Ecole internationale des Sciences du Traitement de l'information

INFORMATIQUE

Mastermind

Auteurs :
M. Johan Moncoutié
M. Baptiste Urgell
M. Luca Ordronneau

Encadrant :
Pr. Romain Dujol

Version du
18 janvier 2019

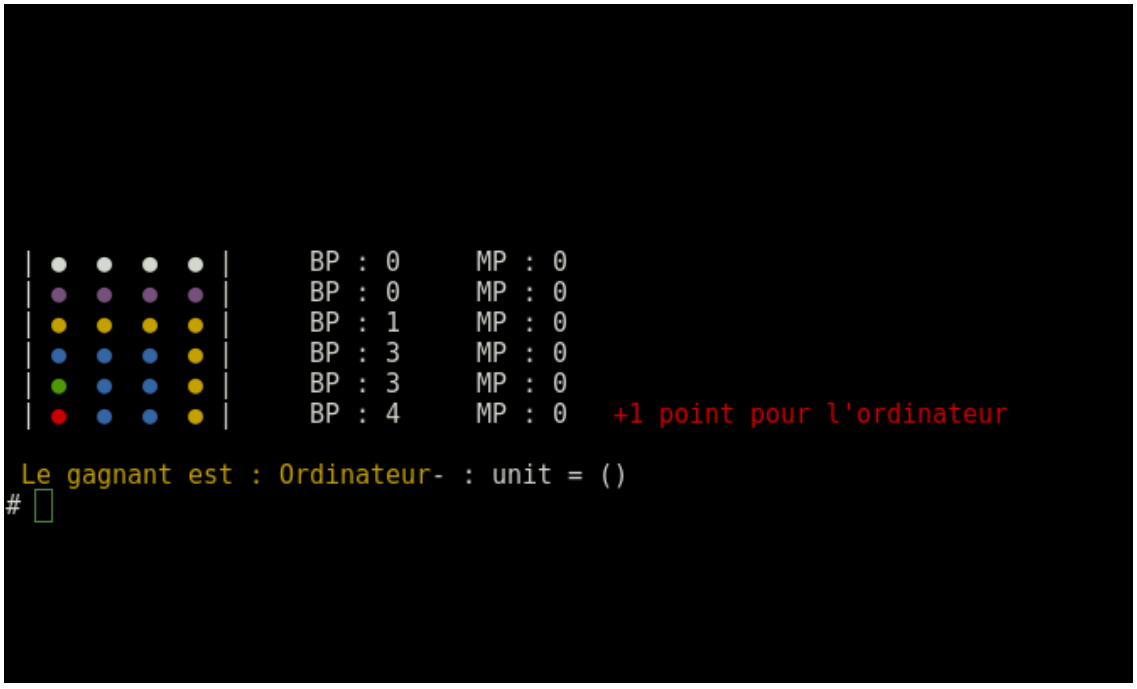
Table des matières

1 Introduction

L'objectif de ce projet est de coder le jeu de société : Mastermind. Ce jeu de réflexion et de déduction se joue à deux joueurs. Le principe est simple :

- Un des joueurs choisit un code de quatre pions à l'aide de six couleurs (il est possible d'utiliser plusieurs fois la même couleur), sans le montrer à son adversaire.
- L'autre joueur a dix tentatives pour deviner le code. À chaque tentative, le créateur du code donne le nombre de pions bien placés ainsi que le nombre de pions mal placés dans la proposition par rapport au code qu'il a créé.
- Si le code n'a pas été trouvé à l'issue des dix tentatives, le créateur du code remporte la partie. Sinon, l'autre joueur remporte la partie. Pour des raisons d'équité, chaque joueur doit avoir le même nombre d'occasions de créer le code : on joue donc un nombre pair de parties, en alternant à tour de rôle.

Ce jeu dispose de trois modes : PvE ("mastermind"), PvP ("mastermind2") et IAvIA ("mastermind3").
De plus, il propose quatre types d'IA différentes : naïve, naïve aléatoire, Knuth, Knuth (min min).



Simulation de l'algorithme naïf

2 Structuration du code

Nous avons structuré le code en quatre fichiers :
— code.ml
— IA.ml
— affichage.ml
— projet.ml

Deux des quatre fichiers énoncés ont été imposé par le sujet ("code.ml" et "IA.ml"). À cela, nous avons ajouté deux fichiers ("affichage.ml" et "projet.ml") pour permettre une bonne répartition du travail.
En effet, c'est "projet.ml" qui est notre fichier principal. Dans ce fichier, nous procédons de la manière suivante pour faire marcher le jeu :
— L'IA ou l'humain créée le code
— Ensuite la personne n'ayant pas créée le code fait un nombre donné de tentatives jusqu'à ce qu'il trouve sinon on passe à l'autre joueur
— La partie se termine lorsque le nombre de partie tombe à zéro, le gagnant est ensuite déterminé comme étant celui qui a remporté le plus de points

3 Choix des algorithmes

Premièrement nous avons choisi des algorithmes assez simplifiés et assez décomposés dans le sens où nous avons divisé le code en plusieurs fonctions réalisant des tâches bien précises. Sur les fichiers avec la signature imposée, cette décomposition était imposée aussi mais l'implémentation de ces fonctions nécessitaient elles mêmes des sous-fonctions.

3.1 Affichage en général

Pour l'interface utilisateur, les entrées, sorties et l'affichage en général, nous avons fait le choix d'utiliser uniquement l'interface proposé dans le terminal. Pour que l'utilisateur puisse rentrer des codes afin de jouer, il doit rentrer son code en l'écrivant de la forme : |rouge|bleu|..|, de cette façon il rentre tous les pions en même temps et peut réessayer un code en appuyant sur la flèche du haut et en modifiant les couleurs.

3.1.1 affichage.ml

Les affichages dans ce fichiers sont faits de manière simple avec les fonctions oCaml de base :
— print_string
— print_newline

3.1.2 afficher_code

Cette fonction écrit dans le terminal un code, qui est à la base une liste d'entier, sous la forme de pions avec des couleurs associées. Elle appelle la fonction string_of_code qui elle transforme le code en chaîne de caractère.

3.2 IA.ml

3.2.1 choix knuth

Le principe de la méthode a été trouvé sur internet. Elle marche sur le principe suivant, on calcule le poids de chaque code possible c'est à dire le nombre maximum de code qui ne pourront pas être le vrai code en fonction de chaque réponse grâce à la fonction poids_test et poids. Elle utilise en plus un système d'élagage pour améliorer le temps d'exécution. On pourrait encore l'améliorer en mettant le premier tour en variable lors de l'exécution du fichier.

3.2.2 filtre

Un seul filtre pour tous les choix possibles. La suppression du code essayé est fait avant le filtre-naïf pour éviter de tester tous les codes et ainsi gagner en complexité.

3.3 code.ml

3.3.1 compare

Il a été choisi de comparer les codes en fonction de la somme de leurs chiffres.

3.3.2 code_of_string

Cette fonction transforme la chaîne de caractère rentrée par l'utilisateur de la forme : |rouge|bleu|..| en liste d'entiers avec des entiers correspondants aux couleurs.

3.3.3 tous

Imaginée pour pouvoir créer toutes les compositions possibles en fonction du nombre de chiffres différents et de la longueur choisie.

3.3.4 toutes_reponses

Utilise le nombre maximum de chiffres par code pour pouvoir créer toutes les possibilités, donc fonctionne pour n'importe quelles conditions.

3.3.5 reponse

Cette fonction va comparer deux codes pour pouvoir renvoyer le nombre de pions bien placés et le nombre de pion mal placés. Dans un premier temps elle va comptabiliser le nombre de pion bien placés et les supprimer des deux codes. Ensuite avec le reste des pions, l'un des deux codes va être parcouru et pour chaque pion une recherche va être effectuée dans l'autre code et le pion supprimé s'il y est trouvé.

3.4 projet.ml

3.4.1 run_tentatives

Pour tenter un code, il y a deux fonctions, une pour l'humain et une pour l'IA. Le code est différent, cependant les deux fonctions suivent le même schéma, c'est à dire que l'on commence par faire tenter un code par l'humain (saisie manuel) ou l'IA (choix du code selon la méthode donné). Ensuite, ce code est analysé via la fonction "reponse" et selon cette même réponse soit la partie se termine s'il a trouvé le code, soit la partie continue jusqu'à ce que le joueur n'ai plus de tentatives.

3.4.2 run_parties

Pour jouer une partie, il y a deux fonctions selon le mode de jeu, une où les réponses sont calculées de manière automatique (true) et une où les réponses sont fournies par le joueur humain (false) dans le cas où c'est au joueur humain de créer le code. Elles suivent le même schéma tout de même. En effet :

- Selon le nombres de parties restantes, à chaque fin de parties les joueurs alternent grâce à l'opérateur modulo.
- Ensuite on appelle la fonction "run_tentatives" IA ou humain selon le joueur qui doit tenter les codes.

4 Répartition du travail

Dans ce projet, nous nous sommes répartis le travail en fonction des tâches qu'il nous restait à faire. Nous avons commencé le projet sans stratégie de répartition puis au fur et à mesure que le projet avançait les parties se sont peu à peu déterminées. Nous avons donc séparé le travail en 3 parties avec une partie plutôt calculatoire qui inclut les algorithmes d'IA et les calculs de réponses ou des codes possibles, une autre partie qui fait le lien entre ces calculs en les appelant au bon endroit afin de pouvoir faire des tours, des tentatives, et les divers types de fonctionnements demandés (PvP,PvIA). La dernière partie est donc les entrées-sorties prenant en compte l'affichage ainsi que les interactions humain/ordinateur.

5 Bilan du projet

Pour conclure, ce projet nous a donc appris à pouvoir séparer des tâches au sein d'un travail. De plus il a fallut transposer sur ordinateur en oCaml (langage fonctionnel) un problème mathématique en algorithme notamment avec la méthode de Knuth. La signature obligatoire de certains modules nous ont aidé sur plusieurs aspects, pour la compréhension des fonctions entre les différents membres du groupe, pour la conception des fonctions et pour la répartition du travail. Ce projet nous a apporté une expérience supplémentaire dans le travail en groupe mais aussi dans le langage oCaml.