5  g5e4 d6c7
  3      10487     30  g5e4 d6c7 c1e3
  4     184442     10  g5e4 d6c7 c1e3 c8b7
  5     647714     26  c2a4 d6c7 g2d5 e6d5 c1e3
  6    8400851     16  g2d5 d6d5 c1f4 b8a8 f4e5 c8b7
  7   33024284     27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 16109 ms
ply      nodes  score  pv
  1        131     20  c1e3
  2       4717      5  g5e4 d6c7
  3      10487     30  g5e4 d6c7 c1e3
  4     184442     10  g5e4 d6c7 c1e3 c8b7
  5     647714     26  c2a4 d6c7 g2d5 e6d5 c1e3
  6    8400851     16  g2d5 d6d5 c1f4 b8a8 f4e5 c8b7
  7   33024284     27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 15797 ms
ply      nodes  score  pv
  1        131     20  c1e3
  2       4717      5  g5e4 d6c7
  3      10487     30  g5e4 d6c7 c1e3
  4     184442     10  g5e4 d6c7 c1e3 c8b7
  5     647714     26  c2a4 d6c7 g2d5 e6d5 c1e3
  6    8400851     16  g2d5 d6d5 c1f4 b8a8 f4e5 c8b7
  7   33024284     27  g2e4 c8d7 e4g6 h7g6 g5e4 d6c7 c1e3
Time: 15765 ms

Nodes: 33024284
Best time: 15765 ms
Nodes per second: 2094784 (Score: 8.615)
Opening book missing.
tscp>
```

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 remanier 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 a avant tout visé à 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.