

Projet IPI : stackchess

Jeu d'échec à empilement de pièces

Table des matières

Implémentation du plateau et des pièces.....	3
Le plateau de jeu.....	3
Les pièces.....	3
Récupération des caractères de choix.....	3
Les coordonnées des cases.....	3
Les choix utilisateurs.....	4
Boucles principales de jeu.....	4
Les fonctions de déplacement.....	5
Les validations d'un déplacement.....	5
Les déplacements.....	6
Limites du jeu.....	6
Taille du plateau.....	6
Match nul par placement des pièces.....	6
Match nul par type des pièces.....	7
Défaite par impossibilité de déplacement.....	7
Match nul par situations cycliques.....	7

Implémentation du plateau et des pièces

Le plateau de jeu

Je représente le plateau du jeu d'échec par un tableau à **une** dimension. Mon plateau est de **taille variable** `n`, je crée donc mon tableau avec :

```
tab = (stack *) malloc (n*n*sizeof(stack)) ;
```

J'ai besoin d'une implémentation de **pires** pour pouvoir empiler les pièces. Mes pires seront donc des **tableaux dynamiques de pièces**, pour une meilleure optimisation de la mémoire :

```
typedef struct stack {  
    piece      *tab_stack ;  
    int        summit ;  
    int        size ;  
} stack ;
```

On peut ensuite initialiser le plateau avec les pièces à leurs positions de départ avec une fonction `init(tab, n)` ; et l'afficher avec une fonction `print_game(tab, n)` ;

Les pièces

On représente les pièces par leur type (P pour pion, F pour fou etc.) et leur couleur (N pour noir, B pour blanc). Je définis donc mon type `piece` :

```
typedef struct piece {  
    char        nom[3] ;  
} piece ;
```

Le nom est de **taille 3** : deux cases pour le **type** et la **couleur**, une case pour le `'\0'`.

Récupération des caractères de choix

Les coordonnées des cases

Le jeu étant à taille variable, une coordonnée est constituée d'une **lettre** et d'un **nombre à 1 ou 2 chiffres**. Je choisis d'imposer une **taille maximale** de **26** en ayant la lettre comme contrainte. J'implémente deux fonctions permettant respectivement de vérifier la validité d'une coordonnée, et de convertir une coordonnée utilisateur en un indice valable dans mon tableau (par une correspondance entre matrice et tableau 1D) :

```
int is_valid_indice(char str[4], int n) ;  
int get_indice(char str[4], int n) ;
```

J'utilise `str` de taille 4 car une coordonnée peut au maximum être de taille 3, plus le `'\0'`.

Pour traiter une coordonnée utilisateur, je traite d'abord la **lettre** :

```
col = str[0] - 'a' ;
```

puis je crée un pointeur tmp ne contenant plus que le **nombre** et j'utilise :

```
line = n - strtol(tmp, NULL, 10) ;
```

Ainsi je récupère l'**indice** correspondant dans **mon tableau**, qui sera $n * line + col$.

Mon problème ici était la récupération du nombre lorsqu'il avait 2 chiffres : je récupérais le nombre avec `str[1]` et `str[2]` mais il fallait faire une disjonction de cas si $n \geq 10$ et vérifier que les deux caractères étaient bien des nombres. L'implémentation est plus simple et efficace avec `strtol`.

Les choix utilisateurs

Le joueur choisit entre sélectionner une case (c), déplacer une pile (d), ou abandonner (a). Il doit aussi entrer un nombre de pièce à déplacer dans le cas échéant. Je récupère le **choix** et le stocke dans une variable c avec :

```
fflush(stdin) ;
scanf("%1s", buf) ;
c = buf[0] ;
```

Le `fflush(stdin)` permet d'enlever tout caractère du `stdin` qui serait resté après un `scanf`. Dans le cas du **nombre de pièce**, j'utilise :

```
fflush(stdin) ;
scanf("%2s", buf) ;
nbr_deplacement = strtol(buf, NULL, 10) ;
```

J'utilise le **%2s** car on ne peut déplacer théoriquement que **52 pièces** d'un coup (2 chiffres).

Boucles principales de jeu

Le jeu **s'arrête** lorsqu'un joueur n'a **plus de pièces** ou qu'un joueur **abandonne**. A l'aide de 3 variables, j'implémente donc la boucle principale du jeu :

```
while (abandon && nbr_black && nbr_white) {
    print_game() ;
    if (player_color) printf(« Les blancs jouent\n ») ;
    else printf(« Les noirs jouent\n ») ;
    /* déroulement du jeu */
}
```

A l'aide d'une variable `player_color`, je gère le **passage** du joueur des **blancs** au jouer des **noirs**. Il faut ensuite gérer la boucle de sélection : tant qu'un joueur choisit c, il faut afficher le plateau de jeu et le contenu de la case sélectionnée. Pour cela, j'utilise un `while`

avec les instructions `continue` et `break`. Il faut aussi une fonction `count_pcs` qui **mette à jour** les variables `nbr_black` et `nbr_white` lorsqu'il y a un déplacement :

```
while(1) {
    /* instructions de selection/choix */
    if (choix=='c') {
        print_game() ;
        continue; }
    else if (choix=='a') {
        abandon = 0 ;
        break ; }
    else if (choix=='d') {
        /* instructions pour le deplacement */
        player_color = 1 - player_color ;
        count_pcs(tab, n, &nbr_white, &nbr_black) ;
        break ;
    }
}
```

Les fonctions de déplacement

Les validations d'un déplacement

Pour valider un déplacement, il faut vérifier que le déplacement soit **légal** pour toutes les pièces, mais aussi que les pièces ne passent **pas au-dessus** d'autres pièces, d'où les fonctions respectives :

```
int is_valid_deplacemt(stack *tab, int n, int nbr, int old, int new)
int is_jumping(stack *tab, int n, int old, int new)
```

Pour la fonction `is_valid_deplacemt`, je vérifie que toutes les pièces de la première jusqu'à la `nbrème` puisse se déplacer à l'aide d'un `switch` :

```
switch(c) {
    case 'T' : if (/* deplacement autorise de la tour */)
        break ;
        else return 0 ;
        break ;
    case 'F' : if (/* deplacement autorise du fou */)
        break ;
        else return 0 ;
        break ;
    etc.
}
```

Je vérifie les déplacements pour chaque pièce à l'aide des variables `old` et `new` symbolisant la case de **départ** et **d'arrivée**. Je **converti ces indices** en **coordonnées** dans une **matrice** : `old` est en case (`old_i`, `old_j`) et `new` en case (`new_i`, `new_j`) de l'équivalent

matriciel du tableau.

On a : $\text{old_i} = \text{old}/n$, $\text{old_j} = \text{old}\%n$, $\text{new_i} = \text{new}/n$, $\text{new_j} = \text{new}\%n$

Ainsi, il est aisé de vérifier que `old` et `new` sont sur une même ligne, colonne, diagonale etc.

Les déplacements

Lors d'un déplacement, le joueur va soit déplacer sa pile de pièces sans autre effet, soit retirer du plateau les pièces ennemies correspondantes. J'implémente donc les fonctions :

```
void deplacer(int n, stack *tab, int old, int new) ;  
void manger(int n, stack *tab, int old, int new) ;
```

Pour **déplacer** une pile de pièce en conservant l'ordre, j'utilise une pile temporaire `tmp`. Je **dépile** `n` éléments de la case `old` en **empilant** sur `tmp`, puis je **dépile** `tmp` en **empilant** sur la case `new`. Pour « **manger** » les pièces adverses, il suffit de **dépiler** le contenu de la case `new` d'abord puis de déplacer la pile.

Limites du jeu

Taille du plateau

Comme dit dans la section coordonnées des cases, la lettre est une composante de la coordonnée, j'ai donc **limité la taille** du plateau à **26** pour ne pas avoir à définir une nouvelle façon de noter les coordonnées. Il aurait été possible de numérotter les colonnes 'aa', 'ab' après la 'z', mais cela implique de changer aussi l'implémentation des fonctions `is_valid_indice` et `get_indice`. La question aurait encore pu se poser après la colonne 'zz', il aurait donc fallu **généraliser le processus**.

Match nul par placement des pièces

Il est possible d'arriver à une situation de match nul qui ne dépend pas du type des pièces mais de leur placement. En effet, prenons la situation où un joueur possède **2 pions** l'un sur l'autre, et l'ennemi possède **2 pions** situés sur la **même colonne** ou une colonne distante de plus de 1 colonne par rapport au premier joueur. En théorie, les joueurs disposent des types de pièces qui leurs permettent de manger des pièces adverses. Mais en pratique, **ni l'un ni l'autre** ne pourront **manger** de pions adverses du fait du placement des pions. Ce cas de « match nul par placement » n'est pas pris en compte, un joueur doit abandonner.

Match nul par type des pièces

Les situations de match nul à cause des types des pièces **ne devraient pas se produire**.

Un match nul par type de pièce apparaît quand les deux joueurs ne possèdent que des pièces à déplacements incompatibles, ils ne peuvent donc pas empiler deux pièces pour pouvoir « manger » une pièce adverse. En effet : **supposons** qu'une telle situation se présente. Cela implique que dans une **situation précédente**, un joueur a **mangé au moins une pièce** de l'autre car les joueur débutent avec toutes les pièces du jeu. **Contradiction** : les joueurs ne possèdent que des pièces à déplacements incompatibles, il est impossible pour un joueur d'avoir réussi à manger une ou plusieurs pièce de l'autre dans une situation précédente. Ce cas précis de match nul ne peut donc pas apparaître.

En revanche, un **cas problématique** s'en rapprochant est le suivant : si un joueur possède cette pile : Pion – Tour – Fou, il est **impossible** pour lui de **manger** une pièce adverse (les 3 pièces sont incompatibles et les 2 du dessus aussi) alors qu'il possède théoriquement des pièces compatibles. Il doit donc **changer l'ordre des pièces**, mais cela n'est pas toujours **possible** selon la **disposition** des pièces adverses, il se trouve donc dans une **impasse**. Ce cas n'est pas traité dans le jeu et nous amène au cas suivant.

Défaite par impossibilité de déplacement

De manière analogue à l'impasse due à l'ordre de la pile, un joueur peut se trouver dans la situation où il ne peut **déplacer aucune pièce** à cause des **placements des pièces adverses**. Dans cette situation, le joueur ne peut effectuer aucune action durant son tour, il ne peut donc qu'abandonner. Ce cas « d'obligation d'abandon » n'est pas traité dans le jeu.

Match nul par situations cycliques

Le dernier cas posant problèmes apparaît lorsque les deux joueurs possèdent des pièces à déplacements compatibles, qui théoriquement ne donnent pas lieu à un match nul par placement, mais qui pour autant ne se mangeront jamais. Par exemple : si les **deux joueurs** possèdent **deux tours**, il semble d'abord improbable qu'un joueur laisse ses tours seules et ne les empilent pas. Si toutefois il en décide ainsi, il ne laissera jamais une tour sur la même ligne / colonne que la pile de 2 tours de l'ennemi (il se ferait manger le tour suivant). Il en résulte une situation de **déplacements infini** où il est impossible pour l'un comme pour l'autre de manger l'ennemi.

On peut imaginer une **solution** à ces problèmes avec une **limite de temps** et compter le **nombre de pièces** que chaque joueur à manger, avec un cas d'égalité.