

Linechec v0.16

Gérald Dumas

11 février 2004

Table des matières

I	Structures et règles du jeu.	7
1	Les informations nécessaires.	9
1.1	Les pièces.	9
1.2	Structure d'une pièce.	9
1.2.1	La variable "type" :	10
1.2.2	La variable "couleur" :	10
1.2.3	La variable "ponderation" (Pas encore utilisée a ce stade du développement) :	10
1.2.4	La variable "place" :	10
1.2.5	La variable "nbre_cases_jouables" :	10
1.2.6	La variable "deplacement_initial" :	10
1.3	Les déclarations de macros.	10
1.4	Le tableau " <i>table_piece [33]</i> " :	11
1.5	L'échiquier.	12
1.5.1	Pourquoi 144 cases?	12
1.5.2	L'initialisation de l'échiquier.	12
1.6	Les différentes rosaces de déplacement.	13
1.6.1	Les tableaux "rosace[][]" et "corres_piece[]".	13
1.6.2	Les déplacements du pion.	13
1.6.3	Les déplacements du cavalier.	14
1.6.4	Les déplacements de la tour.	15
1.6.5	Les déplacements du fou.	15
1.6.6	Les déplacements de la dame.	15
1.6.7	Les déplacements du roi.	15
1.7	Structure d'une case de l'échiquier.	15
1.7.1	La variable <i>pondération</i> (<i>pas encore utilisée à ce stade du développement</i>).	16
1.7.2	La variable <i>type_piece</i>	16
1.7.3	La variable <i>couleur_piece</i>	16
1.7.4	La variable <i>couleur_case</i>	16
1.7.5	La variable <i>pos_table_piece</i>	16
1.7.6	Les variables <i>x</i> et <i>y</i>	16
2	Les outils nécessaires.	17
2.1	Gestion d'une liste chaînée.	17
2.1.1	La déclaration d'une liste.	17
2.1.2	Ajouter un élément à la fin de la liste.	17
2.1.3	Se placer en début de liste.	18
2.1.4	Se placer en fin de liste.	18
2.1.5	Passer à l'élément suivant de la liste.	18
2.1.6	Passer à l'élément précédent de la liste.	18

2.1.7	Enlever un élément de la liste.	18
2.1.8	Déterminer le nombre d'élément de la liste.	19
2.1.9	Déterminer le nombre d'élément en fonction d'une couleur.	19
2.1.10	Trier la liste.	19
2.1.11	Vider la liste.	19
2.1.12	Conclusion.	20
2.2	Gestion d'un arbre k.aire.	20
2.2.1	Qu'est-ce qu'un arbre k.aire?	20
2.2.2	Comment s'effectue la lecture d'un tel arbre?	20
2.2.3	Un arbre dynamique.	20
2.2.4	L'intérêt d'un arbre k.aire.	20
2.2.5	Les fonctions de gestion d'un arbre.	21
2.2.6	Utilisation des fonctions.	21
2.2.6.1	Détails de la structure <code>_noeud</code>	21
2.2.6.2	Création de la bibliothèque d'ouverture.	21
2.2.6.3	Ajout d'un noeud.	22
2.2.6.4	Se replacer à la racine de l'arbre.	22
2.2.6.5	Pointer sur le premier enfant du noeud courant.	22
2.2.6.6	Pointer sur l'enfant suivant du noeud courant.	22
2.2.6.7	Pointer sur l'enfant suivant du noeud courant.	22
3	Les règles du jeu.	23
3.1	La liste des coups jouables.	23
3.1.1	Structure d'un coup jouable.	23
3.1.2	La variable <code>pos_piece_tab</code>	23
3.1.3	La variable <code>coup</code>	24
3.1.4	La variable <code>couleur</code>	24
3.1.5	La variable <code>valeur_coup</code>	24
3.1.6	La variable <code>nbre_coups_adverse</code>	24
3.1.7	Le drapeau <code>case_occupee</code>	24
3.1.8	Le drapeau <code>coup_ami</code>	24
3.2	La détermination des coups jouables.	24
3.3	Les coups cloués.	25
3.3.1	Le test du roi en echec.	25
3.3.2	La fonction d'élimination des coups cloués.	27
4	Le déplacement d'une pièce.	29
4.1	Séquencement d'un déplacement :	30
4.2	La fonction de déplacement d'une pièce.	30
II	Les fonctions annexes	33
5	La bibliothèque d'ouverture.	35
5.1	Les types de bibliothèque.	35
5.2	Construction de la bibliothèque.	35
6	La gestion des fichiers.	37
6.1	Les fonctions associés :	37
6.1.1	La fonction de chargement :	37
6.1.2	La fonction de traduction d'une partie pgn :	38
6.1.3	La fonction d'enregistrement :	38
6.1.4	L'état d'une partie :	38

7	Résolution de problème.	39
7.1	Mat en x coups.	39
7.1.1	La mobilité adverse.	39
7.1.2	Echec et mat.	39

III	Le moteur. (ne pas prendre en compte pour l'instant)	41
------------	---	-----------

8	Quoi prendre en compte ?	43
8.1	Le nombre de cases jouables total.	43
8.2	La position de la pièce sur l'échiquier.	43
8.3	La prise d'une pièce.	43
8.4	La valeur de la prise.	43
8.5	La fourchette.	44
8.6	La tour menaçante.	44
8.7	La mise en échec du roi.	44

Première partie

Structures et règles du jeu.

Chapitre 1

Les informations nécessaires.

1.1 Les pièces.

Nous allons commencer par les pièces. Il existe 6 types de pièce par couleur :

1. le pion
2. la tour
3. le cavalier
4. le fou
5. la dame
6. le roi

Sur l'échiquier est disposé, au départ du jeu, par couleur, 8 pions, 2 tours, 2 cavaliers, 2 fous, 1 dame et 1 roi. Ensuite, il sera possible de changer des pions en d'autres types dans certaines phases du jeu.

1.2 Structure d'une pièce.

Il va donc nous falloir une structure capable de définir une pièce :
(déclaration dans `echec_var.h`)

```
struct _piece
{
    char type;
    int couleur;
    int ponderation;
    int place;
    char nbre_cases_jouables;
    bool deplacement_initial;
};
typedef _piece type_piece;
```

1.2.1 La variable “type” :

La variable entière “type” détermine si cette pièce est un pion, une tour, etc ... Il existe des macros qui reprennent les différentes valeurs de ces pièces :

```
#define pion      1
#define cavalier 2
#define tour     3
#define fou      4
#define dame     5
#define roi      6
```

Prenons un exemple :

Je veux déclarer un fou. Il me suffira de déclarer une variable de type “type_piece” en mettant la valeur type=4. Ainsi, lorsque j’aurai besoin de savoir quelle piece c’est, il me suffira de comparer. Le fait de déclarer une macro permet, après 2000 lignes de codes, de ne pas être trop perdu. Ce type d’encodage sera utilisé le plus souvent possible. Ca rallonge un peu mais ca permet une relecture aisée ainsi qu’une maintenance facilité.

1.2.2 La variable “couleur” :

Quoi dire de plus qu’elle aura pour valeur 0 ou 128. Comme pour les pièces, j’ai déclaré 4 macros pour rendre le listing plus lisible. Ainsi il existe les macros blanc, blanche, noir et noire qui existent pou respecter la langue et qui ont respectivement comme valeur 0 et 128 en fonction des couleurs.

1.2.3 La variable “ponderation” (Pas encore utilisée a ce stade du développement) :**1.2.4 La variable “place” :**

Elle indique la position de la pièce sur l’échiquier. L’échiquier est un tableau à 1 dimension. Cette variable correspondra donc à une case de ce tableau. Pour la description de ce tableau voir la section 1.5 page 12.

1.2.5 La variable “nbre_cases_jouables” :

Elle indique le nombre de cases jouables de la pièce à un instant donné (pourra être utilisée ultérieurement lors des évaluations des coups).

1.2.6 La variable “deplacement_initial” :

Cette variable booléenne indique si cette pièce n’a pas encore été déplacé. C’est intéressant pour le pion, le roi et la tour. Si le pion n’a pas encore été déplacé, il peut avancer de deux cases d’un coup. Pour le roi et la tour, c’est une des conditions pour autoriser le roque. Ce type d’information charge un peu les pièces mais facilite les recherches (on utilise un peu plus de mémoire mais on gagne en rapidité).

1.3 Les déclarations de macros.

(déclaration dans echech_var.h et echech_var.c)

```
/* déclaration des variables et macros du jeu */
#define pion      1
#define cavalier 2
```

```

#define tour      3
#define fou       4
#define dame     5
#define roi      6
#define noire    1
#define noir     1
#define blanche  0
#define blanc   0
#define vide     0
#define bord    -1
#define CROISSANT 0
#define NBRE_COUPS_ADVERSE 2

```

On peut remarquer 4 déclarations supplémentaires non encore détaillées :

- vide : valeur d’une case sur l’échiquier non occupée par une pièce.
- bord : valeur d’une case indiquant que l’on est sorti de l’échiquier (voir section 1.5.2 page suivante).
- CROISSANT,NBRE_COUPS_ADVERSE : ces définitions seront utilisées pour la fonction de tri d’une liste contenant les coups jouables.

1.4 Le tableau “*table_piece [33]*” :

Il existe un tableau regroupant toutes les pièces du jeu. Ce tableau comporte donc 32 pièces classées dans un ordre bien précis de façon à pouvoir retrouver une pièce sans chercher (toujours penser au gain de temps).

(déclaration dans `echec_var.h`)

```
type_piece table_piece [33];
```

L’initialisation s’effectue dans une fonction appelée `echec_init(void)`. Cette fonction regroupe toutes les initialisations nécessaires au jeu. Elle sera donc exécutée en premier. elle utilise une petite fonction en complément, `initialisation_piece(char piece, char couleur, int position)` à laquelle on transmet le type de la piece, la couleur de la piece et sa position sur l’échiquier. Elle renvoie une variable du type “type_piece” entièrement initialisée.

Dans le tableau `table_piece[]`, les pièces sont classées comme suit :

- De 1 à 8 : tous les pions blancs.
- En 9 et 10 : les tours blanches.
- En 11 et 12 : les cavaliers blancs.
- En 13 et 14 : les fous blancs.
- En 15 : la dame blanche.
- En 16 : le roi blanc.
- De 17 à 24 : tous les pions noirs.
- En 25 et 26 : les tours noires.
- En 27 et 28 : les cavaliers noirs.
- En 29 et 30 : les fous noirs.
- En 31 : la dame noire.
- En 32 : le roi noir.

Remarque : ce tableau ne sera jamais trié. Si une piece vient à disparaître du jeu, la valeur de son type sera mise à zéro. Ainsi, pour savoir s’il y a une pièce, il suffira de regarder la valeur de “type”. L’intérêt majeur est de pouvoir trouver les rois sans aucune recherche : le roi blanc en position 16 et le roi noir en position 32. Nous verrons que de nombreux tests sont effectués sur les rois. Ainsi, nous gagnerons un temps non négligeable.

1.5 L'échiquier.

L'échiquier est un tableau composé de 144 cases appelé *echiquier*]].

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80	81	82	83
84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107
108	109	110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129	130	131
132	133	134	135	136	137	138	139	140	141	142	143

1.5.1 Pourquoi 144 cases ?

Un échiquier est normalement constitué de 64 cases (8x8). Il y a deux méthodes principales pour contrôler les déplacements des pièces : vectorielle ou numérique.

La méthode vectorielle utilise 2 variables pour désigner un déplacement particulier d'une pièce. Par exemple pour le pion, son déplacement est (0,1). 0 déplacement vers la droite, 1 déplacement vers le haut.

La méthode numérique utilise une seule variable. On part de la case occupée et on soustrait ou on additionne un nombre pour arriver à la case jouable suivante de la pièce. J'ai décidé d'utiliser la deuxième méthode de façon purement arbitraire. Ainsi, pour chaque type de pièce est défini une rosace de déplacement.

La pièce ayant le déplacement le plus particulier est le cavalier. A cause de son déplacement, si un cavalier est situé au bord de l'échiquier, son déplacement le fait regarder deux cases en dehors de l'échiquier, quelque soit le bord concerné. Ainsi, ce n'est plus un tableau de 8x8 mais un tableau de 12x12, ce qui donne un tableau de 144 cases.

1.5.2 L'initialisation de l'échiquier.

Nous verrons que chaque case de l'échiquier est en fait une structure. Dans cette structure on trouve une variable appelée `type_piece`. Nous utiliserons cette variable pour savoir où nous nous situons sur l'échiquier et qu'elle est l'occupation de la case concernée.

En partant de ce principe, nous allons initialiser les cases correspondant aux 2 lignes supérieures, aux 2 lignes inférieures, aux 2 colonnes de gauches ainsi qu'aux 2 colonnes de droites avec la variable `type_piece=-1`. Si vous vous reportez à la section 1.3 page 10 , vous verrez qu'il existe une macro "bord" égale à -1. Donc la variable `type_piece=bord`.

Toutes les autres cases seront initialisées avec `type_piece=vide` (macro `vide=0`).

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	0	0	0	0	0	0	0	0	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

1.6 Les différentes rosaces de déplacement.

1.6.1 Les tableaux "rosace[][]" et "corres_piece[]".

Chaque type de pièce se déplace d'une manière propre. Chaque type de déplacement s'appelle une rosace. Tous ces rosaces sont regroupées dans un tableau à deux dimensions "int rosace[7][10]" déclaré dans echecs_var.h.

Le classement des pièces dans le tableau rosace[][] est le suivant :

- 0 : pion blanc
- 1 : pion noir
- 2 : cavalier
- 3 : roi
- 4 : tour
- 5 : fou
- 6 : dame

Dans le tableau "corres_piece" on retrouve la valeur de la pièce en fonction de sa position dans le tableau "rosace[][]" :

- 0 : 1 (PION)
- 1 : 1 (PION)
- 2 : 2 (CAVALIER)
- 3 : 6 (ROI)
- 4 : 3 (TOUR)
- 5 : 4 (FOU)
- 6 : 5 (DAME)

Ainsi, ces deux tableaux permettent de réduire le code (boucles imbriquées).

1.6.2 Les déplacements du pion.

Le pion peut, selon la couleur, soit monter, soit descendre, soit prendre sur les côtés, et, cas particulier, prendre en passant.

Mis à part la prise en passant qui est un cas particulier qui fait appel au coup précédent de l'adversaire, il est facile de trouver la rosace pour chaque couleur.

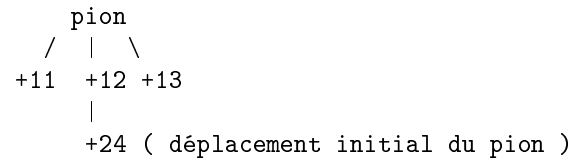
Pour le pion blanc :

```

-24 ( déplacement initial du pion )
  |
-13 -12 -11
  \ | /
  pion

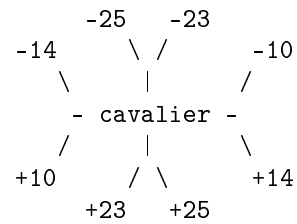
```

Pour le pion noir :



1.6.3 Les déplacements du cavalier.

Le cavalier à un déplacement un peu particulier. Il effectue d'abord un déplacement linéaire perpendiculaire à l'un des bord de l'échiquier d'une case puis un déplacement d'une case en oblique. La première case peut être occupée par une pièce amie ou non. La rosace est la même quelque soit la couleur.



1.6.4 Les déplacements de la tour.

La tour à un déplacement linéaire perpendiculaire aux bords de l'échiquier. La rosace est la même quelque soit la couleur.

```

      -12
      |
-1 - tour - +1
      |
      +12

```

A la différence du pion, cette rosace devra être testée en boucle jusqu'à arriver au bord de l'échiquier, à la rencontre d'une pièce adverse ou amie.

1.6.5 Les déplacements du fou.

Comme la tour, le fou à un déplacement linéaire mais en diagonal cette fois. La rosace est identique quelque soit la couleur.

```

-13      -11
  \      /
   fou
  /      \
+11      +13

```

Comme pour la tour, cette rosace devra être testée en boucle jusqu'à arriver au bord de l'échiquier, à la rencontre d'une pièce adverse ou amie.

1.6.6 Les déplacements de la dame.

La dame regroupe les déplacements de la tour et du fou combiné.

```

-13  -12  -11
  \   |   /
-1 - dame - +1
  /   |   \
+11  +12  +13

```

Bien entendu, comme la tour et le fou, cette rosace devra être testée en boucle jusqu'à arriver au bord de l'échiquier, à la rencontre d'une pièce adverse ou amie.

1.6.7 Les déplacements du roi.

Le roi à la même rosace que la dame, à la différence près qu'il ne peut se déplacer que d'une seule case à la fois. Une exception cependant existe pour les petits et grands roques.

```

      -13  -12  -11
          \ | /
( grand roque ) -2 -1 - roi - +1 - +2 ( petit roque )
          / | \
      +11  +12  +13

```

1.7 Structure d'une case de l'échiquier.

Chaque case est une structure regroupant des informations sur son occupation ou non, ainsi que des informations doublons sur les pièces positionnées. Toutes ces

informations permettent d'accélérer les calculs en évitant des recherches, et donc des boucles, inutiles.

Cette structure est déclarée dans le fichier `echecs_var.h`.

```
struct _echiquier
{
    int ponderation;
    short int type_piece;
    short int couleur_piece;
    short int couleur_case;
    int pos_table_piece;
    short int x;
    short int y;
} echiquier [143];
```

1.7.1 La variable *ponderation* (pas encore utilisée à ce stade du développement).

1.7.2 La variable *type_piece*.

Elle donne la position de la pièce sur l'échiquier. Si elle vaut 0 (macro vide), alors il n'y a pas de pièce sur cette case. Si elle vaut -1 (macro bord), alors on est sorti de l'échiquier.

1.7.3 La variable *couleur_piece*.

Elle donne la couleur de la pièce (comme on s'en serait douté).

1.7.4 La variable *couleur_case*.

Elle est utilisée pour l'affichage. Ce n'est pas très correct d'initialiser une variable qui ne sert pas pour le moteur du jeu mais ça évite une double déclaration. Cette variable pourra disparaître dans l'avenir.

1.7.5 La variable *pos_table_piece*.

Elle indique la position de la pièce dans le tableau `table_piece[]`. Information redondante mais évite un tas de boucles de recherche. => gain de temps.

1.7.6 Les variables *x* et *y*.

Elles correspondent aux coordonnées de la case sous forme algébrique. Elles ont un rôle de commodité et évite ainsi de nombreuses boucles de recherche.

Chapitre 2

Les outils nécessaires.

Lorsque les coups jouables vont être calculés pour les 2 couleurs à chaque position de la partie, il sera nécessaire de les sauvegarder. Soit on utilise un tableau, mais cette méthode oblige à déclarer un tableau très grand alors que la plupart du temps le nombre de coups jouables sera limité, soit on utilise une liste doublement chaînée. Cette deuxième méthode a l'avantage d'adapter sa taille, donc l'allocation mémoire, en fonction du nombre de coups jouables. Ce sera donc celle-ci qui sera utilisée. La librairie glib incluse dans le paquetage GTK intègre la gestion complète de ce genre de liste. Elle sera donc mise à contribution.

2.1 Gestion d'une liste chaînée.

Voici un petit résumé des fonctions de la glib qui gèrent une liste doublement chaînée.

2.1.1 La déclaration d'une liste.

Pour déclarer une liste, il suffit d'écrire

```
GList *liste=NULL;
```

Une fois fait, on pourra ajouter un élément à cette nouvelle liste.

2.1.2 Ajouter un élément à la fin de la liste.

Pour ce faire, il y a la fonction

```
GList *g_list_append(GList *list, gpointer *donnee);
```

Elle insère un élément à la fin de la liste.

Pour pouvoir l'utiliser, il faut d'abord avoir la donnée à insérer. Le type *gpointer* est un pointeur sans type. Donc vous pouvez attacher n'importe quel type de donnée à cette liste.

Exemple :

```
1 : GList *liste=NULL ;
2 : char *donnee ;
3 : donnee=(char*)malloc(6) ;
4 : strcpy(donnee,"coucou") ;
5 : liste=glist_append(liste,donnee) ;
```

Comme vous avez pu le remarquer, la donnée à eu droit à une allocation mémoire en ligne 3. Ce petit programme n'est pas très juste puisque je ne contrôle pas si l'allocation mémoire à été réalisée, mais ce n'est pas le but de l'exemple. Maintenant, la liste "liste" possède un élément.

Cette opération devra s'effectuer à chaque fois que l'on voudra ajouter un élément. Une fois tous les éléments insérés, l'intérêt premier est de pouvoir accéder aux différents éléments qui la compose.

2.1.3 Se placer en début de liste.

Pour ce faire on utilisera la fonction

```
glist *g_list_first(GList *list);
```

2.1.4 Se placer en fin de liste.

La fonction suivante nous y emmènera :

```
glist *g_list_last(GList *list);
```

2.1.5 Passer à l'élément suivant de la liste.

Son prototype est

```
GList *g_list_next(GList *list);
```

2.1.6 Passer à l'élément précédent de la liste.

Comme pour passer à l'élément suivant, il existe une fonction qui le permet :

```
GList *g_list_previous(GList *list);
```

2.1.7 Enlever un élément de la liste.

Il est intéressant de pouvoir enlever un élément de la liste. Pour se faire on a le prototype suivant

```
GList *g_list_remove(GList *list, gpointer *donnee);
```

Cette fonction "enlève" l'élément actuellement pointé de la liste mais ne libère pas la mémoire éventuellement allouée. Il faudra donc le faire de façon explicite.

exemple :

```
liste_tmp=g_list_next(liste);
free(((char *) (liste->data)));
liste=g_list_remove(liste,((char *) (liste->data))->donnee);
liste=liste_tmp;
```

Dans cet exemple, on donne à liste_tmp la valeur du pointeur de l'élément suivant. Ensuite, on libère la mémoire de la donnée que l'on va supprimer. L'utilisation de la fonction g_list_remove(); ne fait que changer les pointeurs des données précédentes et suivantes. La dernière ligne fait pointer la liste liste sur l'ancien élément suivant qui est devenu l'élément actuel.

2.1.8 Déterminer le nombre d'élément de la liste.

Il est souvent nécessaire de savoir combien d'élément compose une liste. Voila la fonction faite pour nous :-)

```
int g_list_length(GList *list);
```

2.1.9 Déterminer le nombre d'élément en fonction d'une couleur.

Cette fonction est plus spécifique au jeu qu'à une liste chaînée standard. Elle ne fait pas partie de la glib. Elle est déclarée dans le fichier source `echecs_liste_chaine.c`. Son prototype est

```
int longueur_par_couleur_dans_liste(GList *liste_coups_jouables, int couleur);
```

Vous lui transmettez la liste à compter et la couleur "noire" ou "blanche" en utilisant les macros définies (0 pour blanc et 128 pour noir). Elle vous renvoie le nombre adéquat.

2.1.10 Trier la liste.

Il sera nécessaire de pouvoir trier et reclasser la liste en fonction de différents paramètres. Ainsi, il existe la fonction

```
GList *trier_liste(GList *liste, int variable_triee, int ordre);
```

Il suffit de transmettre à cette fonction la liste à trier *liste*, de lui dire quelle variable on désire trier. Pour finir on lui dit dans quel ordre on désire que cela soit fait en utilisant la macro CROISSANT. Elle nous renvoie alors le premier pointeur de la liste remise en ordre.

Remarque : cette fonction fait appelle à une fonction annexe appelée *gint compare(gconstpointer a, gconstpointer b)*; Elle fait partie du fonctionnement *trier_liste()*. Actuellement, cette fonction n'accepte que les macros NBRE_COUPS_ADVERSE pour la variable utilisée comme tri et CROISSANT pour l'ordre. Ses possibilités pourront, selon les besoins futurs, s'accroître.

2.1.11 Vider la liste.

Pour vider complètement une liste, il existe la fonction

```
void g_list_free(GList *list);
```

Cette fonction opère comme la *g_list_remove()* mais pour tous les éléments de la liste. Donc elle ne vide pas la mémoire éventuellement allouée pour les données que l'on a mis dans cette liste. Ce sera à nous de le faire.

J'ai donc écrit une fonction *GList *vider_liste(GList *liste_a_supp)*; qui supprime effectivement la liste. Elle se trouve elle aussi dans `echecs_liste_chaine.c`.

A la sortie de cette fonction, le pointeur de la liste transmis en argument vaut NULL. Dans le cas contraire, la fonction a eu problème (utile pour déboguer le code).

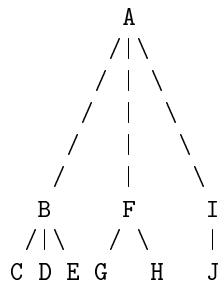
2.1.12 Conclusion.

Il existe d'autres fonctions de la glib permettant d'affiner la gestion d'une liste chaînée. Je vous encourage à consulter la documentation sur le site www.gtk.org pour de plus amples détails.

2.2 Gestion d'un arbre k.aire.

2.2.1 Qu'est-ce qu'un arbre k.aire ?

Le principe de base est une hiérarchie Père-fils. Reprenons le principe de la liste chaînée. Chaque élément pointe sur un élément suivant. Nous appellerons ici l'élément suivant fils et l'élément actuel Père. Un arbre, à la différence d'une liste, est que chaque père peut avoir plus d'un fils. L'arbre le plus connu est l'arbre binaire. Chaque père possède 2 fils. Un arbre k.aire est un arbre qui, pour un père donné, peut avoir k fils. Le premier élément de l'arbre, ici A, s'appelle la racine et chaque père porte le nom de noeud.



Dans cet arbre, le père A a pour fils B, F et I. Si l'on se place sur B, alors B est le père de C, D et E.

2.2.2 Comment s'effectue la lecture d'un tel arbre ?

La lecture s'effectue toujours de gauche à droite. Ainsi, en partant de la racine, on lira A, puis B, puis C. Arrivé à C, on s'aperçoit que ce père n'a pas de fils. Nous allons donc remonter d'un niveau, soit le père B, et prendre le fils suivant, ici D. Lorsque nous arrivons à D, il n'y a plus de fils pour D. On remonte d'un niveau et on passe au fils suivant, soit E. Arrivé à E, E n'a pas de fils. On remonte à B : B n'a plus de fils. On remonte encore d'un cran et on prend le fils suivant de A etc...

Si nous reprenons la schématique ci-dessus, la navigation donnera : A->B->C->D->E->F->G->H->I->J.

2.2.3 Un arbre dynamique.

Il sera nécessaire de pouvoir faire évoluer cet arbre de façon dynamique. C'est à dire pouvoir agrandir une branche pendant le court du jeu. Si par exemple J a, à un moment donné, 3 nouveaux fils, il faudra pouvoir les rajouter.

2.2.4 L'intérêt d'un arbre k.aire.

Nous utiliserons ce type d'arbre pour la bibliothèque d'ouverture. Chaque noeud portera donc des données liées à chaque coup de la bibliothèque. Mais nous en parlerons au chapitre 5 page 35.

2.2.5 Les fonctions de gestion d'un arbre.

Pour qu'un arbre soit utilisable, il lui faut des fonctions permettant de le traiter en profondeur. Il y aura donc une fonction pour :

- créer un arbre.
- ajouter un noeud à la position courante de l'arbre.
- revenir à la racine de l'arbre.
- passer au 1er fils du noeud courant.
- remonter au noeud précédent de l'arbre.
- passer au fils suivant du noeud courant.

2.2.6 Utilisation des fonctions.

Toutes les fonctions nécessaires se trouvent dans `echecs_arbre_kaire.c`. Tout d'abord, nous avons besoin d'une structure regroupant les informations nécessaires à la gestion d'un arbre. Cette structure s'appelle `_noeud` et est déclarée dans `echecs_var.h` :

```

struct _noeud
{
    gchar *nom;           // Nom de l'ouverture
    gchar *coup;         // Coup à jouer en format texte
    short int gagnant;   // Indique la couleur gagnante pour ce coup.
    _noeud *parent;     // Pointeur sur le noeud parents du noeud courant
    _noeud *enfant;     // Pointeur sur le 1er noeud enfant du noeud courant
    _noeud *fils_suivant; // Pointeur sur le fils suivant du même noeud parent
};
extern _noeud *arbre_biblio;

```

2.2.6.1 Détails de la structure `_noeud`.

- Le champ "nom" correspond au nom du fichier duquel on a extrait le coup.
- Le champ "coup" correspond au coup, en format texte.
- Le champ "gagnant" permet de ne pas regarder les coups qui n'apporteraient pas un gain pour la couleur jouée par l'ordinateur.
- Le champ "parent" pointe sur le noeud père.
- Le champ "enfant" pointe sur le premier enfant du noeud actuel s'il existe (=NULL dans le cas contraire).
- Le champ "fils_suivant" pointe sur le "frère" du noeud actuel s'il existe (=NULL dans le cas contraire).

On déclare, en globale, un pointeur "arbre_biblio" qui sera la bibliothèque d'ouverture utilisée dans le jeu.

2.2.6.2 Création de la bibliothèque d'ouverture.

Pour se faire on utilisera la fonction

```
_noeud *creer_arbre(_noeud *arbre);
```

On déclare un pointeur de type `_noeud`, ici `arbre_biblio` (déclaré dans `echecs_var.h`). On alloue un emplacement mémoire pour ce pointeur puis on utilise la fonction de création. Voici un exemple :

```

_noeud *arbre_biblio=NULL;
arbre_biblio=(_noeud*)malloc(sizeof(_noeud));
arbre_biblio=creer_arbre(arbre_biblio);

```

A partir de maintenant on dispose d'un pointeur initialisé qui correspond en fait au pointeur racine d'une bibliothèque.

2.2.6.3 Ajout d'un noeud.

La fonction qui permet d'ajouter un noeud à la position courante à comme prototype

```
_noeud *ajouter_enfant_position_courante_arbre(_noeud *arbre, char *nom, char *coup);
```

Si on reprend l'exemple décrit à la section 2.2.1 page 20, le pointeur `arbre_biblio` créé ci-dessus correspond à A. En utilisant la fonction d'ajout d'un noeud, on va en fait ajouter le noeud B. En admettant que le noeud B existe déjà, la fonction créera le noeud C. Voici un exemple :

```
arbre_biblio=racine_arbre(arbre_biblio);
arbre_biblio=ajouter_enfant_position_courante_arbre(arbre_biblio, nom,coup);
```

La première ligne permet de s'assurer que l'on est bien à la racine de l'arbre. La deuxième ligne ajoute un fils à la racine. Les pointeurs `nom` et `coup` sont de type `gchar`. Ainsi, il suffit de se placer sur le noeud auquel on veut ajouter un fils puis on utilise la fonction d'ajout.

2.2.6.4 Se replacer à la racine de l'arbre.

Il est toujours intéressant de pouvoir se replacer à la racine ne serait-ce que pour être sûr d'où on commence à traiter l'arbre.

Le prototype de la fonction est

```
_noeud *racine_arbre(_noeud *arbre);
```

Vous en avez un exemple d'utilisation à la section précédente. Une fois à la racine, il faut des fonctions, hormis l'ajout de noeud, qui permettent de naviguer dans cet arbre.

2.2.6.5 Pointer sur le premier enfant du noeud courant.

Lorsque l'on veut aller sur l'enfant le plus à gauche du noeud actuellement pointé, on utilise la fonction

```
_noeud *noeud_enfant_suivant_arbre(_noeud *arbre);
```

En retour, la valeur de la fonction est le pointeur du premier enfant ou NULL s'il n'existe pas d'enfant.

2.2.6.6 Pointer sur l'enfant suivant du noeud courant.

Si on veut remonter au père du noeud actuellement pointé, on utilisera plutôt

```
_noeud *noeud_parent_precedent_arbre(_noeud *arbre);
```

Cette fonction nous renverra le pointeur du père. Si on n'est à la racine de l'arbre et que l'on utilise cette fonction, on restera à la racine. (je vois pas d'ailleurs où on pourrait aller!)

2.2.6.7 Pointer sur l'enfant suivant du noeud courant.

Si on veut aller sur l'enfant suivant, c'est à dire le "frère" du noeud actuellement pointé, on utilisera plutôt

```
_noeud *fils_suivant_arbre(_noeud *arbre);
```

Cette fonction nous renverra le pointeur du "frère" suivant ou NULL s'il n'existe pas.

Chapitre 3

Les règles du jeu.

Maintenant que nous avons une structure représentant une pièce et un tableau regroupant toutes les pièces du jeu, il va falloir que le programme sache comment les déplacer. C'est à dire lui apprendre les règles du jeu.

Une liste a été créé dans laquelle on retrouvera tous les coups jouables de chaque couleur. Chaque élément de cette liste est une structure qui regroupe quelques informations au sujet du coup jouable.

3.1 La liste des coups jouables.

Pour le fonctionnement général du jeu, une liste doublement chaînée est déclarée en globale. Elle est aussi utilisée par l'interface graphique. Pour des fonctions plus avancées telles que la résolution de problème, on pourra utiliser des listes locales. Nous allons voir que les fonctions de traitement et de création vont nous le permettre.

La liste principale est déclarée dans le fichier d'entête `echecs_var.h`.

```
GList *liste_coups_jouables;
```

C'est une liste doublement chaînée (voir section 2.1 page 17).

3.1.1 Structure d'un coup jouable.

Chaque élément d'une liste est une structure qui s'appelle `_coups_jouables`;

```
struct _coups_jouables
{
    int pos_piece_tab;
    int couleur;
    short int coup;
    short int valeur_coup;
    short int nbre_coups_adverse;
    bool case_occupee;
    bool coup_ami;
};
```

3.1.2 La variable `pos_piece_tab`.

Elle représente la position de la piece dans le tableau `table_piece[]`.

3.1.3 La variable *coup*.

Elle correspond au coup jouable (position sur l'échiquier).

3.1.4 La variable *couleur*.

Elle permet, en toute évidence, de faire le tri lorsque cela est nécessaire.

3.1.5 La variable *valeur_coup*.

Cette donnée sera utilisée par la suite lors de l'évaluation de chaque coup. Elle nous permettra de classer la liste en fonction de cette valeur.

3.1.6 La variable *nbre_coups_adverse*.

Cette variable est utilisée lors de la recherche d'un mat en x coups.

3.1.7 Le drapeau *case_occupee*.

Il est utilisé pour signifier que ce coup est occupée par une pièce.

3.1.8 Le drapeau *coup_ami*.

Il indique ,lorsque le drapeau *case_occupee* est à vrai, si la pièce qui occupe le coup est une pièce amie ou non.

Remarques : Les drapeaux *case_occupee* et *coup_ami* sont utilisés pour la détermination des coups réellement jouables.

3.2 La détermination des coups jouables.

Plusieurs fonctions travaillent la liste des coups jouables pour y insérer ou y supprimer des coups jouables.

La plus importante d'entre elles est la fonction

*GList *cases_jouables(GList *liste_coups_jouables, type_piece pieces, int pos_table).*

On lui transmet la liste des coups jouables sur laquelle on veut ajouter des coups jouables, une structure représentant la pièce à calculer, et la position de cette pièce dans le tableau *table_piece[]*. En retour, elle renvoie la liste des coups jouables mise à jour. Elle prends en compte toutes les possibilités de déplacement pour chaque type de pièce avec la prise en passant,les roques et la promotion.

Cette fonction fait elle-même appelle à deux fonctions annexes :

- *GList *ajout_coup_dans_liste(GList *liste_coups_jouables, int couleur, int position_piece_tab, int coup, bool case_occupee, bool coup_ami)* ; Elle permet d'initialiser une structure complète d'un coup jouable en fonction des paramètres transmis et d'ajouter cette nouvelle structure à la liste des coups jouables.
- *bool case_attaquee_par_couleur(GList *liste_coups_jouables, int case_a_tester, int couleur)* ; Elle renvoie vrai si la case que l'on teste est occupée par une pièce de la couleur *couleur*. Cette fonction utilise la liste des coups jouables pour son test. Elle est utilisée pour le calcul des coups jouables des rois.

Enfin, il existe une troisième fonction : `GList *calcul_cases_jouables(GList *liste_coups_jouables);`. Elle fait calculer tous les coups jouables de toutes les pièces. Elle est lancée à chaque fois qu'un coup est joué en partenariat avec la suppression complète de la liste.

A la sortie, on dispose donc d'une liste comportant tous les coups jouables des deux couleurs. Cependant, il va peut-être falloir en supprimer certains. Il existe deux règles aux échecs qui interdisent de déplacer une pièce si ce déplacement met en échec son roi et que l'on ne peut déplacer une pièce si son roi est en échec que pour le couvrir. Si une de ces deux règles ne peut être respectée, on dit alors que la pièce est clouée. enfin il est interdit aux rois de se trouver dans des cases adjacentes.

3.3 Les coups cloués.

Rappelons donc les règles. Il est interdit de mettre en échec son roi par le déplacement d'une pièce, y compris par le propre déplacement du roi bien sûr, et il n'est pas non plus autorisé de jouer un coup qui ne couvrirait pas son roi lorsque celui-ci est en échec. Il est aussi interdit de placer un roi sur une case adjacente au roi adverse. Pour ce faire il existe une fonction appelée

```
GList *elimination_des_coups_cloues(GList *liste_coups_jouables, int couleur);
```

Cette fonction fait appel de façon exhaustive à une autre fonction qui teste la mise en échec d'un roi.

3.3.1 Le test du roi en échec.

La fonction qui teste si un roi est en échec s'appelle

```
gboolean le_roi_est_en_echec(int couleur_roi);
```

et se trouve dans le fichier `echecs_mis_en_echec.c`. On lui transmet la couleur du roi que l'on veut tester et nous renvoie vrai si le roi considéré est en échec.

Il existe plusieurs méthodes permettant d'effectuer ce test. La plus simple est de scruter la liste des coups jouables adverse pour voir si un coup ne correspondrait pas à la case occupée par le roi. Cette méthode implique qu'à chaque test, les coups de l'adversaire doivent préalablement être calculés. Une autre méthode consiste à partir du roi à tester, de lui appliquer la rosace des autres pièces et de vérifier si la pièce adverse de la rosace appliquée ne s'y trouve pas. Ça complique un peu la programmation mais évite le recalcul permanent des coups jouables adverses => un gain de temps non négligeable. Cette dernière méthode sera donc celle utilisée.

Exemple : on applique la rosace du cavalier au roi blanc et on regarde s'il n'y a pas un cavalier adverse aux cases trouvées.



3.3.2 La fonction d'élimination des coups cloués.

Maintenant que nous avons une fonction de test de mis en echec du roi performante, nous pouvons nous concentrer sur la fonction d'élimination des coups cloués. Prenons un exemple en partant d'un déplacement d'une pièce blanche :

1. On déplace une pièce blanche.
2. On simule chaque coup jouable des noirs. A chaque fois, on vérifie si le roi noir est en echec. Si oui, on supprime le coup simulé. A la fin, il ne restera que les coups ne mettant pas en echec ou couvrant un echec effectif. Avec ce système, on peut déjà voir qu'il sera facile de savoir si on est mat ou pat en regardant le nombre de coups jouables final.
3. Aux noirs de jouer.

Le point 2 est la fonction `GList *elimination_des_coups_cloues(GList *liste_coups_jouables, int couleur);`. Voici son déroulé :

1. On simule le nième coup jouable (en évitant ceux avec le drapeau `coup_ami=true`) de la couleur considérée.
2. On teste si le roi est en echec.
3. Si oui, on supprime le coup jouable de la liste considérée.
4. On passe au coup jouable suivant jusqu'à la fin de la liste.

Chaque simulation signifie que l'on déplace le coup sur l'échiquier. Ce qui veut dire qu'avant de passer au coup suivant, il faut remettre le jeu comme avant. On replace le coup simulé précédent à sa place. Comme la fonction de test `bool le_roi_est_en_echec(int couleur_roi);` n'utilise pas la liste des coups jouables mais seulement le tableau `echiquier[]`; il n'est pas nécessaire de recalculer les coups jouables après chaque déplacement. On voit maintenant l'intérêt de la méthode utilisée pour le test de mis en échec. La couleur transmise permet de n'éliminer que les coups cloués d'une couleur.

Chapitre 4

Le déplacement d'une pièce.

Après plusieurs recherches comment optimiser le calcul des coups jouables, il s'avère, pour l'instant que la méthode la plus simple et la meilleure. Donc, lors d'un déplacement d'une pièce, la liste des coups jouables, sera entièrement effacée pour être entièrement recalculée. C'est pas très fin, mais pour l'instant, c'est efficace.

Soit la position de l'échiquier suivante :



C'est aux blancs à jouer. Les blancs jouent Bb5.



4.1 Séquencement d'un déplacement :

1. On supprime le fou en f1 du tableau *echiquier[]*;
2. On place le fou en b5 dans le tableau *echiquier[]*;
3. On change la position du fou dans le tableau *table_piece[]*;
4. On supprime la liste des coups jouables. Les coups jouables actuellement contenus dans la liste sont les anciens coups qui correspondent à l'emplacement du fou en f1. Pour se faire on utilise la fonction *GList *vider_liste(GList *liste_a_supp)* ; qui se trouve dans le fichier *echecs_liste_chaine.c*.
5. On calcule une nouvelle liste avec la fonction *GList *calcul_cases_jouables(GList *liste_coups_jouables)* ; qui se trouve dans le fichier *echecs_coups_jouables.c*.

4.2 La fonction de déplacement d'une pièce.

Cette fonction s'appelle

*int deplacer_piece(GList *liste_coups_jouables, type_piece piece_a_jouer, int coordonnee_echiquier) ;*

Elle se trouve dans le fichier *echecs_deplacer_piece.c*. On lui transmet la structure de la pièce à déplacer ainsi que la case d'arrivée. Elle gère tous les actions, le simple déplacement, la prise, la prise en passant, le petit et grand roque. Elle détermine l'action à faire en fonction du type de la pièce et de la case d'arrivée. Elle effectue en faite tous les points repris dans la section précédente.

Pour finir, elle renvoie un entier qui correspond à l'action effectuée :

- 0 : déplacement.
- 1 : prise.
- 2 : echec.

- 3 : petit roque.
- 4 : grand roque.
- 5 : prise en passant.
- 6 : prise par une pièce autre qu'un pion pouvant prêter à confusion (2 cavaliers par exemple)
- 7 : déplacement d'une pièce autre qu'un pion pouvant prêter à confusion.
- 8 : promotion simple.
- 9 : promotion avec une prise.

Cela permet de gérer un affichage du résultat dans le format que vous voulez.

Cette fonction fait appelle à une autre fonction annexe contenue elle aussi dans le fichier `echecs_deplacer_piece.c` :

- *gboolean recherche_coup_dans_liste(GList *liste_coups_jouables, int couleur, int position_piece, int coup)* ; elle renvoie vrai si la pièce à la position *position_piece* dans le tableau *table_piece* ; à un coup jouable.

Deuxième partie

Les fonctions annexes

Chapitre 5

La bibliothèque d'ouverture.

Une partie d'échec génère une quantité énorme de situations. Cependant, pour les 10 premiers coups (nombre arbitraire), on peut considérer que la quantité de combinaison au départ est limitée. Ces coups de départ sont regroupés dans une ouverture. Il existe des dizaines d'ouvertures. Chaque ouverture correspond aux meilleurs coups pour les blancs ou pour les noirs pour obtenir une position dominante ou pour la meilleure défense. Les grands maîtres utilisent ces ouvertures. Ils les connaissent toutes et les modifient selon les situations. Ces modifications sont alors appelées variantes.

Le but d'une bibliothèque d'ouverture est de permettre à l'ordinateur de jouer les meilleurs x premiers coups sans les calculer en piochant dans cette base de donnée.

5.1 Les types de bibliothèque.

Il existe deux types de bibliothèque :

1. séquentielle.
2. positionnelle.

Pour la bibliothèque séquentielle, on parcourt l'arbre tant que l'on trouve un coup en réponse au coup adverse. Lorsqu'il n'existe pas de coup correspondant au coup adverse, on sort de la bibliothèque et les coups sont alors calculés.

Pour la bibliothèque positionnelle, à chaque coup de l'adversaire on scrute toute la base de donnée pour trouver la position échiquienne actuelle. Si elle existe, on joue le coup correspondant. Dans le cas contraire, on calcule le coup. L'intérêt de ce fonctionnement, c'est que la bibliothèque peut être utilisée tout le long de la partie.

Pour des raisons de simplicité, j'ai opté pour la première solution. Le fonctionnement de l'arbre k.aire (voir section 2.2.1 page 20) utilise donc le principe d'une bibliothèque d'ouverture séquentielle.

Je n'ai pas la prétention de vous fournir une bibliothèque d'ouverture. Donc il va falloir la construire.

5.2 Construction de la bibliothèque.

Le but est de pouvoir compenser le défaut qu'elle a par rapport à une bibliothèque positionnelle. Pour ce faire, il faut que la bibliothèque soit la plus grosse possible. Comment faire simple et efficace? Tout simplement en utilisant des parties d'échec déjà jouées.

La bibliothèque se construit en utilisant les parties enregistrées au format pgn (voir chapitre 6 page suivante) dans le répertoire *bibliotheque*. Ainsi, à chaque fois que vous ajouterez une partie dans ce répertoire, la bibliothèque augmentera.

Il existe une fonction dans le fichier *echecs_chgt_biblio.c* pour construire une bibliothèque :

```
void chargement_bibliotheque();
```

Elle même fait appelle à une sous-routine *void creation_bibliotheque(gchar *nom-fichier);* pour son fonctionnement.

Chapitre 6

La gestion des fichiers.

Tout programme d'échec doit être capable de gérer une partie enregistrée. Il existe un format international de sauvegarde d'une partie d'échec. Ce format porte l'extension "pgn". En voici un exemple tiré du site <http://www.echecs.com> :

```
[Event "www.echecs.com"]
[Site "?"]
[Date "2003.02.28"]
[Round "?"]
[White "machin"]
[Black "truc"]
[ECO "A10"]
[Result "1/2-1/2"]
[PlyCount "46"]
1. c4 Nc6 2. Nc3 e5 3. e4 Bc5 4. g3 d6 5. Bg2 f5 6. Nge2 Nf6 7. exf5
Bxf5 8. Bxc6+ bxc6 9. d4 exd4 10. Nxd4 Bd7 11. Qe2+ Kf7 12. Be3 Re8 13.
0-0-0 Rb8 14. f3 Qc8 15. Qd3 Qa6 16. Rhe1 Bb4 17. Bd2 Bxc3 18. Bxc3 Qxa2
19. Rxe8 Rxe8 20. Nc2 Re6 21. Kd2 Qa6 22. c5 Qxd3+ 23. Kxd3 a6 1/2-1/2
```

L'idéal est donc de pouvoir traiter ce format.

6.1 Les fonctions associés :

- il existe 2 fonctions principales pour gérer le format pgn :
- *GList *chargement_fichier_pgn(GList *liste_partie, gchar *nom_fichier);*
- *void enregistrement_fichier_pgn(gchar *nom_fichier);*

On les trouvera dans le fichier *echecs_gestion_fichier_pgn.c*.

6.1.1 La fonction de chargement :

Pour l'utiliser, vous aurez besoin d'un pointeur GList initialisé à NULL et le nom du fichier complet avec son extension et son arborescence. En retour, on obtient une liste GList contenant tous les coups sous format texte et le nom du fichier. Voici la structure associée à la liste que vous pouvez retrouver dans *echecs_var.h* :

```
struct _coup
{
    char *nom ;
    char *coup ;
}
```

Le nom du fichier, intégré à chaque coup de la liste, sera utile pour la bibliothèque d'ouverture. Si vous transmettez un nom de fichier erroné, la fonction renvoie un pointeur NULL.

Un fois chargée, cette liste ne nous est pas très utile en l'état. Il existe donc une fonction plus globale qui s'occupe de tout.

6.1.2 La fonction de traduction d'une partie pgn :

Nous utiliserons la fonction

```
int incorporation_pgn_dans_jeu(gchar *nomfichier);
```

On lui donne le nom de fichier complet avec son extension et son arborescence. Elle s'occupe du chargement, en utilisant la fonction de chargement vu auparavant et de la traduction de ce fichier en partie utilisable dans le jeu.

Elle renvoie un entier nous permettant de connaître l'état du chargement :

- -1 : impossibilité de charger la partie.
- 0 : la partie à été entièrement intégrée.
- x : n° du coup injouable ou incompréhensible.

Cette fonction utilise pour son propre fonctionnement 3 sous-fonctions :

- *type_piece recherche_piece_a_jouer(gchar *coup, int coup_arrive);*
- *int recherche_coup_arrive(gchar *coup);*
- *type_piece recherche_coup_valide(GList *liste_coups_jouables, int couleur, int piece_jouee, int coup_arrive);*

6.1.3 La fonction d'enregistrement :

Elle permet d'enregistrer la partie en cours. On lui transmet le nom de fichier complet, avec son extension et son arborescence. Son prototype est

```
void enregistrement_fichier_pgn(gchar *nom_fichier);
```

Cette fonction ne contrôle pas si une partie est effectivement en cours. Si vous voulez l'utiliser, il vous faudra y adjoindre un test d'état de la partie.

6.1.4 L'état d'une partie :

Il existe, pour gérer l'état du jeu, deux booléens déclarés dans `echecs_var.h` :

- `bool partie_en_cours;`
- `bool partie_enregistree;`

`partie_en_cours` vaut TRUE dès qu'un coup à été joué.

`partie_enregistree` vaut TRUE si la partie est enregistrée.

Chapitre 7

Résolution de problème.

Le but de ce chapitre est d'expliquer les fonctions mise en oeuvre pour rechercher un mat en x coups mais aussi d'expliquer les principes de fonctionnement.

7.1 Mat en x coups.

Rechercher un mat en x coups revient à calculer tous les coups possibles, sur x niveaux, jusqu'à trouver, ou pas, la ou les bonnes combinaisons. On voit tout de suite qu'il va y avoir un nombre de calcul très important. Il serait bon de trouver des méthodes qui permettent de limiter ce nombre.

7.1.1 La mobilité adverse.

Il est admis (par statistique) que plus la mobilité d'une couleur augmente, plus la mobilité adverse diminue. Et plus la mobilité adverse diminue plus les chances de mat pour l'autre augmente.

En partant de ce constat il pourrait être intéressant de classer les coups de la couleur qui fait mat en fonction de la mobilité adverse. Ainsi, on commencerait à simuler les coups qui génèrent les plus petits coups adverses pour aller jusqu'au plus grand.

Bien sûr, selon la position de départ, cette méthode peut ne rien faire gagner du tout si la solution se trouve sur le dernier des coups jouables au niveau 1. Mais en règle générale, elle est valable. Elle sera donc utilisée : tous les coups jouables de la couleur qui doit faire mat seront triés en ordre croissant en fonction de la mobilité adverse pour tous les niveaux de recherche.

7.1.2 Echec et mat.

Lors d'un echec et mat, il se passe deux actions : "echec" et "mat". Il ne peut y avoir mat sans echec. Ainsi, les coups jouables de la couleur qui doit faire mat au niveau $-1/2$ qui ne font pas echec sont inutiles. On pourra ainsi supprimer une multitude de calculs inutiles.

Troisième partie

Le moteur. (ne pas prendre en compte
pour l'instant)

Chapitre 8

Quoi prendre en compte ?

- Le nombre de cases jouables total pour la couleur.
- La position sur l'échiquier de la pièce (au centre ou sur les côtés).
- Si c'est une prise.
- La valeur de la pièce prise.
- S'il est possible de faire une fourchette.
- Si une tour est sur la deuxième rangée adverse, changer sa pondération (entre 7 et 8). A voir ...
- Est-il possible de mettre le roi en echec ?

8.1 Le nombre de cases jouables total.

Cette donnée est déjà calculée par la fonction `calcul_coups_jouables()`. Le résultat se trouve dans la variable globale `nbre_total_cases_jouables_blanc` et `nbre_total_cases_jouables_noir`.

8.2 La position de la pièce sur l'échiquier.

Il va falloir pondérer chaque case de l'échiquier. Les quatre cases du centre sont les plus importantes. Nous partons donc de celles-ci pour décroître en forme de toile d'araignée jusqu'au bord de l'échiquier. Cette donnée pourra être rattachée à la structure `echiquier`. De cette manière elle sera facilement récupérable.

8.3 La prise d'une pièce.

Cette évaluation en comporte en réalité plusieurs. La prise d'une pièce est toujours intéressante. Cependant il faut prendre en compte le risque d'être à son tour pris. Dans un tel cas, il faut regarder si cet échange est équitable et si, dans le futur, il nous est profitable.

8.4 La valeur de la prise.

Cette notion est englobée dans la prise d'une pièce. Il faut qu'en même penser au cas où il est possible de prendre deux pièces adverses sans être pris par la suite. Il y a un choix à faire. La valeur de la pièce adverse prise va devenir prépondérante.

8.5 La fourchette.

Cette notion est une des plus importantes dans le jeu d'échec. Il est donc nécessaire de soigner la fonction qui va s'en occuper. Il existe deux types de fourchette :

1. Celle qui met en echec le roi.
2. Celle qui ne met pas le roi en echec.

Le deuxième type est intéressant parce qu'il va faire rentrer une nouvelle donnée qu'il faudra prendre en compte : l'équilibre des forces en présence.

Exemple :

Si sur l'échiquier il reste pour les blancs 2 cavaliers, un fou, 3 pions et le roi, et pour les noirs 1 cavalier, 1 fou, 1 tour, 3 pions et le roi, il y a un avantage pour les noirs puisque la tour vaut presque 2 fous ou 2 cavaliers. Il peut être intéressant alors, s'il est possible de prendre une pièce au noir, de le faire, histoire de rééquilibrer les forces.

8.6 La tour menacante.

Si au moins une tour blanche est sur la ligne 7 ou 8, il faut augmenter sa pondération. De même pour les noirs sur la ligne 2 ou 1. Cette pondération doit s'apparenter à celle de la dame. Une tour sur une de ces lignes provoque lente mais sûre de l'adversaire. Elle à donc un intérêt très important.

8.7 La mise en échec du roi.

Le point ultime du jeu : venir chatouiller les moustaches du roi adverse. L'attaque de cette pièce est bien entendu à prendre en compte. Il faudra cependant trouver un juste milieu. Parfois la mise en echec direct du roi n'apporte rien de bon ou supprime la possibilité de mat deux coups après. A méditer.