DS 3

Informatique de tronc commun, classe de PC

Julien Reichert

Ce devoir consiste en des questions de cours suivi de différents problèmes sur le thème de la théorie des jeux ou de l'apprentissage.

Partie 1 : Questions de cours

Question 1.1 : Définir la notion de matrice de confusion.

Question 1.2 : Définir la notion de jeu positionnellement déterminé.

Question 1.3 : Définir la notion d'heuristique.

Partie 2 : Tri minmax

On considère le tri par comparaison d'une liste (de taille connue) en Python comme un jeu entre Ève (qui souhaite trier la liste le plus vite possible et dont la stratégie s'apparente à un algorithme écrit à l'avance - ce qui correspond à l'étymologie de « programmer ») et Adam, qui souhaite ralentir la progression du tri en répondant aux comparaisons d'Ève de manière la plus défavorable possible. Ce faisant, la liste en argument du tri peut être considérée comme jamais révélée par Adam (on est dans le cadre de jeux à information imparfaite).

Par exemple, pour la liste `[1, 3, 4, 2]`, une partie pourrait être formée ainsi :

- Ève : le premier élément est-il inférieur ou égal au deuxième ?
- Adam : Oui.
- Ève : le troisième élément est-il inférieur ou égal au quatrième ?
- Adam : Non.
- Ève : le premier élément est-il inférieur ou égal au quatrième ?
- Adam : Oui.
- Ève : le troisième élément est-il inférieur ou égal au deuxième ?
- Adam : Non.
- Ève : le deuxième élément est-il inférieur ou égal au quatrième ?
- Adam : Non.

La partie s'arrête car Ève peut conclure que l'ordre des éléments est : premier, quatrième, deuxième et troisième.

Nous allons réaliser ce jeu, certes de manière non optimale en termes de complexité, en remplaçant l'arène par une structure de mémoire pour Ève contenant la liste des permutations encore possibles au vu des informations fournies. **Chaque réponse élimine donc des permutations possibles et rapproche Ève de la réponse quant à la façon de trier la liste, sauf bien entendu si la question qu'elle a posée n'a pas d'intérêt.**

Dans quelle mesure Adam a-t-il un rôle ? La question est légitime. En fait, puisque la connaissance de la liste n'est pas encore fournie à Ève au moment où le tri est programmé, on peut laisser à Adam le soin de répondre ce qu'il veut, à condition de rester cohérent, et donc de choisir la liste qui l'arrange le plus à chaque étape. **Ainsi, le nombre de permutations que la réponse d'Adam permet d'éliminer sera toujours supposé inférieur au nombre de permutations que la réponse d'Adam ne permet pas d'éliminer.**

Nous faisons donc face à une situation illustrant le minmax.

On supposera dans cette partie qu'il n'y aura jamais d'égalités entre les éléments, ce qui facilitera le travail au niveau de la fin de la partie et permettra de ne pas avoir à se poser la question d'utiliser des comparaisons larges ou strictes.

Exercice 2.1 : Rappeler sans preuve combien il existe de permutations d'un ensemble de taille n.

Pour raccourcir les énoncés suivants, on appellera *peen* une permutation de l'ensemble des entiers de `0` à `n-1`. On suppose par ailleurs que `n` sera toujours supérieur ou égal à deux.

Exercice 2.2 : Écrire une fonction permettant d'engendrer toutes les peen (le nombre `n` sera le seul argument).

Exercice 2.3 : Écrire une fonction prenant en argument une liste de peen (supposée non vide), deux indices distincts, notés respectivement `i` et `j` et entre `0` et `n-1`, et un booléen `b`. Cette fonction renverra la sous-liste des peen dans la liste de départ pour lesquelles l'élément d'indice `i` est inférieur à l'élément d'indice `j` si, et seulement si, le booléen `b` est vrai. **On prendra garde à l'esthétique des comparaisons et à ne pas écrire de code lourd.**

Exercice 2.4 : Écrire une fonction prenant en argument une liste de peen (supposée non vide) et deux indices distincts comme dans la fonction précédente, et renvoyant le booléen qui maximise la taille de la sous-liste qui serait renvoyée par la fonction précédente (en cas d'égalité, on pourra renvoyer n'importe quel booléen).

Exercice 2.5 : Écrire une fonction prenant en argument une liste de peen et renvoyant un couple d'indices qui minimise la taille de la sous-liste qui serait renvoyée par la fonction de l'exercice 2.3 si le booléen renvoyé par la fonction 2.4 était utilisé avec ce couple d'indices. Il faudra éventuellement se servir d'une version légèrement modifiée de la fonction précédente.

Exercice 2.6 : En déduire une fonction appelée *tri minmax* qui filtre les peen possibles jusqu'à ce qu'il n'en reste qu'une. Attention à la façon d'utiliser les fonctions précédentes !

En pratique, on peut voir le tri d'une liste comme un jeu à condition de ne pas avoir de liste déjà formée en argument, ce qui n'est pas le cas ici au vu de la consigne.

Exercice 2.7 : Rappeler la formule de Stirling et s'en servir pour donner un équivalent en $+\infty$ du nombre de tours effectués dans le jeu que l'on simule par la fonction précédente.

Exercice 2.8 : Estimer cependant la complexité de la fonction de l'exercice 2.6 et comprendre qu'on évitera d'employer un tel algorithme de tri.

Partie 3 : Jeu des bâtonnets

Rappelons les règles du jeu présenté en cours : vingt-et-un bâtonnets sont disposés en ligne, et deux joueurs sont amenés à en retirer entre un et trois à chacun de leurs tours (qui s'alternent de manière classique). Le joueur qui ne peut pas jouer a perdu, ce qui signifie que celui qui prend le dernier bâtonnet (possiblement avec un ou deux autres) a gagné. Le joueur qui commence est désigné arbitrairement.

Nous allons modéliser l'arène du jeu comme un dictionnaire, dont les clés seront des couples formés d'un booléen (`True` si c'est au tour d'Adam et `False` si c'est au tour d'Ève) et d'un entier entre `0` et `21` (le nombre de bâtonnets restants) et les valeurs sont des listes de sommets accessibles depuis ce sommet.

Exercice 3.1 : Écrire une fonction sans argument qui crée l'arène associée au jeu.

Exercice 3.2 : Écrire une fonction qui prend en argument une arène de même structure que celle de la fonction précédente (mais pas forcément l'arène de la fonction précédente) et une liste de sommets de l'arène en tant qu'objectif d'accessibilité d'Ève et qui renvoie la liste des sommets gagnants dans le jeu d'accessibilité associé. On pourra commencer par rappeler la construction employée et ce que dit le cours à son sujet, si la programmation s'avère trop difficile.

Exercice 3.3 : Que renvoie la fonction de l'exercice précédent sur l'arène du jeu des bâtonnets ? (On peut donner une description ou énoncer intégralement la valeur de retour.)

Exercice 3.4 : Citer un exemple de partie qui pourrait être jouée dans le jeu des bâtonnets si Ève commence et les deux joueurs jouent au mieux de leur intérêt.

Exercice 3.5 : Écrire une fonction qui prend en argument une partie jouée dans le jeu des bâtonnets et qui détermine le nombre de fois où un joueur était dans un sommet gagnant et n'a pas joué au mieux de ses intérêts, de sorte qu'un sommet gagnant de son adversaire ait été atteint.

Partie 4 : Intelligence artificielle - extension de kNN

On redonne ci-dessous une version de l'algorithme des k plus proches voisins.

```
def kNN(les_points, point_sup, k, dist):
    distances = [(dist(point[0], point_sup), point[1]) for point in les_points]
    # Il manque quelque chose ici
    etiquettes = dict()
    for _, etiq in distances[:k]:
        if etiq in etiquettes:
            etiquettes[etiq] += 1
        else:
            etiquettes[etiq] = 1
    rep = etiq # Initialisation pertinente plutôt que None
    for e in etiquettes: # Moche de réutiliser le nom etiq, mais ça marcherait
        if etiquettes[e] > etiquettes[rep]:
            rep = e
    return rep
```

Question 4.1 : La ligne consistant en un commentaire « `Il manque quelque chose` » a remplacé une ligne de code cruciale. Compléter (en une ou plusieurs lignes, en appelant éventuellement une fonction qu'on écrira / existante).

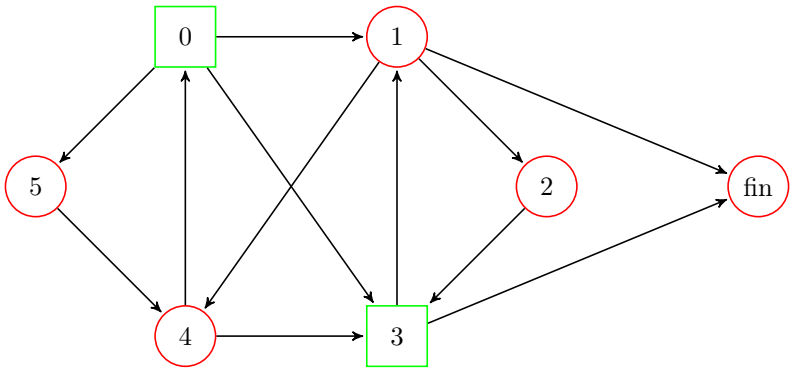
Question 4.2 : Écrire une fonction `dist` qui puisse être compatible avec la fonction `kNN`. Écrire alors un élément possible de la liste `les_points`, en expliquant la structure.

Question 4.3 : Que vaut précisément `etiq`, dans la ligne qui initialise `rep`, par rapport aux arguments de la fonction ?

Question 4.4 : On souhaite modifier la fonction de sorte que plus un des k plus proches voisins est proche, plus il va contribuer pour le choix de son étiquette. La formule suggérée est de le compter pour l'inverse de la distance au point supplémentaire, en traitant le cas particulier d'une distance nulle : dans ce cas l'étiquette sera forcément retenue (mathématiquement cohérent...). On admettra que les points fournis sont distincts deux à deux. Écrire une nouvelle version de la fonction `kNN`. On pourra signaler les modifications à faire, pour aller plus vite.

Partie 5 : Une condition de gain exotique

On considère l'arène ci-dessous, pour un jeu dont la condition de gain est un peu spéciale : un joueur gagne une partie si, et seulement si, il sélectionne depuis un de ses sommets un coup qui mène vers le sommet « fin » alors que la somme des étiquettes des sommets visités jusque là (y compris le sommet de départ) est un nombre premier.



Exercice 5.1 : Existe-t-il des parties infinies ? Si oui, en donner un exemple.

Exercice 5.2 : Considérons la partie 5 4 0 3 fin. Qui gagne ?

Exercice 5.3 : Le jeu est-il déterminé ? Existe-t-il des stratégies gagnantes positionnelles ?

Exercice 5.4 : Un sommet peut-il être gagnant pour un joueur en début d'une partie mais plus par la suite ? Et si les joueurs jouent au mieux de leur intérêt ?

Exercice 5.5 : Pour chaque sommet de départ, présenter les parties possibles, en excluant les coups non optimaux (dans la mesure où un autre coup mènerait à une victoire immédiate ou le prochain coup de l'adversaire le mènerait à une victoire immédiate). On pourra par exemple faire une arborescence, et se servir d'arguments arithmétiques...

Exercice 5.6 : En déduire la liste des sommets depuis lesquels chaque joueur a une stratégie gagnante en début de partie.

Exercice 5.7 : Écrire en Python une fonction prenant en argument une partie finie en tant que liste de sommets (des nombres de 0 à 5 ou la chaîne de caractères `"fin"`) et déterminant si cette partie est gagnée par Ève (valeur de retour `1`), par Adam (valeur de retour `-1`) ou nulle (valeur de retour `0`). Si la partie est mal formatée (en particulier on doit vérifier que `"fin"` n'apparaît qu'une fois et en dernier), il faut déclencher une erreur, par exemple avec `assert false`.