PG108 - Unix / Langage C
Projet : Résolution de Mastermind

Yannick Bornat - yannick.bornat@enseirb-matmeca.fr
Malek Ellouze - malek.ellouze@enseirb-matmeca.fr
Rémi Giraud - remi.giraud@enseirb-matmeca.fr
Jean-Charles Henrion - jhenrion@enseirb-matmeca.fr

2023 - 2024

1 Introduction

Le projet consiste à écrire un programme capable de jouer au Mastermind. Dans un premier temps, nous nous contenterons d'écrire les fonctions de gestion, d'affichage, de proposition de combinaison. Dans un second temps, nous nous attaquerons à l'algorithme de résolution automatique par le programme.

Règles du jeu. Les règles du jeu Mastermind sont disponibles notamment ici [1]. Dans notre cas, on considérera d'abord le cas standard avec 6 valeurs possibles, 4 cases et 12 tentatives.

Structure du code. Ce projet devra être implémenté dans un seul fichier source nommé *mastermind.c*. Commentez votre code, vos fonctions, vos tests, en expliquant vos choix lorsque cela s'avère pertinent (quand vous avez choisi de créer une fonction auxiliaire, choisi un type de données ou qu'il vous était laissé le choix d'une solution pour répondre à un problème).

Rendu. Ce projet sera évalué par un rendu de votre code à chaque fin de séance via thor [2]. Il vous est donc fortement conseillé de sauvegarder les versions intermédiaires de votre programme lorsque vous avez réussi une étape importante. Vous rédigerez également un court rapport pour lequel vous est fourni un exemple de trame [3].

2 Mise en place

Dans cette partie, l'objectif est d'avoir une bonne compréhension des règles, d'appréhender les structures qui vont nous permettre de coder le programme, de mettre en place un affichage correct du déroulement de la partie dans le terminal, pour finalement permettre à l'utilisateur de jouer contre la machine.

2.1 Structure

Le programme sera contenu dans un seul fichier source appelé *mastermind.c*.

On aura une fonction `main` qui contiendra la boucle principale des tests de propositions, les choix d'options (joueur ou machine qui propose, mode de jeu, lecture depuis un fichier, etc).

1. `https://fr.wikihow.com/jouer-au-Mastermind`
2. `https://thor.enseirb-matmeca.fr/ruby`
3. `https://remi-giraud.enseirb-matmeca.fr/teaching/course.php?course=PG108`

Pour les paramètres du jeu, on peut passer par des constantes de preprocessing (ou macros) :

```
#define VALUE 6
#define PIN 4
#define CHANCES 12
```

Pour les combinaisons, on peut soit manipuler des entiers positifs, donc naturellement stockés dans des tableaux d'`unsigned char` avec des valeurs strictement inférieures à `VALUE` :

```
unsigned char solution[PIN] = {2,2,0,3};
```

On peut aussi les manipuler sous forme de chaînes de caractères. Le choix d'une solution vous appartient. Aucune n'est strictement meilleure que l'autre.

```
char solution[PIN+1] = {'2','2','0','3','\0'};
```

Enfin, la fonction `main` aura une structure semblable à celle-ci :

```
int main(int argc, char agrv[]) {
  //ask game mode to the player
  //define a solution (random, from file, player entry, ...)
  //variable initialization
  for (int i=0; i<CHANCES; i++) {
    //display remaining number of chances
    //get a proposition
    //compare proposition to solution
    //get and display result
    //if (proposition equals the solution)
      break; //victory
  }
  //display winning or loosing message
  return 0;
}
```

2.2 Affichage

Avant d'avoir un programme totalement fonctionnel, nous allons d'abord travailler sur l'affichage d'une partie pré-enregistrée, pour ensuite développer les briques de base.

Des fichiers contenant le déroulé de parties sont mis à votre disposition ici[4] : *game_1.txt* (gagnante), *game_2.txt* (perdante). On rappelle qu'on peut donner un argument lors de l'appel du programme avec `./prog fichier`

game_1.txt :

```
5524
1122
X___
1300

____
1525
XXO_
1115
O___
5524
XXXX
```

La première ligne contient la solution.
Puis les lignes suivantes alternent proposition et résultat.
Pour le résultat (affiché indépendamment de la position) :
`X` : bonne valeur bien placée
`O` : bonne valeur mal placée

Le fichier s'ouvre (en lecture) avec `fopen` et on peut facilement lire ligne par ligne avec `fscanf`.

- Adaptez le `main` pour lire dans les fichiers et faire les affichages simulant le déroulement d'une partie (essais restants, test de victoire, message de fin, etc.)

4. https://remi-giraud.enseirb-matmeca.fr/teaching/course.php?course=PG108

2.3 Vérifications

Comme pour tout programme, on doit le sécuriser en anticipant des valeurs qui ne seraient pas conformes. Dans le cas du jeu, cela peut en effet arriver d'entrer une mauvaise valeur. Dans le fichier *game_3_with_fail.txt* (gagnante), des mauvaises combinaisons ont été entrées.

- Adaptez le programme pour vérifier à chaque étape à l'aide d'une fonction `verif_proposition` que la proposition entrée est valide. Si ça n'est pas le cas, on permet d'entrer une nouvelle proposition sans décompter un essai.
- Votre programme fonctionne-t-il sur le fichier *game_3_with_fail.txt* ?
- Attention, êtes-vous sûr de ce qui se passe quand vous tentez d'écrire avec `fscanf` une ligne de plus de 4 caractères dans un tableau de 5 octets ? Proposez une solution pour s'assurer que la lecture n'engendre pas de problème.

2.4 Calcul du résultat

Maintenant que l'affichage et la vérification sont fonctionnels, une première brique à implémenter est celle du calcul du résultat. Celui-ci peut sembler trivial mais il faut faire attention à quelques subtilités.

On compare donc 2 tableaux d'entiers de taille `PIN` avec des valeurs comprises entre 0 et `VALUE-1`. Pour les valeurs identiques et bien placées on affichera un X, puis pour les valeurs identiques à une mauvaise position on affichera O. Attention, les valeurs dans la proposition ou la solution ne doivent être considérées qu'une seule fois. On fera donc en sorte qu'une valeur déjà bien positionnée ou présente à une autre position ne soit plus comptée.
Par exemple : la solution [2,3,0,0] et la proposition [0,3,3,5], donnent comme résultat [X,O,_,_].

- Affichez le résultat depuis une fonction `compute_result` qui le calcule depuis la solution et une proposition.
- Obtenez-vous exactement les mêmes résultats que dans la ligne correspondante des fichiers ?
- Prenez soin d'expliquez votre algorithme dans le rapport.

2.5 L'utilisateur joue

À présent pour pouvoir jouer au Mastermind, il suffit de pouvoir générer une solution aléatoire et de pouvoir lire les propositions depuis l'entrée standard.

La solution aléatoire peut se créer facilement avec la fonction `random` et l'opérateur modulo. Quant à la proposition, celle-ci peut s'obtenir avec `scanf` ou la même fonction `fscanf` qu'utilisée précédemment pour lire dans le fichier mais en considérant l'entrée standard `input`. On pensera à appeler la fonction de vérification `verif_proposition`. Pour le choix du mode de jeu, on considérera qu'en l'absence d'argument, c'est au joueur d'entrer les propositions, sinon on lit le déroulé depuis le fichier.

- La solution varie-t-elle à chaque exécution du programme ? Est-ce normal ?
- Vérifiez bien que vous n'avez pas de bugs dans le calcul des résultats (comparez avec les fichiers d'exemple).

3 Résolution par le programme

À présent on va s'intéresser à l'algorithme de résolution du jeu par la machine. Pour tester les deux modes de jeu (l'utilisateur devine une combinaison aléatoire, et la machine résout une combinaison proposée par l'utilisateur), on demandera à l'utilisateur de choisir en début de programme. On demandera également quel type de résolution par la machine on souhaite.

3.1 Propositions aléatoires

Pour déjà tester votre fonction *main*, on peut faire jouer la machine de manière aléatoire (fonction `machine_random`). La programmation de cette fonction est assez directe.

- Le programme arrive-t-il à gagner une partie ?
- Après combien d'essais ?
- Quel est le nombre total de combinaisons possibles ?

3.2 Résolution optimale

Nous attaquons enfin le gros du travail algorithmique. Tout d'abord, sachez qu'il est toujours possible de gagner au Mastermind, dans le cas de 6 couleurs, 4 cases, en 5 coups ou moins ! L'algorithme de résolution de Knuth, appelé "five guess", est décrit en détails ici [5].

Idée générale

On part d'une liste de candidats solutions (toutes les combinaisons), et à chaque proposition faite au jeu, on élimine des candidats jusqu'à ce qu'il n'en reste qu'un.

Pour une proposition donnée qui a obtenu un résultat ou score, par exemple [X,O,O,_], tous les candidats valides sont ceux de la liste qui ont le même résultat lorsque comparés à la dernière proposition (puisque cette liste doit contenir la solution). Cela permet d'éliminer progressivement des candidats. On peut alors donner comme nouvelle proposition un candidat sélectionné aléatoirement dans la liste et on itère (3.2.2).

Pour aller plus loin, on peut proposer des combinaisons, pas nécessairement parmi la liste de candidats mais qui permettent de converger encore plus rapidement. Pour cela, on va chercher la proposition qui, dans le pire cas (pour un prochain score obtenu), nous donnera un nombre limité de candidats restants (3.2.3).

3.2.1 Calcul du score

Tout d'abord, pour quantifier plus facilement la notion de résultat, on introduit le calcul d'un score. Ce score se calcule entre deux combinaisons, par exemple la solution S et une proposition P, de la manière suivante :

$$score(S, P) = 10 * |\mathrm{X}| + |\mathrm{O}| \tag{1}$$

avec |X| le nombre de bonnes valeurs bien placées, et |O| le nombre restant de bonnes valeurs mal placées. Par exemple un résultat [X,O,O,_] donne un score de 12. L'ensemble des scores possibles avec 4 cases est logiquement s = {0, 1, 2, 3, 4, 10, 11, 12, 13, 20, 21, 22, 30, 40}. Le score maximal de 40 est donc le critère de succès de la partie.

- Implémentez le calcul de score dans une fonction `compute_score`.
- Affichez les scores à côté des résultats pour vous permettre de vérifier votre fonction.

5. `https://www.apmep.fr/IMG/pdf/10-Mastermind_C.pdf`

3.2.2 Élimination de candidats

La deuxième façon de jouer consiste à éliminer progressivement des combinaisons en partant de l'ensemble des combinaisons possibles. À chaque proposition faite et score obtenu (comparaison à la solution), on ne garde que les candidats qui obtiennent le même score avec la proposition.

Comment coder cette liste de candidats ? Le plus simple est sans doute de ne pas créer une structure contenant toutes les combinaisons mais seulement de manipuler un tableau d'indices entiers de taille le nombre de combinaisons possibles. Si une combinaison correspondant à un indice fait partie des candidats, on met la valeur de la case à 1, 0 sinon. Selon comment vous manipulez le parcours du tableau, par exemple avec plusieurs boucles et une variable d'indice incrémentée, il peut être très facile d'avoir accès à l'ensemble des combinaisons tout en ayant accès à la valeur de l'indice correspondant dans le tableau 1D. Une telle correspondance est illustrée dans le tableau de la section suivante.

Remarque : Le nombre de combinaisons possibles restant limité, on peut se permette cette gestion exhaustive et le parcours de toutes les combinaisons, même si le nombre de candidats diminue fortement avec les itérations. Une autre solution serait d'utiliser les listes chaînées... Libre à vous de proposer une méthode différente (à expliquer dans le rapport).

Algorithme 1 : Élimination de candidats

```
Initialiser à 1, un tableau CAND de longueur le nombre de combinaisons possibles
P = proposition aléatoire
s = score(S, P)      %avec S la solution
essai = 1
tant que (s != 40) ET (essai < Nbr_essai)
    Mettre à 0 les cases dans CAND pour les combinaisons C telles que score(C,P) ≠ s
    P = proposition aléatoire (ou la première, la dernière) parmi CAND (case à 1)
    s = score(S,P)
    essai = essai + 1
```

- Implémentez l'algorithme dans une fonction appelée `machine_eliminate`.
- Vérifiez qu'à chaque nouvelle proposition, la condition sur le score soit respectée.
- La machine arrive-t-elle à gagner ?

3.2.3 Choix de la meilleure proposition

On peut encore faire mieux ! Pour cela on peut on peut proposer des combinaisons, pas nécessairement parmi la liste de candidats mais qui permettent de converger encore plus rapidement. Pour cela, on va chercher la proposition qui, dans le pire cas (pour un score obtenu), nous donnera un nombre limité de candidats restants.

Comment obtenir cette proposition ? On va calculer pour toutes les combinaisons possibles, le score obtenu avec chaque candidat restant, et compter le nombre de candidats obtenant le même score. Pour cela on va manipuler un tableau de taille Nbr_combinaisons x Nbr_scores.

Une fois ce tableau obtenu, pour chaque combinaison (chaque ligne) on va retenir le plus grand nombre, c'est-à-dire, si on prend cette combinaison comme nouvelle proposition, et qu'on obtient un score, il nous restera dans le pire cas ce nombre de candidats ayant ce même score. On choisit alors naturellement comme proposition la combinaison qui aura le plus petit nombre de candidats restants dans le pire cas. Le nouveau score obtenu nous permettra d'éliminer à nouveau des candidats de la liste et on répète le procédé jusqu'à obtenir un seul candidat restant.

Pour plus de simplicité on peut manipuler un tableau organisé de la manière suivante, de taille Nbr_combinaisons x (score_max+1). On aura nécessairement des colonnes avec des scores impossibles mais cela ne perturbera pas la suite de l'algorithme.

		scores														
combinaison	i	0	1	2	3	4	5	6	...	10	11	12	13	14	...	40
[0,0,0,0]	0						0	0						0		
[0,0,0,1]	1						0	0						0		
[0,0,0,2]	2						0	0						0		
[0,0,0,3]	3						0	0						0		
[0,0,0,4]	4						0	0						0		
[0,0,0,5]	5						0	0						0		
[0,0,1,0]	6						0	0						0		
...	...						0	0						0		
[5,5,5,5]	1295						0	0						0		

Dans le cadre d'une résolution optimale, l'algorithme de recherche de meilleure proposition vient se substituer aux lignes 2 et 7 de l'Algorithme 1. Celui-ci peut être résumé comme suit :

Algorithme 2 : Recherche de meilleure proposition

```
1: Si un seul candidat C restant
2:     Retourner C
3: Créer un tableau TAB de 0 de taille Nbr_combinaisons x (score_max+1)
4: i = 0
5: Pour chaque combinaison P possible
6:     Pour chaque candidat C restant
7:         s = score(P,C)
8:         TAB[i,s] = TAB[i,s] + 1
9:     i = i + 1
10: Créer un tableau MAX de taille Nbr_combinaisons
11: i = 0
12: Pour chaque combinaison P possible
13:     MAX[i] = maximum de la ligne TAB[i, :]
14: Retourner P = combinaison correspondant au minimum de MAX
```

- Implémentez l'algorithme dans une fonction appelée `machine_optimal`.
- Quels doivent être les types des tableaux TAB et MAX ?
- Celui-ci fait-il mieux que précédemment ?

3.2.4 Comparaison nombre de coups

À ce stade, vous disposez donc de 3 méthodes de résolution différentes. On peut comparer ces méthodes tout d'abord en termes de nombre de coups moyen et de nombre de coups maximum sur un ensemble de parties. On rajoutera donc une boucle à notre programme pour tester sur un nombre suffisant de parties avec pour chacune une solution générée aléatoirement.

- Complétez le tableau suivant. Est-ce que vos résultats vous semblent cohérents ?

Nombre de coups	Aléatoire	Élimination	Élimination + Meilleure proposition
Moyen			
Max			

4 Pour aller plus loin

Liste non exhaustive d'améliorations possibles qui peuvent être développées en parallèle.

4.1 Temps moyen de jeu (+)

Que ce soit pour l'utilisateur ou pour la machine, on peut également calculer le temps moyen de jeu pour chaque partie. Après chaque partie, affichez le temps mis par la machine ou le joueur ainsi que la moyenne si vous effectuez plusieurs parties.

- Ajoutez une ligne au tableau de la section 3.2.4.

4.2 Couleur (+)

Pour un aspect visuel plus poussé, on peut intégrer de la couleur aux affichages. Selon les systèmes, il peut exister plusieurs façons de faire. Sous linux, pour modifier la couleur d'affichage, par exemple dans un `printf` depuis du code C, on utilise la balise `\033[1;<colorcode>m]`. Une liste des codes couleur est disponible ici [6].

4.3 Nombre de couleurs (+)

Le nombre de chiffres/couleurs a été fixé à 6 (`VALUE`) comme dans le jeu original. Si ça n'est pas déjà le cas, adaptez votre code pour pouvoir modifier ces valeurs et pouvoir jouer au jeu uniquement en modifiant la valeur de `VALUE`.

- Quelles sont les limites potentielles de cette valeur ?

4.4 Chiffres → lettres (++)

Vous pouvez remplacer les chiffres (0,1,2,3,4,5) par des lettres :

Y (jaune), R (rouge), G (vert), B (bleu), W (blanc), K (noir)

L'utilisateur ne tapera donc plus des chiffres mais des lettres. Ce changement va entraîner quelques modifications dans le code puisque l'on passe d'entiers ou de caractères contigus et facilement convertibles en entiers, à des caractères.

- Expliquez la façon de procéder que vous avez choisie et pourquoi.

4.5 Nombre de cases (+++)

Même modification ici pour pouvoir jouer avec un nombre de cases `PIN` différent. Ce changement peut s'avérer plus problématique selon comment a été construit votre programme. Si vous avez la présence de multiples boucles pour parcourir la liste de candidats, vous aurez sans doute besoin de créer la fonction `ind2base_sequence` qui considère un indice (décimal) et retourne la combinaison correspondante (dans la base définie par `VALUE`). Il s'agit de la fonction qui vous fait passer de la 2ème à la 1ère colonne du Tableau de la section 3.2.3. Pour construire cette fonction, vous pouvez vous inspirer de l'algorithme de conversion d'un entier décimal en binaire.

- Quelles sont les limites potentielles de cette valeur ?
- Pour quelle(s) partie(s) du programme cela pourrait-il poser problème ?

4.6 D'autres idées ?

Libre à vous de proposer d'autres améliorations de votre programme (log avec profil, high score, indices donnés au joueur, temps limite pour jouer, liste chaînée de candidats, ...).

6. `http://sdz.tdct.org/sdz/des-couleurs-dans-la-console-linux.html`