

Introduction à l'I.A.

Architecture d'un programme

D'échecs de compétition

Optimisation de vitesse

Guideline

ESIEE

Groupe E3FI

V2

Jan 2020

Pascal Tang

Objectif

Le travail consiste à modifier le code du programme TSCP afin d'améliorer ses performances.

Dans ce document, je vais donner 2 pistes principales d'amélioration des performances du programme TSCP étudié en cours. Si vous en trouvez d'autres, vous pouvez bien évidemment les proposer. Vous êtes libres d'implémenter comme vous le souhaitez tant que la fonction d'évaluation ou l'algorithme de recherche reste identique au programme original.

Le travail initial demandé est de rajouter des structures pour améliorer la vitesse (Optimisation 1).

L'Optimisation 2 est prévue pour ceux qui ont fini l'Optimisation 1 ou sont bloqués sur l'Optimisation 1.

Optimisation 1 : Rajout de structures supplémentaires

Les structures de données qui ont été choisies pour l'écriture du programme TSCP imposent de « scanner » 64 cases afin de détecter les pièces présentes sur l'échiquier, ce qui est sous-optimal.

L'amélioration consiste à maintenir un tableau de 32 entiers (ou 1 tableau de 33 entiers ou 2 tableaux de 16 entiers...Selon votre implémentation) qui va stocker les coordonnées des pièces de l'échiquier. Au lieu donc de scanner tout l'échiquier (64 cases), on scanner la position des pièces (16 positions par couleur).

Le but de l'exercice consiste à modifier/rajouter le code aux endroits où la structure de l'échiquier est modifiée : `init_board()`, `makemove()`, `takeback()`, `bench()` si on utilise `bench()`...

Exemple d'implémentation

D'autres implémentations sont possibles :

```
#define PIECE_DEAD (-1)
int pospiece [33]; // le tableau des positions des pièces de l'échiquier
int board[64];     // L'échiquier contenant les index de pospiece
```

Au début de la partie le tableau board[] pourrait être initialisée ainsi :

- Toutes les pièces blanches sont comprises entre 1 et 16
- Toutes les pièces noires sont comprises entre 17 et 32.
- On s'arrange pour que les coordonnées du roi blanc soient stockées dans pospiece[1]
- On s'arrange pour que les coordonnées du roi noir soient stockées dans pospiece[17]

18	19	20	21	17	22	23	24
25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
2	3	4	5	6	7	8	9
10	11	12	13	1	14	15	16

Ce qui correspond à :

pospiece[17]=A8 ;

pospiece[18]=B8 ;

pospiece[19]=C8 ;

...

pospiece[9]=A0 ;

On remarque que si board[i] est non nul, on a toujours : pospiece[board[i]]== i

A chaque mouvement de pièces, il faudra remettre à jour les tableaux pospsiece[] et board[].

Si on déplace le pion de e2 en e4, on aura :

Pour le tableau board[] :

18	19	20	21	17	22	23	24
25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	6	0	0	0
0	0	0	0	0	0	0	0
2	3	4	5	0	7	8	9
10	11	12	13	1	14	15	16

Pour le tableau pospsiece[] :

...

Pospsiece[6]=E4 ; // E4 est la nouvelle coordonnée de la pièce 6

Capture de pièces

Lorsqu'une pièce est capturée, elle disparaît de l'échiquier et ses coordonnées sont mises à PIECE_DEAD pour bien marquer que les coordonnées ne sont plus utilisables.

Pospsiece[i]=PIECE_DEAD

Pour gagner en vitesse : il faudra modifier les fonctions suivantes :

Board.cpp :

- attack()
- in_check()
- gen()
- gencaps()

Eval.cpp :

- eval()

Etape 1 :

Data.h/data.cpp :

Rajouter la déclaration de vos structures de données.

Etape 2 :

Modifier la fonction init_board(), afin d'initialiser vos structures de données conformément à la position initiale de l'échiquier. Vous pouvez utiliser des memsets pour éviter les boucles.

Pour cela créer une fonction void syncBoard(), qui va mettre les tableaux pospiece[] et board[] à jour par rapport à la représentation de l'échiquier de TSCP.

```
void syncBoard()
{
    memset(board, 0, sizeof(board));
    for (int i = 0; i <= 32; i++)
        pospiece[i] = PIECE_DEAD;

    for (int i = 0; i < 64; ++i) {
        if (color[i] != EMPTY) // On trouve une pièce sur l'échiquier
        {
            // MAJ des tableaux pospiece[] et board[]
            // A terminer de coder
        }
    }
}
```

Etape 5 :

Modifier et optimiser la fonction fonction attack() afin d'accélérer le programme. La modification consiste à réduire la boucle externe de 64 à 16 itérations.

Il faut absolument commencer par la fonction `attack()` car la modification/optimisation de cette fonction n'impactera pas le nombre de nœuds parcourus, l'évaluation ou les variantes (=lignes de réflexion). Ce qui va être utile pour savoir si on a un bug ou non. Le bug ici serait principalement un oubli d'update des tableaux de positions.

Il faut donc comparer avec les sorties du programme avec ceux du programme original : si vous n'obtenez pas les mêmes sorties, ce n'est pas la peine d'aller plus loin : le programme est buggé. Généralement, vous avez oublié de synchroniser les tables (rajouter ou enlever une pièce) quelque part...Mettez en place des tests (ex. asserts...) pour fixer le problème.

Pour obtenir les gains de performances, vous pouvez utiliser la commande « bench » disponible sur la console de TSCP. Il affichera la vitesse en nœuds/sec et un score (Ici 6).

```
Tom Kerrigan's Simple Chess Program (TSCP)
version 1.81c, 2/3/19
Copyright 2019 Tom Kerrigan

"help" displays a list of commands.

Opening book missing.
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: 19405 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: 19442 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: 19500 ms

Nodes: 28757562
Best time: 19405 ms
Nodes per second: 1481966 (Score: 6.094)
Opening book missing.
tscp>
```

Etape 6 :

Finir l'optimisation en modifiant les fonctions :

- `gen()`
- `gencaps()`
- `eval()`

La modification des fonctions ci-dessus est triviale si vous avez réussi l'étape 5.

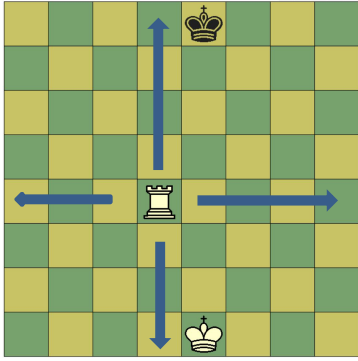
Attention : la modification de ces fonctions va modifier l'enveloppe de recherche et le nombre de nœuds parcourus est susceptible d'être différent du programme original.

Optimisation 2 : Utiliser des tables prégénérées

Les tables prégénérées sont utilisées pour éviter les boucles inutiles.

La fonction `attack()` est une fonction très utilisée. Elle est « simple » mais n'est pas très rapide et handicape les performances du programme.

Exemple :

	<p>Pour savoir si la tour blanche attaque le roi noir, le programme <code>tscp</code> va scanner les lignes et les rangées jusqu'à rencontrer les bords ou une pièce.</p> <p>L'idée est d'éviter de scanner si l'on sait à priori que la case à attaquer (ici le roi noir) ne se trouve pas sur la ligne ou la rangée de la tour.</p> <p>Ceci est aussi vrai pour toutes les autres pièces (sauf les pions).</p>
---	--

L'idée est de prégénérer des tables pour chaque pièce (sauf les pions) un tableau à 2 dimensions `[source][Destination]` contenant des 0 (zone non atteignable) ou 1 (zone atteignable). Dans l'exemple ci-dessus, la table à prégénérer pour la tour en D4 est :

0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0

Exemple d'implémentation :

Utiliser un tableau à 3 dimensions:

```
char canAttack[6][64][64] ; // canAttack[TypeDePiece][Source][Destination]
```

Par exemple :

```
canAttack[ROOK][D5][E8] donnera 0
canAttack[ROOK][D5][E5] donnera 1

canAttack[QUEEN][D5][E5] donnera 1
canAttack[KNIGHT][D5][E5] donnera 0
```

Remarques :

- Utiliser des bitboards et des masques pour une performance optimale. A faire seulement si vous avez validé avec la table canAttack() !

Etape 1 :

Déclarer les tables dans data.h et modifier la fonction attack(int sq, int s) afin qu'elle prenne en compte les tables prégénérées. (i.e. si pour une piece autre qu'un pion, on scanne l'échiquier que si la table canAttack[..][..][..] renvoie 1).

La table canAttack[][][] permet d'éviter de dérouler des boucles inutilement si l'on sait à priori que la case de destination ne sera jamais attaquée par une pièce située sur une case source.

Etape 2 :

Dans la fonction main() appeler une fonction initAttacktables() ;

Par exemple :

```
...
initAttackTables ();
init_board();
open_book();
...
```

Etape 3 :

Commencer l'écriture de la fonction initAttackTables() ;

```
void initAttackTables()
{
    memset(canAttack,1,sizeof(canAttack)); // Pour test
}
```

A ce stade, à cause du memset(), on devrait obtenir le même nombre de nœuds, le même score et les mêmes variations.

Etape 4 :

Terminer l'écriture de la fonction `initAttackTables()` ;

```
void initAttackTables()
{
    //memset(canAttack,1,sizeof(canAttack)); // On enlèvera cette ligne lorsque tout
                                           // marche

    // génération de la table pour les pièces autre que les pions
    // Utiliser les fonctions existantes pour générer la table.
    // ne pas réinventer la roue !
    ...
}
```

Vérifier que le programme va plus vite avec « bench » (cf. plus haut).