Informatique *DM*
Programmation fonctionnelle avec OCaml Ipesup

Arborescente de Monte-Carlo (MCTS) pour un mini-Go

Contexte général

On considère un jeu inspiré du *Go* sur un plateau carré (le *goban*).
Deux joueurs, Noir et Blanc, jouent à tour de rôle en posant une pierre sur une intersection libre de la grille.

Exemple

On représente le goban par une grille 3×3 (lignes 0–2, colonnes 0–2). Un point « `.` » signifie vide, « `N` » une pierre noire, « `B` » une pierre blanche.

Position initiale :

```
. . .
. . .
. . .
```

Tour 1 — Noir joue (0,0) :

```
N . .
. . .
. . .
```

Tour 2 — Blanc joue (1,1) :

```
N . .
. B .
. . .
```

On visualise l'alternance Noir/Blanc et l'effet concret des coups.

Le but, dans ce mini-cadre, n'est pas de reproduire toutes les règles officielles du Go, mais de disposer d'un environnement cohérent pour expérimenter et programmer une recherche arborescente de Monte-Carlo (MCTS) en OCaml.

Objectifs

Ce que doit faire votre programme.
Depuis l'état courant du goban, votre programme doit choisir la prochaine coup à jouer. Pour décider, il se comporte comme un *cerveau qui imagine de nombreuses parties futures* : il simule beaucoup de fins de partie possibles pour *chaque* coup initial, observe qui gagne le plus souvent, puis joue le coup estimé le plus prometteur.

État et coups.
Un *état* est donné par la position des pierres sur toutes les intersections du goban.
Un *coup* est la pose d'une pierre du joueur courant sur une intersection libre et *légale*.
Les critères de légalité exacts (captures, interdits, etc.) sont encapsulés dans des fonctions utilitaires fournies (voir dessous).

Lorsqu'aucun coup légal n'est disponible, la partie est terminée et un *vainqueur* est déterminé par la fonction `gagnant` (elle encode la règle de terminaison choisie : « plus de coups » ou « score/territoire » selon la variante).

Lecture intuitive

Gardez ce fil rouge pendant tout le DM : la MCTS ne "prédit" pas le futur, elle l'*échantillonne*. À partir de l'état présent, elle essaie beaucoup de suites de coups « plausibles », mesure combien de fois Noir ou Blanc l'emporte, et utilise ces statistiques pour *préférer* un coup de départ par rapport aux autres.

MCTS — vue d'ensemble La MCTS s'appuie sur une structure d'exploration qui enregistre les différentes *variantes de jeu* déjà testées, avec des *compteurs* (nombre d'essais, nombre de succès pour Noir) et une *préférence initiale* (`prior`). Une itération se décompose ainsi :

1. **Sélection** : à partir de l'état initial, on enchaîne des *choix successifs* parmi les coups connus, en prenant à chaque étape l'option jugée la plus prometteuse par une *valeur d'action* (définie plus en bas).
2. **Extension** : on *ajoute* à la structure d'exploration les prochains coups possibles, avec leurs priorités.
3. **Simulation (playout)** : depuis l'état courant obtenu, on simule la fin de partie en appliquant une *stratégie par défaut* jusqu'à l'arrêt, puis on détermine le gagnant.
4. **Mise à jour** : on met à jour les compteurs associés à tous les choix effectués pendant l'itération (essais +1 ; succès Noir +1 si Noir gagne la simulation).

En répétant un grand nombre d'itérations, on affine les estimations et on choisit un coup à jouer depuis l'état initial (par exemple, l'option la plus fréquemment retenue).

Ce que signifie « décision finale »

À la fin de votre budget d'itérations, vous devez effectivement jouer un coup sur le vrai plateau. On choisit la coup la plus visitée (celle qui a servi le plus souvent dans les simulations). Ensuite, la partie continue : l'adversaire répond, et vous relancez la MCTS depuis le *nouvel* état pour choisir la prochaine coup.

Types, conventions et fonctions fournies

On utilise les définitions et conventions suivantes (supposées disponibles pour les exercices) :

— **Types de base.**
```
type joueur = Noir | Blanc
type intersection = Libre | Pierre_noire | Pierre_blanche
type goban = intersection array array
type position = int * int
```
— **Taille du goban.** Le goban est un tableau `n x n`. Les indices de lignes et colonnes sont des entiers `0..n-1`. La taille `n` est fixée.
— **Joueur courant.** À tout état est associé le joueur qui doit jouer. La fonction utilitaire `js : joueur -> joueur` renvoie l'adversaire (`Noir` ↔ `Blanc`).

— **Fonctions utilitaires (fournies).**

```
(* Génération de coups et politique par défaut *)
val liste_coup_prior : goban -> joueur -> (position * float) list
(* coups légaux avec une priorité/prior $\in [0,1]$ *)

val strategie_defaut : goban -> joueur -> position
(* choisit un coup; peut lever Pas_de_coup_valide si aucun coup n'est possible *)

(* Mécanique du jeu *)
val joue   : goban -> position -> joueur -> goban
(* nouveau goban après le coup; FONCTIONNEL (pas d'effet de bord) *)

val gagnant : goban -> joueur
(* sur un goban terminal, renvoie le vainqueur : Noir ou Blanc *)
```

Remarques importantes.

— **Pureté / effets.** `joue` est *fonctionnelle* : elle renvoie un nouveau goban sans modifier l'argument. Évitez toute mutation accidentelle du plateau source.
— **Légalité des coups.** La légalité est gérée par les utilitaires : `liste_coup_prior` ne renvoie que des coups autorisés. `strategie_defaut` peut lever `Pas_de_coup_valide` lorsque aucun coup n'existe.

Structure d'exploration et valeur d'action

Pour enregistrer les choix testés et leurs statistiques, on utilise les types suivants :

```
type etiquette = { mutable n:int; mutable v:int; prior:float; pos:position }
type plateau = step of etiquette * arbre list
```

Interprétation :

— Une entrée de la structure d'exploration est un enregistrement `etiquette` : `n` (nombre d'essais), `v` (succès pour Noir), `prior` (préférence initiale), `pos` (coup choisi depuis la situation précédente).
— Un `step (e, opt)` code un choix (`e`) et une liste d'options suivantes (`opt`) accessibles si l'on applique ce choix.

Pourquoi ces compteurs ?

`n` et `v` servent à mesurer ce qui marche : si, en partant d'un coup donné, beaucoup de simulations finissent gagnantes pour Noir, alors $\frac{v}{n}$ sera grand et la MCTS *reviendra plus souvent* explorer ce coup. Le `prior` donne un léger avantage initial à des coups jugés plausibles avant d'avoir des données.

Valeur d'action (point de vue Noir). Pour une entrée avec compteurs (n, v) et préférence `prior`, on utilise :

$$V^N = \begin{cases} \dfrac{v}{n} + \dfrac{\texttt{prior}}{1+n} & \text{si } n > 0, \\ \texttt{prior} & \text{si } n = 0. \end{cases}$$

Le point de vue Blanc s'obtient par symétrie (*succès Blanc* $= n - v$).

Politique de sélection (enchaînement de choix). Lorsqu'il revient à Noir de jouer, on retient l'option de V^N maximal ; si c'est à Blanc, on retient l'option de V^B maximal. En cas d'égalité, on peut départager par `prior`, puis par l'ordre d'apparition (tri stable).

Extension. Depuis une situation non encore détaillée, on obtient `liste_coup_prior goban joueur` et, pour chaque (`pos`, `prior`), on *ajoute* une option de la forme

```
step { n=0; v=0; prior; pos }, [].
```

Simulation d'une partie. On applique `strategie_defaut` pour jouer des coups successifs jusqu'à lever `Pas_de_coup_valide`. On évalue le vainqueur via `gagnant` et on retourne `Noir` ou `Blanc`.

Mise à jour des compteurs. Après la simulation, on augmente `n` pour chaque choix effectué dans l'itération ; si la simulation est gagnée par `Noir`, on augmente aussi `v`.

Exemple

On part d'un plateau vide 3×3, Noir doit jouer.
1) Sélection. Les options visibles depuis l'état initial avec priorités (donnée par `liste_coup_prior`) sont, par exemple : $((0,0) : 0{,}8), ((1,1) : 0{,}6), ((2,2) : 0{,}4)$.
Elles n'ont jamais été essayées ($n = 0, v = 0$) $\Rightarrow$ leur valeur d'action pour Noir vaut la *priorité*. Noir choisit donc $(0,0)$ (la plus grande).

2) Extension. On regarde la situation juste après ce choix (Noir en $(0,0)$, à Blanc de jouer) : on ajoute alors, pour cette situation, le coups plausibles pour `Blanc` (renvoyés par `liste_coup_prior`), par exemple $((0,1) : 0{,}7), ((1,0) : 0{,}5), ((1,1) : 0{,}5)$.

3) Simulation (playout). À partir du plateau courant, on joue une partie imaginaire avec la stratégie par défaut (ex. : « première case libre en balayant ligne par ligne »), jusqu'à blocage. Une simulation possible (en supposant que Noir a déjà joué $(0,0)$) :

$$
\begin{array}{lcl@{\quad}lcl}
 & & N\ B\ . & & & N\ B\ N \\
\text{Blanc } (0,1) & \to & .\ .\ . & \text{Noir } (0,2) & \to & .\ .\ . \\
 & & .\ .\ . & & & .\ .\ . \\
 & & N\ B\ N & & & N\ B\ N \\
\text{Blanc } (1,0) & \to & B\ .\ . & \text{Noir } (1,1) & \to & B\ N\ . \\
 & & .\ .\ . & & & .\ .\ . \\
 & & N\ B\ N & & & N\ B\ N \\
\text{Blanc } (1,2) & \to & B\ N\ B & \text{Noir } (2,0) & \to & B\ N\ B \\
 & & .\ .\ . & & & N\ .\ . \\
 & & N\ B\ N & & & N\ B\ N \\
\text{Blanc } (2,1) & \to & B\ N\ B & \text{Noir } (2,2) & \to & B\ N\ B \\
 & & N\ B\ . & & & N\ B\ N
\end{array}
$$

Plateau plein : Noir a 5 pierres, Blanc 4 $\Rightarrow$ Noir gagne pour cette simulation.

4) Mise à jour. On augmente les compteurs des choix utilisés pendant l'itération. Ici, on n'a effectivement « engagé » le coup $(0,0)$: ses compteurs passent de $(n, v) = (0,0)$ à $(1,1)$ (un essai de plus, une victoire pour Noir).
Sa valeur d'action pour Noir devient $V^N = v/n + \text{prior}/(1+n) = 1 + 0{,}8/2 = 1{,}4$.

Bilan. Une itération = choix depuis l'état initial → extension → simulation complète → mise à jour des compteurs.
En répétant ces itérations, on favorise progressivement les options gagnantes et, à la fin, on joue l'option la plus visitée (ou la mieux notée selon la règle fixée).

Énoncé final — Ce que votre programme doit faire

Construire un programme qui, à partir d'un état courant du goban, choisit la prochaine coup à jouer en utilisant une MCTS, selon un max d'itérations donné.

Exercice

Interface attendue. Votre programme doit demander/interpréter :

1. Couleur du joueur courant (`Noir` ou `Blanc`).
2. Taille du goban `n` et état actuel du goban ($n \times n$).
3. Budget de calcul (nombre max d'itérations MCTS, entier positif).

Traitement.

- Construire une MCTS conforme aux définitions ci-dessus (*sélection* → *extension* → *simulation* → *mise à jour*).
- Utiliser la valeur d'action fournie (point de vue Noir/Blanc) pour la sélection.
- Répéter les itérations jusqu'à épuisement du budget de calcul.
- Décision finale : choisissez la coup la plus visitée (n maximal).

Sortie attendue.
La position du coup à jouer, ou un message `Pas_de_coup_valide` s'il n'existe aucun coup légal.

Contraintes techniques.

- Ne modifiez pas les types/fonctions fournies (`liste_coup_prior`, `strategie_defaut`, `joue`, `gagnant`).
- Pas d'effets de bord sur le goban source (utilisez le goban renvoyé par `joue`).

Exemple

Entrée :

```
Joueur ? Noir
Taille ? 3
Goban (3 lignes) :
. ..
.B.
...
Budget d'itérations ? 10000
```

Sortie :

```
Coup conseillé : (0,0)
```