TP1 - Autour du Mastermind

Objectifs

- Retravailler les boucles, les tests et les listes abordées en première année ;
- Comprendre un algorithme et le traduire dans un langage de programmation.

Le Mastermind est un jeu de société pour deux joueurs dont le but est de trouver un code. C'est un jeu de réflexion et de déduction, inventé par Mordecai Meirowitz en 1973, qui a connu un vif succès à sa sortie. Il a été depuis décliné en plusieurs variantes augmentant sa difficulté initiale. Ce jeu a fait l'objet de diverses études mathématiques.

Déroulement du jeu

Le jeu d'origine se joue avec un joueur appelé *codificateur* et un joueur appelé *décodeur*. Le codificateur dispose de pions de N couleurs différentes (**numérotées de 1 à** N). Il commence par placer P pions dont il a choisi les couleurs sans qu'ils soient vus de l'autre joueur à l'arrière d'un cache qui les masquera à la vue de celui-ci jusqu'à la fin de la partie. Il doit prendre soin de ne pas révéler la configuration qu'il a choisie, c'est-à-dire la couleur et la position des pions. Rien ne l'empêche de choisir plusieurs pions d'une même couleur dans une même configuration.

Le décodeur doit trouver quels sont les P pions choisis par le codificateur, c'est-à-dire leurs couleur et position. Le décodeur propose une configuration : il place P pions dans les trous de la première rangée la plus proche de lui, voir figure (a) ci-dessous. Une fois les pions placés, le codificateur indique :

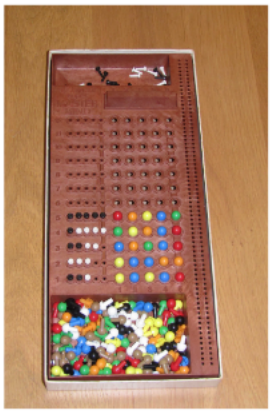
- le nombre de pions de la bonne couleur bien placés en utilisant le même nombre de fiches noires à côté de la rangée proposée par le décodeur ;
- puis le nombre de pions de la bonne couleur, mais mal placés, en insérant le même nombre de fiches blanches à côté des fiches noires.

S'il n'y a aucune correspondance, le codificateur ne met aucune fiche.

Ces deux nombres constituent le score de la configuration proposée par le décodeur.

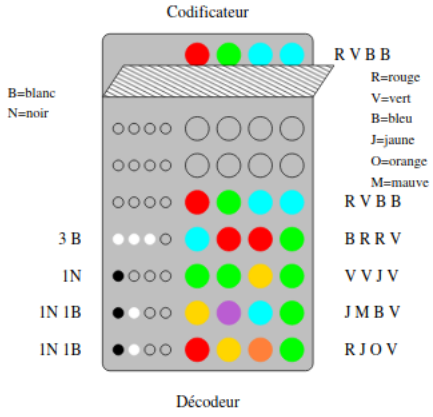
Une fois que le codificateur a indiqué le nombre de fiches noires et le nombre de fiches blanches associés à la configuration proposée par le décodeur, ce dernier propose une nouvelle configuration. Le processus se répète jusqu'à ce que le décodeur ait trouvé la bonne configuration ou jusqu'à ce qu'il ait proposé C configurations qui ne correspondent pas à la configuration choisie par le codificateur.

La figure 1 ci-dessous donne des exemples de déroulement de parties de Mastermind. Le Mastermind classique fonctionne avec 8 couleurs et 5 positions et la configuration doit être trouvée en au plus $C = 12$ coups. **Pour simplifier, nous considérerons dans ce TP que l'on dispose de** $N = 6$ **couleurs et** $P = 4$ **positions. Vous testerez les fonctions implémentées dans ce TP avec ces valeurs, déjà définies dans des variables globales.**



(a) Vue du jeu côté décodeur

(b) Vue du jeu côté codificateur



(c) Schéma du jeu

Travail préliminaire

Récupérer le fichier Python `TP_1_mastermind.py` mis à disposition par votre professeur, puis ouvrez-le dans Spyder. Il contient l'ensemble des programmes que vous devrez compléter pendant le TP.

Première situation

Dans cette première situation, le joueur (humain) tente de découvrir une configuration de pions cachée par la machine. La machine doit d'abord commencer par générer une configuration aléatoire.

Question 1. Écrire une fonction `genere_config()` qui renvoie une liste de taille P représentant une configuration aléatoire selon une loi uniforme sur l'ensemble des configurations. On rappelle que N désigne le nombre de couleurs disponibles (numérotées de 1 à N) et P le nombre de pions. Vous pourrez utiliser la fonction `randint` de la bibliothèque `random` (déjà importée), telle que l'appel `randint(a,b)` renvoie un nombre entier aléatoire x tel que $a \leq x \leq b$.

Question 2. Écrire une fonction `nb_bien_places(c1, c2)` prenant en arguments deux configurations, et renvoyant le nombre de pions bien placés entre deux configurations, c'est-à-dire les pions de même couleur et de même position entre les deux configurations (nombre de fiches noires).

Par exemple, si `c1 = [1, 1, 3, 6]` et `c2 = [3, 1, 1, 2]`, alors votre fonction devra renvoyer 1 car seul le deuxième pion est bien placé.

Pour calculer le nombre de fiches blanches, on commence par calculer le nombre de pions en commun entre les deux configurations c_1 et c_2 (l'ordre des pions dans la configuration n'a pas d'importance). On procède de la façon suivante : si la couleur i apparaît n_i fois dans c_1 et m_i fois dans c_2, alors elle est commune $\min(n_i, m_i)$ fois dans les deux configurations. Il suffit alors de calculer la somme de ces minima pour l'ensemble des couleurs possibles.

Question 3. Compléter la fonction `nb_communs(c1, c2)` calculant le nombre de pions communs entre les deux configurations c_1 et c_2. N'oubliez pas de tester votre fonction !

Puisque le nombre de pions en commun décompte également les couleurs bien placées, il convient d'y retrancher le nombre de fiches noires pour obtenir le nombre de fiches blanches.

Question 4. À l'aide des deux fonctions précédentes, écrire une fonction `score(c1, c2)` renvoyant le nombre de fiches noires et de fiches blanches (dans cet ordre) qui sont déterminées lorsque deux configurations c_1 et c_2 sont comparées. Pour rappel, le nombre de fiches noires correspond au nombre de pions étant bien placés (et donc de la même couleur) et le nombre de fiches blanches correspond au nombre de pions de la bonne couleur qui ne sont pas bien placés.

Avec l'exemple précédent (`c1 = [1, 1, 3, 6]`, `c2 = [3, 1, 1, 2]`), votre fonction `score` devra renvoyer `(1, 2)` (un pion bien placé et trois en commun, donc deux fiches blanches).

Question 5. Compléter la fonction interactive `joueur_devine()` permettant d'effectuer une partie de Mastermind dans laquelle le codificateur est la machine et le décodeur est le joueur humain. Cette fonction comprend la génération d'une configuration à faire deviner, la saisie des propositions successives de configurations par l'utilisateur et les réponses correspondantes de la machine via les fiches noires et blanches.

Question 6. Tester la fonction précédente pour jouer (un peu) au Mastermind.

Deuxième situation

On étudie maintenant le cas où le joueur joue le rôle du codificateur et la machine celui du décodeur. Le principal atout de l'ordinateur est sans doute sa vitesse de calcul, il est donc possible d'envisager de passer en revue de manière exhaustive l'ensemble des configurations possibles, de `[1, 1, 1, 1]` à `[6, 6, 6, 6]` dans le cas du Mastermind classique.

Question 7. Observer rapidement la fonction `mystere`. Quelle est sa particularité ? Vérifier que cette fonction renvoie bien l'ensemble des configurations possibles.

Valeurs par défaut

Dans cette fonction, on a fourni des valeurs par défaut pour les deux paramètres. Cela signifie que l'appel `mystere()` (sans argument !) correspondra à une partie classique de Mastermind, où les valeurs par défaut $N = 6$ et $P = 4$ ont été choisies.

Supposons que la machine a proposé une configuration, nommée `tentative`, et a reçu en retour du codificateur le score de sa proposition (c'est-à-dire le nombre de fiches noires et le nombre de fiches blanches), stocké sous la forme d'un couple nommé `sc`. L'idée fondamentale de l'algorithme est la suivante : si le décodeur fait l'hypothèse qu'une proposition `c` est la bonne, alors nécessairement les nombres de fiches noires et de fiches blanches déterminés en comparant `c` et `tentative` doivent correspondre à `sc`. L'ordinateur peut alors éliminer toutes les configurations candidates `c` ne vérifiant pas ce critère.

☐ **Question 8.** On considère l'exemple suivant :

- les configurations candidates sont `[2, 5, 1, 4]`, `[1, 6, 4, 3]`, `[1, 1, 1, 4]`, et `[1, 4, 1, 5]` ;
- la dernière tentative était `[2, 4, 1, 1]`, pour laquelle le score obtenu est `sc = (2, 1)` (deux fiches noires et une fiche blanche).

Déterminer les configurations ne pouvant correspondre à la configuration cachée, et celles à conserver.

☐ **Question 9.** Écrire une fonction `reduction_configurations_possibles` (dont les arguments sont détaillés dans le fichier Python) conservant dans une nouvelle liste les configurations candidates pouvant correspondre à la configuration cachée.

☐ **Question 10.** Compléter la fonction `machine_devine` permettant d'effectuer une partie où la machine choisit une configuration qu'elle essaye ensuite de deviner.

☐ **Question 11.** Déterminer expérimentalement le nombre maximal de tentatives nécessaire à l'ordinateur pour gagner à coup sûr. Comparer au nombre maximal C de tentatives autorisées.

Pour aller plus loin : l'algorithme « five-guess » de Knuth

On propose de démontrer que dans le cas du jeu étudié (6 couleurs et 4 pions), la configuration secrète peut toujours être trouvée en 5 tentatives ou moins, d'où le nom « five–guess » donné à cet algorithme.

Si dans la fonction précédente on change de proposition initiale, la répartition du nombre de coups nécessaires n'est plus la même. Même si cela ne remet pas en cause le résultat précédent (réussite en 9 tentatives ou moins), cette observation suggère que la proposition initiale a une influence sur la suite. Parmi toutes les propositions candidates, y-a-t-il des choix plus judicieux que d'autres ?

Il semble assez naturel de retenir, parmi toutes les configurations candidates restantes à une étape donnée, celle (ou l'une de celles) qui minimise le nombre de candidats restants. L'idée de Donald Knuth consiste à choisir à chaque étape, une proposition qui minimise le maximum (minimax) de propositions restantes en tenant compte des différents scores possibles :

Pour chaque configuration c_1 candidate faire
    Pour chacun des scores s possibles ($s \in \{(0,0),(0,1)\ldots,(4,0)\}$) faire
        Déterminer le nombre de candidats c_2 obtenant le score s en les comparant avec c_1
    Attribuer au candidat c_1 un poids égal au plus grand des nombres précédents
Choisir comme nouvelle proposition l'une de celles ayant le plus petit poids

☐ **Question 12. (difficile)** Écrire une fonction `prochaine_tentative(config_restantes)` prenant en argument un ensemble de configuration restantes et renvoyant le candidat choisi selon cette méthode. Vous vérifierez notamment qu'avec l'ensemble de toutes les configurations possibles, votre fonction doit renvoyer la liste `[1, 1, 2, 2]`.

☐ **Question 13.** Modifier la fonction `machine_devine` pour qu'elle utilise la méthode de Knuth précédente. Vérifier alors expérimentalement que le nombre de propositions effectuées est toujours inférieur ou égal à 5. Cette méthode vous semble t-elle accessible pour un joueur humain ?