Sylvain Archenault
Yves Houpert

PROJET INFORMATIQUE :

Langage C :

Le Jeu De Dames

Projet GM3
Janvier 2005

Table des matières

Chapitre 1

Introduction.

Au cours de ce projet d'informatique, nous avons réalisé un jeu de dames en C. L'intérêt principal de ce projet est de mettre en pratique les connaissances que nous avons apprises lors du cours de C et de programmation structurée.

Nous allons d'abord présenter les règles du jeu de dames que nous allons programmer. Ensuite nous expliquerons le fonctionnement de chaque module de code. Puis dans un nouveau chapitre nous parlerons des problèmes que nous avons rencontrés.

Nous avons ensuite consacré un chapitre à l'intégration des dames dans notre programme. Pour finir nous présenterons les améliorations apportées au projet.

Chapitre 2

Présentation du Jeu.

2.1 Règle du jeu de dames.

- Les blancs commencent
- Le pion avance d'une seule case et ne peut pas reculer. A son tour le joueur doit jouer un pion ou une dame. S'il ne peut pas jouer, il a perdu la partie.
- Le joueur gagne la partie si son adversaire ne peut plus jouer, s'il abandonne ou s'il dépasse sont temps de jeu (jeu à la pendule). Il y a match nul en cas de commun accord, si la partie ne peut se terminer (20 coups maximum 3 dames contre 1 par exemple) ou si la même position se répète trois fois.
- Le pion prend en sautant par-dessus le pion adversaire (jamais par-dessus son propre pion). On ne peut pas passer par dessus deux pions accolés. Il pourra sauter successivement plusieurs pions adverses dans le même tour, il s'agit d'une rafle. La prise est aussi possible en arrière
- La prise est obligatoire et le joueur adverse à le droit de faire rejouer le joueur fautif tant qu'il n'a pas joué à son tour (le coup serait alors déclaré couvert).
- La prise prioritaire est celle où il y a le plus de pions a prendre (la dame comptant alors pour un pion).
- Lors de la prise de plusieurs pions, n'enlever les pions qu'une fois la rafle effectuée. Un pion prenant peut passer deux fois sur la même case vide mais en peut pas passer deux fois sur le même pion.
- Le pion ne peut être promu dame que s'il parvient exactement sur une case de la ligne de fond adverse. Y passer lors d'une rafle ne suffit donc pas.
- Une dame peut se déplacer sur la ligne entière en avant ou en arrière sauf si bien sûr elle est bloqué par ses pions ou au moins deux pions adverses accolés.
- La prise obligatoire et prioritaire est aussi valable pour la dame. Celle-ci prend un pion en passant par-dessus, peut s'arrêter à la case souhaitée après le pion, est amenée à bifurquer sur la ligne traverse si un autre pion est à prendre et ainsi de suite jusqu'a la fin de la rafle effectuée.

2.2 Adaptation des règles.

Lors de ce projet, nous avons essayé d'être le plus fidèle possible aux règles du jeu. Néanmoins, essentiellement pour une raison de temps, toutes les règles n'ont pu être implémentées. Nous pensons en particulier à la répétition d'une position ou encore si la partie ne peut se terminer.

Nous n'avons pas non plus proposer aux joueurs une partie nulle. La raison est que du fait que les deux joueurs jouent ensemble et se voient, nous n'avons pas vu l'intérêt de cette fonctionnalité. En ce qui concerne le temps le dépassement du temps de jeu, à l'initialisation nous demandons à l'utilisateur de spécifier un temps de réflexion maximum.

Toute fois, nous avons attaché une très grande importance au coup obligatoire. Nous avons essayé d'avoir la meilleur jouabilité possible, notamment lors du choix des différents coups obligatoires possibles, et du mouvement des pièces.

Chapitre 3

Analyse du Problème.

Après une première analyse, nous avons remarqué qu'il était judicieux d'utiliser des sous programmes pour une conserver une certaine ergonomie. Nous avons donc séparé le code en plusieurs fichiers. Tous les prototypes sont également stockés à part dans les fichiers affichage.c,damier.c, jeu.c, cpu.c et bien sûr main.c.

Cela permet une meilleure visualisation des tâches effectuées.

3.1 Les structures.

Pour faciliter la lecture et l'écriture du programme nous avons décidé d'utiliser des *symboles* pour notamment l'état d'une case : *PION_BLANC, PION_NOIR, CASE_VIDE*, mais aussi pour connaître le type d'un joueur (*CPU, HUMAIN*).

```
# define PION_BLANC 1
# define DAME_BLANC 2
# define PION_NOIR -1
# define DAME_NOIR -2
# define CASE_VIDE 0
# define HUMAIN 0
# define CPU 1
```

Nous avons également choisi de créer des structures pour faciliter la programmation. Ces structures sont stockées dans le fichier structures.h et sont définies ainsi :

- La structure case, qui symbolise une case, elle a donc pour élément deux entiers, un pour la ligne, un autre pour la colonne :

```
typedef struct {
      int lig,col;
} Case;
```

- La structure deplacement, qui symbolise un deplacement, cette structure possède deux éléments, deux case, une étant la case initiale, l'autre la case finale :

```
typedef struct {
      Case case_i,case_f;
} deplacement;
```

- La structure element, qui est un élément d'une liste chaînée. En effet, dans notre programme, pour représenté les rafles, nous avons choisis d'utiliser comme structure de données les listes chaînées.

```
struct element {
     Case c;
     listeCases suivante;
};
typedef struct element *listeCases;
```

– La structure joueur, qui symbolise un joueur, elle a pour élément le nom du joueur, son type *CPU* ou *HUMAIN*, ainsi que la couleur des pions du joueurs :

```
typedef struct {
      char *nom;
      int type;
      int couleur;
} joueur;
```

– La structure damier, qui symbolise la jeu, elle contient le nombre de pions de chaque couleur, et un tableau représentant la position des pions sur le plateau :

```
typedef struct {
      int plateau[10][10]; // Représente l'état du   plateau
      int nb_ noir;        // Représente le nombre de pion noir
      int nb_ blanc;        // Représente le nombre de pion blanc
} damier;
```

3.2 La conception et l'initialisation du damier.

La réalisation du damier est réalisée dans le fichier affichage.c. L'idée principale pour l'afficher est la suivante :

Le damier est un tableau à deux dimensions. Si une case du damier a pour valeur PION NOIR, c'est à dire qu'elle est occupée par un jeton noir, elle prend la valeur X. Si elle a pour valeur PION BLANC, elle aura la valeur 0.Enfin, si la case est vide, elle aura " " pour valeur.

Pour chaque case de chaque ligne, on effectue ce test.Le damier peut être alors affiché sans problème.

- Aperçu du code :

```
void affiche_ligne(int i,int damier[10][10]) {
int j;
printf(" %2d ",i+1);
for(j=0;j<10;j++) {
      switch(damier[i][j]) {
            case PION_BLANC :
                  printf("| O ");
                  break;
            case PION_NOIR :
                  printf("| X ");
                  break;
            case DAME_NOIR :
                  printf("| [X] ");
                  break;
            case DAME_BLANC :
                  printf("| [O] ");
                  break;
            default :
                  printf("| ");
                  break;
      }
}
```

```
printf("| %2d\n",i+1);
printf(" |-----------------------------------------|\n");
}
/*
        Fonction qui affiche le damier en totalité
*/
void affiche_damier(int damier[10][10]) {
int i;
printf("      a    b    c    d    e    f    g    h    i    j  \n");
printf("    |-----------------------------------------|\n");
for(i=0;i<10;i++)
     affiche_ligne(i,damier);

printf("     a    b    c    d    e    f    g    h    i    j  \n");
}
```

Pour initialiser, le damier, il suffit de donner la valeur PION NOIR et PION BLANC aux cases où les pions doivent se trouver en début de partie.

- Aperçu du code :

```
void init_damier(damier* jeu) {
jeu->nb_blanc = 20;
jeu->nb_noir = 20;
int i,j;
//On initialise le tableau
for(i=0;i<10;i++)
     for(j=0;j<10;j++)
          jeu->plateau[i][j] = 0;
//On remplie la partie haute du damier par des pions noirs
for(i=0;i<4;i++) {
          //Si i est pair, la première case doit être noir (active)
          if(i%2 == 0) j=1;
          else j=0;

          for(j;j<10;j+=2)
                  jeu->plateau[i][j] = PION_NOIR;
}
//On remplie la partie basse du damier par des pions blancs
for(i=6;i<10;i++) {
          //Si i est pair, la premier case doit être noir (active)
          if(i%2 == 0) j=1;
          else j=0;

          for(j;j<10;j+=2)
          jeu->plateau[i][j] = PION_BLANC;
}
}
```

3.3 Le déplacement des pions.

Cette partie demande une certaine rigueur car de nombreux tests doivent être effectués pour bien vérifier qu'un déplacement puisse se réaliser. En effet, on doit vérifier par exemple que le joueur indique une case qui existe sur le damier, qu'il joue bien un de ses pions ou qu'il ne se trompe pas de case en sélectionnant une case vide. Ensuite on doit vérifier si son déplacement est correct, c'est à dire que le pion se déplacera vers une case vide du plateau et non une case déjà occupée ou qui n'existe pas.

On voit donc que l'on a toute une batterie de tests à effectuer rien que pour valider le déplacement simple d'un pion.

- Aperçu du code :

```
void deplace_pion(int couleur,deplacement d,int prise,damier* jeu) {
     jeu->plateau[d.case_i.lig][d.case_i.col] = CASE_VIDE;
     jeu->plateau[d.case_f.lig][d.case_f.col] = couleur;
     if(prise == 1) {
          if(couleur == PION_BLANC)
                 jeu->nb_noir--;
          else jeu->nb_blanc--;
          jeu->plateau[(d.case_f.lig + d.case_i.lig)/2]
[(d.case_f.col + d.case_i.col)/2]=CASE_VIDE;
     }
}
int test_deplacement(int couleur,deplacement d,int damier[10][10]) {
//Le pion sélectionné n'est pas un pion du joueur
if(damier[d.case_i.lig][d.case_i.col] != couleur)
       return -1;
//Case occupé
if(damier[d.case_f.lig][d.case_f.col] != CASE_VIDE)
       return -1;
//Mauvaise Ligne
if(d.case_f.lig != d.case_i.lig - couleur)
       return -1;
//Mauvaise colonne
if(d.case_f.col != (d.case_i.col+1) &&
d.case_f.col != (d.case_i.col-1) ) {
       return -1;
return 0;
}
```

3.4 Algorithme de recherche de la rafle maximale.

Une des parties les plus dures de ce projet est sans aucun doute de bien programmer la prise des pions. En effet, un grand nombre de règles doivent être respectées :

- si une prise est possible, elle doit être effectuée.
- un pion peut prendre vers l'avant mais aussi vers l'arrière.
- la rafle doit être toujours prioritaire.

Toutes ces conditions doivent être remplies. De nombreuses procédures ou fonctions, et plusieurs tests sont donc nécessaires. On a divisé le problème en deux. Nous avons programmé une fonction coup_obligatoire qui parcourt tout le damier . Pour chaque pion trouvé du joueur , on appelle une deuxième fonction max_pion_prises qui calcule le nombre maximum de prises pour ce pion. Il ne nous reste plus qu'à enregistrer la rafle qui prend le plus de pion.

La fonction max_pion_prises a été assez difficile à programmer. Voici son principe. Cette fonction est appelée avec comme argument une “case “. C’est a partir de cette case que l’on doit calculer la ou les rafles maximales. Tout d’abord, grâce à la fonction pion_prise, on connaît le nombre de pions adversaires que notre pion peut prendre. On effectue une copie du damier afin de pouvoir simuler les prises. Pour chaque pion “prenable “, on effectue la prise dans un damier “virtuel”, et on rappelle notre fonction max_pion_prises avec comme argument, cette fois la case d’arrivée de la prise. Ensuite nous devons enregistrer les rafles maximales. Il y a plusieurs cas à envisager. Le plus facile est quand le nombre de pions pris est supérieur au nombre maximal de pions pris par les chemins précédents. En effet il suffit de mettre chaque rafle trouvée dans un tableau de rafles. Si on obtient plusieurs fois le même nombre de pions pris par des rafles différentes, on doit ajouter les nouvelles rafles à notre tableau de rafles maximales.

- Aperçu du code de max_pion_prises :

```
int max_pion_prises(Case case_i, int couleur, damier* jeu,
listeCases* rafle_max,int* n) {
int prises = 0; //0 si on peut pas prendre en case_i, 1 sinon
int prises_p; //Nombre de prises max en n_case[i]
int prises_max = 0; //Nombre max de prises possibles trouvé
int np; //Nb de possibilité de prendre le max de pions en n_case[i]
int nb; //Contient le nombre de possibilité de prises en case_i
//Variable d'indice, de boucle
int i,t,ind=0;
//Variable servant au calcul
Case n_case[4];
deplacement d;
damier cjeu;
listeCases* r;
if ((nb = pion_prise(case_i.col,case_i.lig,couleur,
jeu->plateau,n_case))!=0) {
       //Le pions situé en case_i peut prendre nb pions
       //On créé un copie du jeu qui va nous permettre de simuler les
coups sans affecter le damier originel
       cjeu = copie_damier(jeu);
       //On peut prendre un pion -> prises =1
       prises++;
//Pour chaque possibilité de prises, on calcul le nombre de
prises maximales que l'on peut faire
       for(i=0;i<nb;i++) {
              //On initialise le nombre de façon de faire
              np = 1;
              // On alloue de la place
              r = (listeCases*) malloc(sizeof(listeCases));
              r[0] = (listeCases) malloc(sizeof(struct element));
//On crée une variable déplacement et on effectue le deplacement
              d.case_f = n_case[i];
              d.case_i = case_i;
              deplace_pion(couleur,d,1,&cjeu);

              //Calcul du nombre de prises max a partir de cette position
              prises_p = max_pion_prises(n_case[i],couleur,&cjeu,r,&np);

              if(prises_p == prises_max && prises_p>0) {
// Il y a d'autre possibilité de prendre prises_max pion,
mais en passant par un autre endroit
```

```
                    for(t=0;t<np;t++) {
// Pour chaque nouvelle possibilité on crée une nouvelle cases dans rafle_max
                    rafle_max[ind] = (listeCases) malloc(sizeof(struct element));
                   //On lui affecte les valeurs
                   rafle_max[ind]->suivante = r[t];
                   rafle_max[ind]->c = case_i;
                   ind++;
                   }
                // Correction de l'ind
                ind--;
// On ajoute au nombre de possibilité les np que l'on vient de trouver
                *n=*n+np;
                }
               if(prises_p == 0 && prises_max==0) {
                     // On ne peut plus prendre de pion
                     // On crée le dernier element de la liste
                     r[0]->c = n_case[i];
                     r[0]->suivante = 0;
                     // On créé une nouvelle case dans le tableau
                     rafle_max[ind] = (listeCases) malloc(sizeof(struct element));
                     // On lui affecte les valeurs correspondante
                     rafle_max[ind]->suivante = r[0];
                     rafle_max[ind]->c = case_i;
// L'indice augmente de 1 car on a trouvé un moyen de faire prises_max
                     ind++;
                     *n =ind;
                }

              if(prises_p > prises_max) {
              // On a trouvé np moyen de faire plus de prises
              //Il y a np moyen
                  *n = np;
                   prises_max = prises_p;
               //On met dans le tableau tous les possibilités qu'on a trouvées
                   for(ind=0;ind<np;ind++) {
                        rafle_max[ind] = (listeCases) malloc(sizeof(struct element))
                        rafle_max[ind]->suivante = r[ind];
                        rafle_max[ind]->c = case_i;
                   }
              }
         }
}
else {
//Si on ne peut pas prendre, on est en fin de liste, d'où suivante pointe vers nil
   for(t=0;t<ind;t++)
       rafle_max[t]->suivante = 0;
}
//On retourne le nombre de prises maximales
return prises + prises_max;
}
```

Nous avons vu ici uniquement les principales fonctions qui permettent d'obtenir des rafles cohérentes qui respectent les règles du jeu. Nous verrons dans la partie " problèmes rencontrés ", les aspects plus détaillés de l'obtention et de la réalisation de la rafle maximale.

3.5 Faire jouer l'ordinateur

Une autre étape du projet consistait à faire jouer l'ordinateur. Dans un premier temps, on admettra que l'ordinateur jouera " bêtement " . c'est à dire qu'il effectuera le premier coup possible. Si il y a plusieurs coups obligatoires, il réalise le premier. On a décidé de créer 4 fonctions pour ce problème :

→ une fonction test deplacement qui renvoie un déplacement si le pion (en (i,j)) peut bouger, ou -1 sinon.
Il y a simplement 2 cases à tester. Pour les blancs, la coordonnée horizontale diminue ; \"couleur \" étant égale à 1 (PION _ BLANC), la nouvelle coordonnée horizontale est donc i-couleur. Pour les noirs, c'est pareil. En ce qui concerne les coordonnées verticales, il est évident que les deux seules possibilités sont j+1 et j-1. On teste donc la case qui se situe en (i-couleur, j+1) et celle en (i-couleur, j-1).

- Aperçu du code de testdeplacement :

```
/*
          Renvois un deplacement, si le pion en (i,j) peut bouger, 0 sinon

          param :
                 plateau : matrice contenant les pions du jeu
                 i,j : ligne et colonne de la pièce a bouger
                couleur : indique la couleur des pions de l'ordinateur
*/
deplacement* testdeplacement(int plateau[10][10],int i,int j,int couleur) {
deplacement* d = (deplacement*) malloc (sizeof(deplacement));
if(i - couleur >= 0 && i - couleur < 10 && j+1 <10 &&
                                  plateau[i-couleur][j+1] == CASE_VIDE) {
                  d->case_i.lig=i;
                  d->case_i.col=j;
                  d->case_f.lig=i-couleur;
                  d->case_f.col=j+1;
                 return d;
}
if(i - couleur >= 0 && i - couleur < 10 && j-1 >= 0 &&
                                    plateau[i-couleur][j-1] == CASE_VIDE) {
                  d->case_i.lig=i;
                  d->case_i.col=j;
                  d->case_f.lig=i-couleur;
                  d->case_f.col=j-1;
                  return d;
}
return 0;
}
```

→ une fonction trouvercoup qui renvoie le premier coup possible pour l'ordinateur. On parcourt le damier à la recherche des pions du joueur. On teste si le pion peut bouger grâce à la fonction précédente testdeplacement. Dès qu'un pion peut bouger, on retourne son déplacement sous la forme d'une liste chaînée.

- Aperçu du code de trouvercoup :

```
listeCases trouvercoup(damier* jeu, int couleur) {
int i=0,j,nbpion=0,max;
deplacement *d;
listeCases coup = (listeCases) malloc(sizeof(struct element));
```

```
if(couleur == PION_BLANC)
       max = jeu->nb_blanc;
else max = jeu->nb_noir;
while(nbpion<max) {
         for(j=0;j<10;j++) {
                if(jeu->plateau[i][j] == couleur) {
                //On a trouvé un pion
                      nbpion++;
                      if((d=testdeplacement
                                   (jeu->plateau,i,j,couleur))!=0) {
                      //On crée notre liste
                             coup->c = d->case_i;
                             coup->suivante = (listeCases) malloc
                                                 (sizeof(struct element));
                             coup->suivante->c = d->case_f;
                             coup->suivante->suivante = 0;
                             return coup;
                      }
                 }
                else {
                        if(jeu->plateau[i][j] == 2*couleur) {
                        //On a trouvé une dame
                                nbpion++;
                                if((d=testdeplacement_dame
                                            (jeu->plateau,i,j,couleur))!=0) {
                                //On crée notre liste
                                       coup->c = d->case_i;
                                       coup->suivante = (listeCases) malloc
                                                          (sizeof(struct element))
                                       coup->suivante->c = d->case_f;
                                       coup->suivante->suivante = 0;
                                       return coup;
                                }
                             }
                  }
           }
i++;
}
return 0;
}
```

→ Une fonction jouer_cpu qui fait jouer l'ordinateur. Cette fonction ne présente aucune particularité, elle utilise juste les 2 fonctions précédentes.

- Aperçu du code de jouer_cpu :

```
int jouer_cpu(damier* jeu,int couleur,int nb_coup,listeCases* rafle) {
listeCases coup;
deplacement d;
//Si il y a plusieurs coups possibles, on prend le premier
if(nb_coup>0)
       faire_coup(rafle[0],couleur,jeu);
else {
       //Pas de coup possible
```

```
            if((coup=trouvercoup(jeu,couleur))==0)
                    return -1;
            else {
                    d.case_i = coup->c;
                    d.case_f = coup->suivante->c;
                    deplace_pion(couleur,d,0,jeu);
            }
}
return 1;}
```

3.6 La partie est finie.

A chaque tour, il faut tester si le jeu . La fonction *int jeu_fini(damier* jeu, int couleur)* joue ce rôle.A chaque fin de coup elle vérifie si le jeu peut continuer en testant si le nombre de pions blancs ou noirs et égal à 0. Elle teste également si l'ordinateur peut encore jouer en cherchant "un premier coup possible " (si il n'y a pas de premier coup possible, l'ordinateur ne peut plus jouer).Si une des deux conditions n'est pas remplie, la partie est finie, sinon le jeu continue.

- Aperçu du code de jeu_fini(damier* jeu, int couleur) :

```
int jeu_fini(damier* jeu, int couleur) {
if(jeu->nb_blanc==0)
        return PION_BLANC;
if(jeu->nb_noir==0)
        return PION_NOIR;
if(trouvercoup(jeu,couleur) == 0) {
            printf( "Fin du match \n ");
            affiche_damier(jeu->plateau);

            if(couleur == PION_NOIR)
                        printf( "Les noirs ne peuvent plus bouger ! \n  ");
            else printf( "Les blancs ne peuvent plus bouger ! \n ");
            return couleur;
}
return 0;
}
```

3.7 La saisie d'un déplacement.

La gestion des déplacements est effectuée par la fonction *selec_pion.* On appelle la fonction *lecture_deplacement* qui lit un déplacement ou une action à effectuer (Chap 6.). Cette fonction nous retourne différentes valeurs : -1 si la saisie est erronée, 3 si l'utilisateur veut quitter, si il a chargé un fichier et 0 si le déplacement est valide. Dans ce cas on vérifie la validité du déplacement par l'intermédiaire de la fonction *test_deplacement* ; puis on déplace le pion.

- Aperçu de selec_pion :

```
int selec_pion(int couleur,damier* jeu) {
int ok=0;
int i,prise;
deplacement d;
do {
       switch(lecture_deplacement(&d,jeu,couleur)) {
```

```
            case -1 :
                  printf("Erreur de deplacement\n");
                  break;
             case 0 :
                   if(jeu->plateau[d.case_i.lig][d.case_i.col] == couleur) {
                   if((prise = test_deplacement(couleur,d,jeu->plateau)) == -1) {
printf("Impossible de déplacer le pion %c,%d en %c,%d\n",
d.case_i.col+'a',d.case_i.lig+1,d.case_f.col+'a',d.case_f.lig+1);
                         }
                        else
                             ok = 1;
                   }
                   break;

              case 3 :
                    return -1;
                    break;
               case 4 :
                     return 2;
                     break;
                default :
                      break;
            }
} while(ok == 0);
deplace_pion(couleur,d,prise,jeu);
return 1;
}
```

3.8 Le programme principal

Le programme principal gère essentiellement l'initialisation des joueurs, la gestion des tours, et vérifie si la partie est terminée.

Nous allons expliquer succinctement la gestion des tours. Tout d'abord, on recherche les coups obligatoires par l'intermédiaire de la fonction *coup_obligatoire.* Ensuite, si le joueur est "humain" , nous lui demandons de saisir un coup grâce à *selec_pion* si il n'y a pas de coup obligatoire, sinon on lui demande de choisir un coup parmi la liste des coups obligatoires possibles. Dans le cas d'une partie contre l'ordinateur, on appelle la fonction *jouer_cpu.* Pour finir, on change de joueur et on continue jusqu'à la fin de la partie qui nous est indiquée par la fonction *jeu_fini* ou que le joueur veuille quitter.

- Aperçu de main.c :

```
do {
  coup = coup_obligatoire(joueur_c->couleur,jeu,rafle,tmp,&n);
  if(joueur_c->type==HUMAIN) {
    affiche_damier(jeu.plateau);
    if(joueur_c->couleur == PION_NOIR)
      printf( "\n C'est à %s de jouer avec les noirs ( %d pions) \n",
joueur_c->nom,jeu.nb_noir);
    else
      printf( "\nC'est à %s de jouer avec les blancs ( %d pions) \n",
joueur_c->nom,jeu.nb_blanc);
    printf("Nombre de prises maximum : %d \n",coup);
    if(coup > 0 )
      affiche_choixcoup(rafle,n,joueur_c->couleur,&jeu);
```

```
      else {
         if(selec_pion(joueur_c->couleur,&jeu)==-1)
           quitter=1;
      }
    }
    else {
      if(jouer_cpu(&jeu,joueur_c->couleur,coup,rafle)==-1) {
        printf( "L'ordinateur abandonne, vous avez gagner!!! \n");
        quitter=1;
      }
}
//On change de joueur
if(joueur_c == joueur1)
joueur_c = joueur2;
else joueur_c = joueur1;
} while(quitter==0 && (statut = jeu_fini(&jeu,joueur_c->couleur))==0);
```

Chapitre 4

Les problèmes rencontrés

Dans ce chapitre, nous allons aborder les difficultés auxquelles nous nous sommes heurtés lors de la réalisation de ce projet.

4.1 La recherche du coup obligatoire

Cette partie a été la plus dure à réaliser. La recherche de la prise maximale est complexe à obtenir. En effet, il n'est déjà pas évident de trouver la prise maximale, mais il est encore plus difficile d'enregistrer toutes les rafles équivalentes (même nombre de prises). Nous avons tout d'abord fait un algorithme qui recherche la rafle maximale sans se préoccuper des égalités. C'est une fois celui-ci effectué qu'on l'a amélioré pour qu'il puisse gérer les diverses possibilités de rafles.

Au cours du développement de cette algorithme, nous avons rencontré plusieurs problèmes. Au début, on effectuait les déplacements dans la même strucure damier, sans supprimer les pions pris, nous avions donc une boucle infinie. En effet, dans le cas de la simulation de la prise d'un pion, il ne faut pas que lorsque la prise est effectuée, le pion puisse à nouveau prendre le pion qu'il vient de prendre. La solution est de simuler virtuellement chaque possibilité. Nous avons dû écrire une fonction damier *copie_ damier(damier*)* qui fait la copie du damier. On simule donc " virtuellement " la prise, le pion pris disparaît du damier, et on fait de nouveau une recherche d'une possibilité de prise.

Du fait qu'on utilise des listes chaînées pour symboliser les rafles, nous nous sommes heurtés à beaucoup de problèmes d'allocation de mémoire. Grâce à l'utilisation du debugger gdb, la résolution de ces problèmes s'est faite relativement facilement. De même la prise en compte des rafles équivalentes nous a causé plusieurs problèmes. En effet, on n'a plus à faire à une simple liste chaînée, mais à un tableau de listes chaînées.

4.2 Le choix de la prise maximale

On sait que dans le jeu de dames, la rafle maximale, si elle existe doit être jouée. Par contre, si il existe plusieurs rafles effectuant le même nombre de prises, le joueur a le choix d'effectuer celle qui semble la plus pertinente. Nous avons donc créé des fonctions qui permettent ce choix.

→ la fonction affiche _ rafle qui affiche la liste des cases lors d'un coup obligatoire.
→ la fonction affiche _ choixcoup qui affiche la liste des coups obligatoires.

- Aperçu du code de affiche_choixcoup :

```
/*
Fonction qui propose le choix d'un coup obligatoire
*/
void affiche_choixcoup(listeCases* rafle, int n, int couleur, damier* jeu) {
int ok=0,i;
```

```
char buffer[50];
do {
           printf( "Plusieurs coups sont possibles %d : \n",n);
           for(i=0;i<n;i++) {
                   printf(" %d : ",i+1);
                   affiche_rafle(rafle[i]);
           }
          if(scanf( " %d ",&i) ==1 && i>0 && i<=n) {
               faire_coup(rafle[i-1],couleur,jeu);
               ok = 1;
          }
           //Vide le tampon
         else fflush(stdin);
} while(ok == 0);
}
```

Chapitre 5

Introduction des Dames.

Nous parlerons dans ce chapitre de l'introduction des dames. Cette introduction s'est faite en deux étapes, tout d'abord nous avons programmé les fonctions de déplacement et de promotion du pion en dame. Cette partie fut traitée assez facilement. Nous avons eu un peu plus de mal à calculer la rafle maximale pour les dames. Notre idée de départ était d'adapter l'algorithme programmé pour les pions, mais comme les dames ne se déplacent pas de le même façon, nous avons eu quelques difficultés à l'implémenter pour les dames.

Toute cette partie du jeu a été traitée en dernier. En effet, il fallait tout d'abord maîtriser les autres paramètres du projet (recherche de la prise maximale par exemple).Cette partie du code se trouve dans dames.h, dames.c.

5.1 Fonctions de déplacement et de promotion

5.1.1 Promotion

Le principe est d'appeler dans le main à chaque fin de tour la fonction faire_dames :

```
void faire_dames(damier* jeu,int couleur) {
int i,j;
if(couleur==PION_NOIR) {
     i=9;
     j=0;
}
else {
     i=0;
     j=1;
}
for(j;j<10;j+=2) {
     if(jeu->plateau[i][j] == couleur)
          jeu->plateau[i][j] = 2*couleur;
}
}
```

En fait, nous testons la présence de pions du joueur sur la ligne de promotion, si c'est le cas, on transforme le pion en dame, ce qui est équivalent à une multiplication par 2.

5.1.2 Déplacement

En ce qui concerne le déplacement des dames, c'est le même principe que pour les pions à la seule différence qu'un déplacement n'est pas vérifier par la fonction test_deplace_pions mais par la fonction test_deplace_dames(damier* jeu, int couleur, deplacement d) :

```
int test_deplace_dame(damier* jeu, int couleur,deplacement d) {
int i=d.case_i.lig,j=d.case_i.col;
if(jeu->plateau[i][j] != couleur*2) {
           printf("%c%d n'est pas une dame\n",j+'a',i+1);
           return -1;
}
while(i!=d.case_f.lig || j!= d.case_f.col) {
            if(d.case_f.lig>d.case_i.lig)
                       i++;
            else
                       i--;

            if(d.case_f.col>d.case_i.col)
                       j++;
            else
                       j--;

            if(jeu->plateau[i][j] != CASE_VIDE) {
                       printf("%c%d n'est pas vide\n",j+'a',i+1);
                       return -1;
            }
}
return 1;
}
```

Cette fonction teste toutes les cases situées entre la case initiale et finale et retourne -1 si une de ces cases n'est pas valide.

Puis la fonction deplace_dame permet tout simplement de déplacer la dame. Elle est est similaire la fonction déplace_ pion :

```
void deplace_dame(damier* jeu, int couleur,deplacement d, int p, Case prise) {
jeu->plateau[d.case_i.lig][d.case_i.col] = CASE_VIDE;
jeu->plateau[d.case_f.lig][d.case_f.col] = 2*couleur;
if(p==1 && (jeu->plateau[prise.lig][prise.col]==-couleur ||
jeu->plateau[prise.lig][prise.col]==-couleur*2)) {
               jeu->plateau[prise.lig][prise.col] = CASE_VIDE;
               if(couleur == PION_BLANC)
                                   jeu->nb_noir = jeu->nb_noir -1;
             else
                                   jeu->nb_blanc = jeu->nb_blanc - 1;
     }
}
```

5.2 Adaptation de la recherche du coup obligatoire

5.2.1 Modification de la fonction max_prise

Vient alors le problème de la recherche du coup obligatoire. Nous allons adapter l'algorithme écrit pour les pions aux dames. Seulement, le déplacement des dames lors d'une rafle est beaucoup plus complexe. En effet, il faut à chaque fois parcourir la diagonale dans le sens de la prise et vérifier s'il existe un pions de la couleur adverse prenable. C'est a dire qu'il y a un pion soit devant la dame, soit dans la diagonale gauche, ou dans la diagonale droite. Si de tels pions existent, alors nous rappelons notre fonction max_dame_prise. C'est donc une fonction récursive.

Le principe de l'algorithme diffère un peu de celui utilisé pour les pions. En effet, comme on l'a vu, on regardait si on pouvait faire une prise (grâce a pion_prise), si oui on la faisait dans un damier "virtuel", puis on rappelait max_prises avec comme argument la case finale de la prise. Avec les dames, la fonctions max_dame_prise est appelée avec comme argument la case qui se situe juste avant la pion "prenable", et la direction dans laquelle ce pion se trouve. De manière récursive, on obtient donc une rafle, que l'on stocke toujours de la même manière, c'est-à-dire dans une liste chaînée.

La modification de la fonction max_prises qui calculait la rafle maximale pour les pions s'est faite en deux étapes. Tout d'abord, la fonction pion_prise par la fonction dame_prise. Cette fonction fonctionne exactement de la même manière. Ensuite, du fait que l'on est amené à tester toutes les cases d'une diagonale, on doit indiquer à notre fonction quelle diagonale. Pour cela, on a rajouté en argument une "case", dans laquelle nous stockons l'évolution de l'état des lignes dans le membre lig, celui des colonnes dans le membre col. Par exemple, si la dame se déplace dans la diagonale vers le haut à gauche, on a dans case.lig : 1, case.col : -1.

5.2.2 Modification de la fonction coup_obligatoire

Nous avons dû modifier notre fonction coup obligatoire pour qu'elle puisse gérer les dames, on parcourt toujours tout le damier, mais cette fois lorsque l'on tombe sur un pion on appelle la fonction "pion", qui gère le calcul de la rafle maximale pour ce pion. Si c'est une dame, on appelle la fonction "dame".

Dans cette fonction, on recherche, tout d'abord, les pions "prenables", grâce à une nouvelle fonction, prise_possible qui nous donne la liste de ces pions. Pour chaque pion "prenable", on déplace dans un damier 'virtuel", notre dame sur la case précédant ce pion. On appelle ensuite max_dame_prises, avec comme argument, cette case, et la direction dans laquelle se trouve le pion à prendre.

Ensuite la gestion de la rafle maximal, s'effectue comme pour les pions.

La dernière chose que nous devons faire, est de proposer au joueur, où il veut déplacer sa dame après la dernière prise. En effet, une dame peut s'arrêter où elle veut après une prise. Nous avons modifié la fonction faire_coup pour qu'elle puisse gérer cette possibilité.

```
if(jeu->plateau[d.case_f.lig][d.case_f.col] == 2*couleur) {
          ind = 0;
          affiche_plateau(jeu->plateau);
          d.case_i = d.case_f;
          tableau[0] = d.case_i;
          d.case_f.lig+=i;
          d.case_f.col+=j;

        while(d.case_f.lig >= 0 && d.case_f.lig <10 && *d.case_f.col >= 0
&& d.case_f.col <10 && jeu->plateau[d.case_f.lig][d.case_f.col]
== CASE_VIDE) {
                      ind++;
                      tableau[ind] = d.case_f;
                      d.case_f.lig+=i;
                      d.case_f.col+=j;
                      printf("%d%d\n",d.case_f.lig,d.case_f.col);
        }

        do {
                  printf("Ou voulez vous que la dame arrive?\n");
                  for(i=0;i<=ind;i++)
                  printf("%d. %c%d ",i+1,tableau[i].col + 'a',
tableau[i].lig + 1);
                  printf("\n");
                  scanf("%d",&j);
        }while(j<1 || j>ind+1);
```

```
                d.case_f = tableau[j-1];
                deplace_dame(jeu,couleur,d,0,d.case_i);
    }
```

Chapitre 6

Les améliorations apportées au projet

6.1 Ajout de quelques fonctionnalités bien pratique

Afin de bénéficier d'une meilleure ergonomie de jeu, nous avons pensé que mettre en place quelques options supplémentaires serait intéressant. Il suffit de taper "aide " pour afficher la liste des commandes disponibles : "sauvegarder ", "charger ", "afficher le damier " et "quitter " . La fonction *int lecture_deplacement(deplacement* d,damier* jeu,int couleur)* nous permet de bien gérer cette partie du jeu.

Les options dont le joueur peut bénéficier dans le menu sont :

- sauvegarde une partie : *affiche_sauver(damier*jeu,int couleur).*
- charger une partie préalablement enregistrée : *affiche_load(damier*jeu).*
- afficher le damier : *affiche_plateau(int plateau[10][10]).*
- quitter.

On peut noter que les fonctions de sauvegarde et de chargement d'un fichier peuvent être pratiques pour l'utilisateur, mais elles nous ont également été très utiles dans les tests que nous avons dû effectuer pour rendre le jeu efficace. Ces fonctions nous ont facilitées la tâche car nous avons pu sauvegarder une situation particulière (longue à redéfinir si on doit la recréer coup après coup ! !par ex : une situation où l'on peut choisir la rafle maximale ! ! !) afin de tester plus facilement notre code et voir si celui-ci était performant dans ce cas précis.

6.2 Limite de temps

6.2.1 Présentation du problème

Nous avons également ajouté comme nouvelle fonctionnalité, le dépassement du temps de jeu. C'est à dire que si l'utilisateur dépasse le nombre de secondes maximum auxquelles il a droit, il a perdu la partie. Cette limite est rentrée par l'utilisateur lors de l'initialisation du jeu.

Pour programmer cette fonctionnalité, nous nous sommes heurtés aux problèmes de faire tourner deux tâches en même temps. En effet, d'un coté on devait gérer le tour du joueur, c'est à dire la saisie des déplacements ou des actions ; et d'un autre côté un timer qui calcule le temps écoulé. Pour cela, on avait deux possibilités.

La première était non pas d'utiliser un "read bloquant" comme le scanf qui met le processus dans un état "blocked"(C'est à dire qu'on ne peut plus rien faire tant que l'utilisateur n'a pas rentré les données qu'on attend), mais d'utiliser un "read non bloquant", c'est à dire qu'on lit toutes les secondes dans stdin, les données rentrées par l'utilisateur. Et si il met trop de temps, alors on quitte la partie, c'est son adversaire qui gagne. Cette solution, bien que pas mauvaise ne nous a pas paru judicieuse pour plusieurs raisons. Tout d'abord parce que son utilisation ne nous a pas apparue très simple ; en effet si pendant la première seconde, l'utilisateur rentre la première case de son déplacement et une partie de la case finale, comment savoir s'il a fini sa saisie ? De plus l'utilisation de sleep rend l'exécution du programme moins fluide. Ensuite, du fait qu'il existe plusieurs possibilités d'action, nous devions écrire des "read non bloquant" un peu partout, et

cela rendait difficile la gestion du temps totale du tour. Nous préférions employer un contrôle du temps sur le tour du joueur en globalité.

La deuxième possibilité, celle que nous avons choisies, nous permet de faire cela. Il s'agit d'utiliser des threads. Des threads sont une alternatives à la programmation multiprocessus en offrant un parallélisme plus léger et une plus grande facilité pour partager des tâches au sein d'un programme (processus) ainsi que la communication entre ces threads. Pour simplifier, les threads permettent de faire tourner en mêmes temps, sur un même processeur, deux fonctions du même processus (programme). C'est en cela que les threads répondent parfaitement à nos demandes.

6.2.2 Résolution grâce aux pthreads.

Il existe plusieurs librairies permettant d'implémenter les threads dans un programme. En fait, la gestion des threads dépend grandement de l'OS utilisé. Même au sein des mêmes OS, il peut y avoir des différences de gestion. Par exemple la gestion des threads dans les noyaux GNU\Linux 2.6.x et 2.4.x est légèrement différente.

Du fait, que nous développons notre programme sous GNU\Linux (Noyau : 2.6.9), nous avons utilisé une des api proposée sous GNU\Linux, à savoir l'API des pthreads (POSIX threads)[1].

La programmation est assez simple. Nous avons deux fonctions, une qui gère le tour du joueur, et une autre qui compte le temps écoulé. Chaque fonction est lancée dans un thread séparé par l'api *pthread_create()*

6.2.2.1 La fonction tour()

Au sein de cette fonction, nous appelons la procédure *tour_joueur*, qui appelle suivant le cas *choix_rafle*, ou *selec_pion* (Voir Annexe B), une fois cette fonction terminée. On change la valeur de la variable tour_fini, qui va nous permettre de synchroniser nos deux threads.

Le prototype des fonctions pouvant être appelées lors d'un *pthread_create()* est *void* ma_fonction(void* argument)*. Par conséquent on a dû créer une structure afin de passer à notre fonction les arguments dont elle a besoin.

- Aperçu de la fonction tour :

```
void *tour(void* a) {
argument* arg = (argument*) a;
int ret = 0;
if(tour_joueur(arg->jeu,arg->couleur,arg->n,arg->rafle,arg->coup) == -1)
        ret = -1;
pthread_mutex_lock(&mutex);
          if(tour_fini == 0) tour_fini = 1;
pthread_mutex_unlock(&mutex);
pthread_exit((void *) ret);
}
```

6.2.2.2 La fonction temps

Cette fonction est relativement simple. Nous avons un compteur i initialisé à zéro. Puis dans une boucle, on teste si le thread "tour" n'est pas fini grâce à la variable *tour_fini*. Si ce n'est pas le cas on attend une seconde grâce à la fonction *sleep()*, on incrémente notre compteur de 1. On recommence tant que i est inférieur au nombre de secondes limite.

- Aperçu de la fonction tour :

```
void* temps() {
int i = 0;
while(i<=time_limit) {
```

[1] Pour plus d'information, veuillez consulter l'article d'Eric Lacombe dans GNU Linux Magazine n°63, sur le multithreading qui nous a servi de référence. Le tutorial disponible ici : *http ://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html* nous a également été très utile.

```
            pthread_mutex_lock(&mutex);
                    if(tour_fini == 1)
                            pthread_exit((void *) 1);
            pthread_mutex_unlock(&mutex);
            sleep(1);
            i++;
        }
    pthread_exit((void *) -1);
    }
```

6.2.3 Les limites

La gestion des threads étant très dépendante de la machine et notamment du système d'exploitation, cette gestion du temps ne rend pas notre code standard. En effet il ne peut tourner que sur des machines de type GNU\Linux (Unix à vérifier) possédant l'API pthreads. On a donc laissé le choix à l'utilisateur de compiler le jeu avec ou sans cette fonctionnalité. Grâce au makefile, on a le choix de compiler le programme avec ou sans cette gestion du temps (voir Annexe A). Si l'on veut compiler le programme avec cette fonctionnalité, il suffit de compiler le programme avec *"make THREAD=1* ", sinon un *"make* " suffit.

On peut remarquer que cette fonctionnalité ralentit un peu la fluidité du programme. En effet, lorsqu'un joueur rentre un déplacement, le thread "temps "a toutes les chances de se trouver dans un état "blocked ", c'est à dire de se trouver dans le "sleep ", et donc on doit attendre la fin de ce sleep. Par conséquent, on remarque un léger ralentissement après la saisie d'un déplacement ou d'une action.

Chapitre 7

Conclusion

Ce jeu de dames fût très intéressant à réaliser, car il nous a permis de bien comprendre le fonctionnement des pointeurs, des allocations dynamiques de mémoire et des listes chaînées. On a pu remarquer que une programmation structurée était vraiment efficace lorsque la longueur et la complexité du code devient conséquente.

Nous avons essayé d'améliorer le projet en programmant quelques fonctionnalités, à savoir la possibilité d'enregistrer la partie, de charger une partie préalablement enregistrée, de réafficher le damier,etc.

Cette version est la version 0.6 de notre jeu et nous avions eu l'idée de compléter ce rapport en montrant l'évolution du projet à l'aide des versions 0.1,0.2,0.3,0.4,0.5 mais nous n'avons pas eu le temps.

Il serait très intéressant d'améliorer le jeu de l'ordinateur pour l'instant, il n' a aucune chance de gagner ! ! ! ! Ensuite, il est évident que les algorithmes de recherche des coups obligatoires pourraient être améliorés, en rapidité mais aussi en gestion de la mémoire. Néanmoins, il n'y a pas de ralentissement dans le jeu lorsque l'ordinateur recherche ces coups obligatoires du fait qu'il n'y pas énormément d'opération à effectuer, malgré la complexité des algorthimes. Enfin, la jouabilité serait accrue si l'interface graphique était plus agréable. Ces améliorations pourraient être apportées au projet au cours d'un futur projet d'informatique, en java par exemple.

Annexe A

Le Makefile

```
OPT = -g
CC = gcc
ifdef THREAD
   PTHREAD = -D_REENTRANT -pthread
   DEF_THREAD = -DTHREAD
endif
dames : affichage.o dames.o damier.o cpu.o jeu.o main.o
        $(CC) $(PTHREAD) $(OPT) affichage.o dames.o damier.o cpu.o jeu.o main.o -o
affichage.o : affichage.c affichage.h structures.h
        $(CC) $(DEF_THREAD) $(OPT) affichage.c -c
dames.o : dames.c dames.h
        $(CC) $(OPT) dames.c -c
damier.o : damier.c damier.h
        $(CC) $(OPT)  damier.c -c
cpu.o : cpu.c cpu.h
        $(CC) $(OPT)  cpu.c -c
jeu.o : jeu.h jeu.c dames.o
        $(CC) $(OPT)  jeu.c -c
main.o : main.c
        $(CC) $(DEF_THREAD) $(OPT)  main.c -c
clean :
        rm -rf core dames *.c~ *.h~ *.o
```

Annexe B

Listes des fonctions

Nom :	Fichier :	Description :
affiche_aide	affichage.h	Affiche la liste des commandes possibles.
affiche_choixcoup	damier.h	Fonction qui affiche le liste de coups possibles.
affiche_damier	affichage.h	Fonction qui affiche le damier.
affiche_ligne	affichage.h	Fonction qui affiche la ligne i du damier.
afficher_load	affichage.h	Charge une partie à partir d'un fichier.
affiche_plateau	affichage.h	Ré-affiche la plateau.
affiche_quitter	affichage.h	Fonction appelée lorsque le joueur décide de quitter le jeu.
affiche_rafle	damier.h	Fonction qui affiche une rafle.
afficher_sauver	affichage.h	Sauvegarde dans un fichier la partie.
copie_damier	dames.h	Copie la structure du jeu pour effectuers les simulations de coup.
coup_obligatoire	jeu.h	Fonction renvoyant le coup obligatoire (-1 si pas de coup obligatoire).
dame	dames.h	Fonction qui calcule le coup obligatoire pour la dame, et adapte les tableau de rafles r et pr, en fonction du resultat, ainsi que le nombre de façon de faire : n.
dame_prise	dames.h	Renvoie dans n_case la liste des possibilité de prises d'une dame a partir c,l suivante la direction ic,il.
deplace_dame	dames.h	Deplace la dame suivant d, en prenant le pion situé en prise.
deplace_pion	jeu.h	Fonction qui déplace un pion
faire_coup	jeu.h	Fonction qui effectue un rafle
faire_dames	dames.h	Fonction appellé a chaque fin de tour qui promoue un pion en dame.
init_damier	damier.h	Fonction initialisant le jeu.
jeu_fini	cpu.h	Fonction verifiant que la partie est finie.
jouer_cpu	cpu.h	Fait jouer l'ordinateur.
lecture_deplacement	damier.h	Fonction qui lit un déplacement.
max_dame_prises	dames.h	Fonction calculant le nombre de prises max a partir de case_i.
max_pion_prises	jeu.h	Fonction renvoyant le nombre maximum de prises en partant d'une case
pion	jeu.h	Fonction qui calcule le coup obligatoire pour la dame, et adapte les tableau de rafles r et pr, en fonction du resultat, ainsi que le nombre de façon de faire : n.

pion_prise	jeu.h	Fonction renvoyant les prises possibles a partir de la case c,l
prise_possible	dames.h	Fonction qui regarde combien de pions la dames en ca peut prendre, et les stockes dans les stocke dans les tableaux de cases : d et a.
selec_pion	damier.h	Gere la saisie du joueur : déplacement d'un pion ou action.
temps()	main.c	Fonction qui attend time_limit secondes, éxécutée dans le second thread.
test_deplace_dame	dames.h	Vérifie si le deplacement d est valide.
testdeplacement	cpu.h	Renvoit un deplacement, si le pion en (i,j) peut bouger, null sinon.
testdeplacement_dame	cpu.h	Renvoit un deplacement, si la dame en (i,j) peut bouger, null sinon.
test_case	damier.h	Test si une case est bonne.
test_deplacement	damier.h	Test si un déplacement est bon.
tour	main.c	Fonction que gère le tour d'un joueur, exécutée dans le premier thread.
tour_joueur	damier.h	Fonction gérant le tour d'un joueur.
trouvercoup	cpu.h	Renvoie le premier coup possible.