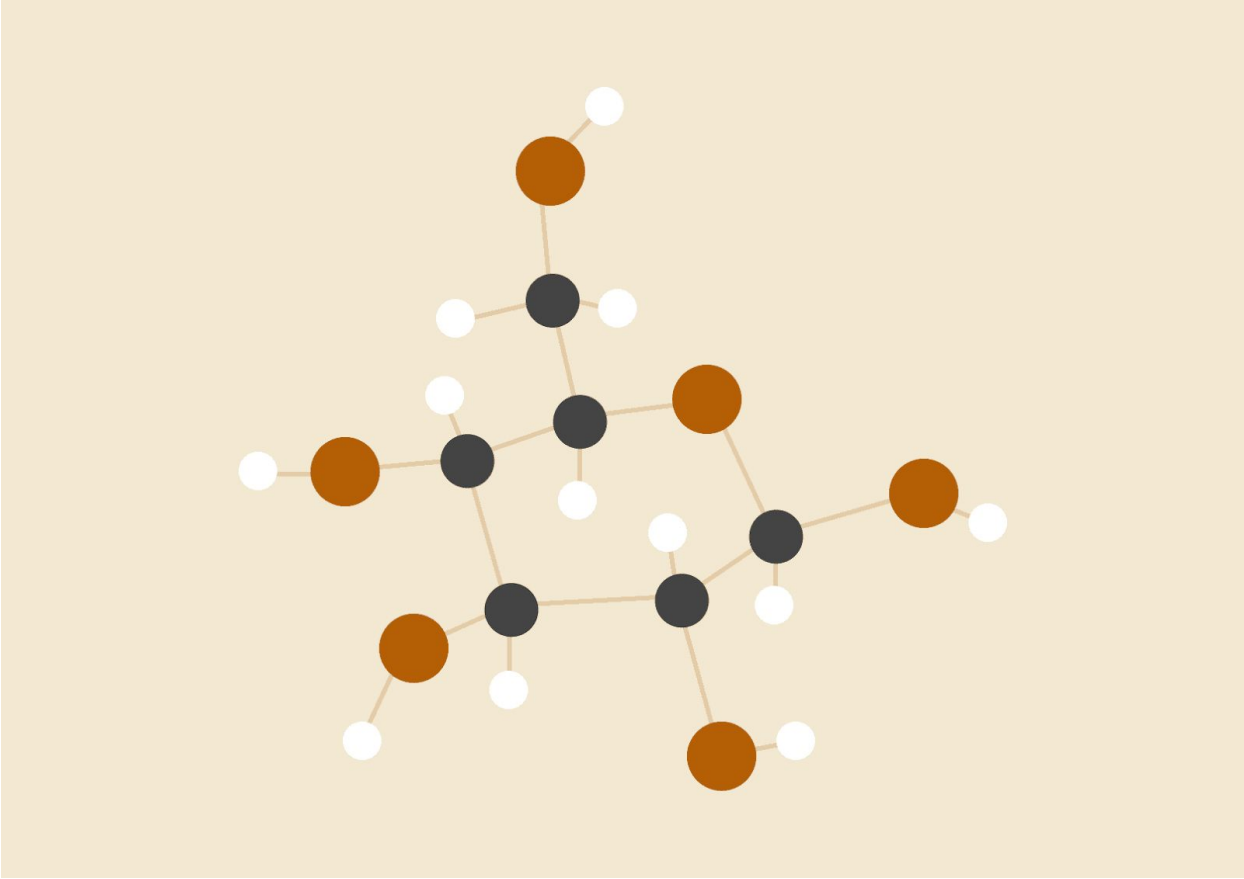
Rapport d'optimisation de vitesse

I.A. Architecture d'un programme d'échecs



Guillaume DEMEYERE - Anas TULIMAT

07/02/2021

E5FI– Groupe 2

INTRODUCTION

L’objectif de ce TD est de modifier le code du programme TSCP afin d’améliorer ses performances.
Dans ce TD, nous implémentons deux méthodes d'amélioration des performances du programme. L'idée principale est de réduire la taille des boucles « for » permettant d’accéder aux pièces de l’échiquier et à leurs mouvements possibles.

DÉROULEMENT

Nous avons commencé ce TD par implémenter la première méthode d'optimisation qui consiste à maintenir un tableau de 32 entiers qui va stocker les positions des pièces de l’échiquier. Donc au lieu de scanner 64 cases de l'échiquier , on scannera la position des pièces (16 positions par couleur).

La difficulté consiste à maintenir à jour, en parallèle, chacune des deux représentations de l’échiquier : celle utilisée par le programme d’origine, et celle permettant un accès optimisé.

Nous avons posé problème, avant tout, l’équivalence entre la représentation originelle de l’échiquier (tableaux piece[] et color[]), et notre représentation optimisée (pospiece[] et board[]).

En particulier, la mise à jour de pospiece et de board est assez complexe dans les fonctions makemove() et takeback(), notamment quand il s’agit de restaurer des pièces qui ont été capturées à un coup précédent, que ce soit par une prise directe ou par une prise en passant. Le roque aussi est un cas à envisager en détail.

Pour débugger les nombreuses erreurs de notre code d’origine, la fonction checkBoard() s’est révélée vitale, et il a fallu la rendre aussi exhaustive que possible. Vérifier la cohérence entre les deux représentations n’était pas suffisant: il a fallu que nous ajoutions du code pour vérifier la cohérence entre board et pospiece eux-mêmes.

La deuxième méthode d'optimisation nous a paru plus simple en comparaison.

L'explication donnée dans l'énoncé du TD était très claire, en plus vous nous avez montré durant le cours comment implémenter la méthode initAttackTables() afin d'utiliser des tables régénérées.

PROBLÈME RENCONTRÉS

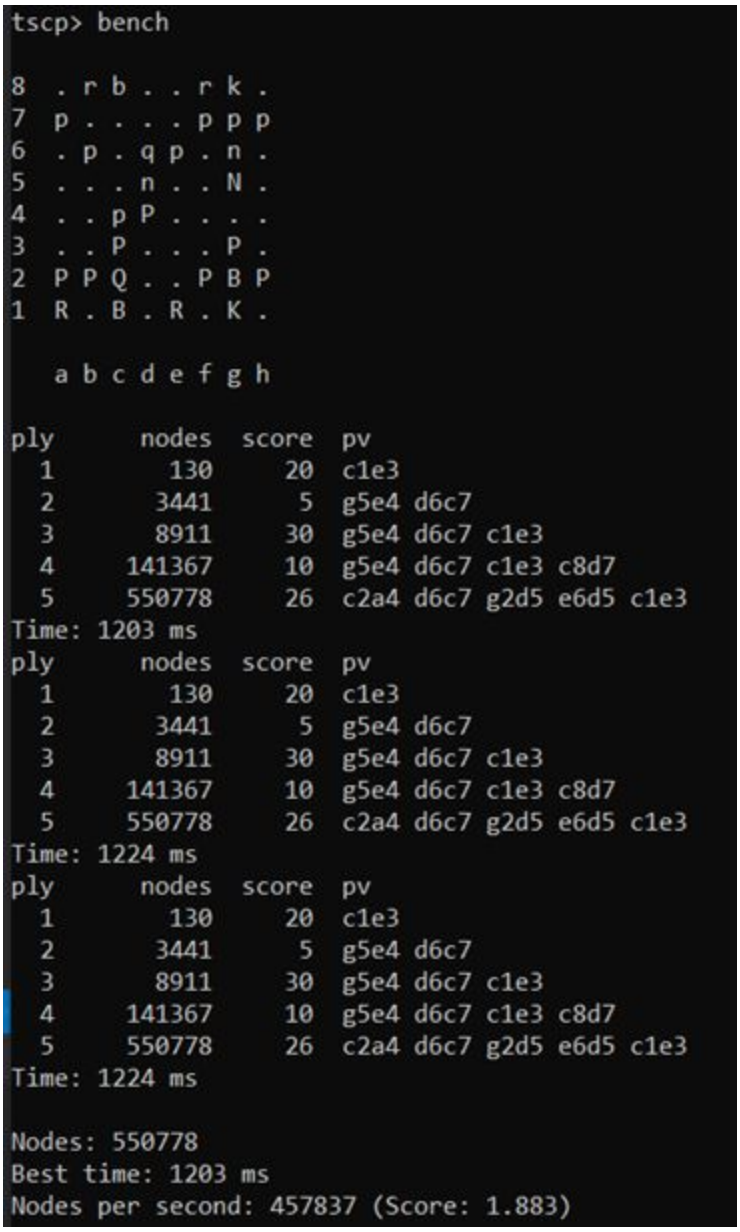
Nous avons rencontré plusieurs problèmes lors de ce TD.
Notre difficulté principale a été de bien comprendre le fonctionnement du programme et les structures de données utilisées, ce qui nous a pris beaucoup de temps.
Nous avons également ressenti un peu de difficulté à manipuler le langage C au sein de ce programme.

La deuxième difficulté rencontrée a été l'implémentation des méthodes makemove() et takeback() car il y avait des cas spécifiques à gérer comme le roque et la prise en passant qui n'était pas simple à gérer.

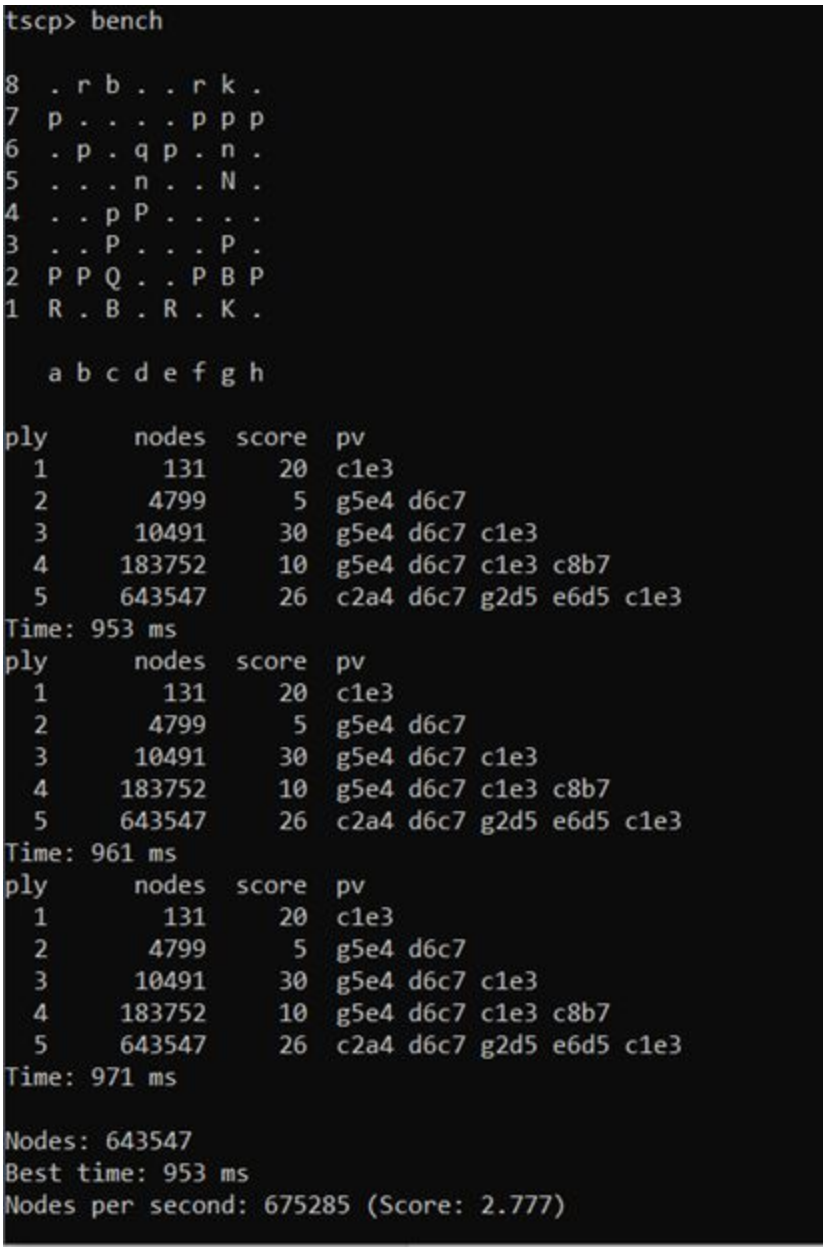
Le debug en langage C n'a pas été facile non plus, nous avons donc utilisé le système de debug intégré du Visual Studio Code, et nous avons implémenté également la méthode checkBoard() afin de vérifier l'état de l'échiquier.

AMÉLIORATION DES PERFORMANCES

Voici les performances (bench) du programme TSCP avant le TP, évalué sur notre ordinateur, en mode debug (de toute évidence, en mode release, les performances sont bien meilleures dans les deux cas: notre code optimisé nous fait alors atteindre un score de 8.81 dans bench, contre 4.8 à l'origine)



À la fin du TP, nous relançons la même évaluation sur notre code optimisé:



L’amélioration est notable, mais modeste :

- pour le meilleur temps, nous passons de 1.2 secondes à 0.95.
- Le nombre de nœuds exploré par seconde augmente d’environ 200 000.
- Le score global passe de 1.9 à 2.8.

RÉSULTATS

Dans Arena, nous testons notre programme amélioré, en le faisant affronter le programme TSCP originel.

Voici le résultat d'un tournoi de 10 parties, avec une demi-seconde de réflexion à chaque coup ("Time per move = 0.5"), pour chaque opposant:

----------Chess_A0_Fin_TP2----------

Chess_A0_Fin_TP2 - Original TSCP : 5.0/10 5-5-0 (0101010101) 50% ±0

----------Original TSCP----------

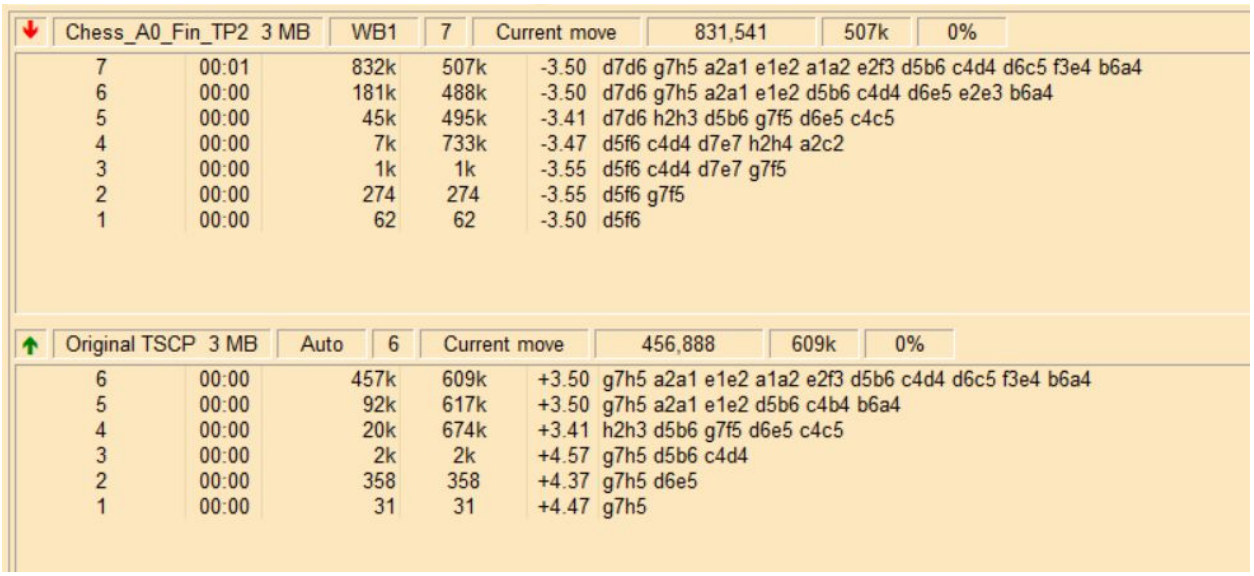
Original TSCP - Chess_A0_Fin_TP2 : 5.0/10 5-5-0 (1010101010) 50% ±0

L'amélioration ne saute pas aux yeux ! Toutefois, cela permet de vérifier déjà que notre programme continue de jouer correctement. Peut-être l'amélioration apportée, qui doit pouvoir permettre une plus grande exploration de l'arbre de recherche en un temps limité, n'est-elle significative qu'avec un temps de réflexion plus long ?

Nous relançons le test avec 2 secondes de réflexion par coup.

Ci-dessus: début du second tournoi de 10 parties

Cette seconde itération n'est pas plus concluante. Nous avons parfois l'impression que notre programme explore plus de nœuds (cf. capture d'écran ci-dessous), mais cette impression ne semble qu'éphémère, voire illusoire, et ne se traduit pas par une supériorité claire.



Voici le résultat final, après une demi-heure d'exécutions:

----------Chess_A0_Fin_TP2----------

Chess_A0_Fin_TP2 - Original TSCP : 4.5/10 4-5-1 (1010=01010) 45% -35

----------Original TSCP----------

Original TSCP - Chess_A0_Fin_TP2 : 5.5/10 5-4-1 (0101=10101) 55% +35

CONCLUSION

En demi-teinte. Après des dizaines d'heures de débug et de tests intensifs, notre programme fonctionne, et semble réellement plus performant. Bench() montre une amélioration des performances notable, de l'ordre de +50%.

Pour autant, nos tests Arena n'ont pas permis d'identifier ce gain. Nos tests ont-ils été insuffisants ? Maladroits ? Notre compilation a-t-elle posé problème ? Nous manquons de temps pour répondre à toutes ces questions.