Chapitre 18 : Intelligence artificielle et théorie des jeux

Table des matières

List of Algorithms

1 Introduction

L'expression *intelligence artificielle* est une manière floue de désigner des algorithmes chargés comme tous les autres de résoudre des problèmes. Certains d'entre eux doivent jouer à des jeux, d'autres doivent répartir des données en plusieurs catégories (on parle de *classification*) ou déterminer des valeurs numériques associées à des paramètres d'entrée (on parle de *problème de régression*). Quelques traits communs à ces algorithmes sont l'usage d'heuristiques afin d'essayer d'obtenir des réponses les meilleures possibles en temps raisonnable et l'exploitation d'une grande quantité de données afin d'en construire une représentation (on parle de l'*apprentissage d'un modèle de données*) qui sera exploité pour construire la réponse de l'algorithme. Dans ce chapitre, on se limite à l'étude des problèmes de classification et de la théorie des jeux.

2 Apprentissage supervisé

2.1 Algorithme des k plus proches voisins

2.1.1 Introduction

Pour résoudre un problème de classification, la méthode de l'apprentissage supervisé consiste à exploiter des données dont on connait déjà la classe afin de construire un algorithme de classification prenant les caractéristiques d'une donnée en entrée et renvoyant la classe à laquelle cette donnée appartient probablement.

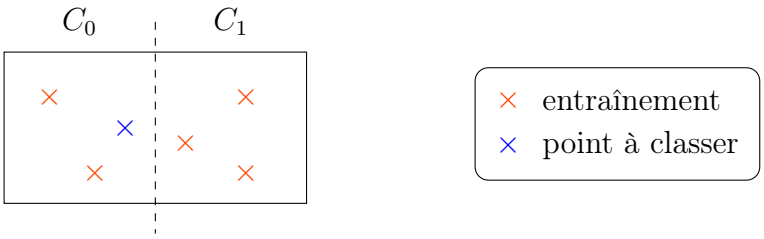
Les données manipulées sont en général représentées par des points de $\mathbb{R}^d$ ou d est souvent grand. Par exemple, si on veut reconnaître des caractères manuscrits, l'algorithme peut prendre en entrée une image de 28×28 pixels en 256 niveaux de gris contenant un scan du caractère manuscrit à reconnaître, donc la donnée est représentée par un point de $[\![0\ ;\ 255]\!]^d$, où $d = 28^2 = 784$.

Si on veut répartir dans C classes des données de $\mathbb{R}^d$, le problème consiste à construire un algorithme réalisant une fonction $\mathbb{R}^d \longrightarrow [\![0\ ;\ C-1]\!]$ en exploitant une heuristique s'appuyant sur un ensemble de couples $(x, y) \in \mathbb{R}^d \times [\![0\ ;\ C-1]\!]$, où y est la classe de x, appelées données d'entrainement.

2.1.2 Algorithme des k plus proches voisins

Idée : il est possible que des données proches appartiennent à la même classe, donc on pourrait pour un point donné, renvoyer la classe du point déjà étiqueté le plus proche.

Problème : la donnée d'entrainement la plus proche peut appartenir à une autre classe que celle du point, par example si les données d'entrainement sont mal réparties, ou bruitées, ou si le point considéré est proche de la frontière entre deux classes.



Pour éviter cet écueil, on considère plutôt la classe majoritaire parmi les classes des k données d'entrainement les plus proches du point d'entrée, pour un k fixé.

On parle alors de l'algorithme des k–plus proches voisins (kNN pour k–*nearest neighbors*).

Variables d'ajustement :

- En cas d'égalité, il faut choisir une classe, par exemple au hasard parmi les classes majoritaires.
- Le nombre k de voisins : si k est trop faible, l'algorithme sera trop sensible au bruit sur les données et si k est trop grand, l'algorithme renverra surtout la classe majoritaire parmi les données d'entraînement, donc effectue une mauvaise généralisation.
- La notion de distance : on utilise souvent la distance de Minkowski

$$d(x, x') = \left(\sum_{i=1}^{d} |x_i - x'_i|^p \right)^{\frac{1}{p}}$$

qui donne la distance de Manhattan pour $p = 1$, et la distance euclidienne pour $p = 2$. Le programme se limite à la distance euclidienne.

Pour déterminer les k plus proches voisins d'un point donné, on peut exploiter une file de priorité :

Algorithm 1: k plus proches voisins (kNN)

Input: données d'entraînement $(x_i, y_i)_{i \in [\![1\ ;\ N]\!]}$
Input: point à classer x

```
F ← file de priorité max vide;
for i de 1 à k do
    Insérer i dans F avec la priorité d(x, x_i);
for i de k + 1 à N do
    if d(x, x_i) < d(x, x_{max F}) then
        Extraire le max de F;
        Insérer i dans F avec la priorité d(x, x_i);
C ← {C_i | i ∈ F};
return un élément le plus fréquent de C;
```

Complexité : $\mathcal{O}(N \log k)$ en temps, et $\mathcal{O}(k)$ en espace.

Remarque : si on a beaucoup de points à classer, cet algorithme est peu efficace car il nécessite de parcourir l'intégralité des données pour chaque point à classer. On pourrait

plutôt effectuer un prétraitement des données pour rendre plus efficace le calcul des k plus proches voisins d'un point donné.

2.1.3 Arbres k–dimensionnels

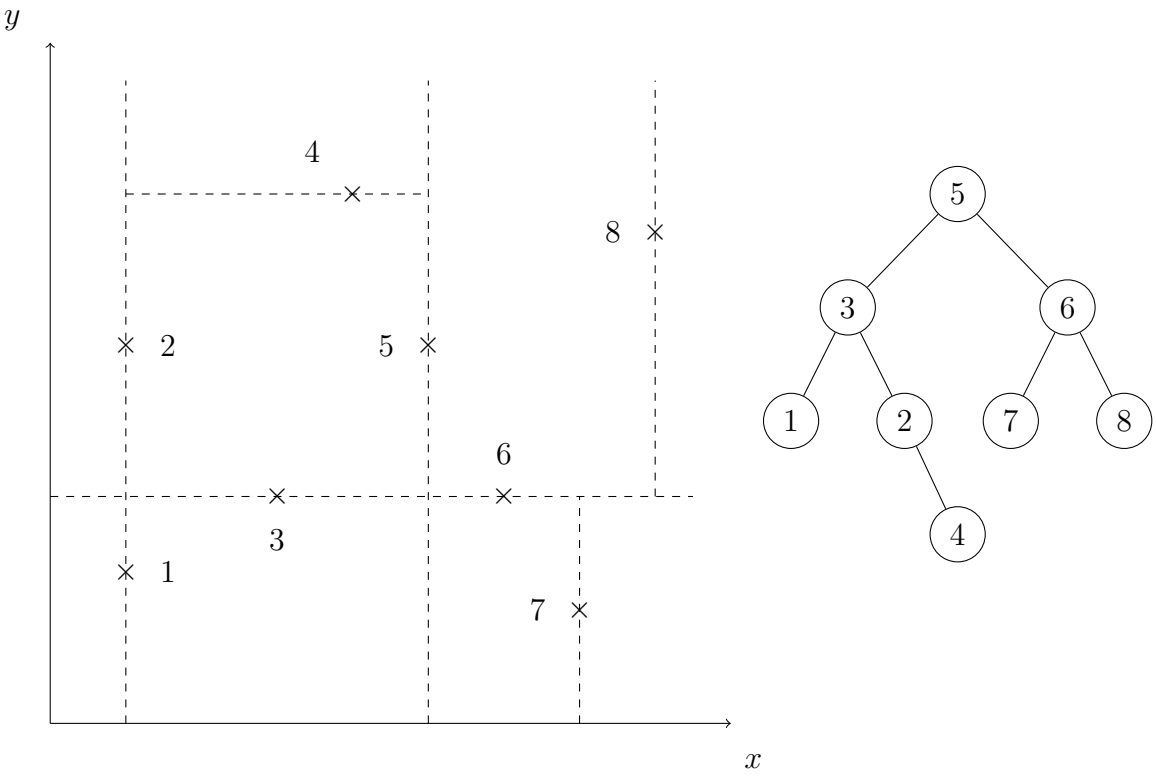
Attention : on ne parle pas du k de kNN, mais plutôt de la dimension de l'espace des données (d en 2.1.1, page 3), mais c'est la lettre k qui est utilisée dans la littérature.

• Définition : la structure d'arbre k–dimensionnel, ou arbre k-d, est une généralisation de la notion d'arbre binaire de recherche : un arbre binaire étiqueté par des éléments de $\mathbb{R}^k$ est un arbre k-d si et seulement si pour tout nœud d'étiquette $x = (x_0, \ldots, x_{k-1})$ de profondeur i,

$$\begin{array}{ll} \forall x' = (x'_0, \ldots, x'_{k-1}) \text{ étiquette du sous-arbre gauche}, & x'_j \leqslant x_j \\ \forall x' = (x'_0, \ldots, x'_{k-1}) \text{ étiquette du sous-arbre droit}, & x'_j > x_j \end{array}$$

où $j = i \mod k$.

• Exemple en dimension 2 :



• Remarque : dans cet exemple, on a fait en sorte de construire un arbre k-d équilibré en choisissant l'élément médian pour la coordonnée associée à la profondeur du nœud comme étiquette.

On écrit l'algorithme `créer_arbre` suivant :

Algorithm 2: Création de l'arbre k–d

```
Function créer_arbre(k, i, l):
    Input: dimension k
    Input: profondeur i
    Input: liste de données l
    if l = [] then
        return l'arbre vide;
    else
        Extraire l'élément x de l médian pour la coordonnée i mod k;
        l_<, l_> ← partition de l suivant le pivot x;
        return Noeud(x, créer_arbre(k, i + 1, l_<), créer_arbre(k,
          (i + 1), l_>));
```

Complexité : à l'aide de l'algorithme de calcul de la médiane en temps linéaire (*cf* chapitre 7, 2.2), la complexité est celle du tri rapide dans le meilleur cas, *i.e* $\mathcal{O}(N \log N)$.

2.1.4 Recherche des n plus proches voisins dans un arbre k–d

Idée : pour trouver les n plus proches voisins d'un point $x \in \mathbb{R}^d$ dans un arbre k-d T de dimension d, on procède initialement comme pour la recherche de x dans un ABR (en comparant la bonne coordonnée à chaque profondeur) puis on remonte dans l'arbre en sélectionnant les n voisins, parfois en redescendant dans un sous-arbre que l'on avait ignoré.

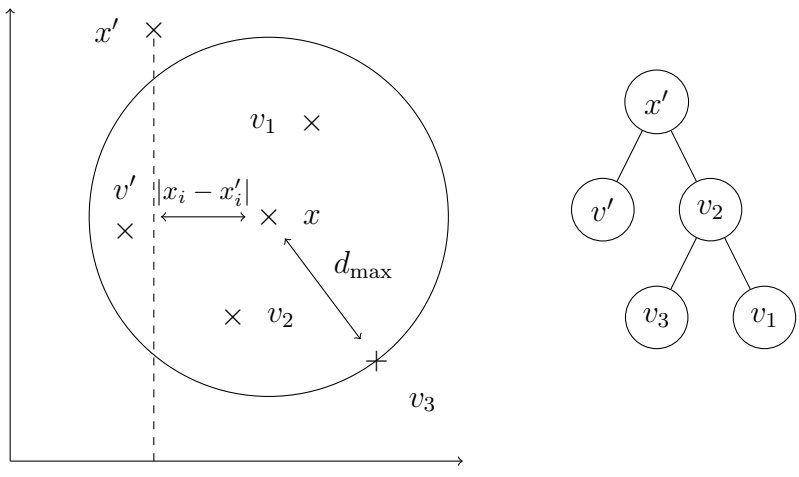
Exemple : on suppose être au niveau d'un nœud d'étiquette x' de profondeur i et tel que $x_i > x'_i$.

On cherche donc récursivement les n voisins dans le sous-arbre droit.

S'il y a moins de n nœuds dans ce sous-arbre, il faudra considérer x' comme voisin et peut-être aussi les nœuds du sous-arbre gauche.

Même si l'appel récursif sélectionne n voisins, on peut devoir considérer x' ou les nœuds du sous-arbre gauche si $|x_i - x'_i|$ est inférieure à la distance maximale entre x et les voisins sélectionnés.

Par exemple, si $n = 3, d = 2(= k)$, et i correspond aux abscisses,



D'où l'algorithme :

Algorithm 3: Recherche des n plus proches voisins dans un arbre k–d

```
1  Function recherche_voisins(x, T, n):
2      F ← file de priorité max vide;
3      visite(F, x, T, 0, n);
4      return les éléments de F;
5  Function visite(F, x, T, i, n):
6      if T = Noeud(x', l, r) then
7          if x_i ⩽ x'_i then
8              t_1, t_2 ← l, r;
9          else
10             t_1, t_2 ← r, l;
11         visite(F, x, t_1, i + 1 mod d, n);
12         if |F| < n ou priorité max F ⩾ |x_i − x_j| then
13             if d(x, x') < priorité max F then
14                 Extraire max F;
15                 Insérer x' dans F avec la priorité d(x, x');
16             visite(F, x, t_2, i + 1 mod d, n);
```

Dans le pire cas, on visite les N nœuds de l'arbre (donc on n'a rien gagné par rapport au premier algorithme) mais le plus souvent on ne visite que de l'ordre de $\log N$ nœuds.

2.2 Arbres de décision

2.2.1 Introduction

Un arbre de décision est un outil permettant d'implémenter un algorithme de classification dont le fonctionnement est le suivant : étant donné un point à classer, on descend récursivement dans l'arbre en effectuant à chaque nœud un test sur une coordonnée dont le résultat détermine la branche à parcourir. La feuille atteinte donne la classe du point.

2.2.2 Définition (*arbre de décision*)

Un *arbre décision* est un arbre étiqueté tel que les étiquettes des nœuds internes sont des coordonnées et celles des feuilles sont des classes, de telle sorte que les fils d'un nœud donné correspond aux différentes valeurs possibles pour la coordonnée associée au nœud.

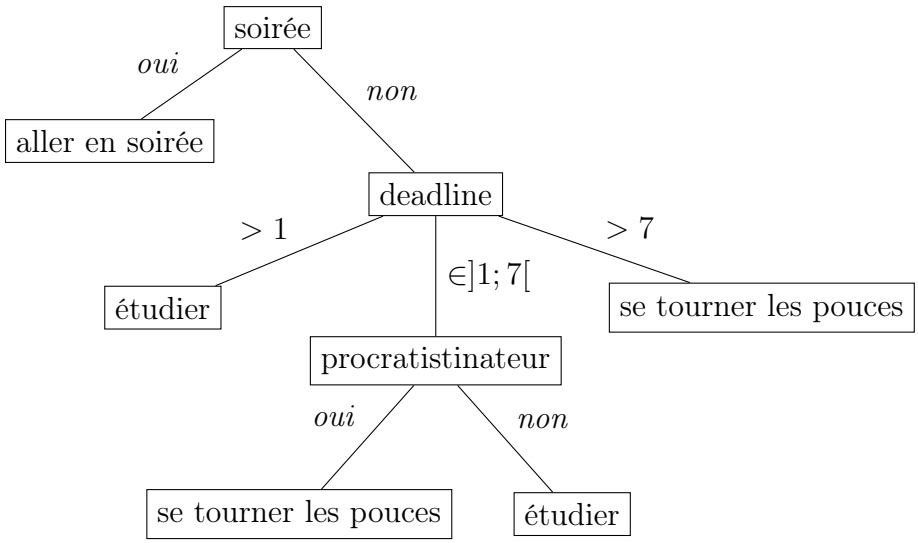
Si la coordonnée est *catégorique*, *i.e* ne peut prendre qu'un nombre fini de valeur, alors il y a autant de fils que de valeurs. Si la coordonnée est *numérique*, les fils correspondent à des intervalles de valeurs disjoints.

2.2.3 Exemple

Un étudiant veut savoir comment passer la soirée, sachant qu'il y a trois activités possibles : aller en soirée, étudier, ou se tourner les pouces. On représente les étudiants par des triplets dont les coordonnées sont :

- un booléen indiquant s'il y a une soirée organisée par son cercle d'amis ;
- Un réel > 0 indiquant le nombre de jour avant la prochaine deadline (exemple : devoir) ;
- Un booléen indiquant si l'étudiant est procrastinateur.

On pourrait construire l'arbre suivant :



L'étudiant $(V, 0.2, F)$ va en soirée
L'étudiant $(F, 4, V)$ va se tourner les pouces.

Remarque : dans le cadre de l'apprentissage supervisé, la question est de savoir comment construire un arbre de décision à partir de données étiquetées. Il faut donc choisir la structure de l'arbre et les tests effectués à chaque nœud. Il est possible de construire un arbre qui classe parfaitement toutes les données d'entraînement ou alors de s'autoriser des erreurs pour tenir compte d'un éventuel bruit sur les données d'entraînement.

2.2.4 Entropie de Shannon

• Idée : pour construire l'arbre de décision, une idée serait de placer en racine la coordonnée qui discrimine au mieux entre les différentes classes pour le jeu de données et de construire les sous-arbres récursivement. On utilise pour cela la notion d'*entropie de* Shannon, qui est une mesure de l'information contenue dans un jeu de données.

• Définition (*entropie de* Shannon) : on considère un ensemble S de N données réparties dans C classes $C_0, \dots, C_{C-1}$.

L'entropie de SHANNON de S est la quantité

$$H(S) = -\sum_{i=0}^{C-1} \frac{|C_i|}{N} \log_2 \frac{|C_i|}{N}$$

• Remarque : L'entropie de SHANNON est l'espérance du nombre de bits d'informations que l'on obtient en tirant aléatoirement uniformément un point de S.
En effet, $\frac{|C_i|}{N}$ est la probabilité d'obtenir un élément de la classe C_i, et $-\log_2 \frac{|C_i|}{N} = \log_2 N - \log_2 |C_i|$ est la différence entre le nombre de bits nécessaires pour représenter l'intégralité des données et le nombre de bits nécessaires pour distinguer entre elles les données de la classe C_i. C'est donc le nombre de bits qui reste afin de donner de l'information sur l'appartenance à la classe C_i.
En particulier,

- Si $C = 1$, alors $H(S) = 0$: la représentation des données sert uniquement à les distinguer entre elles et n'apporte aucune information sur la classe.
- L'entropie est maximale lorsque les données sont uniformément réparties $\left(i.e \text{ quand } \forall i \in [\![0\ ;\ C-1]\!],\ |C_i| = \frac{N}{C}\right)$. Dans ce cas, $H(S) = \log_2 C$, et on obtient autant de bits que nécessaire pour distinguer les C classes.

• Définition (*gain d'un attribut*) : On considère un ensemble S de N données en dimension d, et une coordonnée (ou un attribut) $i \in [\![1\ ;\ d]\!]$ dont on note m le nombre de valeurs possibles.
Le *gain* de l'attribut i est la quantité

$$G(S,i) = H(S) - \sum_{j=1}^{m} \frac{|S_j|}{N} H(S_j)$$

où $S_1, \ldots, S_m$ est la partition de S suivant les valeurs de la coordonnée i.

• Remarque : le gain de l'attribut correspond à l'espérance de la perte d'entropie lorsque l'on fixe la valeur de l'attribut.

$$\begin{aligned} G(S,i) &= H(S) - \sum_{j=1}^{m} \frac{|S_j|}{N} H(S_j) \\ &= \sum_{j=1}^{m} \frac{|S_j|}{N} H(S) - \sum_{j=1}^{m} \frac{|S_j|}{N} H(S_j) \\ &= \sum_{j=1}^{m} \frac{|S_j|}{N} (H(S) - H(S_j)) \end{aligned}$$

où $\frac{|S_j|}{N}$ est la probabilité de tirer un point dont la coordonnée i prend la $j^{\text{ème}}$ valeur, et $H(S) - H(S_j)$ correspond à la différence entre l'entropie du jeu de données initial et l'entropie des données restantes après le choix de la $j^{\text{ème}}$ valeur.

Plus un attribut est discriminant pour les classes, plus l'entropie des données restantes après le choix d'un valeur pour l'attribut doit être faible. On cherche donc à maximiser le gain de l'attribut.

• Proposition (Inégalité de Gibls) :

> Si $(p_i)_{i\in[\![0\ ;\ k-1]\!]}$, et $(q_i)_{i\in[\![0\ ;\ k-1]\!]}$ sont deux distributions de probabilité sur un ensemble de cardinal k, alors
>
> $$-\sum_{i=0}^{k-1} p_i \log_2(p_i) \leqslant -\sum_{i=0}^{k-1} p_i \log_2(q_i)$$

□

$\forall x \in \mathbb{R}_+^*,\ \ln(x) \leqslant x-1$ (concavité de ln).

Donc

$$\begin{aligned}\sum_{i=0}^{k-1} p_i \ln\left(\frac{q_i}{p_i}\right) &\leqslant \sum_{i=0}^{k-1} p_i\left(\frac{q_i}{p_i}-1\right)\\ &= \sum_{i=0}^{k-1}(q_i-p_i)\\ &= 1-1=0\end{aligned}$$

Donc

$$\sum_{i=0}^{k-1} p_i \ln q_i \leqslant \sum_{i=0}^{k-1} p_i \ln p_i$$

ce qui conclut car $\dfrac{-1}{\ln 2} < 0$. ■

• Corollaire :

> Si S est un ensemble de N données en dimension d réparties dans les classes $C_0, \dots, C_{C-1}$, et si $i \in [\![1\ ;\ d]\!]$ est un attribut pouvant prendre une valeur, alors $G(S,i) \geqslant 0$.

□

$$\begin{aligned}G(S,i) &= H(S) - \sum_{j=1}^{m} \frac{|S_j|}{N} H(S_j)\\ &= -\sum_{k=0}^{C-1} \frac{|C_k|}{N} \log_2 \frac{|C_k|}{N} - \sum_{j=1}^{m} \frac{|S_j|}{N}\left(-\sum_{k=0}^{C-1} \frac{|C_k\cap S_j|}{|S_j|} \log_2 \frac{|C_k\cap S_j|}{|S_j|}\right)\\ &= -\sum_{k=0}^{C-1}\sum_{j=1}^{m} \frac{|C_k\cap S_j|}{N} \log_2 \frac{|C_k|}{N} - \left(-\sum_{j=1}^{m}\sum_{k=0}^{C-1} \frac{|C_k\cap S_j|}{N} \log_2 \frac{|C_k\cap S_j|}{|S_j|}\right)\end{aligned}$$

Or

$$\log_2 \frac{|C_k\cap S_j|}{|S_j|} = \log_2 \frac{|C_k\cap S_j|}{N} - \log_2 \frac{|S_j|}{N}$$

Donc

$$G(S,i) = -\sum_{k=0}^{C-1}\sum_{j=1}^{m}\frac{|C_i\cap S_j|}{N}\left(\log_2\frac{|C_k|}{N}+\log_2\frac{|S_j|}{N}\right)+\sum_{j=1}^{m}\sum_{k=0}^{C-1}\frac{|C_k\cap S_j|}{N}\log_2\frac{|C_k\cap S_j|}{N}$$

On note $(p_{j,k})_{(j,k)\in[\![1\ ;\ n]\!]\times[\![0\ ;\ C-1]\!]}=\left(\frac{|C_k\cap S_j|}{N}\right)_{j,k}$ et $(q_{j,k})_{j,k}=\left(\frac{|C_k|\,|S_j|}{N^2}\right)_{j,k}$ et on remarque que $(p_{j,k})_{j,k}$ et $(q_{j,k})_{j,k}$ sont des distributions de probabilité, donc l'inégalité de Gibls donne :

$$-\sum_{j,k}p_{j,k}\log_2 p_{j,k}\leqslant-\sum_{j,k}p_{j,k}\log_2 q_{j,k}$$

d'où $G(S,i)\geqslant 0$. ■

2.2.5 Algorithme ID3

• Principe : l'algorithme ID3 (pour *Iterative Dichotomiser 3*) consiste à construire un arbre de décision en choisissant pour racine l'attribut de gain maximal et en construisant récursivement les sous-arbres en considérant comme jeu de données la partition des données selon la valeur de l'attribut choisi.

• Cas binaire : on énonce l'algorithme ID3 dans le cas où les attributs sont binaires.

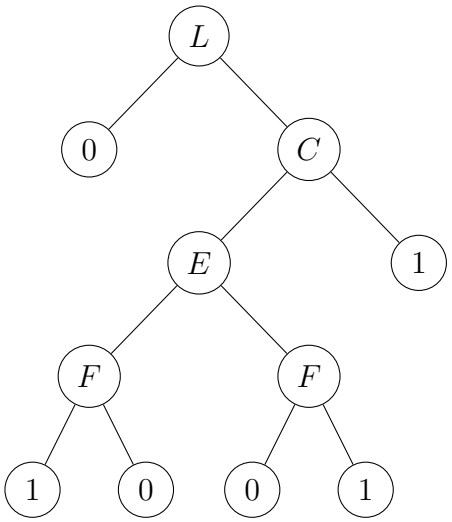
Algorithm 4: ID3, cas binaire

Input: L'ensemble S des données
Input: L'ensemble A des attributs que l'on s'autorise à utiliser dans les nœuds

if $S=\varnothing$ **then**
  Construire une feuille contenant la classe la plus représentée parmi les données qui ont servi à construire son père;
else if tous les éléments de S appartient à la même classe C **then**
  Construire une feuille contenant C;
else if $A=\varnothing$ **then**
  Construire une feuille contenant la classe la plus représentée parmi les données de S;
else
  $i\leftarrow$ attribut de A qui maximise $G(S,i)$;
  $S_0\leftarrow$ données de S dont l'attribut i vaut 0;
  $S_1\leftarrow$ données de S dont l'attribut i vaut 1;
  Construire le nœud d'étiquette i, de sous-arbre gauche `ID3`$(S_0, A\setminus\{i\})$, de sous-arbre droit `ID3`$(S_1, A\setminus\{i\})$;

• Exemple :

Attributs				Classe
E	C	F	L	R
1	1	1	1	1
1	1	1	0	0
1	1	0	1	1
1	1	0	0	0
1	0	1	1	1
1	0	1	0	0
1	0	0	1	0
1	0	0	0	0
0	0	1	1	0
0	0	1	0	0
0	0	0	1	1
0	0	0	0	0



2.3 Analyse des résultats

2.3.1 Introduction

Les algorithmes d'apprentissage dépendent de paramètres (par exemple le nombre k de voisins dans kNN ou une heuristique d'apprentissage d'arbres de décision telle qu'ID3) qu'il faut ajuster selon le cas d'application afin d'obtenir le meilleur classificateur possible.

Choisir les meilleurs paramètres pour un algorithme d'apprentissage passe par l'essai de plusieurs valeurs de ces paramètres et par une évaluation du classificateur obtenu.

On a donc besoin d'un moyen de mesurer els performances d'un classificateur.

2.3.2 Jeu d'entraînement, jeu de test

Dans le cadre de l'apprentissage supervisé, on dispose de données étiquetées par le résultat attendu, donc il est possible d'évaluer les performances d'un classificateur en comparant le résultat qu'il renvoie et le résultat attendu sur chacune des données étiquettées.

Cependant, il faut éviter d'utiliser les mêmes données pour l'apprentissage et pour les tests car l'algorithme est censé renvoyer de bons résultats sur les données d'entraînement. On n'aurait donc pas une bonne mesure de la capacité de l'algorithme à généraliser le modèle qu'il apprend. Il est alors d'usage de répartir les données en deux jeux : un jeu d'entraînement qui contient les données qui serviront à l'apprentissage, et un jeu de test qui contient les données qui serviront aux mesures de performance.

Construire ces jeux de données est un enjeux majeur car de mauvais choix peuvent donner des résultats grossièrement faux. Par exemple, si on applique l'algorithme ID3 à un jeu d'entraînement dont toutes les données appartiennent à la même classe, l'arbre de décision obtenu est une feuille réduite à cette classe donc le classificateur est une

fonction constante qui se trompera sur toutes les données d'autres classes. Disposer d'une grande quantité de données d'entraînement est aussi important.

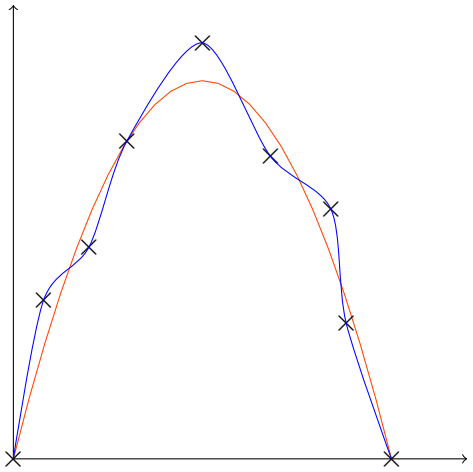
Les données sont en général réparties entre les deux jeux avec de l'ordre de 80% de données d'entraînement et 20% de données de test. Une manière de respecter la répartition des classes dans le jeu d'entraînement consiste à tirer les données d'entraînement avec un tirage aléatoire uniforme.

2.3.3 Sur-apprentissage / sous-apprentissage

L'objectif de l'apprentissage est de deviner des règles implicites qui régissent la distribution des classes, à partir d'un jeu de données, et de généraliser ces règles pour construire un modèle permettant de déterminer la classe de données pour lesquelles on ne dispose pas d'information. Le sur-apprentissage consiste en une mauvaise généralisation liée à une trop grande précision sur le jeu d'entraînement.

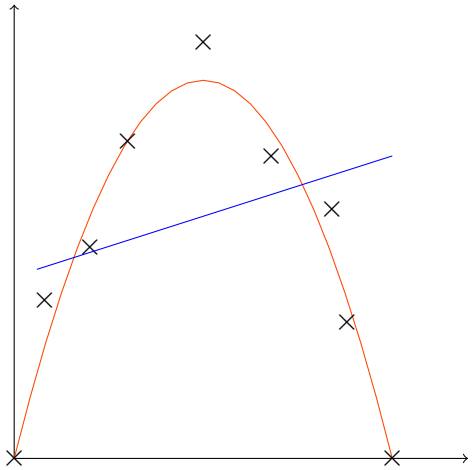
Par exemple, pour un algorithme de régression, si les données sont bruitées, il y a sur-apprentissage si la fonction produite par l'algorithme "colle" trop aux données d'entraînement et reproduit le bruit des données au lieu d'apprendre uniquement la tendance générale.

Par exemple, il est toujours possible d'obtenir une erreur nulle sur les données d'entraînement par interpolation, mais cela ne garantit pas la qualité de la régression en dehors de ces données.



Il y a sous-apprentissage lorsque l'algorithme ne tient pas assez compte des données d'entraînement et n'apprend donc pas assez de règles pour obtenir un bon résultat.

Par exemple : régression linéaire sur les mêmes données.



Pour des algorithmes d'apprentissage qui ajustent leurs paramètres en cours d'apprentissage, il est alors nécessaire d'avoir un troisième jeu de données, appelé jeu de validation, qui sert à détecter le sur-apprentissage afin d'interrompre l'algorithme. Une répartition fréquente des données entre jeu d'entraînement, jeu de validation, et jeu de test est de l'ordre de 60% / 20% / 20%.

2.3.4 Matrice de confusion

• Remarque : les résultats obtenus par exécution du classificateur résultant de l'apprentissage sur les données du jeu de test permettent de calculer le taux d'erreur, *i.e* le rapport entre le nombre de tests pour lesquels le résultat est faux et le nombre total de tests.

Cependant, ce taux ne donne qu'une information partielle sur les performances de classification car il n'indique rien sur la nature des erreurs. Dans le cadre des problèmes de classification, la notion de *matrice de confusion* permet d'effectuer une analyse plus poussée.

• Définition (*matrice de confusion*) : on considère un problème de classification à C classes $C_0, \dots, C_{C-1}$.

La *matrice de confusion* d'un classificateur pour un jeu de test donné est la matrice de taille $C \times C$ telle que $\forall i, j \in [\![0\ ;\ C-1]\!]$, le coefficient (i, j) de la matrice est le nombre de données de test appartenant à la classe C_i pour lesquelles le classificateur a renvoyé la classe C_j.

• La matrice suivante pour un problème à trois classes :

$$\begin{pmatrix} 6 & 0 & 2 \\ 0 & 5 & 0 \\ 1 & 1 & 5 \end{pmatrix}$$

Les réponses correctes sont situées la diagonale donc le taux d'erreur est la somme des valeurs qui ne sont pas sur la diagonale divisée par la somme de toutes les valeurs. Ici : $\frac{4}{20} = \frac{1}{5}$, soit 20%.

La matrice indique également :

- que le classificateur ne se trompe jamais sur les données de la classe C_1 ;
- que lorsque le classificateur se trompe sur une donnée de la classe C_0, c'est qu'il y a confusion uniquement avec la classe C_2 et qu'il y a $\frac{2}{8} = \frac{1}{4}$ soit 25% des données sur lesquelles il y a confusion ;
- le taux d'erreur pour les données de chaque classe ;
- que lorsque le classificateur répond C_0, il donne une bonne réponse dans $\frac{6}{7}$, soit environ 85.7% des cas ;
- etc.

2.3.5 Mesures de précision dans le cas d'un classificateur binaire (H.P)

• Dans le cas d'un problème de classification binaire, le classificateur n'a que deux réponses possibles, que l'on peut assimiler à "vrai" et "faux". C'est donc un algorithme qui essaie de résoudre un problème de décision.

La matrice de confusion pour un tel algorithme est de taille 2×2, et si on assimile la classe C_0 à "vrai" et la classe C_1 à "faux", on dit que le coefficient

- $(0, 0)$ représente les vrai positifs ;
- $(0, 1)$ représente les faux négatifs ;
- $(1, 0)$ représente les faux positifs ;
- $(1, 1)$ représente les vrai négatifs.

Dans ce cas, il est d'usage de définir certaines quantités, appelées *mesures de précision*, qui permettent d'évaluer la qualité du classificateur.

• Définition : étant donné une matrice de confusion 2×2, on définit :

- la *sensibilité* : c'est le taux de vrais positifs, *i.e* le rapport entre le nombre de vrais positifs et le nombre de données positives (vrais positifs et faux négatifs)

  Cela représente la capacité du classificateur à reconnaître des données de la classe C_0.

- La *spécificité* : c'est le taux de vrais négatifs, donc cela représente la capacité du classificateur à reconnaître les données de la classe C_1.

- La *précision* : c'est le rapport entre le nombre de vrais positifs et le nombre de réponses positives de classificateur. Cela représente la fiabilité du classificateur lorsqu'il répond "vrai".

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

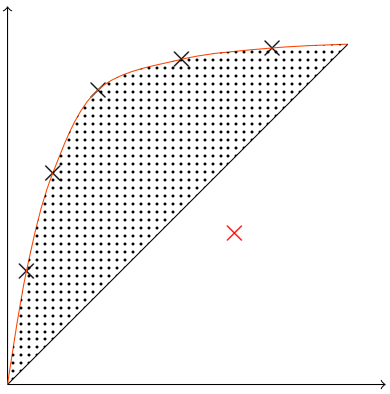
Sensibilité : $\dfrac{a}{a+b}$

Spécificité : $\dfrac{d}{c+d}$

Précision : $\dfrac{a}{a+c}$

• Remarque : au-delà de ces mesures, on peut également tracer la courbe ROC (*Receiver Operator Characteristic*) qui représente en abscisse le pourcentage de faux positifs et en ordonnées le pourcentage de vrais positifs. Un point correspond à une exécution d'un classificateur sur un jeu de test. On fait en général varier les paramètres d'un algorithme d'apprentissage et on place des points pour chacun des classificateurs obtenus, ce qui permet d'obtenir une courbe, appelée courbe ROC, représentant les performances de l'algorithme d'apprentissage.

Exemple :



La diagonale représente les cas où on renvoie autant de bonnes réponses que de mauvaises. Cela représente donc un algorithme qui ferait un tirage aléatoire pour répondre.

Un point situé sous cette courbe représente un algorithme qui ferait moins bien que le hasard.

On mesure en général la qualité d'un algorithme d'apprentissage par la valeur de l'aire située entre sa courbe ROC et la diagonale.

L'objectif est de trouver un algorithme pour lequel cette aire est maximale puis d'ajuster ses paramètres pour obtenir un classificateur dont le point est proche de celui du coin supérieur gauche.

3 Apprentissage non supervisé

3.1 Classification hiérarchique ascendante

3.1.1 Introduction

Dans le cadre de l'apprentissage non supervisé, les données dont on dispose ne sont plus étiquetées par l'information que l'algorithme d'apprentissage doit savoir retrouver. L'algorithme doit plutôt inférer de nouvelles connaissances sur les données en découvrant une structure sous-jacente aux données.

Dans ce cadre, le problème de classification est plutôt appelé problème de *regroupement* (*clustering*) car il ne s'agit plus de redécouvrir des classes existantes, mais d'en

former en regroupant les données d'entraînement, de sorte à maximiser la similarité des données au sein des classes et la dissimilarité entre données de classe différentes.

3.1.2 Classification hiérarchique ascendante

L'algorithme de classification ascendante (CHA), ou regroupement hiérarchique ascendant, consiste à partir de données isolées dans des classes singletons et à fusionner progressivement des classes jusqu'à la réalisation d'une condition d'arrêt.

Pour choisir les classes à fusionner, on se donne une mesure de la dissimilarité entre les classes, que l'on appellera abusivement *distance*, et on fusionne les deux classes les plus proches.

Pour définir une distance sur les classes, on prend une distance d sur les données, et on peut par exemple choisir parmi les distances suivantes :

– La distance minimale :

$$d(C_1, C_2) = \min_{(x,y)\in C_1\times C_2} d(x, y)$$

qui correspond à la notion naturelle de distance en mathématiques ;

– La distance maximale :

$$d(C_1, C_2) = \max_{(x,y)\in C_1\times C_2} d(x, y)$$

qui permet de créer des classes de faible "diamètre" ;

– La distance des centres de gravité :

$$d(C_1, C_2) = d(b_1, b_2)$$

où b_1 et b_2 sont les (iso)barycentres de C_1, C_2 ;

– La distance de Ward :

$$d(C_1, C_2) = \sqrt{\frac{|C_1|\,|C_2|}{|C_1| + |C_2|}} d(b_1, b_2)$$

Les distances de Ward et des centres de gravité donnent des algorithmes qui sont moins sensibles au bruit sur les données.

Une contrainte importante est la monotonie de la distance : la fusion de deux classes ne doit par réduire la distance aux autres classes, *i.e*

$$\min(d(C_1, C_2), d(C_1, C_3)) \leqslant d(C_1, C_2 \cup C_3)$$

Une fois la distance choisie, l'algorithme s'exprime ainsi :

Algorithm 5: Classification hiérarchique ascendante (CHA)

```
for chaque donnée x do
    Créer une classe réduite à {x};
while condition d'arrêt n'est pas satisfaite do
    C_1, C_2 ← classes qui minimisent d(C_1, C_2);
    Fusionner C_1 et C_2;
return l'ensemble des classes;
```

Il y a plusieurs choix possibles de condition d'arrêt : un nombre prédéterminé de classes à atteindre, un majorant sur la distance entre deux classes que l'on s'autorise à fusionner, un majorant sur la distance entre les données d'une même classe, ...

3.1.3 Avantages et inconvénients

Un inconvénient de cet algorithme est sa complexité : lorsque le jeu de données est grand (ce qui est en général le cas), le temps de calcul explose, ce qui en fait un algorithme peu pratique.

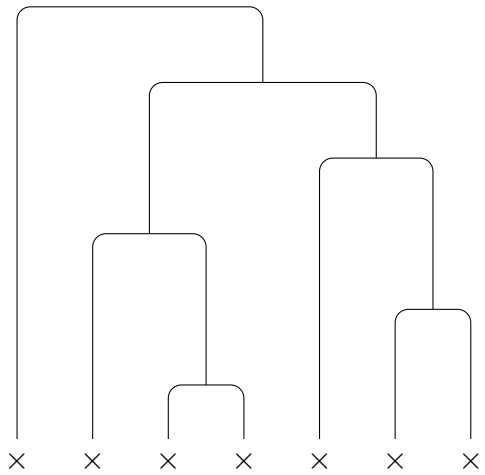
Un autre inconvénient réside dans l'absence de remise en question. Lorsque des données sont réunies dans une même classes, on ne peut plus les séparer.

Le principal avantage de cet algorithme est sa souplesse tirée de la grande variété de distances et de conditions d'arrêt que l'on peut choisir. De plus, on n'a pas besoin de connaître *a priori* le nombre de classes.

3.1.4 Pourquoi *hiérarchique* ?

L'algorithme CHA est qualifié de *hiérarchique* car on peut en tirer une structure arborescente (en fait une forêt). Pour cela, on dessine un *dendrogramme*, *i.e* un graphe représentant l'organisation hiérarchique des fusions.

Exemple :



Remarque : On distingue la classification hiérarchique ascendante de la classification hiérarchique descendante qui consiste à partir d'une seule classe réunissant toutes les

données et à chaque itération à partionner l'une des classes en deux sous-classes. La structure arborescente obtenue ressemble alors aux arbres de décision car les nœuds internes permettent de séparer des données et les feuilles contiennent les classes construites.

3.2 Algorithme des k-moyennes

3.2.1 Introduction

L'objectif de l'algorithme des k-moyennes consiste à répartir les données dans un nombre prédéterminé k de classes (à ne pas confondre avec le k de kNN ou des arbres k-d). L'algorithme fonctionne en plusieurs itérations et les données peuvent changer de classe au cours des itérations.

3.2.2 Algorithme des k-moyennes

On se donne une distance d sur l'espace des données.

Code :

Algorithm 6: k-moyennes

```
μ_0, ..., μ_{k-1} ← k données arbitraires;
while pas de convergence (i.e tant que les μ_i, C_i changent) do
    for chaque donnée x do
        Associer x à la classe C_i telle que d(x, μ_i) est minimale;
    for chaque i ∈ ⟦0 ; k − 1⟧ do
        μ_i ← (iso)barycentre de C_i;
return les C_i;
```

Le principe de cet algorithme est de partir d'un regroupement arbitraire (souvent aléatoire) et de l'ajuster jusqu'à l'obtention d'un résultat "satisfaisant", en considérant que l'on classe les données par rapport à la distance de ces données aux centres des classes. La définition de ce qu'est un résultat satisfaisant vient du problème d'optimisation que l'on cherche à résoudre : étant donné un ensemble fini $D \subseteq \mathbb{R}^p$ et un entier $k \in \mathbb{N}^*$, déterminer une partition $(C_0, \dots, C_{k-1})$ de D, qui minimise

$$\sum_{i=0}^{k-1} \sum_{x \in C_i} d(x, \mu_i)^2$$

où

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

est l'isobarycentre de C_i.

3.2.3 Analyse de l'algorithme

On suppose ici que d est la distance euclidienne.

• Proposition [admise] :

> La fonction objectif
> $$\sum_{i=0}^{k-1} \sum_{x \in C_i} d(x, \mu_i)^2$$
> décroit au cours des itérations et l'algorithme termine.

• Corollaire :

> L'algorithme des k-moyennes détermine un minimum local pour la fonction de coût.

• Remarque : ce minimum n'est pas nécessairement global.

Par exemple : on prend $p = 1$, $D = \{1, 2, 3, 4\}$, et $k = 2$.

On suppose qu'en cas d'égalité $d(x, \mu_0) = d(x, \mu_1)$, le point x est affecté à la classe C_i telle que $\mu_i < \mu_{1-i}$.

Si les points initiaux sont $\mu_0 = 1$ et $\mu_1 = 4$,

$\mu_0 = 1.5,\ \mu_1 = 3.5$.

Les μ_i ne changent pas, l'algorithme se termine.

Le regroupement est de coût $4 \cdot \left(\frac{1}{2}\right)^2 = 1$

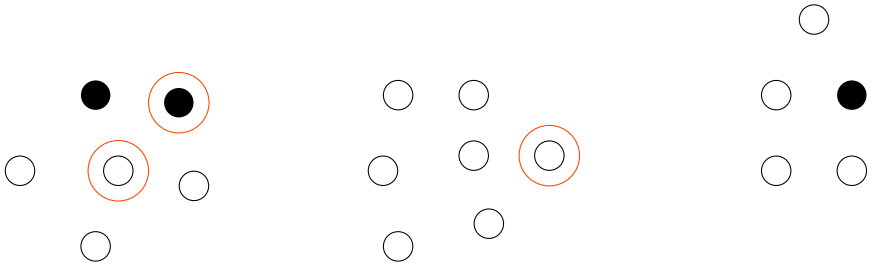
Si les points initiaux sont plutôt $\mu_0 = 2$ et $\mu_1 = 4$, on obtient :

Les μ_i ne changent pas, et l'algorithme se termine.

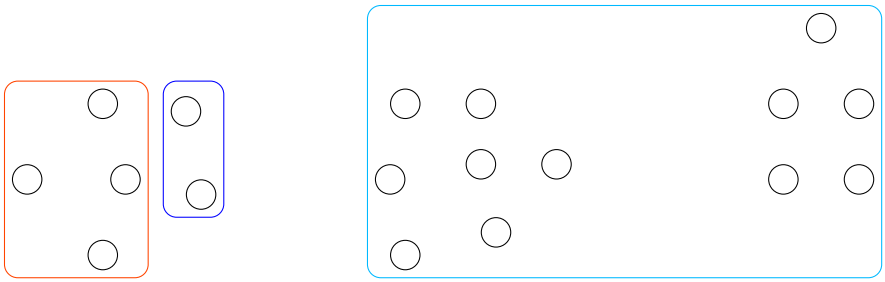
Le regroupement est de coût $1^2 + 0^2 + 1^2 + 0^2 = 2$.

Ce problème peut également survenir lorsque les données ont une répartition "naturelle" entre les classes.

Par exemple, pour $p = 2$ et $k = 3$, avec les données suivantes :



Les points initiaux colorées en noir donnent bien la répartition que l'on attend.
Les points cerclés en orange donnent le regroupement suivant :



En pratique, on exécute plusieurs fois l'algorithme avec une sélection aléatoire des points initiaux et on conserve le "meilleur" résultat. Cela nécessite en général de construire un jeu de test étiqueté ou de définir une mesure de la qualité d'un regroupement, ce qui est la difficulté majeure de l'apprentissage non supervisé.

Avantages :

- L'algorithme des k-moyennes est bien plus rapide que l'algorithme CHA et, de plus, il n'est pas nécessaire d'attendre la convergence pour obtenir un résultat acceptable ;
- Cet algorithme présente également une certaine souplesse :
  + Grand choix de distances possibles ;
  + On peut initialiser l'algorithme de manières différentes (par exemple : tirage aléatoire de la classe initiale de chaque donnée ou tirage aléatoire des k centres parmi les points de l'ensemble convexe des données sans que ces points soient nécessairement des données) ;
  + Possibilité de calculer les "centres" autrement que par un calcul de barycentre.

Inconvénients :

- Le minimum local atteint n'est pas forcément le regroupement recherché ;
- Il faut connaître *a priori* le nombre de classes recherché ;
- On fait l'hypothèse que les données sont réparties dans des classes relativement convexe (*cf* diagramme de Voronoï)

4 Jeux à deux joueurs

4.1 Modélisation

4.1.1 Introduction

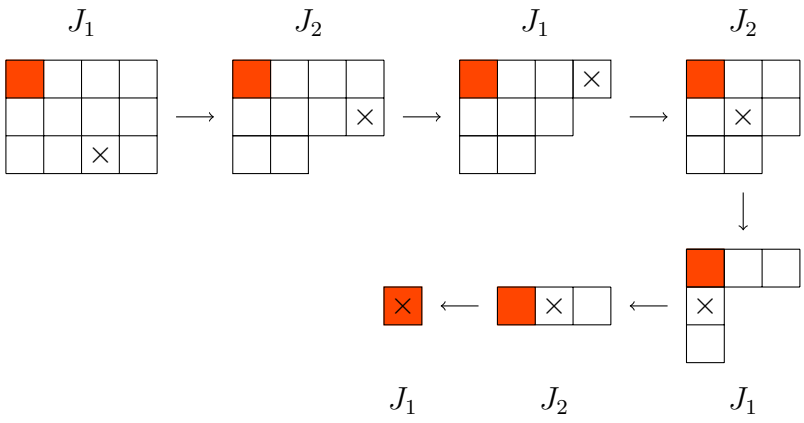
L'objectif de cette partie est d'introduire les concepts de base de la théorie des jeux, en se focalisant sur les jeux à deux joueurs, appelés J_1 et J_2, sans hasard (donc pas de jeux de dés par exemple), à information complète, *i.e* les joueurs ont la connaissance

parfaite de l'état du jeu à tout moment (donc la plupart des jeux de cartes seront pas considérés). On se limite également à des jeux d'accessibilité, *i.e* pour lesquels la condition de victoire est d'atteindre un certain état du jeu. Les notions abordées doivent permettre l'écriture d'un programme capable de jouer au jeu étudié.

4.1.2 Exemple : le jeu de Chomp

• Définition : dans le jeu de Chomp, J_1 et J_2 se disputent une tablette de chocolat rectangulaire, dont le carré dans le coin supérieur gauche est empoisonné. Tour à tour, les joueurs choisissent un carré, et mangent les carrés situé en dessous et à droite de celui-ci. Le joueur contraint à manger le carré empoisonné a perdu.

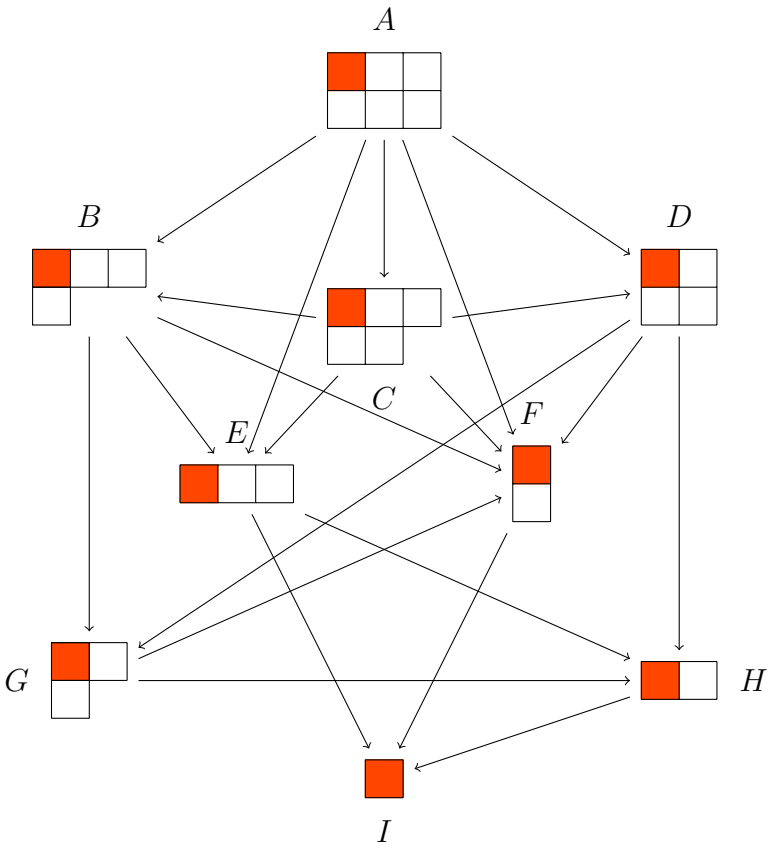
• Exemple : partie sur une tablette 3×4



Ici, c'est J_2 qui gagne.

• Modélisation : on peut représenter le jeu de Chomp par un graphe orienté dont les sommets sont les différents états possibles de la tablette et dont les arcs correspondent aux changements d'état de la tablette associés à des coups légaux. On peut visualiser une partie comme le fait, pour J_1 et J_2, de déplacer tour à tour un jeton d'état en état du graphe en suivant les arcs. Le joueur qui parvient à placer le jeton sur l'état où il ne reste plus qu'un carré a gagné.

• Exemple : graphe du jeu de Chomp sur une tablette 2×3

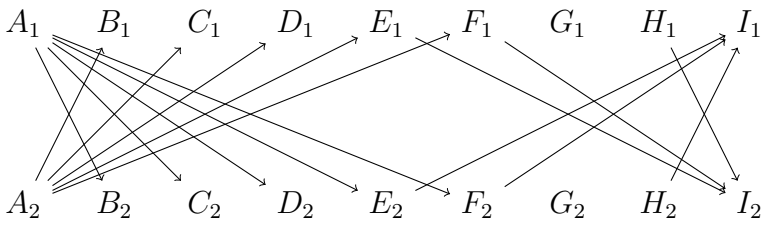


• Remarques :

– Une partie est représentée par un chemin dans le graphe de l'état initial vers l'unique état terminal. Étant donné un tel chemin, le gagnant est déterminé par le joueur initial et la parité de la longueur du chemin.

– On préfère souvent distinguer les états dans lesquels c'est au tour de J_1 de jouer de ceux dans lesquels c'est au tour de J_2. On duplique donc les sommets et on crée un graphe biparti.

Sur le graphe précédant, voici un extrait du graphe obtenu :



Si on suppose que c'est toujours J_1 qui commence, alors on peut simplifier ce graphe.

Par exemple : A_2 est inutile donc C_1 aussi car il devient de degré entrant nul.

4.1.3 Vocabulaire

• Un jeu à deux joueurs est donné par un graphe orienté biparti $G = (S, A)$, aussi appelé *arène*. En notant S_1, S_2 une partition convenable de S, on dit que les éléments

de S_1 (resp. S_2) sont les états *contrôlés* par J_1 (resp. J_2).

Les arcs de G sont appelés des *coups* : un arc (s,t) décrit un coup possible depuis s pour le joueur qui contrôle cet état.

• Une partie étant constituée d'une suite de coups alternant les deux joueurs, on définit une partie depuis l'état s comme un chemin de sommet initial s, maximal. Un tel chemin est alors nécessairement soit infini, soit fini et tel que son dernier sommet est de degré sortant nul. Les sommets de degré sortant nul sont appelés *états terminaux*.

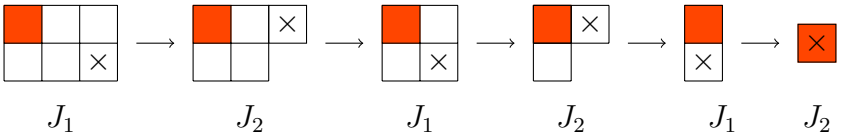
On suppose dans la suite que le graphe G est acyclique, donc qu'il n'existe pas de partie infinie.

• Parmi les états terminaux, on distingue les états gagnants pour J_1, les états gagnants pour J_2, et les états nuls.

Attention, un état gagnant pour J_1 n'est pas forcément contrôlé par J_1 (*cf* Chomp). Cela dépend du jeu.

On appelle *partie gagnante* pour J_1 (resp. J_2) toute partie se terminant dans un état gagnant pour J_1 (resp. J_2) et *partie nulle* toute partie se terminant dans un état nul.

• Exemple : dans le jeu de Chomp, la partie suivante :



est décrite par le chemin $A_1, C_2, D_1, G_2, F_1, I_2$ dans le graphe de 4.1.2 (page 22). C'est une partie gagnante pour J_1. Les états terminaux du graphe sont I_1 et I_2 respectivement gagnants pour J_2 et J_1. Il n'y a donc pas d'état nul ni de partie nulle.

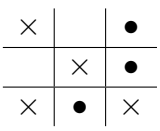
• Exemple : le *tic-tac-toe*

J_1 et J_2 doivent à tour de rôle placer leur symbole personnel (traditionnellement un rond ou une croix) dans une grille 3×3, de sorte à aligner sur une ligne, une colonne, ou une diagonale trois occurrences de leur symbole.

Les états du jeux sont les différents états de la grille, ce qui donne un graphe ayant plus de 5000 sommets, et plus de 16000 arcs.

Dans les états terminaux, la grille n'est pas nécessairement remplie.

Par exemple,



est un état terminal, gagnant pour le joueur dont le symbole est $\times$.

On peut d'ailleurs affirmer que dans toute partie d'état initial la grille vide, et qui mène à cet état, c'est ce joueur qui a commencé.

D'autres états terminaux peuvent correspondre à une grille remplie, comme

•	×	×
×	×	•
•	•	•

pour lequel on peut affirmer que pour toute partie menant à cet état, le joueur dont le symbole est • a commencé et a joué le dernier coup nécessairement sur la dernière rangée.

Dans ce jeu, il existe des états nuls, par exemple

×	×	•
•	×	×
×	•	•

Dans le cas où l'état précédant dans une partie est le suivant :

×	×	•
•	×	×
	•	•

on pourrait déjà affirmer que la partie sera nulle car c'est le joueur dont le symbole est × qui le contrôle, alors que si cet état était contrôlé par l'autre joueur, alors ce dernier gagnerait nécessairement, d'où l'importance de travailler sur le graphe biparti, qui donne la connaissance du joueur au trait.

• Remarque : déterminer à l'avance quels sont les états qui mènent à des parties gagnantes pour l'un des joueurs ou nulles est au cœur de la prise de décision des joueurs. Il est donc nécessaire de trouver des algorithmes efficaces permettant d'évaluer la la situation de jeu afin d'établir une stratégie.

4.2 Stratégies

4.2.1 Stratégies sans mémoire

• Définition : Soit $G = (S, A)$ un jeu à deux joueurs pour lequel on note S_1 l'ensemble des états contrôlés par J_1, et S_2 celui des états contrôlés par J_2.

Pour $i \in \{1, 2\}$, une stratégie sans mémoire pour J_i est une fonction $f : S_i \longrightarrow S$ telle que $\forall e \in S_i$ non terminal, $(e, f(e)) \in A$.

Une partie est jouée selon la stratégie f si, en notant $s_0, \ldots, s_n$ cette partie, $\forall j \in [\![0\ ;\ n-1]\!] \mid s_j \in S_i$, alors $s_{j+1} = f(s_j)$.

• Remarque : la stratégie est dite *sans mémoire* car elle ne dépend que de l'état courant du jeu et pas de ce qui s'est passé avant.

Un objectif est de réussir à construire une stratégie qui garantisse si possible la victoire du joueur qui l'applique, tout en espérant qu'il existe une implémentation efficace pour une telle stratégie.

4.2.2 Stratégies et positions gagnantes

• Définition (*stratégie gagnante*) : Soit $G = (S, A)$ un jeu à deux joueurs pour lequel on note S_1 (resp. S_2) l'ensemble des états contrôlés par J_1 (resp. J_2).

Une stratégie f est dite *gagnante* depuis l'état $s_0 \in S$ pour le joueur i si et seulement si toute partie jouée selon cette stratégie à partir de l'état s_0 est gagnante pour le joueur i.

• Exemple : pour le jeu de Chomp sur une tablette 2×3 (modélisée par le graphe en 4.1.2, page 22), la stratégie suivante est gagnante pour J_1 depuis $A1$:

$$\begin{array}{rcl} f & : & A_1 \longmapsto B_1 \\ & & B_1 \longmapsto G_2 \\ & & C_1 \longmapsto B_2 \\ & & D_1 \longmapsto G_2 \\ & & E_1 \longmapsto I_2 \\ & & F_1 \longmapsto I_2 \\ & & G_1 \longmapsto H_2 \\ & & H_1 \longmapsto H_2 \\ & & I_1 \longmapsto A_1 \end{array}$$

J_2 admet une stratégie gagnante depuis B_2 :

$$\begin{array}{rcl} g & : & A_2 \longmapsto C_1 \\ & & B_2 \longmapsto G_1 \\ & & C_2 \longmapsto B_1 \\ & & D_2 \longmapsto G_1 \\ & & E_2 \longmapsto I_1 \\ & & F_2 \longmapsto I_1 \\ & & G_2 \longmapsto H_1 \\ & & H_2 \longmapsto I_1 \\ & & I_2 \longmapsto A_1 \end{array}$$

• Remarque : si J_1 et J_2 jouent tous les deux leur stratégie, c'est tout de même J_1 qui va gagner car celle de J_2 n'est gagnante que depuis des états vers lesquels J_1 ne va pas jouer.

• Définition (*position gagnante*) : Soit $G = (S, A)$ un jeu à deux joueurs, et $e \in S$ un état du jeu.

L'état e est une *position gagnante* pour le joueur i si et seulement si il existe une stratégie gagnante depuis cet état pour ce joueur.

• Exemple : dans le jeu de Chomp sur une tablette 2×3, les positions gagnantes pour le joueur i sont $A_i, B_i, D_i, E_i, F_i, H_i, C_{3-i}, G_{3-i}, I_{3-i}$.

• Proposition :

> Dans le jeu de Chomp sur une tablette $p \times q$ avec $(p,q) \neq (1,1)$, la position initiale est gagnante pour le premier joueur (on suppose que J_1 commence).

□

On montre d'abord que toute position est gagnante soit pour J_1 soit pour J_2, par récurrence forte sur le nombre de carrés de chocolat.

– S'il ne reste qu'un carré : l'état est terminal et gagnant pour le joueur qui ne le contrôle pas.

– On suppose la propriété vraie pour tous les états à moins de n carrés de chocolat.

On considère un état s à $n+1$ carrés. On note J_i le joueur qui contrôle cet état.

Par hypothèse de récurrence, pour tout coup (s,t) possible, comme t a moins de n carrées, t est une position gagnante soit pour J_1, soit pour J_2.

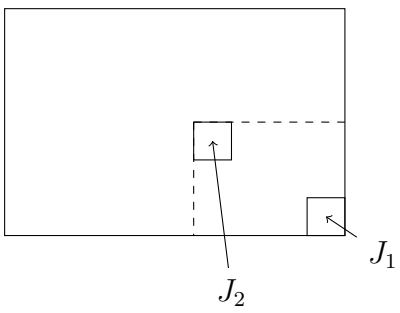
Deux options :

+ Il existe un coup (s,t) tel que t est une position gagnante pour J_i, alors s est une position gagnante pour J_i (il suffit de jouer ce coup puis d'appliquer une stratégie gagnante depuis t) ;

+ Pour tous les coups (s,t), t est une position gagnante pour J_{3-i} : alors s est une position gagnante pour J_{3-i} car quelque soit le coup joué, J_{3-i} peut appliquer une stratégie gagnante depuis l'état atteint.

Remarque : ce raisonnement s'applique à tout jeu sans cycle ni état terminal nul en appliquant une induction bien fondée selon l'ordre topologique inverse sur le graphe.

– On suppose que la position initiale est gagnante pour J_2. Il existe donc une stratégie gagnante pour J_2 depuis la position initiale. En particulier si J_1 ne mange que le carré en bas à droite, il existe un coup pour J_2 qui mène à une position contrôlée par J_1 encore gagnante pour J_2.



Or le carré en bas à droite est inclus dans les carrés situées en dessous et à droite du carré correspondant au coup de J_2.

Ainsi, si J_1 mange plutôt ce carré pour son premier coup, il atteint une position qui était gagnante pour J_2, mais en inversant le joueur qui contrôle cet état. Il peut donc à partir de cet état appliquer la stratégie gagnante de J_2 et gagner la partie : absurde

La position initiale est donc gagnante pour J_1 ■

• Remarque : la technique utilisée dans cette démonstration est appelée *vol de stratégie* :

J_1 vole la stratégie de J_2 en l'appliquant en premier. Ce n'est pas un démonstration constructive donc cela ne permet pas de déterminer une stratégie gagnante.

En revanche, on peut déduire de la première partie de la démonstration un algorithme permettant de calculer une stratégie gagnante.

4.2.3 Attracteur

• Remarque : on se place du point de vue de J_1 pour simplifier, mais tout se transpose à J_2.

• Définition (*attracteur*) : Soit $G = (S, A)$ un jeu à deux joueurs pour lequel l'ensemble des états contrôlés par J_1 (resp. J_2) est noté S_1 (resp. S_2).

On note W_1 l'ensemble est états terminaux gagnants pour J_1.

On définit

$$\begin{cases} A_0 = W_1 \\ \forall n \in \mathbb{N},\ A_{n+1} = A_n \ \cup \ \{s \in S_1 \mid \exists t \in A_n \mid (s,t) \in A\} \\ \qquad\qquad\qquad\quad \cup \ \{s \in S_2 \mid \forall t \in S,\ (s,t) \in A \Rightarrow t \in A_n\} \end{cases}$$

L'*attracteur* de W_1 est l'ensemble

$$\bigcup_{n \in \mathbb{N}} A_n$$

• Théorème :

> Avec les même notations, l'attracteur de W_1 est l'ensemble des points gagnants pour J_1.

□

$\forall s \in S$, on note

$$n_s = \begin{cases} \min\{n \in \mathbb{N} \mid s \in A_n\} & \text{si } s \in \bigcup_{n \in \mathbb{N}} A_n \\ +\infty & \text{sinon} \end{cases}$$

On montre par récurrence forte que $\forall n \in \mathbb{N},\ \forall s \in S \mid n_s = n$, s est une position gagnante pour J_1.

$-$ $n = 0$: si $s \in A_0 = W_1$, s est par définition de W_1 gagnante pour J_1.

$-$ Si la propriété est vraie jusqu'au rang n, soit $s \in S \mid n_s = n + 1$.

Par définition,

$$s \in A_{n+1} \setminus A_n \subseteq \{s \in S_1 \mid \exists t \in A_n \mid (s,t) \in A_n\} \cup \{s \in S_2 \mid \forall t \in S,\ (s,t) \in A \Rightarrow t \in A_n\}$$

$+$ Si $s \in S_1,\ \exists t \in A_n \mid (s,t) \in A$.

Comme $t \in A_n,\ n_t \leqslant n$, donc par H.R, t est une position gagnante pour J_1. Donc s est une position gagnante pour J_1 (J_1 joue (s,t) puis applique une stratégie gagnante depuis t).

+ Si $s \in S_2,\ \forall t \in S,\ (s,t) \in A \Rightarrow t \in A_n$.

De même, $\forall t \in S,\ (s,t) \in A \Rightarrow t$ est une position gagnante pour J_1.

Donc s est une position gagnante pour J_1 car quel que soit le coup joué par J_2, J_1 pourra appliquer une stratégie gagnante depuis l'état atteint.

− Réciproquement, si

$$s \notin \bigcup_{n\in\mathbb{N}} A_n$$

alors $n_s = +\infty$.

On montre que s est soit gagnante pour J_2, soit nulle.

On procède par induction bien fondée selon l'ordre topologique inverse (existe car G est acyclique).

+ Si s est terminal : comme

$$s \notin \bigcup_{n\in\mathbb{N}} A_n$$

alors $s \notin W_1 = A_0$, donc s est un élément terminal nul ou gagnant pour J_2.

+ Si s est tel que

$$\forall s' \notin \bigcup_{n\in\mathbb{N}},\ s' < s$$

selon l'ordre topologique inverse, s' est une position gagnante pour J_2 ou nulle.

∗ Si s est contrôlé par J_2 :

comme $n_s = +\infty,\ \forall t \in S \mid (s,t) \in A,\ n_t = +\infty$ (sinon $n_s \leqslant n_t + 1$).

Alors par hypothèse d'induction, $\forall t \in S \mid (s,t) \in A$ ($i.e$ $t < s$), t est gagnant pour J_2 ou nul, donc quel que soit le coup joué par J_1, il est possible pour J_2 d'obtenir au moins la nulle, donc s est soit gagnant pour J_2, soit nul.

∗ Si s est contrôlé par J_2 :

comme $n_s = +\infty,\ \exists t \in S \mid (s,t) \in A$, et $n_t = +\infty$ (sinon $\exists N \in \mathbb{N} \mid n \forall t \in S \mid (s,t) \in A, n_t \leqslant N$, d'où $n_s \leqslant N + 1$).

Comme $t < s$, par hypothèse d'induction, t est gagnant pour J_2 ou nul.

Donc J_2 obtient au moins la nulle en jouant (s,t).

Donc s est gagnant pour J_2 ou nul.

■

• Remarque : cette démonstration permet de construire une stratégie gagnante pour J_1 (depuis les états de l'attracteur W_1) :

− Si $n_s = +\infty$, le coup importe peu car si J_2 joue parfaitement, J_1 obtiendra au mieux la nulle.

− Si $n_s < +\infty$, jouer un coup $(s,t) \in A \mid n_t < +\infty$.

Il "suffit" pour cela de calculer (efficacement) l'attracteur de W_1. Cela ne fonctionne que si le graphe du jeu n'est pas trop grand.

• Calcul de l'attracteur : On procède par un parcours du graphe (transposé) du jeu en considérant les états selon l'ordre topologique inverse.

On part des états terminaux et on leur associe leur statut : gagnant pour J_1, gagnant pour J_2, ou nul.

Puis on "remonte" dans le graphe : pour chaque état dont le statut de tous les successeurs est connu (garanti par le choix de l'ordre topologique inverse), on détermine le statut de cet état en fonction du joueur qui contrôle l'état et des status des successeurs (comme dans la démonstration).

Complexité :

– Calcul du graphe transposé : $\mathcal{O}(|S| + |A|)$.

– Tri topologique du graphe transposé : $\mathcal{O}(|S| + |A|)$.

– Parcours des sommets selon ce tri, en observant leurs successeurs dans le graphe du jeu : $\mathcal{O}(|S| + |A|)$.

Donc au total, $\mathcal{O}(|S| + |A|)$.

Exo Modification de l'algorithme pour calculer une stratégie à la volée + code.

4.3 Algorithme min-max

4.3.1 Principe

Dans la perspective d'écrire un programme capable de jouer, on se place du point de vue où l'on dispose d'un état donné pour lequel on essaie de déterminer un couple. Contrairement au calcul des attracteurs dans lequel on remonte depuis les états terminaux, on essaie ici d'explorer le graphe à partir de l'état donné pour déterminer son statut, en supposant que l'adversaire joue parfaitement.

Ainsi, on construit un arbre tel que :

– la racine de l'arbre est l'état donné ;

– pour chaque nœud interne, les fils de ce nœud sont les états que l'on peut atteindre en un coup.

En particulier, les nœuds de profondeur paire sont contrôles par le joueur et ceux de profondeur impaire par l'adversaire.

L'algorithme min-max est alors l'algorithme récursif qui consiste à calculer une évaluation numérique pour chaque nœud (1 pour les positions gagnantes, -1 pour les positions perdantes, et 0 pour les positions nulles) de la manière suivante :

– Si le nœud est une feuille, *i.e* un état terminal, on lui attribue l'évaluation associée à son statut.

– Si le nœud est interne, il y a deux possibilités :

+ Si le nœud est de profondeur paire : comme le joueur contrôle cet état et cherche à maximiser ses chances de gain, l'évaluation de nœud est le maximum des évaluations de ses fils (et on peut retenir un coup associé) ;

+ Le nœud est de profondeur impaire : comme l'adversaire contrôle cet état et comme on suppose qu'il joue parfaitement, sachant qu'il cherche à minimiser les

chances de gain du joueur, l'évaluation du nœud est le minimum des évaluations des ses fils.

Remarques :

– C'est une réécriture du calcul des attracteurs :

Pour chaque état contrôlé par le joueur, son évaluation vaut 1 si et seulement si il existe un coup vers un état d'évaluation 1.

Pour chaque état contrôlé par l'adversaire, son évaluation vaut 1 si et seulement si tous les coups mènent vers des états d'évaluation 1.

Un programme qui jouerait selon cet algorithme serait donc beaucoup moins efficace qu'un programme qui précalculerait les attracteurs pour jouer ensuite selon le résultat de ce précalcul. En revanche, cet algorithme est assez simple pour que l'on puisse le modifier de sorte à pouvoir l'appliquer dans des cas où le graphe est trop grand pour réaliser un calcul des attracteurs.

– On parle d'algorithme *min-max* car l'évaluation d'un état se fait *via* un parcours de l'arbre en alternant des calculs de min et de max.

– Il y a souvent plusieurs séries de coups qui mènent d'un même état de départ à un même état d'arrivée, donc certains sous-arbres sont identiques : on peut éviter de recalculer les évaluations en effectuant une mémoïsation (ce qui revient à partager les sous-arbres). Cela consiste à effectuer l'exploration du graphe du jeu directement.

4.3.2 Heuristiques

En pratique, le graphe du jeu est souvent trop grand pour être représenté en mémoire (par exemple : pour les dames, il y a de l'ordre de 10^{32} états, pour les échecs il y a de l'ordre de 10^{52} états, pour le go, il y a de l'ordre de $> 10^{100}$ états). Même la mémoïsation ne permet pas d'appliquer cet algorithme, mais il est possible d'en conserver le principe en fixant une profondeur maximale pour l'exploration.

Le principe le l'évaluation reste le même mais il faut donner une valeur aux nœuds de la profondeur p fixée qui ne sont pas terminaux. On utilise pour cela une fonction h, appelée *heuristique*, des états vers $\mathbb{R} \cup \{\pm\infty\}$.

Sa spécification est la suivante :

– Les états terminaux gagnants prennent la valeur $+\infty$;

– Les états terminaux perdants prennent la valeur $-\infty$;

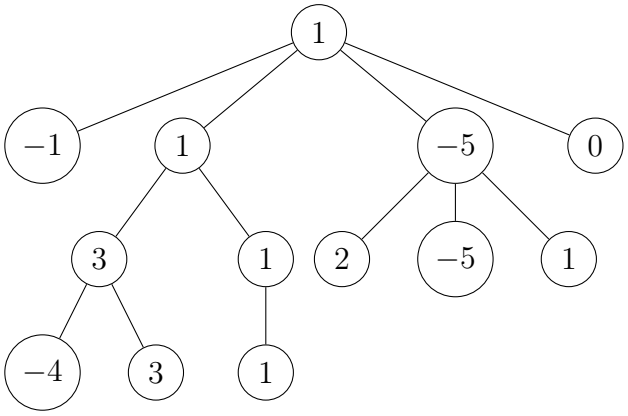
– Les autres états prennent une valeur dans $\mathbb{R}$.

Il est d'usage d'attribuer la valeur 0 aux états terminaux nuls, mais ce n'est pas une obligation. La seule contrainte est que plus la valeur attribuée à un état est grande, plus cet état est jugé favorable au joueur.

Pour jouer à un jeu, il est donc possible d'écrire un algorithme qui implémente l'algorithme min-max, limité à la profondeur p et qui évalue chaque état terminal et chaque état de profondeur p par l'heuristique h.

• Exemple :

– Exécution sur l'arbre suivant, pour lequel les évaluations de l'heuristique sont placées aux feuilles :



Profondeur	
0	max
1	min
2	max
$3 = p$	h

Coup choisi par l'algorithme : de la racine vers 1.

– Heuristique pour le jeu du puissance 4 :

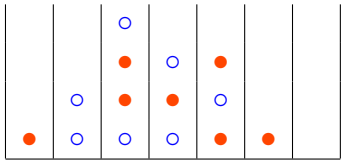
Principe : les joueurs placent des jetons colorés sur une grille 6×7 de la manière suivante : à tour de rôle les joueurs choisissent une colonne sur laquelle il reste une case vide et placent un jeton de leur couleur sur la case vide située le plus en bas sur cette colonne. Un joueur gagne dès qu'il parvient à aligner quatre jetons de sa couleur, horizontalement, verticalement, ou en diagonale.

Idée d'heuristique : on attribue à chaque case la valeur suivante : le nombre d'alignements qu'il est possible de réaliser à l'aide d'un jeton placé sur cette case, de manière indépendante du remplissage actuel de la grille :

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Pour les états non terminaux, l'heuristique attribue l'évaluation suivante : la différence entre la somme des valeurs des cases occupées par des jetons du joueur qui applique l'heuristique et la somme des valeurs des cases occupées par des jetons de l'autre joueur (*i.e* l'adversaire).

Par exemple, pour l'état suivant :



l'heuristique évalue pour le joueur bleu (∘) :

$$4+5+7+6+8+13+11-3-5-4-8-10-11-11=+2$$

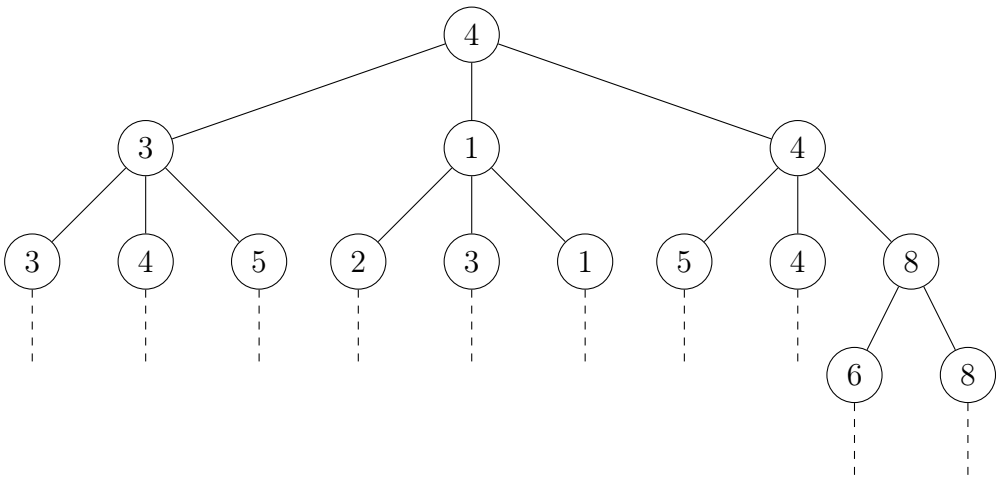
donc une position légèrement favorable pour le joueur bleu.

• Remarque : comme le branchement n'est pas constant, travailler avec la même profondeur pour toutes les branches n'est pas forcément pertinent : certaines branches vont nécessiter beaucoup plus de temps de calcul que les autres. Une possibilité pour équilibrer la charge consiste à appliquer un parcours en profondeur itéré : on augmente progressivement la profondeur en donnant la priorité aux branches qui contiennent plus de nœuds et en s'arrêtant lorsque le temps attribué au programme pour le calcul du coup est écoulé.

4.3.3 Élagage α–β

Principe : de manière similaire à un algorithme par séparation et évaluation, il est parfois possible de ne pas explorer certaines branches de l'arbre en exploitant la connaissance de bornes sur l'évaluation d'un nœud.

Par exemple :



On suppose que l'exploration se fait de gauche à droite. Lorsque le nœud 2 est évalué, comme son père est un nœud min, on sait que l'évaluation de ce dernier sera inférieure ou égale à 2.

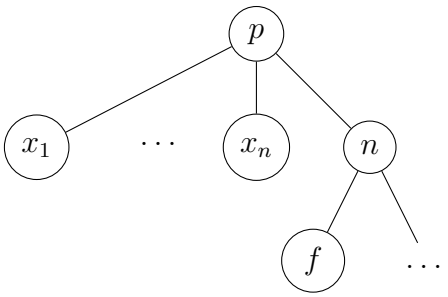
Or la racine est un nœud max dont l'un des fils a déjà une évaluation connu égale à 3. On sait donc que la branche menant au père du nœud 2 ne permettra pas de réaliser le maximum. On peut donc interrompre l'évaluation de cette branche (donc ne pas évaluer les nœuds 3 et 1).

De même, lorsque l'évaluation du nœud 6 est connue, il devient inutile d'évaluer son frère car leur père aura une évaluation supérieure ou égale à 6, donc supérieure au minimum courant nécessaire à l'évaluation du nœud 4.

• Coupure α, coupure β :

On ajoute deux paramètres à l'algorithme min–max, nommés α et β, qui représentent respectivement les bornes inférieures et supérieures de l'intervalle des évaluations "utiles" d'un nœud. Si une évaluation est assurée de sortir de cet intervalle, il est inutile de poursuivre son calcul car elle ne sera pas prise en compte pour les calculs de min et de max.

– Si le nœud courant est un nœud min, alors son père est un nœud max. Dans ce cas, si l'un des fils du nœud courant prend une évaluation inférieure à α, alors c'est aussi le cas de l'évaluation du nœud courant, et il est inutile de continuer.



On considère $\alpha = \max(\text{eval}(x_i))$.

Si $\text{eval}(f) \leqslant \alpha$, alors $\text{eval}(n) \leqslant \alpha$, donc n ne sera pas choisi pour le calcul de $\text{eval}(p)$. On coupe donc les autres branches issues de n.

On parle ici de *coupure* α.

– Si le nœud courant est un nœud max, alors son père est un nœud min. Dans ce cas, si l'un des fils du nœud courant prend une valeur supérieure à β, alors c'est aussi le cas du nœud courant, et on peut élaguer.

Avec le même dessin, on considère maintenant $\beta = \min(\text{eval}(x_i))$.

Si $\text{eval}(f) \geqslant \beta$, alors $\text{eval}(n) \geqslant \beta$, et n ne servira pas au calcul de $\text{eval}(p)$.

Les autres branches issues de n sont élaguées et on parle de *coupure* β.

• Algorithme :

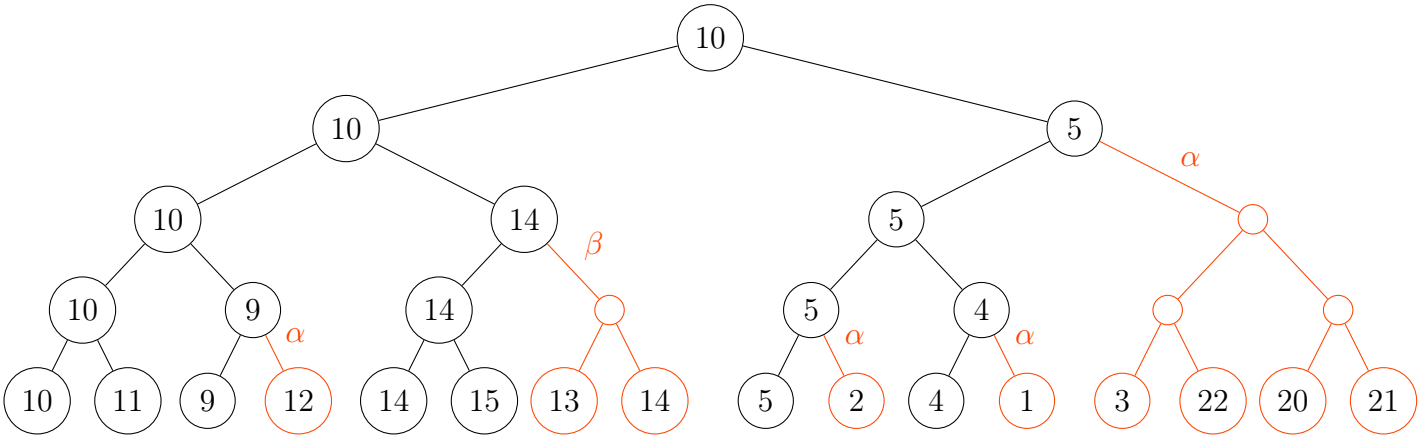
Initialement, on prend $\alpha = -\infty$, et $\beta = +\infty$ car on ne dispose d'aucune information sur les évaluations.

Algorithm 7: Min–max avec élagage α–β

Input: Nœud n (un état du jeu)
Input: Profondeur maximale p
Input: heuristique h
Input: bornes α et β

```
1  if n est terminal then
2      return h(n) ;      // Valeur exacte associée à un état terminal
3  if p = 0 then
4      return h(n) ;      // Estimation associée à un état non terminal
         via l'heuristique
5  if n est un nœud min then
       // Le test nécessite la connaissance de la profondeur
          courante par exemple.
6      M ← +∞ ;                                   // Majorant de eval(n)
7      for chaque fils f de n do
           // pour chaque coup depuis n
8          v ← min-max(f, p − 1, h, α, β);
9          if v < M then
10             M ← v;
11         if M ⩽ α then
12             return M ;                                  // coupure α
13         β ← min(β, M) ;  // permet des coupure β pour les fils non
             encore traités de n, si leur évaluation dépasse le min
             connu
14     return M;
15 else
       // n est un nœud max
16     m ← −∞ ;                                  // minorant pour eval(n)
17     for chaque fils f de n do
18         v ← min-max(f, p − 1, h, α, β);
19         if v > m then
20             ← v;
21         if m ⩾ β then
22             return m ;                                  // coupure β
23         α ← min α, m;
24     return m;
```

- Exemple :



5 Recherche informée

5.1 Contexte

5.1.1 Graphe d'états

De nombreux problèmes peuvent être modélisés à l'aide de graphes : les nœuds sont appelés *états* et correspondent aux états du système que l'on modélise, et les arcs, appelé *transitions*, représentant les actions qu'il est possible d'effectuer pour change d'état. On cherche en général à trouver une suite d'actions permettant de passer d'un état initial à un état cible, *i.e* un chemin entre ces états dans le graphe.

5.1.2 Exemple

– Pour les jeux à un joueur, par exemple le Rubik's cube, les états sont les configurations du jeux, et les arcs les coups légaux.

– Pour la recherche d'itinéraire sur un plan, on peut par exemple considérer que les états sont les intersections, et les arcs les portions de rues reliant deux intersections consécutives (ou alors se placer à une échelle plus fine).

– Pour la planification de trajectoire en robotique, les états peuvent inclure la position dans l'espace du robot, mais aussi sa vitesse, son accélération (*via* une discrétisation) et les arcs peuvent représenter les opérations de contrôle effectuées sur le robot.

– Etc.

5.1.3 Pondération

Dans les problèmes modélisés par un graphe d'états, on cherche souvent à trouver une solution optimale (par exemple : nombre minimal de coups pour résoudre le Rubik's cube, itinéraire le plus court, ...). Cela passe par une pondération des arcs du graphe d'état, qui représentent le coût associé à chaque action Le problème posé est donc celui de la recherche de plus court chemin dans un graphe pondéré.

Problème : les algorithmes de calcul de plus court chemin vus en chapitre 10 5. ne peuvent pas s'appliquer à des graphes trop grands pour être représentés en mémoire ce qui est souvent le cas dans les problèmes d'intelligence artificielle. Même si l'algorithme de FLOYD–WARSHALL ne peut être réalisé dans ce cas, il reste possible d'explorer l'algorithme de DIJKSTRA en changeant les structures de données utilisées : il peut être impossible de conserver une estimation de la distance au sommet de départ pour chacun des nœuds dans un tableau, mais l'usage d'un dictionnaire permet de conserver cette estimation pour les nœuds rencontrés en cours d'exploration. On espère alors trouver le sommet cible avant de saturer la mémoire. Cependant, l'algorithme de DIJKSTRA manque d'efficacité en pratique car on ne sait pas "dans quelle direction" chercher le sommet cible. Il est parfois possible d'exploiter une information supplémentaire pour guider la recherche et limiter le nombre de sommets parcourus. C'est ce que l'on appelle la *recherche informée*.

5.2 Algorithme A^*

5.2.1 Heuristiques

L'information dont on dispose pour guider la recherche est représentée par une fonction, appelée *heuristique*, estimant le coût nécessaire à la résolution du problème depuis chaque état.

• Définition : Soit $G = (S, A, w)$ un GO pondéré par des poids positifs, et $t \in S_n$ un sommet cible.
Une *heuristique* pour la recherche de t est une fonction

$$h : S \longrightarrow \mathbb{R}_+$$

telle que

$$h(t) = 0$$

L'heuristique h est dite *admissible* si et seulement si

$$\forall s \in S,\ h(s) \leqslant d(s,t)$$

où $d(s,t) = \min\limits_{p \text{ chemin de } s \text{ à } t}(w(p))$, *i.e* si et seulement si h ne surestime jamais le coût de la résolution.
L'heuristique h est dite *monotone* si et seulement si

$$\forall (u,v) \in A,\ h(u) \leqslant w(u,v) + h(v)$$

• Proposition :

> Soit $G = (S, A, w)$ un GOP tel que $w : A \longrightarrow \mathbb{R}_+$, $t \in S$, $h : S \longrightarrow \mathbb{R}_+$ une heuristique.
> Si h est monotone, alors h est admissible.

□

Soit $s \in S$, et $s_0 \cdots s_n$ un chemin de s à t de poids

$$\sum_{i=0}^{n-1} w(s_i, s_{i+1})$$

Par récurrence, on montre que

$$\forall i \in [\![0\ ;\ n]\!]\ ,\ h(s_i) \leqslant \sum_{j=i}^{n-1} w(s_j, s_{j+1})$$

– Initialisation :

$$h(s_n) = h(t) = 0 = \sum_{j=n}^{n-1} w(s_j, s_{j+1})$$

– Hérédité :

$$\begin{aligned} h(s_{i-1}) &\leqslant w(s_{i-1}, s_i) + h(s_i) && \text{car } h \text{ monotone} \\ &\leqslant w(s_{i-1}, s_i) + \sum_{j=i}^{n-1} w(s_j, s_{j+1}) && \text{par H.R} \\ &= \sum_{j=i-1}^{n-1} w(s_j, s_{j+1}) \end{aligned}$$

Conclusion :

$$h(s) = h(s_0) \leqslant \sum_{j=0}^{n-1} w(s_j, s_{j+1}) = d(s, t)$$

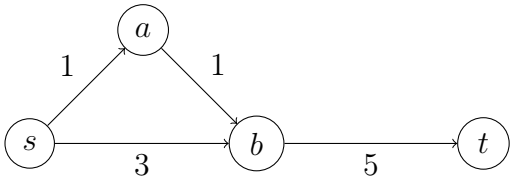
■

• Exemple : la fonction nulle est toujours monotone donc admissible.
Dans le cas de la recherche d'itinéraire, la distance à vol d'oiseau (distance euclidienne) est également monotone (d'après l'inégalité triangulaire).

5.2.2 Algorithme A^*

• Principe : l'algorithme A^* est une adaptation de l'algorithme de DIJKSTRA pour tenir compte d'une heuristique : la priorité d'un sommet rencontré dans la file de priorité n'est plus l'estimation de sa distance au sommet de départ, mais la somme de cette estimation et de la valeur de l'heuristique pour ce sommet, *i.e* une estimation du coût pour passer du sommet initial au sommet cible *via* ce sommet.
Attention : en raison de l'usage de l'heuristique, lorsqu'un sommet est extrait de la file de priorité, on ne connaît pas forcément sa distance au sommet initial, mais seulement un majorant de cette distance. Il peut donc être nécessaire de réinsérer ce sommet dans la file afin d'affiner les estimations.

• Exemple :



avec $h(a) = 6$, et $h(b) = 3$.

Depuis s, a et b sont insérés avec les priorités $w(s,a)+h(a) = 7$ et $w(s,b)+h(b) = 6$. Le sommet b est alors extrait, et on insère t dans la file avec la priorité $w(s,b) + w(b,t) + h(t) = 8$.

Même si t est rencontré, on poursuit l'algorithme car on n'a pas forcément trouvé le plus court chemin de s à t.

Le sommet a est alors extrait et on remarque que $w(s,a) + w(a,b) + h(b) = 5 < 6$, ancienne priorité de b. On réinsère donc b dans la file avec la priorité 5, correspondant au chemin $s \to a \to b$.

L'extraction de b mène à la mise à jour de la priorité de t, qui devient $w(s,a)+w(a,b)+w(b,t)+h(t) = 7$ correspondant bien au poids du plus court chemin de s à t.

• Algorithme :

Algorithm 8: A^*

Input: $G = (S, A, w)$ avec $w : A \longrightarrow \mathbb{R}_+$
Input: heuristique h
Input: source $s \in S$
Input: cible $t \in S$

```
1  d[s] ← 0;
2  p[s] ← s ;                                   // prédécesseur
3  F ← file de priorité min vide;
4  Insérer s dans F avec priorité h(s);
5  u ← s;
6  while u ≠ t et F ≠ ∅ do
7      u ← extraction du min de F;
8      for tout voisin v de u do
9          if d[v] n'est pas défini ou d[u] +w(u,v) < d[v] then
10             d[v] ← d[u] +w(u,v);
11             p[v] ← u;
12             if v ∈ F then
13                 Mettre à jour la priorité de v avec d[v] +h(v);
14             else
15                 Insérer v dans F avec la priorité d[v] +h(v)
16 if u = t then
17     return le chemin de s à t calculé grâce à p;
18 else
19     Échec;
```

Remarque : si l'heuristique est la fonction nulle, on retrouve l'algorithme de DIJKSTRA.

• Proposition :

> Si h est admissible, alors l'algorithme A^* renvoie un plus court chemin de s à t s'il existe.

□

S'il existe un chemin $s_0 \cdots s_n$ de s à t, alors l'algorithme A^* ne peut pas échouer.

Sinon, on considère

$$i = \max\{j \in [\![0\ ;\ n]\!] \mid s_j \text{ a été extrait de la file par l'algorithme}\}$$

– i existe bien car $s_0 = s$ est bien inséré, donc extrait.

– $i \neq n$ car sinon $s_n = t$ est extrait donc u prend la valeur t et la boucle est interrompue et un chemin est renvoyé.

– Lorsque s_i est extrait, comme $(s_i, s_{i+1}) \in A$, on considère s_{i+1}. Dans ce cas, soit s_{i+1} est déjà passé dans la file de priorité (et y est peut-être encore), soit on insère s_{i+1} dans la file. Dans tous les cas, s_{i+1} sera extrait car la file est vide en fin d'algorithme puisque l'algorithme échoue.

Contradiction avec la maximalité de i.

On considère maintenant le cas où l'algorithme renvoie un chemin de s à t dont on note d le poids.

Remarque : au moment de son extraction, t est de priorité `d[t]` $+h(t) =$ `d[t]` $= d$.

On suppose qu'il existe un chemin $s_0 \cdots s_n$ de s à t de poids minimal $d' < d$.

On montre par récurrence que $\forall i \in [\![0\ ;\ n]\!]$, s_i prendra la priorité

$$d(s, s_i) + h(s_i) \leqslant d'$$

avant que t ne soit extrait.

– $i = 0$: $s = s_0$ est inséré avec priorité $h(s) = d(s, s) + h(s) \leqslant d' = d(s, t)$ car h est admissible.

– Hérédité : si s_k prend la priorité $d(s, s_i) + h(s_i) \leqslant d'$ avant l'extraction de t, alors la priorité de s_i est inférieure ou égale à $d' < d$, priorité avec laquelle t est extrait, donc s_i est extrait avant t.

Au moment de l'extraction de s_i, comme $(s_i, s_{i+1}) \in A$, on considère la priorité de s_{i+1}.

Soit s_{i+1} a déjà la priorité

$$d(s, s_{i+1}) + h(s_{i+1}) \leqslant d(s, s_{i+1}) + d(s_{i+1}, t) = d'$$

soit cette priorité est plus grande.

Dans ce cas, `d[`s_i`]` $+w(s_i, s_{i+1}) = d(s, s_{i+1}) <$ `d[`s_{i+1}`]` donc on met à jour la priorité de s_{i+1} avec la bonne valeur.

Ainsi, $s_n = t$ prend une priorité $\leqslant d' < d$ avant son extraction : absurde.

Exo Montrer que l'algorithme termine bien s'il existe un chemin de s à t. ■

5.2.3 Complexité

L'algorithme A^* peut avoir une complexité exponentielle en le nombre d'arcs d'un plus court chemin de s à t en raison des réinsertions de sommets.

Cependant, dans le cas où l'heuristique est monotone, on peut montrer comme pour l'algorithme de DIJKSTRA que lorsqu'un sommet est extrait, on connaît sa distance au sommet source (Exo, preuve d'invariant).

Ainsi, chaque sommet peut provoquer l'insertion / la mise à jour de la priorité de tous ses voisins au moment de sa première extraction, donnant une file de taille $\mathcal{O}(|A|)$, puis chaque extraction ne donnera lieu qu'à l'examen des voisins en $\mathcal{O}(|S|)$ sans insertion supplémentaire. Comme chaque opération sur la file est de complexité $\mathcal{O}(\log|A|) = \mathcal{O}(\log|S|)$, on obtient une complexité

$$\mathcal{O}(|S|\,|A|\log|S|)$$