Chapitre 3
Dictionnaires et algorithmes pour l'intelligence artificielle

Table des matières

1 Dictionnaires

1.1 Dictionnaires, clés et valeurs

Vous avez vu en première année l'utilisation des dictionnaires, et nous allons cette année revenir sur leur principe de fonctionnement.

En première approche, un dictionnaire en Python est très similaire à une liste, la différence étant que les éléments d'une liste sont ordonnées, et donc associés à un entier donnant leur position dans la liste alors que dans un dictionnaire chaque élément (appelé *valeur*) est associé à une *clé* qui peut avoir différents types de données (un entier, une chaine de caractère, etc). Dans un dictionnaire, il n'y a donc pas d'ordre naturel pour l'affichage des entrées (comme pour les bases de données), il est donc peu judicieux de parler du n-ième élément d'un dictionnaire.

1.1.1 Rappels : réalisation et utilisation d'un dictionnaire

En Python, les dictionnaires sont délimités par des accolades (les listes par des crochets et les tuples par des parenthèses).

On peut alors créer un dictionnaire vide :

```
1  mon_dico = {}
```

puis insérer les couples clés/valeurs une à une. Par exemple pour un dictionnaire associant à chaque clé le nom d'un pays, on associera comme valeur le nom de sa capitale.

```
1  mon_dico['France']= 'Paris'
2  mon_dico['Angleterre'] = 'Londres'
3  mon_dico['Allemagne'] = 'Berlin'
4  mon_dico['Italie'] = 'Rome'
```

On peut aussi plus directement renseigner un dico comme :

```
1  mon_dico = {'France':'Paris','Angleterre' : 'Londres'}
```

Comme on peut le voir, les dictionnaires sont des structures de données de type *mutable* : il est possible de changer la valeur associée à une clé (comme dans le cas des listes), par exemple en mettant les noms des capitales dans la langue du pays. Dans ce cas, si la clé existe déjà, la valeur est modifiée, sinon, un nouveau couple clé/valeur est ajouté au dictionnaire.

```
1  mon_dico['Angleterre'] = 'London'
2  mon_dico['Italie'] = 'Roma'
```

Attention, les clés ne sont pas nécessairement toutes du même type (on peut avoir dans un même dictionnaire une clé de type *float* et une clé de type *string* ou encore une clé de type tuple d'entiers).

Comme pour les listes, on récupère le nombre d'éléments en faisant `len(mon_dico)`, et on peut parcourir le dictionnaire de différentes manières :

- selon les clés : `for cle in mon_dico.keys()` ou de manière plus concise `for cle in mon_dico` ;
- selon les valeurs : `for value in mon_dico.values()` ;
- selon les couples : `for cle,value in mon_dico.items()` .

On peut enfin supprimer un élément en appelant `del(mon_dico[cle])` et comme pour les listes, une affectation `dico_2 = dico_1` ne crée pas une copie mais seulement une nouvelle affectation, donc les changements apportés à `dico_1` seront aussi subis par `dico_2`, il faut donc utiliser la syntaxe `dico_2 = dico_1.copy()`.

1.1.2 Principe du hachage

Lors de la création d'un dictionnaire de taille n, Python va affecter une partie de l'espace mémoire du système afin de stocker les valeurs associées chaque clé à chaque adresse mémoire. Pour savoir à quel adresse correspond chaque clé, une opération appelée hachage a lieu sur la clé.

Une *fonction de hachage* en informatique est une fonction qui prend en entrée un argument (de type quelconque) et qui renvoie un entier. Elle n'est pas nécessairement injective, et si deux entrées renvoient le même résultat, on parle de *collisions*.

Un exemple de fonction de hachage est l'attribution d'un numéro de sécurité sociale ou les codes ISBN d'identification d'un livre dans les bibliothèques et librairies (dans ces cas, il faut interdire les collisions !). Un autre exemple plus évident est d'associer à un fichier informatique sa taille en octet. Dans ce cas, il y aura des collisions, mais si deux

fichiers ont des tailles différentes, ils sont forcément différents, on peut donc très rapidement vérifier si deux fichiers sont différents, sans comparer bit par bit les fichiers (mais cette rapidité se paie par un haut taux de collisions).

Ainsi, dans un dictionnaire, chaque clé est hachée pour donner un entier que l'on ramène ensuite entre 0 et $n-1$ (la taille supposée du dictionnaire), par exemple en ne gardant que le reste dans la division euclidienne.

Python possède nativement une fonction de hachage `hash` qui prend en particulier comme argument un nombre ou une chaine de caractère et renvoie un entier compris entre $-(N-1)$ et $N-1$ avec $N = 2^{61}-1$.

Pour un entier `hash(n)` renvoie la valeur de l'entier, pour un flottant écrit comme p/q, alors `hash(p/q) = hash(p)*r` où r est l'inverse de `hash(q)` dans le corps $\mathbb{Z}/N\mathbb{Z}$ (puisque N est premier, quel que soit q entre 1 et N, il existe un unique r entre 1 et N tel que le reste de $q*r$ dans la division euclidienne par N vaille 1). Enfin, pour une chaine de caractère, elle est d'abord converti en nombre par une fonction de type *fnv* dont on ne cherchera pas le détail (il s'agit de changer chaque caractère par son code ascii, puis d'effectuer des opérations dessus, afin que des chaines proches correspondent à des résultats de hachage très différents afin d'éviter des erreurs de frappe). Enfin, la fonction `hash` peut aussi prendre comme argument un tuple composé d'éléments eux-mêmes hachables.

A retenir : avec Python, on peut faire le hachage de nombres ou de chaines de caractères ou de tuples formés de nombres et chaines de caractères.

1.1.3 Principe de fonctionnement des dictionnaires

Lors de la création d'un dictionnaire, les opérations suivantes sont effectuées :

1. lors de la création du dictionnaire `mon_dico = {}` un tableau d'adresses mémoires de taille n donnée est réservée ;
2. lors de l'ajout d'un couple clé/valeur `c,v` par l'appel `mon_dico[c] = v`, la clé `c` est hachée pour donnée un entier entre 0 et $n-1$, qui correspond à l'adresse mémoire dans laquelle va être stockée la valeur `v` ;
3. en cas de collision (si une deuxième clé c_2 donne le même résultat de hachage que c), il n'est pas possible d'affecter la valeur v_2 à l'adresse voulue. On parcourt alors les adresses mémoires vides restantes pour affecter la valeur v_2 à une case vide ;
4. lors de l'appel `d[c]`, la clé est hachée à nouveau, et Python renvoie la valeur stockée dans l'adresse mémoire correspondante ;
5. des mécanismes cachés à l'utilisateur s'assurent que tout fonctionne correctement (gestion des collisions, suppressions des couples, extension du dictionnaire, etc), mais nous ne les expliciterons pas.

Ce principe de fonctionnement explique donc pourquoi les clés ne peuvent pas être de n'importe quel type puisqu'elles doivent pouvoir être hachées : les clés sont donc des nombres, des chaines de caractères ou des tuples de ces objets. On ne peut pas hacher non plus des objets de type mutables (comme les listes) puisque un changement des élements d'une liste devrait changer le résultat de hachage, or on veut pouvoir associer une seule adresse mémoire à chaque clé.

1.1.4 Comparaison avec les listes

On peut alors se demander quel est l'avantage de travailler avec des dictionnaires par rapport aux listes. Pour voir la différence, il faut s'intéresser à comment les éléments d'une liste sont stockés : lors de la création de la liste, des adresses mémoires sont fixées donnant le début de la liste, la valeur stockée dans chaque adresse et l'adresse où trouver le prochain élément de la liste, on parle de *liste chainée*.

Ainsi, pour savoir si un élément x appartient à la liste L (donc l'opération `x in L`), il est nécessaire de parcourir toutes les adresses mémoires les unes après les autres jusqu'à trouver la valeur cherchée x dans une des adresses. A l'inverse, le test de présence d'une clé dans un dictionnaire `c in mon_dico` nécessite seulement de hacher la clé, et de regarder dans l'adresse mémoire correspondante si elle est remplie ou pas.

Pour des dictionnaires ou listes de taille N, le coût temporel d'une telle recherche est donc $\mathcal{O}(N)$ pour les listes contre $\mathcal{O}(1)$ pour les dictionnaires !

On pourra se reporter à l'exercice 2 du TD pour s'en rendre compte.

1.1.5 Dictionnaires et mémoïsation

Puisque les dictionnaires sont très rapides d'accès, il est très intéressant de s'en servir pour stocker le résultat r d'une fonction d'argument A sous la forme d'un couple clé/valeur A/R afin d'y accéder ultérieurement sans avoir à refaire le calcul.

La *mémoïsation* consiste justement à mettre en place un système qui mémorise les arguments d'entrée et le résultat d'une fonction, lors de chaque appel de la fonction et évite ainsi de faire plusieurs fois un même calcul, ce qui permet un gain en temps.

La mémoïsation est particulièrement utile dans le cas de fonctions récursives (cf calcul récursif du n'-ième terme de la suite de Fibonacci en TD).

2 Intelligence artificielle : algorithmes d'apprentissage

Historiquement, la notion d'intelligence artificielle a émergé dès le début de la fabrication des premiers calculateurs (par exemple, *Computing Machinery and Intelligence*, A. Turing, 1950). Le développement des capacités de calculs a permis le développement de cette science avec comme dates notables :

- Années 50 et 60 : développement de l'intelligence artificielle comme domaine de recherche international (conférence au Dartmouth College, 56) ;
- développement des capacités de calcul dans les années 80, développement des algorithmes d'apprentissage automatique, puis des précurseurs des réseaux de neurone (apprentissage par renforcement, *Deep Blue* ;
- Années 2000 : web 2.0 et développement des capacités de calcul change l'échelle, algorithmes d'apprentissage profond (*deep learning*, *AlphaGo*, *AlphaFold*, etc).

Nous voyons dans cet exposé historique très très sommaire les notions qui vont nous intéresser sur ce chapitre : les algorithmes apprenants (*machine learning*), qui appartiennent à la famille de l'intelligence artificielle et dont les algorithmes d'apprentissage profond ne sont qu'une partie.

Nous allons aborder dans ce chapitre 2 types d'algorithmes apprenants, donc la description du fonctionnement nous permettra éventuellement d'adapter de nouvelles manières de traiter un nouveau problème.

2.1 Apprentissage supervisé : algorithme des k-plus proches voisins

Le premier algorithme que nous allons étudier est un algorithme de classification : l'algorithme des k-plus proches voisins.

La question posée est la suivante : en partant d'un ensemble de données d'entrainement pour lesquelles une valeur est donnée, quelle est la valeur à attribuer à une nouvelle donnée inconnue ? C'est un algorithme de classification, et c'est par exemple ce que l'on souhaite faire pour des algorithmes de conduite autonome : en partant d'une base de données de photos (que vous avez contribué à enrichir en validant des captchas), quels sont les éléments qui apparaissent sur une image prise en temps réel par une caméra (passages piétons, feux tricolores, bornes incendie, etc) ?

Le principe de fonctionnement est très simple : on cherche pour la nouvelle donnée à classifier quels sont les k données d'entrainement les plus proches, et on classifie alors la nouvelle donnée dans la classe la plus représentée parmi ces k valeurs. Par exemple, si dans une équipe de rugby amateur on sépare les joueurs selon taille et poids, on peut imaginer voir 4 blocs : des petits maigres (demis de mêlée), des grands maigres (arrières), des petits gros (piliers) et des grands gros (deuxième ligne). Si se présente alors un joueur de 1m95 pesant 110 kg, on l'enverra alors plutôt s'entrainer avec les deuxième lignes qu'avec les demis de mêlée.

Nous allons discuter de l'implémentation de cet algorithme à travers un exemple historique qui a été un des premiers développés à cet effet : la reconnaissance de chiffres manuscrits pour l'encaissement des chèques. Nous verrons en TP un autre exemple historique avec la distinction d'espèces d'iris.

2.1.1 Base de données

La première étape consiste à importer les modules nécessaires à la récupération des données et à leur mise en forme pour exploitation.

```
1  from sklearn.datasets import fetch_openml
2
3  #récupération des données
4  mnist = fetch_openml('mnist_784', version=1)
5
6  # séparer les données entre images X et chiffre y
7  X = mnist.data
8  y = mnist.target.astype(int)
9
10
```

```
11
12 # diviser les données entre données d'entrainement et données de test
13 from sklearn.model_selection import train_test_split
14 X_train, X_test, y_train, y_test = train_test_split(X.values, y.values, test_size=0.99)
```

La base de données $mnist_784$ est une banque d'image en format 28*28 en niveaux de gris (qui correspond ici à X) à laquelle est associée le chiffre entre 0 et 9 représenté (la liste y).

Il est ensuite nécessaire (nous verrons après pourquoi) de séparer aléatoirement des données un ensemble d'images et valeurs correspondantes d'entrainement sur laquelle on va "calibrer" l'algorithme. Les données non sélectionnées serviront elles à tester l'algorithme ainsi réalisé afin de vérifier son fonctionnement avant de l'utiliser sur des données inconnues. Habituellement, on fait une séparation 80-20 où 80% des données servenet à l'entrainement et les 20% restants aux tests. Toutefois, j'ai ici pris pour l'entrainement seulement 1% des données parce que le traitement est lourd et qu'il y a beaucoup de données au départ (56000 images).

2.1.2 Comparaison

Le principe de l'algorithme consiste à trouver quels sont les k données d'entrainement les plus proches, donc ici, quels sont les images les plus ressemblantes.

De manière générale, on va obtenir dans la base de données un ensemble de n valeurs à comparer, donc on travaille dans un espace de dimension n. La distance entre deux vecteurs x_i et y_i dans cet espace sera alors la distance euclidienne $dist(x,y) = \sqrt{\sum_i (x_i - y_i)^2}$.

Note : On peut choisir évidemment d'autres manières de définir la distance (mais elles sont hors-programme), et il faut aussi faire attention à avoir des distributions statistiques comparable (sur l'exemple de l'équipe de rugby, les tailles en m varient de seulement 0,5 alors que les poids en kg varient de 50, donc le critère de taille serait totalement écrasé si l'on ne fait pas attention).

Dans le cas des chiffres manuscrits, les n valeurs à comparer sont les niveaux de gris de chaque pixel, l'espace est donc de dimension 28*28 = 784

On obtient donc comme distance :

```
1 def distance(image1,image2) :
2     dcarre = 0
3     for i in range(len(image1)) :
4         dcarre += (image1[i]-image2[i])**2 #distance euclidienne
5     return dcarre
```

On a gardé ici que le carré de la distance puisque on cherche uniquement les k plus proches (et que la fonction carré est croissante).

2.1.3 Détermination des k-plus proches voisins

On souhaite maintenant déterminer la liste des k plus proches voisins d'une donnée de test parmi les données d'entrainement.

On va donc parcourir l'ensembles des données d'entrainement et mesurer la distance d de chacune avec la donnée de test.

- si on a moins que k distances, on garde cette valeur de la distance d et l'indice i correspondant à la donnée test ;
- si on en a déjà k et que d n'est pas plus grande que ces k là, on élimine la plus grande distance et son indice, et on garde d et i en mémoire à la place. Ici, vu que la liste n'a que k élément, j'ai pris la liberté d'utiliser la fonction sorted, mais un tri par insertion est surement plus judicieux.
- une fois toutes les données d'entrainement passées, la liste mémoire contient les k valeurs correspondant aux distances et indices des données de test. Je peux donc parcourir cette liste afin de connaitre la valeur y associée à chaque indice (ici le chiffre qui a été écrit sur chacune des k images les plus proches).

On obtient :

```
1 def plusproches(k,index) :
2     image = X_test[index]
3     liste_pp = []
4     for i in range(len(X_train)) :
5         d = distance(image,X_train[i])
6         if len(liste_pp) < k : liste_pp.append([d,i])
7         elif d < liste_pp[-1][0] :
```

```
            liste_pp[-1] = [d,i]
            liste_pp.sort()
    valeurs_proches = []
    for j in range(k) :
        valeurs_proches.append(y_train[liste_pp[j][1]])
    return valeurs_proches
```

On peut alors faire des statistiques pour déterminer dans quelle classe affecter la donnée de test (j'ai choisi de prendre la classe la plus représentée, et celle correspondant à la donnée la plus proche si égalité). En utilisant un dictionnaire, on peut faire :

```
def valeur(Liste):
    dico={}
    vmax=0
    for i in Liste:
        if i in dico.keys(): dico[i] += 1
        else: dico[i] = 1
    for i in dico.keys():
        if dico[i] > vmax :
            vmax=dico[i]
            resultat=i
    return resultat
```

2.1.4 Efficacité de l'algorithme

Il est alors possible de comparer le résultat prédit par l'algorithme au résultat escompté (en utilisant la liste y_{test} qui donne la valeur du chiffre dessiné). On peut ainsi déterminer le taux de succès de l'algorithme, et faire varier alors les paramètres (ici le paramètre k) afin d'être le plus efficace possible.

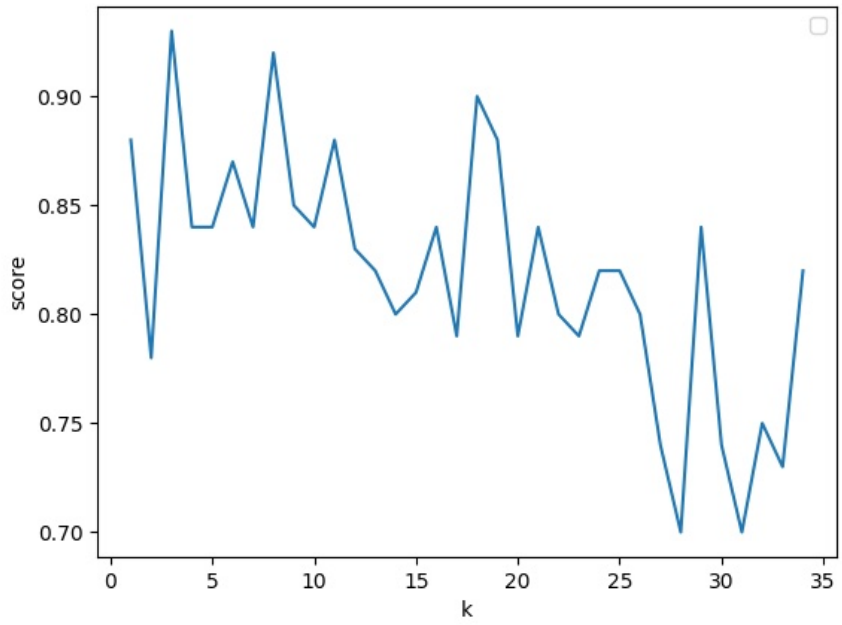

Figure 1 – Evolution du score de l'algorithme en fonction de k (100 tests à chaque valeur de k).

Dans notre cas, on observe :

- un taux de succès porche de 90 %, c'est assez impressionnant vu la facilité avec laquelle on a implémenté l'algorithme sans trop de réflexion (seulement 2 choix : la valeur de k et la définition de la distance) ;
- qu'il y a un optimum pour la valeur de $k = 3$: c'est le comportement attendu qui permet de déterminer la valeur de k à retenir. Si k est trop faible, il y a une forte probabilité que les valeurs retenues ne soit pas assez

représentative de la classe déterminée (surajustement aux données), et s'il est trop grand, un effet de moyennage entre les classes les plus proches peut fausser l'estimation (sous-ajustement).

Une autre manière de quantifier la précision de l'algorithme est de déterminer la *matrice de confusion* où la valeur de M_{ij} correspond au nombre de fois que l'algorithme a prévu la valeur j pour une valeur réelle i : idéalement la matrice de confusion doit se rapprocher le plus possible d'une matrice diagonale. Pour l'algorithme des 3-plus proches voisins et 200 tests, on obtient la matrice :

22	0	0	1	0	0	0	0	0	0
0	20	1	0	0	0	0	0	0	0
0	0	10	0	0	0	0	2	2	0
0	0	0	17	0	1	0	1	0	2
0	0	0	0	15	0	0	0	0	2
0	0	0	2	0	18	1	0	1	0
0	0	0	0	0	0	25	0	0	0
0	3	0	0	0	0	0	17	0	1
0	0	0	1	0	1	0	0	11	0
0	0	0	1	2	0	1	2	0	17

Figure 2 – Matrice de confusion pour 200 tests de l'algorithme des 3 plus proches voisins.

Enfin, en plus de la classification, cet algorithme peut servir à faire de la régression (par exemple estimer la valeur d'un bien en fonction de sa localisation, de sa surface, du nombre de pièces, etc).

2.2 Apprentissage non-supervisé : algorithme des k-moyennes

Le deuxième type d'algorithme que nous allons étudier est un exemple d'algorithme d'*apprentissage non supervisé* ce qui signifie que dans cet algorithme, les résultats attendus ne sont pas connus à l'avance pour les données d'apprentissage. Nous allons étudier un *algorithme de partionnement* (clustering en anglais), ce qui signifie que le but est de regrouper les données de départ en plusieurs groupes dont on espère qu'ils soient pertinents (nous verrons que le choix de l'algorithme ou des paramètres peut rendre la pertinence très relative).

L'algorithme que nous allons étudier est l'algorithme des k-moyennes (k-means en anglais), qui permet de partitionner les données en k groupes différents. Le principe de fonctionnement est le suivant :

- la phase d'initialisation, pour laquelle on choisit k points au hasard parmi les données qui seront les représentants de chaque classe ;
- à chaque étape, on crée k listes, et on ajoute chaque point des données à la liste qui correspond à celui des k représentants qui est le plus proche ;
- on modifie les représentants en leur affectant la valeur moyenne des points des données correspondant à chacune des listes précédentes (d'où le nom des k moyennes) et on recommence ;
- l'algorithme s'arrête lorsque les listes ne changent plus (donc les représentants non plus) : ce point est assuré par le fait qu'une fonction (l'inertie intra-classe, qui quantifie plus ou moins l'écart-type de chaque distribution) est décroissante à chaque itération jusqu'à ce qu'une position stable soit atteinte.

2.2.1 Variantes, avantages, inconvénients, utilité

Par rapport à ce canevas de base, il est possible de varier quelques détails afin d'améliorer la convergence. Par exemple, des données extrêmes peuvent déplacer les moyennes (surtout si il y a peu d'éléments dans le cluster), donc on utilise parfois le point le plus proche de la moyenne comme représentant de la classe (on implémente en fait l'algorithme des k-médoïdes).

Il est aussi possible de modifier le choix des représentants de départ en effectuant une partition aléatoire (on affecte chacun des points de données aléatoirement à une des k listes) avant de calculer alors la première valeur du représentant, il s'agit de la méthode du partitionnement aléatoire, l'autre étant la méthode de Forgy (on peut aussi simuler k points aléatoires).

L'avantage de cet algorithme est qu'il est relativement aisé à mettre en œuvre. A l'inverse cet algorithme ne renvoie que des solutions qui sont des minima locaux des sommes des distances : il est possible qu'une meilleure solution existe que celle trouvée par l'algorithme, mais qu'il n'y accède pas. C'est ce que nous verrons en TP.

3 Jeux d'accessibilité à deux joueurs sur graphes

3.1 Description

On va dans cette partie utiliser ce que vous avez vu en première année sur les graphes afin de trouver des stratégies gagnantes pour une famille de jeux bien particuliers appelés jeux d'accessibilité à deux joueurs. Ils vérifient les propriétés suivantes :

- ce sont des jeux à *information complète* : il n'y a pas d'informations cachées, "tout est sur la table" (ce qui n'est pas le cas des jeux comme la belote ou les dominos) ;
- ce sont des jeux *déterministes* : il n'y a pas de composante aléatoire dû au hasard (donc pas de 421 ou de monopoly par exemple)
- ce sont des jeux joués à tour de rôle, donc pas de chifoumi par exemple.

On trouve comme exemples de jeux qui correspondent à cette description des jeux comme le morpion, puissance 4, les dames, les échecs (occidentaux ou *xiangqi*), le go, etc.

On modélise ces jeux par un graphe $\mathcal{G} = (S, A)$ tel que chaque sommet ($\in S$)correspond à une position atteignable du jeu, et une arête ($\in A$) entre deux états correspond à un coup du jeu. Puisque les deux joueurs J_1 et J_2 jouent à tour de rôle, on peut séparer les états de S en deux sous-ensembles disjoints S_1 et S_2 (tels que S_i correspondent aux états où c'est au joueur J_i de jouer) : on parle de graphe *biparti*. On distingue de plus l'état de jeu initial s_0 (qui par convention appartient à S_1 puisque c'est au joueur J_1 de débuter), ainsi que les états de fin de partie (qui sont de degré sortants 0, c'est-à-dire qu'il n'y a pas d'arêtes sortantes de ce sommet). Dans ces états de fin de partie, il y a les états gagnants pour J_1, ceux gagnants pour J_2 et les matchs nuls.

Exemple avec le jeu de Nim (le jeu des allumettes de Fort Boyard) : le jeu consiste à ce que les joueurs prennent 1, 2 ou 3 allumettes à chaque tour dans un tas qui en contenait N au départ. Celui qui prend la dernière a gagné. On trouve figure 3 le graphe correspondant au jeu de Nim avec 7 allumettes et la possibilité de n'en prendre que 1 ou 2 à chaque tour.

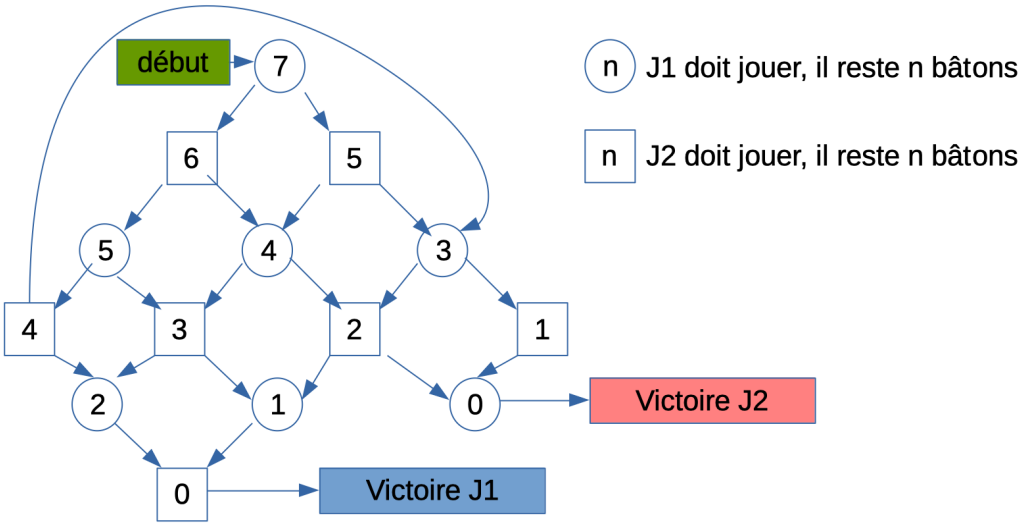

Figure 3 – Graphe du jeu de Nim avec au départ 7 bâtons, et où les règles imposent de n'en prendre qu'un ou deux par tour

Définition

Un jeu d'accessibilité à 2 joueurs est donc défini par :

- un graphe biparti $\mathcal{G} = (S, A)$ selon la partition $S = S_1 \cup S_2$ correspondants aux positions jouables par chaque joueur. On appelle aussi ce graphe *arène* ;
- un état de départ $s \in S_1$;
- une partition de l'ensemble des états finals F telle que $F = V_1 \cup V_2 \cup N$ correspondant aux états de victoire de J_1, de J_2 et aux nulles.

Une *partie* est un chemin de $\mathcal{G}$ qui part de s_0 (donc une suite s_n telle que $\forall n, (s_n, s_{n+1}) \in A$) et qui soit termine en F, soit est infini. Une partie gagnante pour J_i est une partie finie dont la dernière position est dans V_i.

3.2 Stratégies, positions gagnantes

Définition

Pour une arène $\mathcal{G} = (S_1 \cup S_2, A)$, une stratégie pour le joueur J_1 est une application $f : S_1 \to S$ telle que pour tout sommet $s \in S_1 \setminus F$, $(s, f(s))$ appartienne à A.

Ceci veut dire qu'une stratégie correspond à donner pour chaque état où le joueur J_1 doit jouer quel coup jouer parmi les coups jouables. Il s'agit ici d'une stratégie *sans mémoire* puisque le coup à jouer $(s, f(s))$ ne dépend que de l'état s et pas des états précédemment rencontrés pendant la partie (le programme ne traitant que des stratégies sans mémoire, on ne parlera que de stratégie par simplicité).

Par exemple, dans le jeu de Nim, une stratégie peut être de toujours prendre un seul bâton.

Définition

Une stratégie est dite *gagnante* pour un joueur si toutes les parties jouées en suivant cette stratégie sont gagnantes pour ce joueur.

Dans le jeu de Nim, la stratégie consistant à ne prendre qu'un bâton à chaque fois n'est clairement pas une stratégie gagnante (par exemple s'il ne reste que 2 bâtons...), par contre si on peut prendre que 1 ou 2 bâtons, une stratégie gagnante consiste à ne laisser qu'un nombre multiple de 3 bâtons sur la table.

Définition

Une *position gagnante* pour un joueur est une position telle que si la partie débutait en cette position, il y aurait une stratégie gagnante pour ce joueur.

Avec les mêmes règles sur le jeu de Nim, les positions gagnantes sont toutes celles où le nombre de bâtons n'est pas un multiple de 3.

3.3 Détermination de stratégies et positions gagnantes

On considère ici une arène et on souhaite déterminer quelles sont les positions gagnantes de chacun des joueurs. On procède alors de la manière suivante pour le joueur 1 par exemple :

- les positions qui appartiennent à V_1 (les états finals des victoires du joueur 1) sont trivialement des positions gagnantes pour le joueur 1 ;
- pour une position $x \in S_1$ (une position x dans laquelle c'est au joueur 1 de jouer) : si y est une position gagnante du joueur 1 et qu'il existe une arête $(x, y) \in A$ alors x est une position gagnante du joueur 1 (il lui suffit de jouer en y) ;
- pour une position $x \in S_2$ (c'est à l'autre joueur de jouer) : x est une position gagnante pour le joueur 1 si toutes les arêtes $(x, y) \in A$ aboutissent à des positions y gagnantes pour le joueur 1 (le joueur 2 n'a pas d'autres possibilités que de joueur vers une position gagnante).

On peut donc déterminer l'ensemble des postions gagnantes d'un joueur en "remontant" le problème : on crée alors un *attracteur* $\mathcal{A}_1$ qui correspond à toutes les positions gagnantes en construisant :

- $\mathcal{A}_1^0 = V_1$: ceci correspond à toutes les positions gagnantes du joueur 1 en 0 coup ou moins ;
- $\mathcal{A}_1^{n+1} = \mathcal{A}_1^n \cup \{x \in S_1 | \exists y \in \mathcal{A}_1^n, (x,y) \in A\} \cup \{x \in S_2 | \forall y, S(x,y) \in A \implies y \in \mathcal{A}_1^n\}$: on construit ainsi à partir de l'ensemble des positions gagnantes en n coups ou moins celles qui sont gagnantes en $n+1$ coups ou moins.
- $\mathcal{A}_1 = \cup_{n \in \mathbb{N}} \mathcal{A}_1^n$: l'ensemble des positions gagnantes du joueur 1.

On trouve à la figure 4 la construction graphique de chacun des ensembles $\mathcal{A}_1^n$ pour n allant de 0 à 5 dans le jeu de Nim que nous avons étudié (et donc l'attracteur puisque $\forall n > 5, \mathcal{A}_1^n = \mathcal{A}_1^5$).

On voit ainsi que la position initiale est dans l'attracteur, et donc qu'il existe une stratégie gagnante pour le joueur 1 : de manière générale, la stratégie à chaque position consiste à jouer vers une position qui reste dans l'attracteur (ce qui ici correspond à la stratégie gagnante déjà vue, à savoir jouer de telle sorte à ne laisser que des multiples de 3 bâtons).

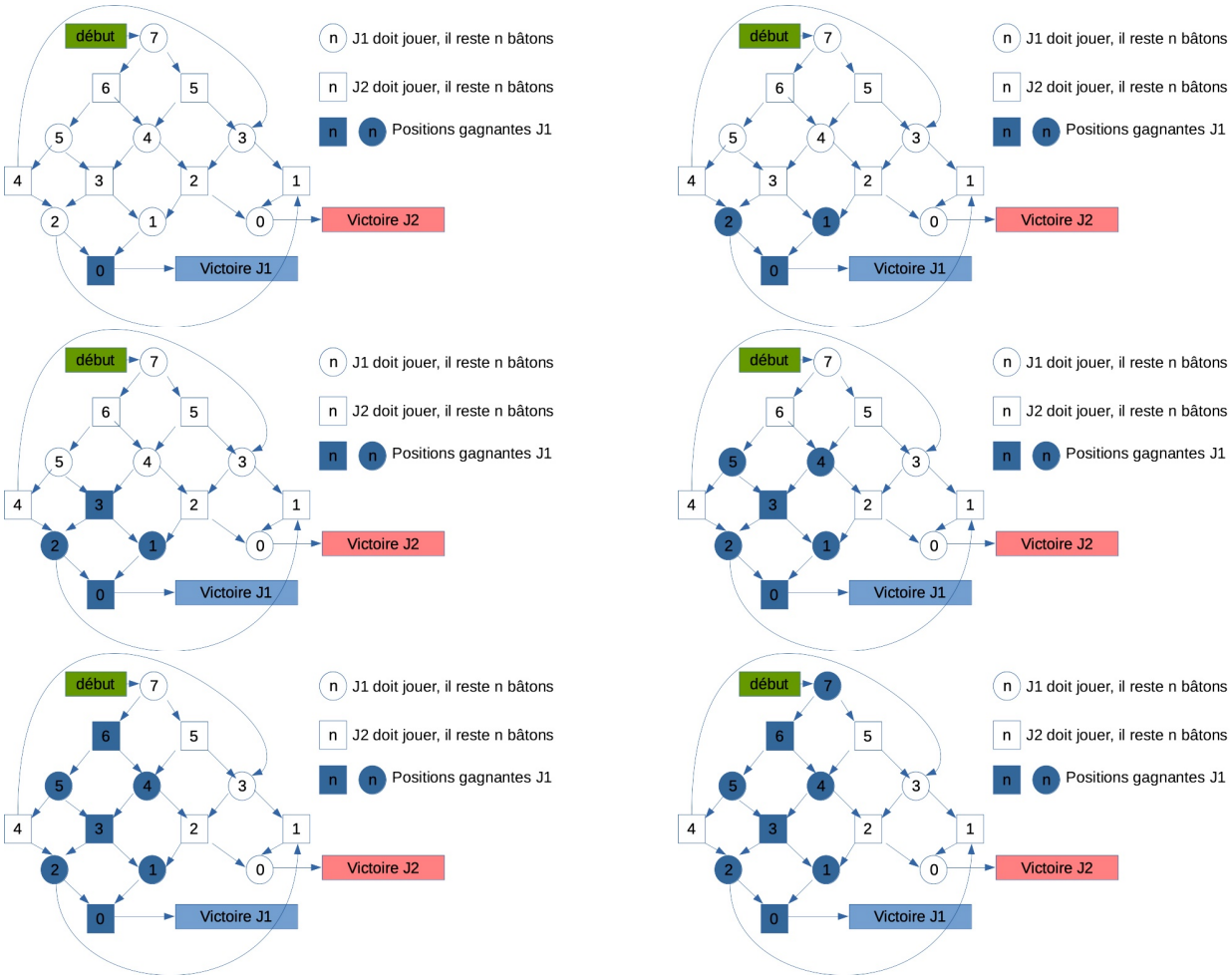

Figure 4 – Construction étape par étape (gauche à droite puis haut en bas) de l'attracteur du jeu de Nim étudié pour le joueur 1.

4 Notion d'heuristique

Nous venons de traiter les jeux d'accessibilité sur graphe dans la partie précédente, et avons vu que le jeu d'échecs par exemple rentre dans cette catégorie. On pourra alors imaginer tracer le graphe correspondant à toutes les positions atteignables, puis le calcul des attracteurs et ainsi déterminer s'il existe une stratégie gagnante pour le jouer qui a les pièces blanches par exemple. Le problème est que le nombre de sommets du graphe est immense (de l'ordre de $4{,}5.10^{46}$ d'après les estimations les plus récentes), et donc le nombre de parties aussi (la première estimation de Shannon est de l'ordre de 10^{120}, mille fois plus pour des estimations les plus récentes). Pour comparaison, le nombre d'atomes estimé dans l'univers observable est de l'ordre de 10^{80}, donc il est physiquement impossible de stocker toutes les parties possibles dans une mémoire d'ordinateur.

Ainsi, on comprend bien que les programmes informatiques comme Stockfish cherchent des stratégies en ne se concentrant que sur certaines branches du graphe et en éliminant rapidement celles qui semblent peu prometteuses.

Pour faire ce genre d'opérations il y a besoin d'introduire une estimation de la qualité d'une position, afin de baisser au maximum la qualité des futures positions adverses en augmentant le plus possible la qualité de ses positions.

La fonction qui estime la qualité d'une position est appelée une *heuristique*. Dans le cadre des jeux, on cherche donc à définir une fonction qui à toute position s associe un nombre réel, auquels on ajoute la possibilité des infinis $-\infty$ pour une position perdante et $+\infty$ pour une position gagnante.

Il existe alors des algorithmes permettant de trouver une stratégie qui soit satisfaisante en explorant les branches qui maximisent la valeur de l'heuristique pour les futures positions possibles du joueur et minimisent celles de l'adversaire afin de jouer en fonction.

Dans le cadre du programme nous allons traiter de la question d'heuristique avec un problème plus simple, le problème du sac à dos. Toutefois, l'idée générale est qu'une heuristique permet d'estimer (plus ou moins grossièrement) la qualité d'une solution d'un problème afin de donner rapidement une solution "convenable" qui peut ne pas être la solution optimale.

4.1 Le problème du sac à dos

Le problème du sac à dos (*knapsack problem* en anglais) est celui auquel est confronté tout randonneur : on souhaite apporter en randonnée une quantité d'objets divers, qui sont plus ou moins utile, en les faisant rentrer dans un sac de taille limitée. Par exemple un couteau suisse est très utile et peu encombrant, au contraire d'un sèche cheveux (et tous les cas intermédiaires entre, comme une gourde ou une clé usb). Le problème consiste donc à maximiser l'utilité des objets emportés en respectant la contrainte de poids/taille.

Formellement, on dispose d'une liste de n objets, caractérisés par leur utilité $v_i > 0$ et leur poids $w_i > 0$. On définit aussi $x_i \in \{0, 1\}$ tel que $x_i = 0$ signifie que l'objet n'est pas pris et $x_i = 1$ que l'objet est pris. Si on appelle W le poids maximal que peut porter le sac (et on supposera que $\forall i, w_i < W$), on cherche donc à maximiser la valeur de $V = \sum_i x_i v_i$ en respectant la contrainte $\sum_i x_i w_i \leqslant W$.

On voit ainsi que si on veut traiter toutes les configurations pour déterminer la solution optimale, il faut étudier 2^n configurations (chacun des x_i prenant les deux valeurs possibles). Ainsi, la complexité de l'exploration totale des configurations est exponentiel en n, donc non-polynomial, et le fait d'avoir rapidement une solution pas trop éloignée de la solution optimale peut être intéressant (c'est par exemple le cas de sociétés de transport maritime qui vont préférer envoyer rapidement les conteneurs même s'ils ne sont pas remplis de manière optimale afin d'encaisser les commandes au lieu de payer pour stocker les marchandises le temps de trouver la solution "parfaite").

4.2 Heuristique : algorithmes gloutons

Les algorithmes gloutons sont un exemple d'heuristique bien particuliers : à chaque étape, on va rajouter l'objet le "meilleur" qui ne fait pas dépasser la contrainte. On passe ensuite au suivant, sans ne jamais enlever d'objets (d'où le nom d'algorithme glouton). Il nous faut donc définir la manière dont la qualité des objets est estimée, c'est bien une heuristique. en fonction de la qualité de l'heuristique, la solution déterminée sera plus ou moins proche de la solution optimale, il est donc essentiel de faire attention au choix retenu pour celle-ci. Pour plus d'exemples, voir TD.

4.3 Pour aller plus loin : les méta-heuristiques

Pour améliorer les solutions données par une heuristique, il est courant d'employer des *méta-heuristiques* (étymologiquement "au-delà" des heuristiques. Une des manières de faire est de partir d'une solution donnée par une heuristique et de voir si un léger changement l'améliore ou pas. Si c'est le cas, on réitère l'opération sur la nouvelle solution améliorée, jusqu'à arriver à un optimum local (on ne trouve pas mieux dans le voisinage, mais on ne peut être sûr puisqu'une fois encore, le but est de ne pas explorer exhaustivement toutes les possibilités).

Nous verrons en TD comment implémenter une méta-heuristique dans le cas du problème de voyageur de commerce : la méthode de descente locale qui reprend exactement la méthode donnée ci-dessus (on cherche à chaque tentative d'améliorer la solution).

Il existe encore d'autres méthodes méta-heuristiques dans laquelle on garde certaines tentatives même si la solution obtenue n'est pas meilleure afin d'éviter de tomber dans des minimums locaux. On peut citer comme exemple la méthode du recuit simulé (si la tentative donne une solution moins bonne, alors le programme a une certaine probabilité de s'arrêter, d'autant plus grande que la nouvelle solution est moins bonne) ou la méthode tabou (dans laquelle on stocke pendant un certain temps les solutions les moins bonnes pour ne pas les réévaluer avant qu'elles ne disparaissent).