Algorithmes sur le jeux d’échec

ld = ligne départ
cd = colonne départ
la = ligne arrivée
ca = colonne arrivée

ValidMove(ld,cd,la,ca) :

Pour toutes les pièces du jeu d’échec :

- position (la,ca) dans l’échiquier
- pas de pièce de ma couleur à l’arrivée (la,ca)

Pion :

- premier mouvement d’une ou deux case
- prise en diagonale
- prise en passant

Cavalier :

- juste 8 positions à vérifier : (ld ± 1, cd ± 2) == (la,ca) ou (ld ± 2, cd ± 1) == (la,ca)

Tour :

- vérifier qu’on est bien sur une ligne ou une colonne : la == ld ou ca == cd
- vérifier l’ensemble des cases entre la case de début et la case de fin sont vides

Fou :

- vérifier qu’on est bien sur diagonale : (la – ld) / (ca – cd) == ± 1
- vérifier l’ensemble des cases entre la case de début et la case de fin sont vides

Reine :

- pas besoin, avec l’héritage de Tour et Fou

On peut mutualiser la vérification de l’ensemble des cases entre la case de début et la case de fin de la façon suivante :

- on fait une fonction/méthode bool verif(ld,cd,la,ca,delta_l,delta_c)
- delta_l et delta_c appartiennent à (-1,0,1)
- pour la vérification, on part de la case (l,c) = (ld,cd)
- pour passer à la case suivante, on fait l = l + delta_l et c = c + delta_c
- pour une tour, (delta_l = 0 et delta_c = ± 1) ou (delta_l= ± 1 et delta_c = 0)
- pour un fou, (delta_l = ± 1 et delta_c = ± 1)

Une façon élégante de faire ceci est de créer une classe abstraite TourFou qui contient la méthode bool verif(ld,cd,la,ca,delta_l,delta_c). Tour et Fou héritent de cette classe et utilise ma méthode verif(ld,cd,la,ca,delta_l,delta_c).

Echec

Il faut vérifier après chaque coup deux échecs :

- mon mouvement a-t-il mis mon roi en échec ? Si oui, il faut annuler ne mouvement.
- mon mouvement a-t-il mis le roi adverse en échec ? Si oui, il faut le signaler.

Pour vérifier un échec, il suffit de vérifier pour chaque pièce d'une couleur si l'une d'entre elles a un ValidMove sur la position du roi adverse. On se rend compte à ce niveau là que savoir à tout moment où sont les rois sur l'échiquier sera pratique.

Echec et mat

Il faut vérifier après chaque coup l'échec et mat. Cela implique :

- le roi est en échec
- aucun mouvement d'aucune pièce de sa couleur ne peut éliminer cet échec.

Il faut donc pour chaque pièce de sa couleur tester chaque mouvement valide et une fois le mouvement effectué, vérifier à nouveau l'échec. Si après un des mouvements, on n'est plus échec, alors il n'y a pas échec et mat.

Pour faire ce travail de manière efficace, ajouter aux pièces du jeu une méthode getMoves ou getValidMoves qui renvoie une liste STL des mouvements possibles permettra de tester une nombre limité de cas. Dans le cas contraire, il faut tester chaque case de l'échiquier.

Comme on est amené à modifier de nombreuse fois l'échiquier, puis à revenir en arrière, deux solutions s'offrent à nous :

- prévoir un mécanisme de retour en arrière, en mémorisant la succession des mouvements par exemple. Ce n'est pas trop compliqué, mais il faut penser à gérer les pièces mangées.
- faire à chaque fois une copie dynamique de l'échiquier, la modifier, puis la libérer quand on en a plus besoin. Méthode bourrine et peu élégante, mais qui fonctionne en faisant un simple constructeur par copie.

Pat (match nul)

Si à son tour de jouer, on ne peut bouger aucune pièce sans que notre roi soit en échec, on est pat.

La vérification du pat est proche de celle de l'échec et mat. On teste tous les mouvements possibles et si à chaque fois on se trouve en échec, alors on est pat.