Mise en route

Rappel préliminaire : l'excellent polycopié sur le langage C de H. Garetta est disponible à l'url `http://www.dil.univ-mrs.fr/~garreta/Polys/PolyC.pdf`

1 Premiers objectifs

L'ensemble des éléments présentés dans cette section devraient être fonctionnels vers le début du mois de mars.

1.1 Aspects fonctionnels

D'un point de vue fonctionnel, la projet devra permettre à deux joueurs humains de jouer l'un contre l'autre, en particulier :

1. la position de départ doit être respectée et le joueur qui a les pions noirs doit commencer à jouer ;
2. chaque joueur doit être identifié par son nom ;
3. un indicateur doit préciser qui est le joueur qui a le trait (c'est-à-dire le joueur à qui c'est le tour de jouer) ;
4. quelle que soit la structure de données utilisée pour représenter l'othellier (c'est-à-dire le damier), le format qui sera utilisé pour rentrer un coup sera du type une lettre, pour indiquer la colonne, et un chiffre pour indiquer la ligne : la coup 'a1' consiste ainsi à poser un pion dans le coin en haut à gauche de l'othellier ;
5. à chaque instant de la partie, le nombre de pions de chaque joueur doit être affiché ;
6. l'ensemble des coups possibles pour le joueur ayant le trait doit être affiché ;
7. le programme doit détecter lui-même la fin de la partie, et indiquer le vainqueur ; le programme devra également gérer la possibilité de recommencer une partie, et éventuellement de changer la couleur des joueurs (en d'autres termes, il ne faut pas que le programme soit relancé pour effectuer ces opérations) ;
8. une fonction pour annuler un (ou plusieurs) coup(s) doit être disponible ; une fonction pour « désannuler «doit être disponible (cela peut être utile si vous voulez étudier une partie entière, ce qui peut nécessiter de reculer et avancer dans la partie) ;
9. il doit être possible d'enregistrer une partie en cours dans un fichier ;
10. il doit être possible de charger une partie à partir d'un fichier : dans ce cas, une fois la partie chargée, on doit être capable d'annuler, si on le désire, tous les coups de la partie.

1.2 Programmation modulaire

D'un point de vue technique, le programme devra comporter des modules différents dont, par exemple :

1. un module pour la gestion (pose d'un pion, retournement d'un pion ou d'un rangée de pion, remise à zéro) de l'othellier ;
2. un module pour les opérations d'enregistrement et de sauvegarde des parties ;
3. un module pour l'affichage et les opérations d'entrée/sortie au clavier (pour l'entrée des coups par les joueeurs) ;
4. un module pour la gestion de la partie (détection de la fin de la partie, gestion du joueur dont c'est le tour de jouer, annulation d'un coup ou de plusieurs).

Bien sûr, vous pouvez décider de choisir des modules complètement différents. Quel que soit votre choix, vous devrez néanmoins être capables d'expliquer et motiver dans votre rapport final votre découpage modulaire.

En aucun cas évidemment, un programme fait d'un seul bloc ne pourra être considéré comme acceptable. De plus, afin de pouvoir tester aisément la réalisation efffectuée, un makefile devra être fourni et la commande 'make' permettra de construire l'exécutable du projet.

2 Othello

2.1 Règles du jeu

Othello est un jeu de réflexion à deux joueurs qui se joue sur un damier de 64 cases, et où les pions sont de deux couleurs différentes. Le but d'une partie est d'avoir au final le plus grand nombre de pions de sa couleur posés sur le damier.

Les règles de ce jeu peuvent se trouver partout sur internet, aussi on se contente de rappeler la position de début de partie et un exemple de cinq coups consécutifs et les damiers correspondant sur la Figure 1

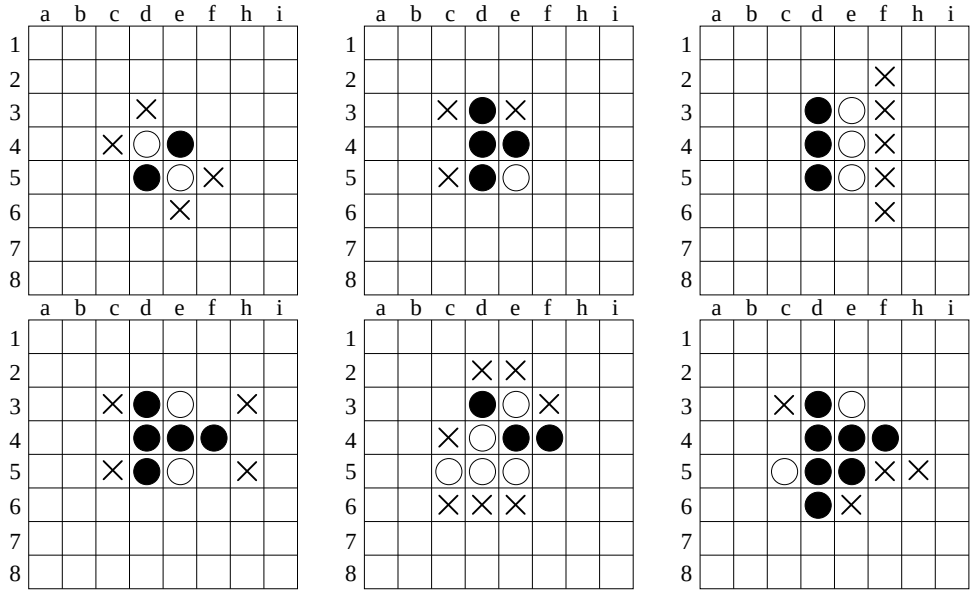

Fig. 1 – 6 premières positions d'une partie d'Othello. La position initiale du jeu est celle en haut à gauche et c'est le joueur noir qui débute la partie. Le premier coup de noir est 'd3', la réplique de blanc est 'e3', puis noir joue en 'f4' et blanc en e5", et le dernier coup joué par noir est 'd6'.

Les deux règles principales du jeu sont que

- les adversaires jouent alternativement en posant un pion de leur couleur sur l'othellier ; un joueur qui ne peut jouer passe son tour ;
- pour pouvoir poser un pion et retourner des pions adverses, le joueur qui a le trait doit nécessairement choisir une case vide qui est adjacente à un pion de la couleur adverse et qui est telle qu'une fois le pion posé dedans, ce pion doit encadrer avec un autre pion de la même couleur une ou plusieurs rangées de pions adjacents adverses (l'adjacence se définit verticalement, horizontalement et en diagonale).

2.2 Othello et l'informatique

Depuis 1995 et le programme Logistello de Michael Buro, l'ordinateur est (beaucoup) plus fort que l'humain au jeu d'Othello. Des serveurs informatique dédiés (type GGS[1]) où les programmes s'affrontent continuellement jour et nuit sont donc désormais les lieux où les meilleures parties se jouent.

Un objectif respectable pour ce projet est donc (à moins que vous ne soyez champions confirmés d'Othello vous-mêmes) de faire un programme qui, à terme, vous batte systématiquement.

3 Structures de données

Dans tout projet informatique, une des choses à faire avant de se lancer dans la programmation proprement dite est de décider d'une façon de représenter les objets du monde réel informatiquement. Cette phase est celle du choix des structures de données. Le choix est laissé libre à chaque groupe de ses structures de données. Cette section suggère néanmoins une manière communément utilisée de représenter un damier.

On notera que l'usage de struct{} est vivement encouragé (i.e. obligatoire) !

3.1 Othellier

Une ébauche de structure pour représenter l'Othellier est la suivante

```
const int MAX_CASE   = 100;
const int NOIR       = 0;
const int BLANC      = 1;
const int VIDE       = 2;
const int BORD       = 3;
const int NB_JOUEURS = 2;
typedef struct{
  int case[MAX_CASE];        /* représente le damier proprement dit        */
  int materiel[NB_JOUEURS];  /* stocke le nombre de pions de noir et blanc */
  ...
} Othellier;
```

Ici le choix est fait de réprésenter le damier comme un tableau à une seule dimension et faisant 100 cases au lieu de 64. Les 36 cases sont des cases qui représentent un bord 'virtuel' du plateau (cf. Figure 2) et permettent de ne pas avoir à multiplier les tests lorsque la validité d'un coup est évaluée (cela se comprendra mieux à l'usage). Rien ne vous empêche évidemment de faire un tableau à double entrée de taille 8×8.

Une bonne pratique en programmation est de donner des noms aux constantes. Ainsi, si othellier est une variable de type Othellier, othellier.case[32]=VIDE permet de dire qu'on veut que la case 32 de l'othellier vaut 2, c'est-à-dire que cette case est vide. Toutes les cases correpondant au bord auront donc une valeur égale à BORD (cf. Figure 2). Pour savoir le nombre de pions que possède le joueur noir, il suffit d'accéder à othellier.materiel[NOIR].

3.2 Autres structures de données

Pour chaque structure de données que vous déciderez vous devrez vous interroger de son opportunité, aussi bien du point de vue informatique que du point de vue réel. Par exemple, même si cela peut paraître dans un premier temps assez tentant de proposer une structure de données de type struct case {...} pour représenter une case d'un othellier, cette modélisation n'est pas optimale aussi bien du point de vue informatique que réel :

[1] Pour ceux intéressés, une recherche sur internet devrait vous fournir des informations.

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

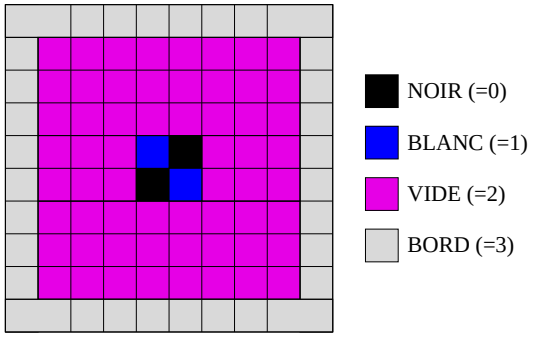

Fig. 2 – A gauche, le damier et les indices qui lui correspondent dans le tableau à une dimension de 100 cases, le bord virtuel est grisé. A droite, le damier qui correspond à la position initiale de jeu en mettant dans chaque case la valeur qui correspond à la couleur (NOIR, BLANC, VIDE ou BORD) l'occupant.

- les seules opérations que l'on peut faire avec une case sont la vider et y mettre un pion, ce que l'on peut par ailleurs effectuer très simplement en utilisant une structure de type othellier telle que celle proposée ci-avant, l'inclusion d'un telle structure ne ferait qu'ajouter de la complexité au programme ;
- et, par ailleurs, on considère plus généralement un othellier (ou n'importe quel damier) dans son entièreté plutôt que comme une adjonction de plusieurs cases.

4 Programmation modulaire et travail de groupe

La programmation modulaire consiste, comme son nom l'indique à découper un projet informatique en plusieurs modules qui se définissent par les fonctionnalités qu'ils offrent. Comme dit précédemment, la gestion du damier et les entrée/sortie au clavier ou dans des fichiers sont des secteurs différents du programme. Il serait donc raisonnable de disposer d'un moyen de séparer les différents morceaux de code ayant trait à ces fonctionnalités.

Pour ce faire, le langage C, comme la plupart des langage de programmation, permet de séparer un programme en plusieurs fichiers, chacun se rapportant à un type de tâche.

4.1 Fichiers en-tête (header)

Avec un minimum d'expérience en C, on sait très vite qu'il y a souvent des instructions qui se placent en tête d'un programme et qui prennent la forme suivante :

```
#include<stdio.h>
#include<stdlib.h>
```

Ces instructions sont des directives qui précisent au compilateur, qui va s'occuper de transformer le programme en langage C en langage machine (comprenez en langage binaire), que certaines des fonctions que vous allez utiliser n'ont pas été écrites par vous-même mais d'autres programmeurs et que les signatures de ces fonctions se trouvent dans les fichiers stdio.h et stdlib.h. (La signature d'une fonction est composée de son nom, ou identifiant, la liste et types de paramètres qu'il est nécessaire de lui fournir pour l'appeler et son type de retour.) De façon simplifiée, ces instructions disent donc au compilateur de ne pas s'étonner de ne pas trouver explicitement la déclaration et la définition de certaines fonctions que vous utilisez dans votre programme mais que ces fonctions existent par ailleurs et qu'elles sont bien programmées.

Pour faire vos modules, vous allez écrire vous-même vos fichiers en-tête. Un fichier correspondra à chacun des modules fonctionnels que vous aurez identifiés. Chaque fichier contiendra les signatures de

fonctions que vous programmerez et, éventuellement la déclaration des types de données (les struct). Par exemple, si l'on veut faire un module pour la gestion de la sauvegarde et du chargement de parties sur et du disque dur, on créera un fichier en-tête iodisque.h du type suivant :

```
#ifndef __IO_DISQUE__
#define __IO_DISQUE__

/* fonction qui sauvegarde une partie sur le disque et qui
renvoie 1 si l'opération a réussi et 0 sinon
*/
int sauvegarde(struct partie *partie, char *nomfichier);

/* fonction qui charge une partie du disque et qui
renvoie 1 si l'opération a réussi et 0 sinon
*/
int charge(struct partie *partie, char *nomfichier);


#endif
```

qui spécifie le type de fonctionnalités qui seront offertes par le module de lecture/écriture sur disque. Un header est donc une sorte de contrat qui dit quelles sont les fonctions qui peuvent être utilisées et ce qu'elles font. Pour que ces fonctions puissent être utilisées dans un fichier toto.c, il est nécessaire que ce fichier ait une ligne du type #include "iodisque.h". Par conséquent, une première phase du projet doit être un processus de réflexion quant aux différentes fonctions que vous devrez écrire. Une fois listées et catégorisées selon le module dont elles sont censées faire partie, elles sont rangées par fichier en-tête (nous considérerons pour ce projet qu'il y aura un fichier en-tête par module).

Comme n'étant que des contrats, ces fichiers header ne font rien que signifier qu'il est possible d'utiliser les fonctions qui y sont déclarées. Mais le vrai travail est de programmer ou définir les fonctions en tant que telles, c'est-à-dire d'écrire le code qui permet d'assurer que les fonctions font effectivement ce qu'elles promettent de réaliser. A chaque fichier en-tête on associe par conséquent un (ou éventuellement plusieurs) fichier(s) qui implémentent les fonctions. Dans ce projet, on écrira le code qui correspond au fichier titi.h dans le fichier titi.c. Et donc, dans le cas du fichier iodisque.h, on créera un fichier iodisque.c qui contiendra par exemple les lignes suivantes :

```
#include "titi.h"

#include <stdio.h>

int sauvegarde(struct partie *partie, char *nomfichier){
  int flag = 0;
  FILE *id=fopen(nomfichier);
  ...
  .../* écriture dans le fichier nomfichier */
  ...
  fclose(id)
  return flag;
}

int charge(struct partie *partie, char *nomfichier){
  int flag = 0;
  FILE *id=fopen(nomfichier);
  ...
```

```
  .../* lecture dans le fichier nomfichier */
  ...
  fclose(id)
  return flag;
}
```

L'intérêt de la séparation du programme en plusieurs modules est qu'une fois que les participants d'un projet ont décidé des fonctions qu'ils devaient programmer et qu'ils les ont réparties en modules (pour nous, un module correspond à un fichier en-tête .h et son fichier .c), la charge d'un module particulier peut être dévolue à un groupe particulier qui, sachant exactement ce que doivent faire les fonctions qu'il doit programmer, peut travailler (i.e. programmer son module) indépendamment de ce qui se passe au sein des autres modules. Le projet est ainsi découpé en plusieurs parties qui peuvent avancer de façon (presque) indépendante.

Même si le projet se fait individuellement dans notre cas, il est important de mettre en oeuvre cette modularité, car celle-ci présente de toute manière des avantages pour la structuration du code, la débogage, la maintenance, etc.

Cette section décrit brièvement comment mettre en œuvre la modularité en C. En d'autres termes, cette section explique comment créer des boîtes à outils dans lesquelles on peut piocher des fonctions d'intérêt. La section suivante donne des liens sur la façon dont on peut exploiter ces modules et les intégrer dans un programme proprement dit.

4.2 Compilation séparée et Makefiles

Une fois les modules programmés, il est nécessaire de disposer d'une procédure pour pouvoir les utiliser dans le programme principal, que nous appellerons othello.c. Pour cela, il est évidemment nécessaire d'insérer les #include idoines au début de othello.c de telle sorte que son contenu ressemble à :

```
#include "iodisque.h"
#include "intelligenceartificielle.h"
#include "ioclavier.h"

#include <stdio.h>

....

int main(int argc, char *argv[]){
  ...
  ...
  ... /* programme principal                    */
  ... /* qui fait appel aux fonctions déclarées */
  ... /* dans les fichiers en-tête              */
  return 1;
}
```

Mais demeure la question de la compilation de ce projet en un seul programme à partir des multiples fichiers .c et .h. L'intérêt du fichier Makefile et de l'utilitaire make est précisément d'aider à cette tâche. En particulier, ces deux outils (bien que le seul programme soit en fait make, mais puisqu'il fonctionne avec un fichier qui s'appelle généralement Makefile, ce dernier est également considéré comme un outil), permettent de ne compiler que les parties qui l'exigent : si un module a été modifié depuis la dernière compilation complète du projet, alors make et le Makefile associé travailleront de concert pour ne recompiler que le module modifié et en intégrer les changement au programme principal.

Pour ce projet, nous nous limiterons à l'écriture de fichiers Makefile assez simples et vous êtes donc invités à consulter le tutoriel très simple de l'url suivante : `http://gl.developpez.com/tutoriel/outil/makefile/`. Ce document ne s'étendra pas plus outre sur l'écriture de Makefile.

5 Tableau de marche indicatif

Voici quelques étapes d'avancement que vous pouvez essayer de respecter pour que votre projet avance de manière satisfaisante sans que vous ayez à y consacrer plus de 2/3 heures par semaine. (Chaque étape correspond à peu près à une semaine.)

1. compréhension du jeu Othello, tests de Makefile avec ceux du tutoriel et vos propres modules
2. réflexion sur les structures de données que vous aurez jugées essentielles
3. réflexion sur les modules à écrire,
4. programmation des modules
5. mise en commun des modules et écriture du programme principal