Documentation sur le jeu d'échecs et son implémentation

Matthias Druelle
Paul Boutes
Damien Cassan

12 octobre 2014

Table des matières

1 Règles des échecs

1.1 Natures et objectifs du jeux d'échecs

Le jeux d'échecs est jouer par deux joueurs, adversaire, disposant chacun de 16 pièces. l'appartenance d'une pièce a un joueur est définit par la couleur cette pièce. Le joueur avec les pièces blanches commence la partie, le jeux se déroule tour par tour, chaque joueur a la possibilité d'effectuer un seul mouvement par tour. Les pièces d'un joueurs ont la possibilité d'éliminer les pièces appartenant a l'autre joueurs en déplaçant une pièce par dessus celle de son adversaire, la pièce éliminé est alors retiré de l'échiquier. Lorsque le roi est en phase de se faire éliminer le joueur auquel appartient ce roi est dit en échec, il doit absolument sortir de cette position d'échec. Si il est incapable de sortir de cette position il est alors en échec et mat et l'autre joueur remporte la partie.

1.2 Disposition de l'échiquier

L'échiquier est composé de 64 cases adjacentes, qui prennent alternativement les couleurs noirs et blanches. Les rangée de 8 cases situé entre les deux joueurs sont appelé colonnes et les rangée orthogonaux a celle-ci sont appelé traverses. On distingue aussi les rangées de cases adjacentes de même couleur appelés diagonales.

1.3 Rôles et mouvements des pièces

- Le roi :
  - Le roi peut se déplacer sur toute les cases adjacente a la sienne a condition qu'il ne se mette pas en condition d'échec ;
  - Il dispose aussi d'un autre déplacement appelé le roque (a compléter).
- La dame :
  - La dame peut se déplacer sur une cases quelconque appartenant a la colonnes, la traverses ou la diagonale a laquelle appartient sa case.
- La tour :

    - La tour peut se déplacer librement sur les colonnes et les traverses effectuant une intersection avec sa case.
- Le fou :
    - Le fou peut se déplacer librement sur les diagonales effectuant une intersection avec sa case.
- Le cavalier :
    - Le mouvement du cavalier se décompose en deux parties, tout d'abord il effectu un premier déplacement sur une colonne ou une traverse, ensuite il effectu un déplacement en diagonale tout en s'éloignant de sa position d'origine. Il n'est pas bloqué par pièces positionné sur son passage.
- Le pion :
    - Le pion avance d'une case sur sa colonne en direction de l'adversaire ;
    - Lors de son premier mouvement il a la possibilité de se déplacer de deux case sur sa colonne ;
    - Il peut prendre les pièces situé à une case sur sa diagonale ;
    - Prise en passant (a compléter) ;
    - Si il atteint la dernière traverse il peut être changer en une autre pièce aux choix du joueur, et effectuer un second mouvement immédiatement.

1.4 Coups particuliers

- La prise en passant :
  Si un pion sur sa case initiale avance de deux cases mais aurait pu être capturé par un pion adverse s'il n'avait avancé que d'une case, la prise est possible comme si le pion n'avait avancé que d'une case.
- La promotion :
  tout pion qui parvient sur sa derniére rangée doit être immédiatement remplacé par une autre piéce de sa couleur (Roi excepté), au gré du joueur.
- Le roque :
  Le roi peut, une fois par partie se refugié sur l'une ou l'autre aile : il se déplace horizontalement de deux cases vers l'une des deux tours tandis que celle-ci saute au-dessus de lui pour venir se placé a coté. On

distingue le Petit Roque et le Grand Roque selon le parcours effectué par la tour. Cette action n'est possible que si le roi n'est pas en echec lorsqu'il effectue le roque ainsi que sur chacune des cases qu'il traverses jusqu'a sa position d'arrivé comprise.

1.5 Notations

L'échiquier comporte un système simple de coordonnées, permettant de désigner chaque case : on attribue à chaque colonne une lettre de a à h, et à chaque rangée un chiffre , de 1 à 8. Chaque case est ainsi identifié par la lettre et le chiffre correspondant à l'intersection d'une colones et d'une rangée.

Un systéme de notation a été mis au point pour conserver et reproduire les parties : on définit le mouvement dd'une pièce en indiquant l'initiale de la pièce (en majuscule), sa case de départ et sa case d'arrivée, séparées par un tiret en cas de simple déplacement et par le signe multiplié en cas de prise. On n'utilise pas d'initiale pour le pion.

On utilise en outre les symbole suivants :

- Le petit Roque : 0-0
- Le grand roque : 0-0-0
- Va à : -
- Prise : x
- Echec : +
- Mat :!=
- Roi : R
- Dame : D
- Fou : F
- Cavalier : C
- Tour : T

2 L'algorithme Minimax

L'algorithme Minimax est un algorithme utilisé pour établir une stratégie de jeu. Il s'applique dans les jeux à somme nulle, opposant deux joueurs et où toutes les informations du plateau sont visibles. On parle de jeu à somme nulle

lorsqu'en cours de partie, un gain pour un joueur constitue nécessairement une perte pour l'autre. La somme des gains des deux joueurs est donc nulle.

2.1 Principe

L'algorithme repose sur le principe suivant : minimiser ses pertes en supposant que l'adversaire veuille maximiser ses gains.

La fonction Minimax est une fonction récursive utilisant trois sous-fonctions : la fonction Min, la fonction Max et la fonction Evalutation.

- La fonction Min retourne à partir d'un arbre de coups possibles le noeud ou les pertes seront les plus faibles.
- La fonction Max retourne à partir d'un arbre de coups possibles le noeud ou les gains seront les plus élevés ;
- La fonction Evaluation retourne un nombre dont la valeur correspond à la position du jeu. Le nombre sera élevé si le jeu est favorable au joueur, il sera plus faible sinon.

2.2 Modelisation

Minimax construit un arbre contenant tous les coups possibles pour un tour. On appelle la profondeur le nombre de niveaux de l'arbre obtenu. PLus la profondeur est grande, plus l'algorithme est efficace. Minimax appelle ensuite Min si il s'agit d'un coup du joueur, ou Max si il s'agit d'un coup de l'adversaire. Min et Max vont ensuite s'apeller mutuellement jusqu'à atteindre les feuilles de l'arbre.

Sur le schéma suivant, les flèches bleues correspondent aux coups possibles du joueur, les flèches jaunes à ceux de l'opposant. L'algorithme est de profondeur 2 et on considère que trois coups sont possibles pour chaque configuration de jeu.

L'algorithme va parcourir l'arbre et utiliser la procédure d'évaluation pour associer à chaque feuille une valeur numérique. On se place ensuite sur les noeuds de profondeur 1 qui prendra la valeur du plus petit de ses fils : on minimise. On remonte ensuite d'un niveau et on prend ce coup ci la valeur du plus grand des fils : on maximise.

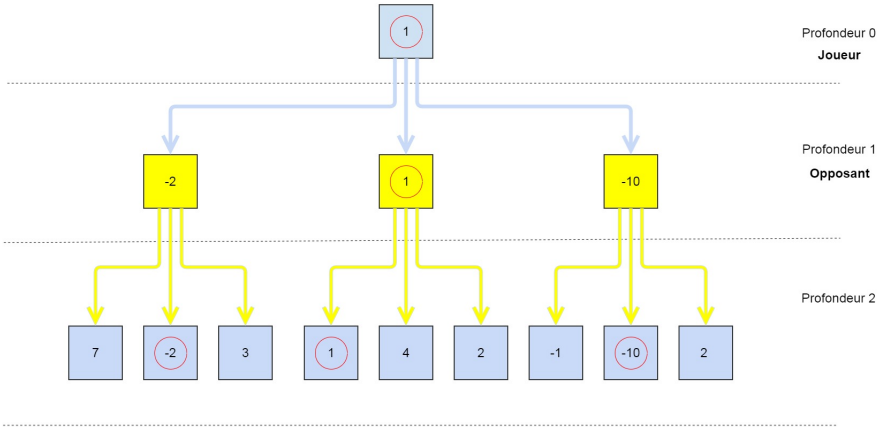

FIGURE 1 – MiniMax

On en déduit la série de coup la plus judicieuse à jouer : celle qui mène à la feuille qui est remontée au niveau 0.

- p est le noeud d'un arbre, il possède n fils (f1, f2, ..., fn).
- minimax(p) = eval(p) si p est une feuille
- minimax(p) = max(minimax(f1), minimax(f2), ... minimax(fn)) si il s'agit d'un coup du joueur
- minimax(p) = mim(minimax(f1), minimax(f2), ... minimax(fn)) si il s'agit d'un coup de son opposant

2.3 Le Minimax associé aux échecs

Le nombre de coup possible aux échecs pour un tour est élevé. Pour une ouverture par exemple, au moins 20 coups différents sont possibles alors que plusieurs pièces n'ont pas la possibilité de bouger. Ainsi pour simuler les differentes possibilités sur les deux premiers tours on doit evaluer prés de 400 coups et pour une profondeur de 3 : 8000 coups. L'efficacité d'une IA Minimax est basée sur la profondeur de l'arbre des coups. Par conséquent, notre algorithme devra travailler sur un arbre relativement important et sera donc probablement coûteux en ressources. Il nous faut donc optimiser l'implementation de notre algorithme Minimax c'est pour cela que nous utiliserons l'elagage alpha-beta et une table de transposition.

2.4 L'evaluation d'un plateau de jeu

la principale difficulté de l'algorithme minimax ainsi que son efficacité repose sur la fonction d'evaluation, qui pour le jeux d'echecs vas à partir de la configuration d'un échequier produire une valeur numerique representant la pertinence de cette configuration. par exemple lorsque le joueur adversaire est en echec l'evaluation de cette configuration vas produire un trés grand nombre pour signaler que cette position est favorable au joueur.

Nous allons donc basé l'evaluation d'un échequier sur differentes données :

- Le nombre et le type de piéce present sur l'echequier :
  - Il faudrait donc attribuer a chaque piéce une valeur en fonction du type de piéce et si lorsque la piéce est retirer du jeux, enlever cette valeur au score du plateau si la piéce appartient au joueur ou sinon l'ajouter si la piéce appartenait au joueur adverse.
- Comparer la configuration actuelle a des ouvertures connues :
  - Nous disposons d'une base de données d'ouverture ou chaque connfiguration est représenté par un hash contenant la position et le type de toutes les piéces presente sur l'echequier, l'ensemble de ces hash serait donc placé dans une table de hashage et la fonction d'evaluation pourait donc ce referé a cette base de donnnées en comparant le hash du jeu actuel avec celui de cette base de données d'ouverture.
- Prendre en compte les situations d'echecs et de pat
  - L'evaluation doit absolument prendre en comptes les situations d'echecs, pat et mat notamement car l'echecs va bloquer la plupart des possibilité de jeu de l'adversaire car il est obligé de se sortir de cette situation mais plus encore c'est une conndition neccesaire à l'echecs et mat qui constitue le seul moyen de remporter la partie.

2.5 Optimisation du Min-Max

2.5.1 Le NégaMax

Le principe du Negamax est relativement simple. Lorsque dans l'algorithme min-max nous testons si nous sommes sur un noeud à niveau pair ou impair, dans le but de maximiser ou de minimiser une évaluation, le Négamax va

toujours chercher à maximiser l'évaluation. Il suffit d'inverser le signe des évaluations à chaque niveau, de ce fait, notre algorithme va toujours chercher à maximiser une évaluation.

Ce principe est assez intérresant, car il nous permet d'alléger l'algorithme min-max, en évitant certaines comparaisons.

2.5.2 L'élagage Alpha-Beta

L'algorithme Alpha-Beta est une amélioration de l'algorithme min-max. Le but de cet algorithme est d'éviter le parcours de branches inutiles dans l'arbre des coups possibles. Comme son noms l'indique, l'algorithme alpha-beta utilise deux paramètres qui sont :

- alpha ;
- beta.

La coupure alpha se fait sur les niveaux Min. Le principe de cette coupure va être qui si la valeur d'un niveau Min est plus petite que la valeur d'un niveau Max supérieur, nous considérons que la valeur du noeud Max supérieur ne changera pas, de ce fait, il est inutile d'aller explorer les branches suivantes du niveau Min.

D'un autre côté, nous avons la coupure béta, cependant, cette coupure s'effectue sur les niveaux Max. Le principe de la coupure béta est le même que pour la coupure alpha, mais pour les niveaux Max.

Nous avons donc deux coupures qui nous permettent d'éviter des parcours inutiles dans l'arbre des coups possibles, ce qui est un gain de temps.

2.5.3 Comment utiliser les variables alpha et béta ?

Tout d'abord, nous devons savoir qu'il y a deux règles :

- sur un niveau max, si la valeur du noeud >= béta, nous effectuons une coupure béta (1) ;
- sur un niveau min, si la valeur du noeud <= alpha, nous effectuons une coupure alpha (2).

C'est grâce à ces comparaisons quer nous pouvons effectuer un élagage, alpha ou béta.

Les variables alpha et béta sont mis à jours lorsque un noeud retourne une évaluation. Sur un niveau Min, la valeur béta va être mis à jour, puis sur un noeud max, la valeur alpha va être mis à jour.

Toutes les valeurs alpha et béta vont remonter aux niveaux supérieurs, et c'est à ce moment là que nous pourrons effectuer des tests avec les règles (1) et (2), et donc procéder à des coupures si besoin. Il faut savoir que les valeurs alpha et béta hérite des niveaux supérieurs.

3 Ouverture

Les ouvertures sont des stratétegies de jeux utilisées en debut de parties, elle permettent notamment de bien positioner ses pièces et de placer les rois en securités. Il existe une multitude d'ouvertures et pour chacune d'entre-elles plusieurs variantes, elle sont souvent nommé en fonction de leur inventeur.

On utilise la notation algebrique pour définir les ouvertures. Les deplacements dans la colonne de gauche correspondent aux mouvements des piéces blanches et ceux de la colonne de droite aux piéces de couleurs noires.

3.1 Implementation

Pour implémenter les bases de données d'ouverture nous avons choisis d'utiliser une table de hachage. Un etat de l'échiquier sera donc réprésenté par un hash et l'IA cherchera la correspondance entre l'etat actuel de l'échiquier et une ouverture dans la base de données.

Format d'un hash :

(Initiale de la piéce - position sous la forme algebrique) : (suivant) ..

Example :

RA1 :DA3 :F2 :C4

Correspond à :

- Roi en position A1 ;
- Dame en position A3 ;
- Fou en position B2 ;

— Cavalier en position C4.

Il faut respecter l'ordre dans lequelle sont placés les piéces dans le hash afin que deux états identique aient toujours le même hash. On définira donc l'ordre suivant :

Roi - Dame - Fou - Tour - Cavalier - Pion

Il faut aussi définir un ordre pour les piéces de même types. On peut par exemple définir un ordre pour les cases :

Suggestions : - On compare les lettres A < B < C … ; - Puis le nombre 1 < 2 ; - -> A8 < B2 < C3 < C6.

On aura donc bessoin :

— D'une fonction comparant deux cases ;
— D'une fonction prenant une liste de piéce et renvoyant un hash.

3.2 Defense Indienne

Cette strategie permet de sécuriser la position de la reine pour le joueur ayant les piéces noires.

1. : d4 Cf6
2. : c4 e6
3. : Cf3 b6

3.3 Defense Benoni

1. : d4 Cf6
2. : c4 c5

3.4 Le mat du berger

Cette ouverture permet d'effectuer un echec et mat en seulement 4 tours.

1. : e4 e5
2. : Dh5 Cc6

3. : Fc4 Cf6
4. : Df7 -> Mat

3.5 La croisade du cavalier

1. : d4 Cf6
2. : Cd2 e5
3. : de5 Cg4
4. : h3 Ce3 # Implémentation du jeu

Pour réaliser ce projet, nous devons penser aux différentes structures, types de données que nous allons utilisés dans notre projet.

3.6 Structures de données

3.6.1 Représentation de l'échiquier

Pour pouvoir réprésenter l'échiquier, nous allons implémenter une structure représentant les cases de l'échiquier.

La structure "Case" sera donc la suivante :

```
struct Case {
      int colonne;
      int ligne;
      Piece *piece;
}
```

— L'entier colonne représente une colone de l'échiquier ;
— L'entier ligne représente une ligne de l'échiquier ;
— La structure Piece représente la pièce présente sur la case à la position colonne et ligne.

La structure "Piece" sera donc la suivante :

```
struct Piece {
     char type;
     Case *case;
     Joueur *joueur;
```

```
}
```

- Le char type représente le type de la pièce (pion, roi, dame, fou, tour) ;
- La *case représente la case sur laquelle se trouve la pièce.
- La *joueur permet d'avoir les différentes informations du joueur à qui appartient la pièce.

De ce fait, grâce à ces deux structures, nous pouvons représenter la grille de notre échiquier.

La grille est donc : `typedef Grille Case[8][8];`

Grâce à cette représentation, nous avons facilement accès aux pièces présentent sur une case, ou à l'inverse, à la position d'une pièce dans la grille.

3.6.2 Représentation du joueur

De plus nous utiliserons une liste de Pièce qui va donc contenir les différentes pièces encore présentent sur la grille, ce qui va nous être utile pour effectuer des opérations sur l'ensemble des pièces, sans faire de parcours inutile à partir de la grille. Par exemple, le calcul de l'echec, ou du pat. Cette liste va être rattaché au joueur, pour indiquer les pièces du joueur présente sur l'échiquier.

La structure "Joueur" sera donc la suivante :

```
struct Joueur {
    int id;
    GList *listePieces;
    Piece *roi;
    Joueur* adverse;
    bool PrisePassant;
    Case *positionPrisePassant;
}
```

- L'entier id représente le joueur 1 ou le joueur 2 ;
- La Glist* listePieces représente la liste des pièces sur l'échiquier de joueur ;
- La Piece* roi permet de garder la position du roi en mémoire, pour calculer l'échec, et le pat plus rapidement ;

— Le Joueur* adverse permet d'avoir accès aux données du joueur adverse durant la partie ;
— Le booleen PrisePassant indique s'il est possible pour le joueur adverse d'effectuer une prise en passant pour le tour actuel.
— La Case* positionPrisePassant permet de retenir l'endroit où il est possible d'effectuer une prise en passant.

3.6.3 Représentation du jeu

Pour représenter les différentes variables du jeu, nous allons les regroupés dans une structure Jeu, qui va donc être :

```
struct Jeu {
    Joueur* blanc;
    Joueur* noir;
    int tour;
    Grille* grille;
}
```

— Le Joueur* blanc représente les pièces blanche ;
— Le Joueur* noir représente les pièces noires ;
— L'entier indique quel joueur doit jouer ;
— La Grille* grille représente la grille de l'échiquier.

4 Implémentation du déplacement

4.1 Prototypes

```
/**
 * Permet d'effectuer le déplacement d'une pièce
 *
 * g  : Représente la grille du jeu, l'échiquier
 * depart  : Représente la case de départ de la pièce
    à déplacer
 * arrive  : Représente la case d'arrivée de la pièce
    déplacée
```

```
 * j  : Représente le joueur
 *
 * retourne 0 si tout va bien, sinon un code d'erreur
 */

int deplacer(Grille* g,Case depart, Case arrive,
   Joueur* j);

/**
 * Génère la liste des positions possibles pour une
    pièce
 *
 * depart  : Représente la case de la pièce a déplacer
 * retourne une Liste de doublé des positions
    possibles pour une pièce
 */

GList* genererPosition(Case depart);

/**
 * Vérifie les positions valide dans la GList*
    listePosition
 *
 * listePosition  : liste de doublé contenant toutes
    les positions possibles pour une pièce
 *  g  : Représente la grille du jeu
 *  j  : Représente un joueur
 *  retourne la liste des déplacements valide (enlève
    les déplacements hors de la grille et sur les
    pions alliés)
 */

GList* check(GList* listePosition, Grille* g, Joueur*
   j);


/**
 * Permet de mettre à jour les pièces déplacés et/ou
    mangés
```

```
 *
 *  g  : Représente la grille du jeu
 *  listePosition  : liste de doublé contenant toutes
    les positions possibles pour une pièce
 *  depart  : Représente la case de départ de la pièce
 *  arrive  : Représente la case d'arrivée de la pièce
 *  j  : Représente le joueur qui déplace sa pièce
 *
 */

void effectuerDeplacement(Grille* g, Glist*
   listePosition, Case depart, Case arrive, Joueur* j)
   ;

/**
 * Permet d'annuler un déplacement d'une pièce
 *
 * pieceDeplace  : Représente la pièce qui a été
    déplacé
 * caseCible  : Représente la case sur laquelle la
    pièce déplacé se trouvait
 * g  : Représente la grille du jeu
 * depart  : Représente la case d'origine de la pièce
    déplacé
 */

void annulerDeplacement(Piece* pieceDeplace, Case*
   caseCible, Grille* g, Case* depart);


/**
 * Permet d'annuler la prise de la pièce
 *
 * piecePrise  : Représente la pièce qui a été prisee
 * arrive  : Représente la case sur laquelle était
    placé piecePrise
 * j  : Représente le joueur
 */
```

```
void annulerPrise(Piece* piecePrise, Case* arrive,
   Joueur* j);

/**
 * Permet de vérifier si le joueur j est en echec
 * On génère toutes les positions possibles des pièces
    , et on compare avec la liste de pièces du joueur

 * j  :Représente le Joueur avec sa liste de pièce
 * retourne vrai si le joueur est en echec, sinon
    retourne non
 */

bool isEchec(Joueur* j);
```

4.2 Pseudo-code

```
int deplacer(Grille* g,Case *depart, Case *arrive,
   Joueur* j) {
    GList *p;
    p <- genererPosition(depart);
    p <- check(p,g,j);

    if (arrive in p) {
        Piece *pieceDeplacer;
        Piece *piecePrise = NULL;

        effectuerDeplacement(g,p,depart,arrive,j);

        if (isEchec(j)) {
            annulerDeplacement(pieceDeplace, arrive, g
               , depart);

            if (piecePrise) {
                annulerPrise(piecePrise, arrive, j);
            }
```

```
                return 1;
            }

        }
        return 0;
}
```

5 Sauvegarde

Le programme doit pouvoir sauvegarder l'etat d'une partie dans un fichier de façon a pouvoir reprendre la partie entre chaque execution du programme.

5.1 Structure du fichier

Nous devons stocker les données suivantes :

- Les piéces :
  - positions
  - type
  - appartenance joueurs
- l'avancement
  - tours
- les joueurs :
  - prisePassant

Nous avons donc choisie de le modéliser de la facon suivante :

```
{
    'Partie' :{
        'Nombre de tours'  :n,
        'PrisePassant' :true/false,
        'PositionPrisePassant' :'A1 '
    },
    'Joueurs1' :{
        'Hash' :'RA1 DB2...'
    },
    'Joueurs2' :{
```

```
        'Hash' :'CA3 :....'
    }
}
```

6 Implementation

Deux choix on été retenue pour implémenter cette sauvegarde :

- Fichier Binaire
  - Avantage :
    - Rapide à parcourir
    - Petite taille du fichier
  - Incovenient
    - Implementation
- Fichier JSON
  - Avantage
    - Lecture facile
    - Facilement utilisable par d'autres programmes
    - Implementer par jsonGlib
  - Incovenient
    - Taille importante des fichiers

6.1 Prototype

```
/**
 * Permet de sauvegarder dans un fichier l'état du jeu
 *
 * fichier  : Nom du fichier qui va être enregistré
 * jeu  : Structure regroupant les joueurs, le tour, et
     la grille de l'échiquier
 *
 * retourne 0 si tout va bien, sinon un code d'erreur
    si le fichier ne s'est pas enregistré
 */

int sauvegarder(char* fichier,Jeu* jeu);
```

```
/**
 * Permet d'ouvrir un fichier et d'initialiser les
    variables associées
 *
 * fichier  : Nom du fichier qui va être chargé
 * jeu  : Structure regroupant les joueurs, le tour, et
     la grille de l'échiquier
 *
 * retourne 0 si tout va bien, sinon retourne un code
    d'erreur s'il y a une erreur lors du chargement du
     fichier
 */

int ouvrir(char* fichier, Jeu* jeu);
```