1 Programmation dynamique

1 Un premier exemple : la chasse au trésor

1.1 Présentation du problème

Afin d'illustrer les difficultés concrètes que les méthodes dites de « *programmation dynamique* » tentent de résoudre, intéressons-nous tout d'abord à un cas concret. Considérons un jeu à un joueur dont les règles sont les suivantes : on dispose d'un tableau de hauteur `H` et de largeur `L` contenant des entiers positifs [1], représentant des « gains ». Par exemple, le tableau représenté ci-dessous, pour lequel `H`= 4 et `L`= 5 :

22	14	42	17	9
37	54	29	19	5
2	27	4	1	12
78	8	34	7	31

On débute le jeu dans la case en haut à gauche, et on se déplace dans le tableau de case en case, mais les seuls mouvements possibles sont vers la case immédiatement à droite ou en-dessous de celle où l'on se trouve. Le jeu s'arrête lorsque l'on atteint la case dans le coin en bas à droite. On remporte alors les gains de l'ensemble des cases visitées, le but étant de maximiser ces gains. Sur notre exemple, le chemin permettant un gain maximal est celui représenté ci-dessous :

22	14	42	17	9
37	54	29	19	5
2	27	4	1	12
78	8	34	7	31

On peut essayer de se convaincre qu'il n'existe pas de solution triviale pour déterminer le chemin optimal. Comme on peut le voir sur l'exemple du dessus, une stratégie gloutonne maximisant le gain à chaque déplacement ne donne pas le meilleur résultat (on se serait déplacé vers la droite depuis la case 54 plutôt que vers le bas, et le gain total aurait été moindre), et chercher à passer par la case contenant le plus grand gain non plus (le meilleur chemin ne passe pas par la case contenant 78).

1. Par simple convention, des entiers relatifs ne changeraient rien dans la pratique.

1.2 Exploration exhaustive

Pour déterminer ce chemin optimal, on peut naturellement envisager de construire *tous* les chemins possibles, déterminer le gain pour chacun, et choisir le plus profitable. Pour ce faire, on va commencer par écrire une fonction `chemins(H, L)` permettant de construire tous les chemins possibles, sous la forme d'une liste de chemins, chaque chemin étant représenté par une liste de caractères, `'B'` correspondant à un déplacement vers le bas, `'D'` un déplacement vers la droite.

Fort heureusement, c'est un problème pour lequel la récursion peut grandement nous aider ! Si le tableau contient au moins deux lignes et deux colonnes, les seuls moyens d'accéder à la dernière case en bas à droite consistent à venir de la case immédiatement à gauche ou de la case immédiatement au-dessus.

Les chemins allant de la case en haut à gauche à la case immédiatement à gauche de la case en bas à droite sont exactement les chemins possibles dans un tableau de taille $H \times (L-1)$, que l'on peut construire avec un appel récursif. Il suffira d'ajouter un `'D'` à la fin des listes correspondant à ces chemins pour construire les chemins possibles dans le tableau de taille $H \times L$ passant par la case immédiatement à gauche de la case finale. De même pour les chemins (distincts) passant par la case immédiatement au-dessus de la case finale.

Reste à considérer les cas particuliers où H et/ou L valent 1 : sur un tableau à une seule case, il n'existe qu'un seul chemin, sans aucun déplacement, si le tableau est de largeur 1, on ne peut atteindre l'arrivée en venant de la gauche, et si le tableau est de hauteur 1, on ne peut le faire en venant du haut. Cela donne :

```python
def chemins(H, L):
    if H==1 and L==1: # Sur un tableau de taille 1x1, il existe
        return [ [] ] # un seul chemin possible, le chemin vide
    res = []
    if H>=2: # si l'on peut terminer avec un mouvement vers le bas
        for ch in chemins(H-1, L):
            ch.append('B')
            res.append(ch)
    if L>=2: # si l'on peut terminer avec un mouvement vers la droite
        for ch in chemins(H, L-1):
            ch.append('D')
            res.append(ch)
    return res
```

On peut s'arrêter un instant pour étudier la complexité de cette fonction. On remarque aisément qu'en dehors des appels récursifs, les opérations effectuées sont toutes de complexité constante ($O(1)$). Durant toute l'exécution de l'algorithme, l'opération effectuée le plus fréquemment est la fonction append. C'est donc le nombre total de ces append qui

donnera la complexité temporelle de la fonction `chemins`.

Il nous faut donc estimer le nombre d'appels à append effectués durant l'*ensemble* des appels. On peut remarquer qu'il y a exactement deux fois plus d'append en général qu'il y a d'append d'un caractère (`'B'` ou `'D'`). Or ces derniers, il est possible de les compter! Chaque `'B'` ou `'D'` apparaissant dans les listes retournées par la fonction provient en effet d'un tel append, et tous les `append('B')` et `append('D')` ont bien pour résultat un élément d'une des listes retournées.

Or on peut voir que chaque chemin a une longueur $(H-1)+(L-1) = H+L-2$ (il faut nécessairement se déplacer exactement $H-1$ fois vers le bas et $L-1$ fois vers la droite), et qu'il y a $\binom{L+H-2}{H-1}$ chemins différents, puisque tout chemin se déplaçant $H-1$ fois vers le bas et $L-1$ fois vers la droite conduit bien au but, aussi cela revient-il à se poser la question de placer $H-1$ déplacements vers le bas parmi $H+L-2$ déplacements au total.

On a donc $2 \times \binom{L+H-2}{H-1} \times (H+L-2)$ appels à append, ce qui donne une complexité considérable! Pour une grille de taille 10×10, cela représente environ 1,75 millions d'appels à append, et pour une grille de taille 100×100, on parle de 9×10^{60} appels, ce qui excède déjà très largement les capacités de la machine. La liste sera également trop vaste pour la mémoire. Laissons cependant ces problèmes de côté pour le moment.

On écrit ensuite une fonction donnant le gain d'un chemin (liste de `'B'` et `'D'`) :

```
def gain(tab, chemin):
    i, j = 0, 0
    g = tab[0][0]
    for depl in chemin:
        if depl=='B':
            i=i+1
        else:
            j=j+1
        g = g + tab[i][j]
    return g
```

Pour déterminer le plus grand gain possible, il suffit d'itérer tous les chemins obtenus avec `chemins`, et de faire appel pour chacun à `gain` pour déterminer ce qu'il permet de récolter, en mémorisant dans `gain_max` le plus grand gain observé (on pourra l'initialiser à 0 puisque tous les gains sont positifs) :

```
def solution(tab):
    H, L = len(tab), len(tab[0])
    gain_max = 0
    for chemin in chemins(H, L):
        gain_max = max(gain_max, gain(tab, chemin))
    return gain_max
```

Si l'on veut également le chemin qui a permi d'obtenir le gain optimal, la fonction peut être modifiée de la sorte :

```
def solution(tab):
    H, L = len(tab), len(tab[0])
    gain_max = 0
    meilleur_chemin = None
    for chemin in chemins(H, L):
        g = gain(tab, chemin)
        if g >= gain_max:
            gain_max = g
            meilleur_chemin = chemin
    return meilleur_chemin, gain_max
```

On obtient bien avec la fonction `solution` le résultat attendu (`T` étant le tableau défini dans l'introduction) :

```
In []: solution(T)
Out[]: (['B', 'D', 'B', 'B', 'D', 'D', 'D'], 220)
```

Cette dernière fonction `solution` mérite une remarque : il est important que la comparaison entre le gain `g` du chemin examiné et `gain_max` soit une comparaison large avec `>=`, car dans le cas, certes pathologique, où toutes les valeurs sont nulles, les gains de tous les chemins seront nuls, donc une comparaison stricte retournerait `None` comme chemin optimal, ce qui n'est probablement pas le résultat souhaité [2].

La complexité de la fonction `solution` est la même que `chemins`, le calcul des coûts des chemins étant de complexité comparable à leur construction, et le calcul du maximum négligeable devant ces coûts. Malheureusement, comme nous l'avons dit, cette approche n'est pas assez efficace pour résoudre le problème, même pour des tableaux de taille raisonnable, et il va falloir trouver d'autres approches.

1.3 Une approche récursive

On peut envisager une autre approche pour écrire une fonction déterminant le meilleur chemin et le gain associé, utilisant la récursion. Rappelons que lorsque l'on parvient à la case $(H-1, L-1)$ de la grille, on ne peut venir que de deux cases : celle immédiatement à gauche, $(H-1, L-2)$ et celle immédiatement au-dessus, $(H-2, L-1)$. Le meilleur chemin menant à $(H-1, L-1)$ sera naturellement le plus fructueux entre le chemin arrivant à $(H-1, L-2)$ et celui arrivant à $(H-2, L-1)$. Le gain, quant à lui, correspondra au gain menant à cette voisine, plus le contenu de la case $(H-1, L-1)$.

2. Alternativement, on aurait pu initialiser `gain_max` avec une valeur strictement négative.

Le raisonnement peu être appliqué à nouveau aux cases $(H-1, L-2)$ et $(H-2, L-1)$, puisqu'on ne peut, à nouveau, arriver sur ces cases qu'en provenance de deux cases (immédiatement à gauche et au-dessus) puisque l'on ne peut se déplacer que vers la droite et vers le bas.

Ainsi, le gain maximal possible, que l'on notera $\text{Gain\_max}(i, j)$, pour un chemin dans un tableau `tab` partant de la case en haut à droite et se terminant sur la case (i, j) (avec $i > 0$ et $j > 0$) s'exprime donc à partir des gains maximaux possibles pour des chemins arrivant sur la case immédiatement au-dessus et sur la case immédiatement à gauche, de la sorte :

$$\text{Gain\_max}(i, j) = \max\big(\text{Gain\_max}(i-1, j), \text{Gain\_max}(i, j-1)\big) + \texttt{tab[i][j]}$$

Lorsque $i = 0$ (resp. $j = 0$), il faut simplement prendre garde au fait qu'il n'y a pas de case au-dessus (resp. à gauche) de la case (i, j) considérée, ce qui donne les trois relations suivantes (toujours pour $i > 0$ et $j > 0$) :

$$\begin{cases} \text{Gain\_max}(i, 0) = \text{Gain\_max}(i-1, 0) + \texttt{tab[i][0]} \\ \text{Gain\_max}(0, j) = \text{Gain\_max}(0, j-1) + \texttt{tab[0][j]} \\ \text{Gain\_max}(0, 0) = \texttt{tab[0][0]} \end{cases}$$

Si seul le gain optimal nous intéresse, et non le chemin associé, on peut donc écrire une fonction `solution_partielle(tab, i, j)` prenant en argument la grille et déterminant $\text{Gain\_max}(i, j)$ de la sorte :

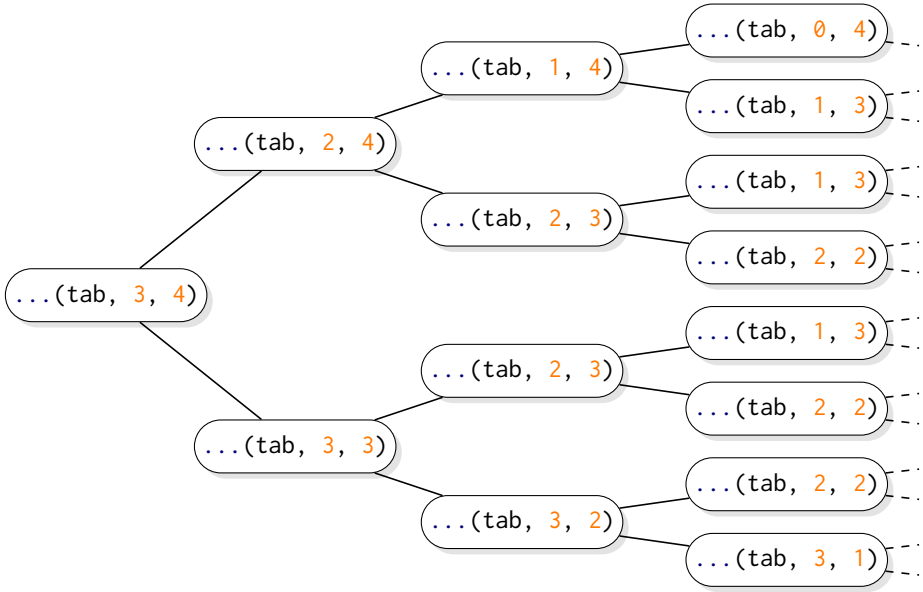
```python
def solution_partielle(tab, i, j):
    if i==0 and j==0:          # Case initiale
        return tab[0][0]
    if i==0:                   # Case de la première ligne
        return solution_partielle(tab, 0, j-1) + tab[0][j]
    if j==0:                   # Case de la première colonne
        return solution_partielle(tab, i-1, 0) + tab[i][0]
    # Autres cases
    return max(solution_partielle(tab, i-1, j),
               solution_partielle(tab, i, j-1)) + tab[i][j]
```

Il ne reste alors, pour répondre au problème, qu'à appeler cette fonction pour déterminer le gain optimal pour un chemin rejoignant la case en bas à droite :

```python
def solution(tab):
    H, L = len(tab), len(tab[0])
    return solution_partielle(tab, H-1, L-1)
```

1.4 Analyse de la complexité

Pour déterminer si la complexité de cette nouvelle approche est meilleure, on peut étudier par exemple les appels récursifs effectués sur une grille de taille 3×4 représentés ci-dessous [3] :



Dans cette séquence arborescente, chaque « branche » partant de l'appel initial, représentant une séquence d'appel récursif, correspond en fait exactement à un chemin possible de la case en haut à gauche à la case en bas à droite. Il y a donc autant de branches que de chemins possibles, autant de « traits » que de déplacements à effectuer dans tous ces chemins.

Même si le contenu de la fonction `solution_partielle`, si l'on excepte le coût des appels récursifs, est bien $O(1)$, la complexité totale de la fonction `solution` reste en $O\left(\binom{H+L-2}{H-1} \times (H+L-2)\right)$. En dehors d'une écriture potentiellent plus simple, on n'a pour l'instant rien gagné en terme d'efficacité (mais cela viendra dans un second temps).

1.5 Obtention du chemin

Si l'on souhaite obtenir le chemin correspondant au gain optimal, en plus ce celui-ci, on peut modifier la fonction `solution_partielle` pour qu'elle retourne non pas seulement le gain, mais un couple (un *tuple*) contenant à la fois le chemin et le gain.

3. On a omis le nom de la fonction, `solution_partielle`, pour ne pas surcharger le graphe, et seuls les premiers appels sont représentés, le graphe d'appel est incomplet.

On y retrouve les quatres cas de la version précédente, même si chaque cas est un peu plus long à traiter.

```python
def solution_partielle(tab, i, j):
    if i==0 and j==0:          # Case initiale
        return [], tab[0][0]
    if i==0:                   # Case de la première ligne
        chemin, gain = solution_partielle(tab, 0, j-1)
        chemin.append('D')
        return chemin, gain+tab[0][j]
    if j==0:                   # Case de la première colonne
        chemin, gain = solution_partielle(tab, i-1, 0)
        chemin.append('B')
        return chemin, gain+tab[i][0]
    # Autre cases
    chemin_b, gain_b = solution_partielle(tab, i-1, j)
    chemin_d, gain_d = solution_partielle(tab, i, j-1)
    if gain_b > gain_d:
        chemin_b.append('B')
        return chemin_b, gain_b+tab[i][j]
    else:
        chemin_d.append('D')
        return chemin_d, gain_d+tab[i][j]
```

La fonction `solution` n'a, quant à elle, pas besoin d'être modifiée.

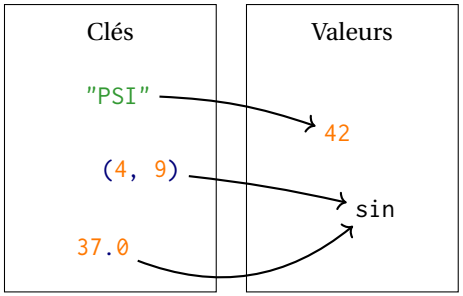
En dehors des appels récursifs, les différentes opérations dans la fonction `solution_partielle` sont toutes en temps constant ($O(1)$), et l'arbre d'appel est identique à celui de la fonction originale. La complexité de cette version modifiée est donc la même que celle de la fonction originale qui ne retournait que le gain. Obtenir le chemin n'est pas ici plus coûteux.

1.6 Mémoïsation des résultats

À y regarder de plus près, cependant, on peut remarquer que la raison du coût important de la fonction précédente est que l'on calcule de nombreuses fois la même chose. Par exemple, `solution_partielle(tab, 1, 3)` apparaît plusieurs fois dans l'arbre des appels représenté précédemment[4]. Si l'on est en mesure de *mémoriser* le résultat obtenu lors du premier appel pour ces paramètres, on pourra s'abstenir de le recalculer (avec moults appels récursifs coûteux) les fois suivantes, ce qui permettra d'économiser grandement sur les calculs à effectuer.

Cette idée de mémoriser, pour une fonction, les résultats obtenus pour un ensemble de paramètres donnés afin d'accélérer les choses la prochaîne fois que l'on fera appel à cette fonction avec les mêmes paramètres est appelé, en informatique, *mémoïsation*.

La solution la plus simple pour implémenter un tel mécanisme en Python est d'utiliser un dictionnaire. Un dictionnaire, rappelons-le, est une structure de données associant des *valeurs* à des *clés*, de la même façon qu'un dictionnaire, dans la vie courante, associe des mots à leur définition ou traduction. Par exemple, on peut vouloir associer à la chaîne de caractères « `"PSI"` » la valeur entière 42, au couple `(4, 9)` la fonction `sin`, et à la valeur 37.0 cette même fonction `sin`, de la sorte :



Comme on le voit, les clés et valeurs peuvent être des objets Python quelconques, la seule condition étant que les clés doivent être des objets immuables (valeurs numériques, booléennes, chaînes de caractères, tuples, etc., mais pas des listes par exemple). Il n'y a pas de condition sur les clés. Les clés sont naturellement toutes distinctes, mais les valeurs peuvent éventuellement ne pas l'être.

Pour définir une telle association en Python, on procède de la sorte :

```python
D = {"PSI": 42, (4, 9): sin, 37.0: sin }
```

La manipulation des dictionnaires ressemblent grandement à celle des listes. Par exemple, on peut obtenir le nombre d'associations mémorisées dans le dictionnaire D (ici trois) en écrivant simplement « `len(D)` ». Pour obtenir la valeur associée à une clé, on utilise la même notation à base de crochets que pour les listes, en indiquant la clé entre les crochets. Ainsi, « `D["PSI"]` » donnera 42, et « `D[(4, 9)]`[5] » comme « `D[37.0]` » donneront tous deux la fonction `sin`.

L'association est à sens unique : il n'est pas possible de déterminer une clé à partir d'une valeur (comme dans un vrai dictionnaire on ne peut aisément retrouver un mot à partir de sa définition !)

4. En fait, un appel `solution_partielle(tab, i, j)` apparaît autant de fois qu'il y a de chemins menant de (i,j) à l'arrivée, trois chemins dans le cas du passage de la case $(1,3)$ à la case $(3,4)$, $\binom{i'+j'-i-j}{i'-i}$ dans le cas du passage de la case (i,j) à la case $(i'j')$.

5. Rappelons que les parenthèses autour d'un tuple ne sont généralement pas requises, et on pourrait écrire également « `D[4, 9]` ».

On peut ajouter une association supplémentaire à un dictionnaire également très simplement [6]. Pour ajouter l'association liant l'entier 17 au booléen **True**, on écrira simplement :

```
D[17] = True
```

Cela permet aussi de modifier une association existante pour une clé déjà présente dans le dictionnaire. En écrivant par exemple

```
D["PSI"] = 54
```

la chaîne de caractère « "PSI" » sera désormais associée à la valeur 54 plutôt qu'à la valeur 42.

Comme on peut ajouter librement des associations à un dictionnaire, il est utile de pouvoir créer un dictionnaire vide, ce qui s'écrit simplement « {} ».

Enfin, on peut tester si une clé est présente ou non dans le dictionnaire en utilisant l'opérateur « **in** ». Ainsi, « "PSI" **in** D » sera évalué à **True** pour notre dictionnaire, tandis que « "MP" **in** D » sera évalué à **False**. Là encore, on peut le faire pour les clés, mais il n'existe pas de méthode pour tester efficacement la présence d'une valeur dans le dictionnaire.

Cette structure de donnée a ceci de remarquable que la plupart des opérations (calcul de la longueur, ajout d'une association, vérification de la présence d'une clé, obtention d'une valeur associée à une clé...) sont *toutes* effectuées en un temps que l'on peut considérer comme constant lorsque l'on souhaite effectuer des calculs de complexité, temps en particulier indépendant du nombre d'associations déjà présentes dans le dictionnaire [7]

Signalons qu'il est possible d'utiliser la construction « **for** ... **in** ... » sur un dictionnaire, et qu'elle permet d'itérer sur les *clés* d'un dictionnaire [8]. Ainsi, le programme suivant affiche toutes les associations mémorisées dans D :

```
for k in D:
    print(k, "->", D[k])
```

Revenons à notre problème de mémorisation des résultats des appels effectués. Pour ce faire, nous allons donc utiliser un dictionnaire, où les clés seront les paramètres de la fonction (ici les coordonnées (i, j)) et les valeurs associées seront les résultats correspondant de la fonction, soit le gain optimal, soit le couple composé du chemin et du gain. Ce dictionnaire doit être créé en-dehors de la fonction, car il doit être préservé d'un appel sur l'autre. Il peut par exemple être transmis à la fonction à chaque appel comme un paramètre. Ainsi, dans le cas d'une fonction mémoïsée, lorsqu'on lui passe un argument :

- si cet argument est une clé du dictionnaire, alors la valeur qui lui est associée est le résultat, on peut retourner directement cette valeur
- si cet argument n'est pas une clé du dictionnaire, alors on n'a jamais été amené à calculer la valeur à retourner dans ce cas ; on effectue donc le calcul, et avant de renvoyer le résultat, on le mémorise, dans le dictionnaire, en l'associant à l'argument utilisé comme clé.

On peut utiliser deux approches pour la mémoïser une fonction `foo` prenant un (ou plusieurs) arguments `args` à l'aide d'un dictionnaire `memo` passé en argument :

```
def foo(args, memo):
    if args in memo:      # déjà calculé ?
        return memo[args] # on retourne le résultat mémorisé
    res = ...             # calcul du résultat (pour la première fois)
    memo[args] = res      # mémorisation du résultat pour la prochaine
    return res                                                  # fois
```

Ou bien, de façon totalement équivalente :

```
def foo(args, memo):
    if args not in memo: # pas encore calculé ?
        res = ...        # calcul du résultat (pour la première fois)
        memo[args] = res # mémorisation du résultat
    return memo[args]    # on retourne le résultat mémorisé
```

Dans le cas de notre jeu à un joueur, si seul le gain nous intéresse, la fonction `solution_partielle` devient donc :

```
def solution_partielle(tab, i, j, memo):
    if (i, j) not in memo:
        if i==0 and j==0:
            memo[0, 0] = tab[0][0]
        elif i==0:
            memo[0, j] = solution_partielle(tab, 0, j-1) + tab[0][j]
        elif j==0:
            memo[i, 0] = solution_partielle(tab, i-1, 0) + tab[i][0]
        else:
            memo[i, j] = max(solution_partielle(tab, i-1, j),
                             solution_partielle(tab, i, j-1)) + tab[i][j]
    return memo[i,j]
```

Petite subtilité ici : tab, qui est toujours le même, n'est pas utilisé comme clé (de toute

6. De même, on peut supprimer l'association liée à la clé 17 en écrivant « **del** D[17] », mais c'est une opération que l'on fera bien plus rarement.

7. En réalité, il peut arriver qu'une opération prenne occasionnellement plus de temps, c'est un temps *en moyenne*, comme c'est aussi le cas de l'opération append sur une liste.

8. Dans les versions récentes de Python, l'itération se fait dans l'ordre chronologique des ajouts des clés, mais cela n'a pas toujours été le cas. Aussi est-il recommandé de considérer que les clés peuvent être énumérées dans un ordre quelconque.

façon, seul les objets immuables peuvent figurer comme clé de dictionnaire), seul le couple (i, j) est utile. Par contre, il faudra avoir un dictionnaire différent pour *chaque* tableau !

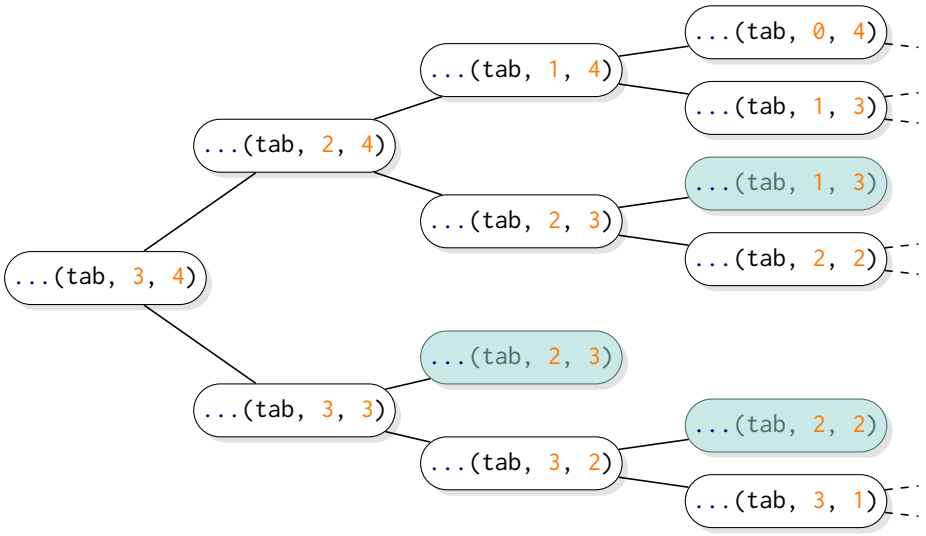
Pour répondre au problème du gain maximal sur l'ensemble de la grille, pour une grille donnée, il ne reste donc qu'à créer un dictionnaire vide pour contenir les valeurs calculées, et faire appel à la fonction précédente :

```python
def solution(tab):
    H, L = len(tab), len(tab[0])
    memo = {}
    return solution_partielle(tab, H-1, L-1, memo)
```

1.7 Gain en complexité

La complexité de cette fonction mémoïsée n'est pas évidente à obtenir. On peut toutefois voir qu'en dehors des coûts liés aux appels récursifs, les opérations ont un coût unitaire ($O(1)$). La complexité de la fonction sera donc directement liée au nombre d'appels à la fonction.

Or, ce nombre a été réduit du fait de la mémoïsation : pour des paramètres (i, j) donnés, on n'effectue des appels récursifs que la première fois que l'on rencontre ces paramètres (lors des appels ultérieurs, on récupère directement et simplement le résultat mémorisé dans le dictionnaire). L'arbre des appels a ainsi été considérablement élagué. Les appels, pour des paramètres (i, j) pour lesquels la réponse est directement tirée du dictionnaire (et donc n'entraînent plus d'appels récursifs), ont été mis en évidence ci-dessous :



Il est ainsi possible de montrer que la fonction n'est plus appelée, dans le pire des cas, que deux fois [9] pour chaque coordonnée (i, j). La complexité temporelle de la fonction sera réduite en $O(\mathrm{H} \times \mathrm{L})$, ce qui est considérablement mieux que la complexité précédente !

La mémoïsation est une stratégie faisant partie des outils de ce que l'on qualifie de « *programmation dynamique* ». Le nom n'est guère parlant [10] mais la programmation dynamique regroupe essentiellement les sratégies de programmation permettant de mémoriser, directement ou indirectement, les résultats intermédiaires des calculs afin de réduire la complexité des algorithmes.

Si l'on souhaite obtenir à la fois le gain et le chemin correspondant, il faut que les valeurs associées aux clés (i, j) dans le dictionnairent contiennent ces deux mêmes données. On peut modifier la fonction `solution_partielle` de la manière suivante :

```python
def solution_partielle(tab, i, j, memo):
    if (i, j) not in memo:
        if i==0 and j==0:
            memo[0, 0] = [], tab[0][0]
        elif i==0:
            chemin, gain = solution_partielle(tab, 0, j-1, memo)
            memo[0, j] = chemin+['D'], gain+tab[0][j]
        elif j==0:
            chemin, gain = solution_partielle(tab, i-1, 0, memo)
            memo[i, 0] = chemin+['B'], gain+tab[i][0]
        else:
            chemin_b, gain_b = solution_partielle(tab, i-1, j, memo)
            chemin_d, gain_d = solution_partielle(tab, i, j-1, memo)
            if gain_b > gain_d:
                memo[i, j] = chemin_b+['B'], gain_b+tab[i][j]
            else:
                memo[i, j] = chemin_d+['D'], gain_d+tab[i][j]
    return memo[i,j]
```

La fonction `solution`, cette fois encore, n'est pas à modifier.

Le lecteur attentif aura remarqué une différence mineure mais notable dans la fonction `solution_partielle` mémoïsée par rapport à la solution originale : les ajouts avec append ont laissé place à une concaténation de listes. C'est en effet indispensable ici, car les appels successifs à `solution_partielle(tab, i, j)` retourneront toujours la *même* liste représentant le chemin optimal arrivant à la case (i, j). Cette liste ne doit pas être modifiée pour obtenir le chemin arrivant par exemple en $(i+1, j)$, car cette même liste pourrait

9. Plus précisément, une seule fois pour les cases de la dernière ligne et de la dernière colonne, deux fois pour chacune des autres cases, ce qui peut se montrer par induction.

10. La légende voudrait qu'il ait été choisi pour son caractère « vendeur » auprès des autorités américaines distribuant les crédits de recherche plus que pour exprimer une démarche bien précise.

également être utilisée pour construire le chemin arrivant en $(i, j+1)$. Il est donc cette fois indispensable de créer une *nouvelle* liste pour ces cases.

Cette modification rend la version retournant le chemin optimal un peu plus coûteuse que celle ne retournant que le gain. En effet, les opérations dans la fonction `solution_partielle`, appels récursifs mis à part, ne sont plus en temps constant, mais peuvent du fait des concaténations avoir un coût de l'ordre des dimensions de la grille, puisque les listes ainsi construites peuvent avoir une longueur allant jusque $H+L-2$. La complexité devient dorénavant $O(H \times L \times (H+L))$, plus importante que précédemment [11], mais toujours considérablement plus raisonnable que dans la version non mémoïsée.

1.8 Une autre approche « dynamique »

Parfois, on sait déterminer à l'avance très exactement la liste des paramètres qu'il faudra fournir à la fonction et l'ordre dans lequel les fournir pour qu'il soit toujours possible de les calculer directement à partir des résultats déjà obtenus.

C'est le cas ici : on peut tout d'abord déterminer Gain_max(i, j) pour tous les couples de la forme $(0, j)$ pour des valeurs croissantes de j de 0 à $L-1$, puis pour tous les couples $(1, j)$, toujours pour des valeurs croissantes de j, et ainsi de suite. Pour chaque couple (i, j), on aura déjà déterminé le résultat pour les couples $(i-1)j$ et $(i, j-1)$

On peut mémoriser les Gain_max(i, j) dans un dictionnaire comme précédemment, mais on peut profiter de connaître par avance les i et j et ranger les résultats dans un tableau de taille $H \times L$. Ainsi, si l'on cherche le gain maximal, on peut écrire :

```python
def solution(tab):
    H, L = len(tab), len(tab[0])
    G = [[0 for j in range(L)] for i in range(H)]
    for i in range(H):  # On remplit le tableau G des Gain_max(i,j)
        for j in range(L):
            if i==0:
                if j==0:
                    G[i][j] = tab[i][j]
                else:
                    G[i][j] = tab[i][j]+G[i][j-1]
            else:
                if j==0:
                    G[i][j] = tab[i][j]+G[i-1][j]
                else:
                    G[i][j] = tab[i][j]+max(G[i-1][j], G[i][j-1])
    return G[H-1][L-1]  # On retourne Gain_max(H-1,L-1)
```

11. Un stockage plus astucieux du chemin, comme nous le verrons dans la section suivante, aurait permis d'éviter ce surcoût.

La fonction précédente construit donc le tableau des Gain_max(i, j), qui correspond, pour notre exemple, à

22	36	78	95	104
59	113	142	161	166
61	140	146	162	178
139	148	182	189	220

On y retrouve notamment le gain maximal recherché en bas à droite. Contrairement à ce qui se passait dans le cas de la fonction récursive mémoïsée, il est aisé de voir ici que la complexité est en $O(H+L)$. En fait, on effectue les mêmes calculs, seul l'ordre change, et il n'est pas surprenant que les complexités sont identiques.

Si l'on souhaite obtenir le chemin, on pourrait stocker également quelle case a permis d'atteindre quelle autre case. Mais on peut aussi procéder différemment, et reconstruire le chemin à rebours, de manière gloutonne, en partant de la dernière case et en remontant jusqu'à la case en haut à gauche, en se déplaçant de case en case vers la gauche ou vers le haut, toujours en direction de la plus grande valeur, comme illustré ci-dessous :

22	36	78	95	104
59	113	142	161	166
61	140	146	162	178
139	148	182	189	220

Pour obtenir ce résultat, il suffit alors de remplacer le `return G[H-1][L-1]` dans la fonction précédente par ce morceau de code :

```python
def solution(tab):
    ...
    i, j = H-1, L-1          # Départ en bas à droite
    chemin = []
    while i!=0 or j!=0:
        if i == 0 or j != 0 and G[i][j-1] > G[i-1][j]  :
            j = j-1
            chemin.append('D')
        else :               # On remonte
            i = i-1
            chemin.append('B')
    return chemin[::-1]      # On renverse le chemin
```

Le principal inconvénient de cette solution, qualifiée « *de bas en haut* », est qu'il est nécessaire pour pouvoir l'appliquer de pouvoir déterminer par avance les sous-problèmes qu'il nous faudra résoudre, et connaître l'ordre dans lequel il est possible de les résoudre. Pour certains problèmes, ce peut être difficile.

En revanche, on sait alors précisément ce qu'il faut calculer, mais également à quel moment un résultat d'un sous-problème n'est plus utile. On peut alors s'en débarasser, et économiser de la mémoire.

Par exemple, pour le problème qui nous occupe, si l'on n'a besoin que du gain maximal, il n'est pas nécessaire de conserver la ligne $i-1$ dès lors qu'on a terminé le calcul de la ligne i. On peut donc ne conserver à tout instant que deux listes [12]

```python
def solution(tab):
    H, L = len(tab), len(tab[0])
    # Construction de la première ligne
    ligne = [tab[0][0]]
    for j in range(1, L):
        ligne.append(ligne[j-1]+tab[0][j])
    # Construction des lignes suivantes
    for i in range(1, H):
        ligne_prec = ligne
        ligne = [ligne_prec[0]+tab[i][0]]
        for j in range(1, L):
            ligne.append(max(ligne[j-1], ligne_prec[j])+tab[i][j])
    return ligne[L-1]
```

2 Vente de ruban

2.1 Présentation du problème

Pour illustrer davantage la notion de programmation dynamique et les deux approches possibles, prenons un autre exemple. Un grossiste dispose d'une longueur $L \geqslant 0$ de ruban. Il propose à la vente plusieurs longueurs de rubans, au nombre de p : le produit numéro i correspond à une longueur l_i de ruban. Par exemple, il propose à la vente des morceaux de ruban de longueur $l_0 = 8$ m, $l_1 = 12$ m et $l_2 = 17$ m. Chacun de ces produits a un prix c_i, par exemple $c_0 = 7$ euros, $c_1 = 11$ euros et $c_2 = 15$ euros.

Dans un premier temps, nous allons nous demander s'il est possible de découper la longueur L de rubans en un ensemble de morceaux de longueur l_i (tous les morceaux n'ayant pas nécessairement la même longueur l_i) de façon à ce qu'il n'y ait pas de chute sans nous préoccuper du prix de vente. Par exemple, il est possible de découper un ruban de longeur L = 50 m en 17 + 17 + 8 + 8, mais il n'est pas possible de découper un ruban de longueur L = 55 m en morceaux de longueur 8 m, 12 m et 17 m sans aucune chute.

Dans un second temps, nous regarderons comment une fonction peut nous proposer une telle découpe, si elle existe. Puis nous étudierons combien de telles découpes différentes sont possibles pour une longueur L donnée. Enfin, nous regarderons comment découper la longueur L de ruban pour obtenir le prix de vente le plus élevé, quitte à avoir une « chute ».

2.2 Possibilité d'une découpe sans chute

On suppose que l'ensemble des longueurs l_i des différents produits est mémorisé dans une liste, par exemple :

```python
options = [8, 12, 17]
```

Pour savoir s'il existe une découpe possible du ruban de longueur $L \geqslant 0$ en un ou plusieurs morceaux de longueurs l_i (possiblement distinctes) sans chute, la solution la plus simple est d'utiliser une fonction récursive. En effet :

- si L = 0, c'est évidemment possible ;
- s'il existe un $i \in [\![0 \,..\, p-1]\!]$ tel que $L \geqslant l_i$ et qu'il est possible de découper $L - l_i$ sans chute, alors il est possible de découper le ruban L sans chute ;
- sinon, c'est impossible.

L'écriture de la fonction est alors simple :

```python
def possible(L, options):
    if L==0:
        return True
    for l_i in options:
        if L>=l_i and possible(L-l_i, options):
            return True
    return False
```

La fonction semble bien retourner les résultats attendus :

```
In []: possible(50, [8, 12, 17])
Out[]: True

In []: possible(55, [8, 12, 17])
Out[]: False
```

L'ennui, c'est que chaque appel à la fonction peut conduire à p appels récursifs. Cela peut conduire, pour des L grands devant les l_i, à un nombre déraisonnablement grands d'appels. Mais beaucoup de ces appels sont effectués pour de mêmes paramètres (par exemple, on risque d'appeler la fonction `possible` pour $L - l_0 - l_1$ et pour $L - l_1 - l_0$). Une

12. On peut même aller un peu plus loin et n'en conserver qu'une seule à tout instant, car le contenu d'une case devient inutile dès que l'on a déterminé le contenu de la case à sa droite et sous elle, mais la fonction est un peu plus complexe à écrire. Notons aussi que si les colonnes sont plus courtes que les lignes, il peut être intéressant, en terme de mémoire, de travailler colonne par colonne.

mémoïsation de la fonction est donc bienvenue. On modifie donc la fonction précédente de la manière suivante :

```
def possible_memo(L, options, memo):
    if L not in memo:
        if L==0:
            memo[L] = True
        else:
            memo[L] = False
            for l_i in options:
                if L>=l_i and possible_memo(L-l_i, options, memo):
                    memo[L] = True
                    break
    return memo[L]

def possible(L, options):
    return possible_memo(L, options, {})
```

2.3 Obtenir une solution de découpe

On peut à présent vouloir obtenir, lorsque la découpe est possible sans chute, un exemple de découpe qui convient. Pour ce faire, nous allons modifier la fonction pour qu'elle ne retourne plus **True** lorsque la découpe est possible, mais une liste de longueurs correspondant à une découpe possible du ruban. Cela devient par exemple :

```
def decoupe_memo(L, options, memo):
    if L not in memo:
        if L==0:
            memo[L] = []
        else:
            memo[L] = False
            for l_i in options:
                if L>=l_i:
                    res = decoupe_memo(L-l_i, options, memo)
                    if res != False:
                        memo[L] = res + [l_i]
                        break
    return memo[L]

def decoupe(L, options):
    return decoupe_memo(L, options, {})
```

On remarquera que l'on a utilisé une concaténation, plutôt qu'un append, dans la fonction précédente, pour les mêmes raisons que dans la section précédente : la liste retournée par `decoupe_memo` pour un `L` donné est toujours la même, et il convient donc de ne pas la modifier !

Grâce à la mémoïsation, la solution proposée ici est raisonnablement efficace, et fournit bien les résultats attendus :

```
In []: decoupe(50, [8, 12, 17])
Out[]: [17, 17, 8, 8]

In []: decoupe(55, [8, 12, 17])
Out[]: False
```

2.4 Nombre de découpes possibles

Cherchons à présent à déterminer combien de découpes sans chutes sont possibles. On peut imaginer, cette fois encore, une approche récursive. On pourrat imaginer le raisonnement [13] suivant :

- si $L = 0$, il existe un unique découpage possible ;
- sinon, on considère chacun des l_i, et on détermine le nombre de découpages de $LL - l_i$, puis on somme les différents résultats obtenus.

Bien évidemment, pour des raisons d'efficacité, comme précédemment, il nous faut mémoïser cette fonction récursive, faute de quoi le nombre d'appels deviendrait vite prohibitif. Cela s'écrirait par exemple de la sorte :

```
def denombre_memo(L, options, memo):   # Attention, cette solution n'est
    if L not in memo:                  # pas tout à fait correcte !
        if L==0:
            memo[L] = 1
        else:
            memo[L] = 0
            for li in options:
                if L >= l_i: # Nombre de découpes de L-l_i ?
                    nb_li = denombre_memo(L-l_i, options, memo)
                    memo[L] = memo[L] + nb_li
    return memo[L]

def denombre(L, options):
    memo = {}
    return denombre_memo(L, options, memo)
```

13. Pas tout à fait correct, nous le verrons.

La fonction semble, à première vue, donner les résultats attendus :

```
In []: denombre(16, [8, 12, 17])
Out[]: 1

In []: denombre(24, [8, 12, 17])
Out[]: 2

In []: denombre(55, [8, 12, 17])
Out[]: 0
```

En effet, pour L = 16 m, la seule découpe possible sans chute est 8 + 8, pour L = 24 m on peut imaginer 8 + 8 + 8 ou 12 + 12, et on a déjà déterminé que pour L = 55 m, il n'y avait pas de découpe sans chute possible. Malheureusement, il y a un problème :

```
In []: denombre(20, [8, 12, 17])
Out[]: 2
```

La fonction retourne 2 car elle a identifié deux solutions : 8 + 12 et 12 + 8. Or il s'agit techniquement de la même solution, avec simplement un ordre différent. De la même façon, pour L = 37 m, la fonction retourne 6 alors que la seule solution est 8 + 12 + 17, mais pour laquelle on peut envisager 6 variations possibles dans l'ordre.

Pour contourner ce problème, on peut modifier notre fonction de la façon suivante : lorsque l'on détermine le nombre de découpages de $L - l_i$, on s'interdit d'utiliser toute longueur l_j avec $j < i$. Cela permet de s'assurer que l'on ne pourra pas compter de façon distincte les solutions qui ne diffèrent que par une permutation. En pratique, on passera seulement une partie de la liste `options` dans les appels récursifs.

Par exemple, lorsque l'on essaie de déterminer le nombre de découpes possibles de 20 avec les longueurs l_i dans `[8, 12, 17]`, on va chercher :

- le nombre de découpes de $20 - 8 = 12$ avec les longueurs dans `[8, 12, 17]` (il y en a une seule, réduite à 12) ;
- le nombre de découpes de $20 - 12 = 8$ avec les longueurs dans `[12, 17]` (il n'y en a aucune, car 8 n'est plus une option envisageable) ;
- le nombre de découpes de $20 - 17 = 3$ avec les longueurs dans `[17]` (il n'y en a aucune, à nouveau).

Il n'y a donc bien qu'une seule solution ! De même, 37 tentera de décomposer $37 - 8 = 29$ avec `[8,12,17]` (une solution, 12 + 17), $37 - 12 = 25$ dans `[12,17]` (aucune solution), et $37 - 17 = 20$ dans `[17]` (aucune solution).

En Python, si l'on utilise l'élément d'index `i` de `options`, on ne conservera donc que les éléments d'index supérieurs ou égaux à `i` pour les appels récursifs (soit `options[i:]`). L'ennui, lorsque l'on implémente cette solution, c'est que `L` n'est plus seul paramètre qu'il faut prendre en compte dans le dictionnaire `memo` : les paramètres sont dorénavant `L` et `options`, puisque le second paramètre peut changer d'un appel à l'autre.

La fonction peut alors s'écrire de la sorte :

```
def denombre_memo(L, options, memo):
    key = L, tuple(options)
    if key not in memo:
        if L==0:
            memo[key] = 1
        else:
            memo[key] = 0
            for i in range(len(options)):
                l_i = options[i]
                if L >= l_i:
                    opt_restantes = options[i:]
                    nb_li = denombre_memo(L-l_i, opt_restantes, memo)
                    memo[key] = memo[key] + nb_li
    return memo[key]

def denombre(L, options):
    memo = {}
    return denombre_memo(L, options, memo)
```

On remarquera que l'on transforme la liste `options` en `tuple`, pour construire la clé du dictionnaire, car les clés d'un dictionnaire sont nécessairement des objets immutables ! Dorénavant, on obtient bien le résultat recherché :

```
In []: denombre(20, [8, 12, 17])
Out[]: 1

In []: denombre(37, [8, 12, 17])
Out[]: 1

In []: denombre(24, [8, 12, 17])
Out[]: 2
```

2.5 Optimisation du prix de vente

Dorénavant, on tolère les chutes, et on cherche à obtenir le prix de vente le plus élevé possible. En général, on souhaiterait vendre des produits maximisant c_i / l_i (le prix au mètre le plus élevé). Dans notre exemple, où les longueurs sont $l_0 = 8$ m, $l_1 = 12$ m et $l_2 = 17$ m

et les prix de vente $c_0 = 7$ euros, $c_1 = 11$ euros et $c_2 = 15$ euros, le produit le plus rentable serait celui de 12 m.

Mais pour L = 35 m, un découpage en deux morceaux de 17 m (avec une chute de 1 m invendable) rapporte 30 euros, plus que toute autre découpe possible.

Pour déterminer le découpage permettant d'obtenir le prix de vente le plus élevé, à nouveau, on envisage une approche récursive :

- on peut toujours jeter tout ce qui reste de ruban (aucun produit) pour un prix de vente total nul ;
- sinon, on considère chacun des l_i, et on détermine récursivment le prix de vente d'un morceau $L - l_i$ auquel on ajoute le prix de vente c_i du morceau de longueur l_i, et on choisit parmi les possibilités celle offrant le prix de vente le plus élevé.

Outre le prix de vente total maximal, on aimerait obtenir la découpe qui permet de l'obtenir. Pour ce faire, toujours avec la mémoïsation indispensable aux performances, la fonction s'écrira par exemple :

```python
def optimal_memo(L, options, memo):
    if L not in memo:
        prix_max = 0
        decoupe_max = []
        for l_i, c_i in options:
            if L >= l_i:
                prix, decoupe = optimal_memo(L-l_i, options, memo)
                if prix + c_i > prix_max:
                    prix_max = prix + c_i
                    decoupe_max = decoupe + [l_i]
            memo[L] = prix_max, decoupe_max
    return memo[L]

def optimal(L, options):
    memo = {}
    return optimal_memo(L, options, memo)
```

La fonction retourne ainsi le prix de vente maximal, et une liste précisant la découpe qui permet de l'obtenir :

```
In []: optimal(35, [(8, 7), (12, 11), (17, 15)])
Out[]: (30, [17, 17])

In []: optimal(40, [(8, 7), (12, 11), (17, 15)])
Out[]: (36, [12, 12, 8, 8])
```

On remarquera que dans la solution précédente, on ne se préoccupe pas d'éviter l'écueil possible d'une découpe d'un morceau de 8 m suivi d'un morceau de 12 m, d'une découpe d'un morceau de 12 m suivi d'un morceau de 8 m, comme précédemment. Ce n'est pas gênant ici, car même si l'ordre diffère, le prix que l'on obtient est le même.

Cela dit, cela peut amener à considérer davantage de découpages, donc une approche comme précédemment réduisant le nombre d'options pourrait être intéressante.

2.6 Approche « du bas vers le haut »

Si les approches récursives avec mémoïsation présentent l'avantage de ne pas être trop complexes à écrire, est-il possible de procéder différemment, et de calculer, par exemple pour ce dernier problème, les découpages optimaux pour différentes longueurs jusqu'à trouver le découpage optimal pour la longueur souhaitée ?

Même si c'est faisable, c'est fréquemment beaucoup plus difficile. Toutefois, si toutes les longueurs l_i sont entières, et que la longueur `L` l'est aussi, on peut chercher à déterminer le découpage optimal pour 1 m, 2 m, 3 m, … jusqu'à parvenir à `L`.

Cela s'écrira par exemple :

```python
def optimal(L, options):
    # L doit être entier, arr[i] contient
    # le gain maximal pour L=i et la découpe correspondante
    arr = [[0, []] for _ in range(L+1)]
    # On remplit le tableau arr
    for i in range(1, L+1):
        for l_i, c_i in options:
            if i>=l_i and arr[i-l_i][0]+c_i > arr[i][0]:
                arr[i][0] = arr[i-l_i][0]+c_i
                arr[i][1] = arr[i-l_i][1]+[l_i]
    return arr[L]
```

La fonction donne évidemment les mêmes résultats que la version récursive :

```
In []: optimal(40, [(8, 7), (12, 11), (17, 15)])
Out[]: [36, [12, 12, 8, 8]]
```

Par contre, la complexité est différente. Ici, elle peut s'écrire $O(L \times p)$ (on rappelle que `L` est ici un entier !). Elle peut parfois être beaucoup plus grande que dans l'approche récursive, car cette dernière ne tentera pas nécessairement de déterminer les découpages optimaux pour *tous* les `i` entiers entre 0 et `L`, mais seulement ceux qui lui sont utiles. Si `L` ou un des l_i n'est pas un entier, il faudra également procéder différemment.

3 Distance d'édition

3.1 Présentation du problème

Intéressons-nous à un dernier exemple : l'estimation de la « distance » entre deux chaînes de caractères. C'est un problème fréquemment utile, par exemple lorsqu'un programme de correction automatique doit remplacer un mot incorrect par un mot présent dans son dictionnaire. Il doit donc chercher, parmi tous les mots de son dictionnaire, le mot le « plus proche » du mot incorrect. Le problème est de définir la notion de « plus proche » sur les chaînes de caractères.

Une solution consiste à déterminer la *distance d'édition* (ou *distance de Levenshtein*) entre deux chaînes. Elle correspond au nombre minimal d'opérations élémentaires que l'on doit effectuer sur une chaîne pour obtenir la seconde.

Ces opérations sont généralement [14] :

- ajout d'un caractère ;
- suppression d'un caractère ;
- substitution d'un caractère à un autre.

Par exemple, pour les chaines `PYTHON` et `YOUHOU`, on peut suivre le chemin suivant :

$$\text{PYTHON} \mapsto \text{YTHON} \mapsto \text{Y}\mathbf{O}\text{THON} \mapsto \text{YO}\mathbf{U}\text{HON} \mapsto \text{YOUHO}\mathbf{U}$$

soit une suppression, une ajout, et deux substitution, et donc au total quatre opérations.

Il n'existe pas de méthode permettant d'effectuer la transformation avec moins d'opérations, aussi la distance entre les deux mots est-elle de 4.

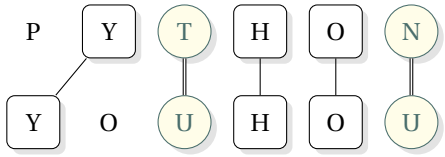
3.2 Calcul de la distance

Pour déterminer cette distance, on peut remarquer les propriétés suivantes :

- si les dernières lettres des chaînes `ch1` et `ch2` sont identiques, la distance entre `ch1` et `ch2` ne peut pas être plus grande que celle entre les chaînes privées de leur dernier caractère ;
- si les dernières lettres des chaînes `ch1` et `ch2` sont différentes, la distance entre `ch1` et `ch2` ne peut pas être plus grande que celle entre les chaînes privées de leur dernier caractère augmentée de 1 (substitution) ;
- la distance entre `ch1` et `ch2` ne peut excéder la distance entre `ch1` privée de son dernier caractère et la chaîne `ch2` augmentée de 1 (ajout) ;
- la distance entre `ch1` et `ch2` ne peut excéder la distance entre `ch1` et la chaîne `ch2` privée de son dernier caractère augmentée de 1 (suppression).

14. On y ajoute parfois l'échange de deux caractères successifs, et on considère parfois des « coûts » différents pour chacune des opérations, voire fonction des caractères impliqués.

Par ailleurs, la distance entre `ch1` et `ch2` correspond nécessairement à l'un des cas précédents. En effet, on peut remarquer que quelle que soit la transformation amenant la chaîne `ch1` vers la chaîne `ch2`, il est possible d'ordonner les opérations de la gauche vers la droite. Pour ce faire, on identifie les caractères qui se « correspondent » (ceux qui seront conservés ou substitués) comme sur l'exemple ci-dessous :



Dans cet exemple de transformation de la chaine `PYTHON` en la chaîne `YOUHOU`, la suite d'opérations peut donc être ordonnée de la sorte :

- suppression d'un `P` ;
- ajout d'un `O` ;
- substitution du `T` de `PYTHON` en `U` ;
- substitution du `N` de `PYTHON` en `U` ;

Une fois cet ordonnancement fait (qui ne change pas le nombre d'opérations dans la transformation), on peut aisément voir que la dernière opération est bien nécessairement l'une des quatre opérations décrites ci-dessus [15].

Pour déterminer la distance entre deux chaînes `ch1` et `ch2` de longueur respectives n et p, on va utiliser le principe de la programmation dynamique, en se basant sur les propriétés précédentes qui permettent de réécrire le problème du calcul de la distance entre deux chaînes à partir du résultat des calculs de distances entre des paires de chaînes plus courtes.

3.3 Implémentation

Ce problème, cette fois, se prête fort bien à une approche de bas en haut [16]. Pour mémoriser les résultats, il nous faut un tableau `tab` à deux dimensions conservant, sur la ligne i et la colonne j, la distance entre la chaîne constituée des i premiers caractères de la chaîne `ch1` et celle constituée des j premiers caractères de la chaîne `ch2` (i et j pouvant être nuls). Le tableau doit donc comporter une ligne de plus que le nombre de caractères dans `ch1`, et une colonne de plus que de caractères dans `ch2`.

La première ligne du tableau se remplit aisément : puisque la première sous-chaîne est de longueur nulle, la seule solution consiste à ajouter tous les caractères de la seconde sous-chaîne, soit sa longueur. Les cases `tab[0][j]` contiennent donc la valeur `j`. Un raisonnement similaire sur la première colonne montre que les cases `tab[i][0]` contiennent la valeur `i`.

15. Sous réserve que l'on évite les séquences d'opérations qui ne font qu'augmenter la distance, comme la suppression d'un caractère suivi d'un ajout, que l'on peut remplacer avantageusement par une substitution.

16. Une approche récursive avec mémoïsation reste toutefois parfaitement possible

Ensuite, il suffit d'écrire que :

- si le caractère au rang $i-1$ de ch1 est égal au caractère au rang $j-1$ de ch2, alors

  tab[i][j] = min(tab[i-1][j-1], tab[i-1][j-1]+1, tab[i-1][j-1]+1).
- si le caractère au rang $i-1$ de ch1 est différent du caractère au rang $j-1$ de ch2, alors

  tab[i][j] = min(tab[i-1][j-1]+1, tab[i-1][j-1]+1, tab[i-1][j-1]+1).

Pour la distance entre les chaînes PYTHON et YOUHOU, le tableau, rempli, contient les valeurs suivantes :

		Y	O	U	H	O	U
	0	1	2	3	4	5	6
P	1	1	2	3	4	5	6
Y	2	1	2	2	3	4	5
T	3	2	2	3	4	5	6
H	4	3	3	3	3	4	5
O	5	4	3	4	4	3	4
N	6	5	4	4	5	4	4

La distance d'édition entre les deux chaînes est alors simplement la valeur se situant en bas à droite dans le tableau. Le calcul de la distance entre deux chaînes peut donc s'écrire ainsi en Python :

```python
def distance(ch1, ch2):
    n, p = len(ch1), len(ch2)
    # On crée un tableau de taille (n+1)*(p+1)
    tab = [[0 for j in range(p+1)] for i in range(n+1)]
    # On remplit la première ligne et la première colonne
    for i in range(1, n+1):
        tab[i][0] = i
    for j in range(1, p+1):
        tab[0][j] = j
    # On renseigne les autres valeurs du tableau, ligne par ligne
    for i in range(1, n+1):
        for j in range(1, p+1):
            if ch1[i-1] == ch2[j-1]:
                tab[i][j] = min(tab[i-1][j-1],
                                tab[i-1][j] + 1,
                                tab[i][j-1] + 1)
            else:
                tab[i][j] = min(tab[i-1][j-1] + 1,
                                tab[i-1][j] + 1,
                                tab[i][j-1] + 1)
    return tab[n][p]
```

De façon évidente, la complexité de cette fonction est $O\big((n+1)\times(p+1)\big)$, correspondant au produit des longueurs des deux chaînes incrémentées d'une unité, puisque toutes les opérations dans la double boucle, qui représente l'essentiel du temps de calcul de la fonction, sont de coût constant ($O(1)$).

Il est par ailleurs possible de retrouver une séquence d'opérations permettant de transformer une chaîne en l'autre en partant de cette case et en remontant jusqu'en haut à gauche, en s'assurant que la série de valeurs traversées forme une suite décroissante. Le long de ce chemin, un déplacement vers la droite correspond à un ajout de caractère, vers le bas à une suppression de caractère, et en diagonale une éventuelle substitution.

Pour notre exemple, le chemin suivant, en gras dans le tableau, est une chaîne de modifications possibles (elle correspond à la suppression du premier P, à la conservation du Y, à l'ajout d'un O, à la transformation d'un T en U, à la conservation des H et O, et à la conversion d'un N en U) :

		Y	O	U	H	O	U
	0	1	2	3	4	5	6
P	**1**	1	2	3	4	5	6
Y	2	**1**	**2**	2	3	4	5
T	3	2	2	**3**	4	5	6
H	4	3	3	3	**3**	4	5
O	5	4	3	4	4	**3**	4
N	6	5	4	4	5	4	**4**

Notons pour terminer que si l'on n'a pas besoin du chemin, comme dans notre tout premier exemple de chasse au trésor, il est parfaitement possible de ne conserver, à tout instant, que deux lignes [17] du tableau lors du remplissage, plutôt que tableau dans son entièreté.

17. Voire, avec quelques efforts supplémentaire, une seule ligne.

2 Apprentissage supervisé

1 Introduction à l'apprentissage supervisé

1.1 Buts poursuivis

L'augmentation de la puissance de calcul des ordinateurs a progressivement ouvert la voie à la réalisation de tâches complexes. Parmi celles-ci, la *classification* de données, où un ordinateur est capable d'identifier des objets (nous le verrons, dans un sens très large) qu'on lui présente, a pris une grande importance dans notre vie quotidienne.

Afin d'être en mesure d'y parvenir, l'ordinateur doit *apprendre* à reconnaître lesdits objets. Parmi les techniques d'apprentissage existantes, la plus élémentaire est l'*apprentissage supervisé*. Dans cette situation, on a présenté préalablement à l'ordinateur une grande quantité d'objets dont la nature est connue (on dit qu'ils ont été préalablement *étiquetés*), afin que la machine puisse établir des règles qui lui permettront, lorsqu'on lui montre un nouvel objet inconnu, de l'identifier comme appartenant à l'une des catégories utilisées lors de la phase d'apprentissage.

Les exemples d'utilisation d'une telle technique sont très nombreuses. On y trouve par exemple (la liste n'étant pas du tout exhaustive) :

- de la classification (identification de plantes, reconnaissance d'obstacles et de panneaux pour la conduite autonome ou assistée, outils de surveillance…) ;
- de la reconnaissance de caractères ou de schémas, pour la numérisation de documents par exemple ;
- de l'identification de musique, tant pour proposer ce service à un particulier qui souhaite connaître le titre d'une chanson que pour détecter l'utilisation « abusive » d'œuvres soumises à des droits d'auteur ;
- de la reconnaissance pour des utilisations notamment liés à la sécurité (à partir de caractéristiques du visage, d'iris, d'empreintes, de voix, de forme d'oreille…) ;
- de l'assistance aux diagnostics médicaux à partir de symptômes ou de résultats d'examen ;
- d'identification de la langue d'un texte pour des outils de traduction automatique ;
- de reconnaissance de phonèmes pour permettre le sous-titrage ou la traduction simultanée, ou bien encore permettre une interface vocale à un service…

1.2 Données

Les données que l'on peut utiliser dans un tel cadre peuvent se trouver également sous bien des formes. Il peut s'agir :

- de données numériques « brutes », comme la représentation binaire d'une image (on parle de *codage rétinien*), d'une piste audio… ;
- de paramètres spécifiques mesurés ou extraits de l'objet (par exemple ses dimensions, son poids…) ;
- d'informations colorimétriques, par exemple la distribution statistique des couleurs de l'objet
- de points d'intérêt particuliers (parfois appelés *marqueurs*), et plus précisément leur nombre, leur position absolue et relative (par exemple les coins des yeux, commissures des lèvres, extrémité du nez et autre dans la reconnaissance de visage) et leur nature ;
- de paramètres descriptifs (qui peuvent être du simple texte, comme la description d'un symptôme)…

Cette fois encore, la liste est loin d'être exhaustive, et la nature des données disponibles aura naturellement un impact important sur la manière de traiter les objets.

1.3 Un exemple

Afin de tester différentes techniques d'apprentissage, il existe des bases de données qui ont été créées spécifiquement pour les mettre à l'épreuve. Elles contiennent un grand nombre d'objets étiquetés, dont une partie pourra être utilisée pour un apprentissage, et le reste, nous le verrons, servira à tester les performances.

Parmi les plus anciennes, on trouve une base de données proposant de classifier des iris. Il s'agit de plantes pour lesquelles il existe plusieurs variétés, représentées ci-dessous :

Iris setosa

Iris virginica

Iris versicolor

L'objectif est de classer une plante a priori inconnue comme membre de l'une de ces trois espèces. On n'utilise pas pour cela directement les images (ce qui est dorénavant possible, mais bien plus difficile), mais quatre données numériques extraites de chacune des plantes : la longueur et la largeur des pétales, et la longueur et la largeur des sépales de la plante. Les sépales (ou calice) correspondent à ce qui se trouve juste en-dessous des

pétales d'une fleur. Si généralement ils sont de moindre taille et usuellement verts pour la plupart des fleurs, ils ressemblent à un autre type de pétale dans le cas des iris : il s'agit, sur les photos ci-dessus, des éléments colorés les plus grands (les plus petits étant les pétales).

Il a été montré que l'on pouvait se servir de ces grandeurs pour classifier les différentes espèces d'iris, et c'est ce que nous allons nous-même tenter dans ce chapitre.

1.4 Méthodes de classification

Les méthodes à notre disposition sont nombreuses. On peut par exemple citer :

- les arbres de décisions (« *cuts* ») où l'identification est faite grâce à un logigramme où l'on est amené à répondre à une série de questions, conduisant à un résultat (les questions peuvent être définies à la main, mais peuvent également être obtenues procéduralement via des techniques d'apprentissage) ;
- les réseaux neuronaux qui, compte tenu de leur souplesse et de leurs résultats ont été très populaires ces trente dernières années, même s'ils peuvent représenter des coûts algorithmiques important et que l'on ne comprend pas forcément toujours bien pour quelle raison ils fonctionnent bien ou mal ;
- les classifieurs bayésiens naïfs ;
- les machines à vecteurs de support (SVM) ;
- des algorithmes plus élémentaires mais néanmoins potentiellement efficaces comme celui des k-plus proches voisins...

C'est cet algorithme des « k-plus proches voisins » que nous allons étudier à présent, afin d'illustrer le principe bien plus général de l'apprentissage supervisé.

1.5 Préparation des données

Lorsque l'on souhaite tester un algorithme de classification, il nous faut diviser les données étiquetées en deux groupes distincts : une partie des données servira à l'apprentissage (ce sont ces données qui permettront à l'algorithme d'en apprendre plus sur la classification), le reste des données servant à tester le bon fonctionnement de l'algorithme. Il est important de disposer de données étiquetées également pour cette seconde étape, car, en comparant les étiquettes aux prédictions de l'algorithme, nous pourrons évaluer son taux de réussite sur des données qu'il n'a pas déjà examinées lors de l'apprentissage.

Le cardinal de chacun des deux ensembles n'est pas toujours simple à choisir. Il doit y avoir suffisamment de données pour l'entraînement, afin que l'algorithme puisse se familiariser avec les variations possibles dans les données, mais il faut également suffisamment de données de tests pour que l'on puisse considérer significatives les statistiques sur les résultats. Dans le cas de notre base d'iris, on dispose de 150 données étiquetés, nous allons en utiliser 60 pour l'apprentissage et 90 pour effectuer les tests.

Cette répartition ne peut pas être faite sans aucune précaution, cependant : il est nécessaire d'avoir des représentants de *chaque* catégorie (on parle généralement de *classe* dans le domaine de l'apprentissage supervisé) en nombre suffisant dans chacun des deux groupes (apprentissage et contrôle). Il y a 50 données pour chacune des trois espèces, nous allons donc nous arranger pour qu'il y en ait 20 de chaque utilisées pour l'apprentissage, et 30 de chaque pour le contrôle.

Les données (fournies par le module Python `sklearn.datasets`) se présentent sous la forme de deux listes : une tableau `data` contenant 150 tableaux de quatre flottants, correspondant aux quatre dimensions précédemment décrites, et une tableau `target` de 150 entiers dans $[\![1\,..\,3]\!]$, chaque entier correspondant à une espèce. Ainsi, les données `data[i]` correspondent à un iris de l'espèce `target[i]`. Pour la répartition, on commence donc par créer un dictionnaire `classes` avec pour clé les identifiants des espèces et pour valeurs associées une liste de données de plantes de cette espèce :

```
classes = {}
for elem, typ in zip(data, target):
    if typ not in classes:
        classes[typ] = []
    classes[typ].append(elem)
```

Dans un second temps, on utilise la fonction `random.shuffle` pour réordonner aléatoirement le contenu de chacune des listes. En effet, on ne sait pas, *a priori*, si les données dans la base sont rangées dans un ordre particulier (par exemple les petites fleurs avant les plus grandes). Cette étape est donc nécessaire pour que les deux ensembles puisse être considérés statistiquement « équivalents ».

```
for k in classes:
    random.shuffle(classes[k])
```

Enfin, on construit quatre listes, `data_ref`, `data_tst`, `target_ref` et `target_tst`, dont le contenu est similaire à `data` et `target`, mais scindé en un groupe d'entraînement (`_ref`) avec 40% des données, et un groupe de contrôle (`_tst`) avec le reste des données :

```
Q = 0.4        # Part (dans [0,1] des données pour l'apprentissage)
data_ref, data_tst, target_ref, target_tst = [], [], [], []
for k in bins:
    nb_ref = int(Q*len(bins[k]))
    nb_tst = len(bins[k])-nb_ref
    data_ref.extend(bins[k][:nb_ref])
    target_ref.extend([k]*nb_ref)
    data_tst.extend(bins[k][nb_ref:])
    target_tst.extend([k]*nb_tst)
```

2 Méthode des k-plus proches voisins

2.1 Plus proche voisin

Dans un premier temps, nous allons mettre en œuvre une méthode de classification très simple, celle du « plus proche voisin ». Pour une donnée inconnue à classifier, nous allons raisonner par similarité, et simplement chercher parmi les données d'apprentissage (dont on connaît la classe) celle qui en est la plus proche.

Pour ce faire, il nous faut pouvoir déterminer la distance entre deux données. Le choix de la distance est un point crucial pour que l'algorithme fonctionne bien. Nous allons simplement utiliser la distance euclidienne [1]. Comme les données sont des tableaux de grandeurs numériques, il est aisé de calculer la distance entre deux données. On définit donc une fonction `dist2` prenant en argument deux données et retournant leur distance :

```
def dist2(elem1, elem2):
    res = 0
    for i in range(len(elem1)):
        res += (elem1[i]-elem2[i])**2.0
    return res
```

On remarquera que l'on ne retourne pas exactement la distance euclidienne mais le *carré* de cette distance. En effet, comme nous allons simplement comparer des distances entre elles et que la fonction racine carrée est une fonction strictement croissante, calculer la racine de chacune des distances est un calcul qui ne nous est pas indispensable.

L'algorithme déterminant la classe par la méthode du plus proche voisin n'est alors guère plus qu'un calcul de minimum. Par exemple :

```
def ppv(elem, data_ref, target_ref):
    meilleur, meilleure_dist = None, None
    for i in range(len(data_ref)):
        elem_ref = data_ref[i]
        typ_ref = target_ref[i]
        d = dist2(elem, elem_ref)
        if meilleur is None or d < meilleure_dist:
            meilleure_dist = d
            meilleur = typ_ref
    return meilleur
```

Grâce à la fonction ppv précédente, on peut déterminer la classe probable d'une donnée `elem`, connaissant les données `data_ref` et `target_ref` qui servent de référence. Par exemple, la fleur numérotée 42 est identifiée comme appartenant à la classe 1, Ce que l'on peut aisément vérifier, puisque l'on dispose d'un étiquetage pour les données servant aux tests. Dans le cas présent, c'est un succès :

```
In []: ppv(data_tst[42], data_ref, target_ref)
Out[]: 1

In []: target_tst[42]
Out[]: 1
```

On remarquera que dans le cadre de cette méthode, il n'y a pas d'étape d'apprentissage à proprement parler [2] : on utilise les données d'entraîntement telles quelles!

2.2 Préparations spécifiques

Compte tenu de l'usage présent de la distance euclidienne, il convient d'être prudent. En physique, on est sensibilisé au besoin de ne pas additionner des mètres et des secondes, ou même des mètres et des millimètres sans une conversion préalable. Or c'est ce que l'on fait éventuellement allégrement ici dans le calcul de la distance, puisque l'on additionne des carrés de paramètres qui ne sont pas forcément comparables entre eux.

On peut donc se heurter à un souci : si une grandeur affiche une plus grande variabilité que les autres, elle risque de jouer un rôle prépondérant dans le calcul de la distance, et la classification ne se fera que sur cette unique grandeur [3]. Il convient donc de s'assurer préalablement que les grandeurs sont bien comparables, par exemple en effectuant une transformation affine pourque chaque paramètre suive une distribution avec une même moyenne et un même écart-type, ce qui peut s'écrire ainsi :

```
def normalisation(data):
    for i in range(len(data[0])): # pour tous les paramètres
        s, s2 = 0.0, 0.0
        for elem in data:
            s += elem[i]
            s2 += elem[i]**2
        moy = s / len(data)                  # moyenne
        etp = (s2/len(data) - moy**2)**0.5   # écart-type
        for elem in data:
            elem[i] = (elem[i]-moy)/etp      # normalisation
```

1. Qui a des défauts en pratique, mais est une bonne base de travail, et suffira pour illustrer les principes de l'algorithme

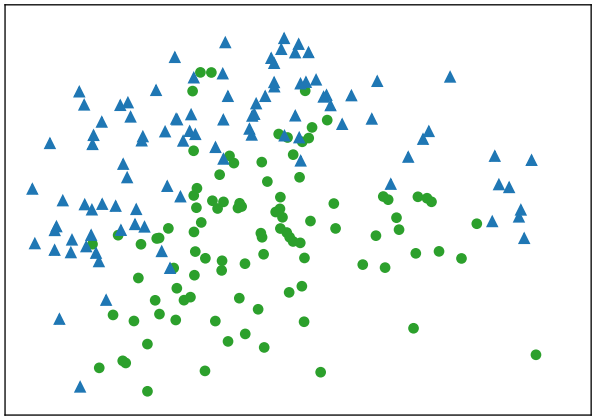
2. On pourrait organiser les données plus efficacement pour accélérer la classification, mais notre objectif étant d'illustrer le principe, les performances ne sont pas notre principal souci.

3. C'est le même problème qui peut intervenir dans le classement d'un concours si certaines disciplines ont des épreuves pour lesquelles l'étalement des notes est plus important!

La fonction `normalisation` précédente s'assure ainsi que tous les paramètres sur lesquelles on va travailler (calculés à partir des quatres dimensions dans le cas des iris) ont une moyenne nulle et un écart-type unitaire [4]. Précisons que, comme elle travaille sur le tableau `data`, elle est à appeler *avant* la répartition des données en un groupe d'apprentissage et un groupe de contrôle !

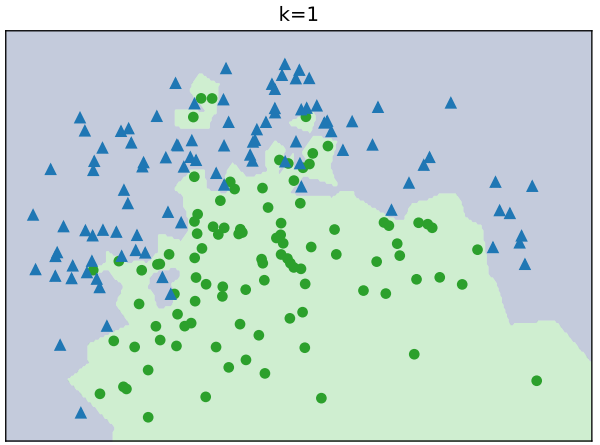
2.3 Interprétation du fonctionnement

Pour mieux appréhender comment se passe la classification, prenons un exemple plus simple. Même si la classification des iris ne fait intervenir que quatre dimensions, la représentation d'un espace à quatre dimensions sur une feuille de papier reste délicate. Considérons donc un ensemble d'objets, appartenant à deux classes distinctes, et pouvant être décrits par deux paramètres. Dans le graphe ci-dessous, les objets servant à l'apprentissage ont été placés en fonction de la valeur de leurs deux paramètres, les classes étant représentées respectivement par des triangles et des disques :



On devine, sur l'exemple précédent, que la classification doit être possible : les disques sont regroupés dans un une zone en bas à droite largement distincte des triangles qui forment un arc par-dessus. Cependant, comme les deux zones ne sont pas clairement séparées, il est raisonnable de s'attendre aussi à quelques erreurs de classification.

L'algorithme du plus proche voisin a pour effet d'attribuer une classe à n'importe quel point du plan : la classe associée à un point (x, y) donnée est celle du point, parmi les objets ayant servi à l'apprentissage, qui se trouve le plus proche des coordonnées (x, y). Si l'on colorie le morceau de plan en tout point en fonction de la classe obtenue, on obtient le résultat suivant [5] :



Le résultat est d'ores et déjà satisfaisant avec deux zones distinctes, mais on remarque que la frontière est très accidentée, et que l'on a quelques « incursions » d'une zone dans l'autre, alors que l'on pourrait espérer un frontière plus lisse entre les deux groupes. La raison est simple à comprendre : la présence d'un seul point a une influence très importante sur son voisinage immédiat.

2.4 Matrice de confusion

Avant d'essayer de résoudre la difficulté précédente, essayons d'estimer l'efficacité de ce premier classifieur sur la base des iris. Pour ce faire, il nous faut savoir combien d'iris, dans la base *de test*, ont été correctement classés (il serait idiot d'essayer de classer les iris de la base d'entraînement qui, par construction, seraient de façon évidente nécessairement tous bien classés, puisqu'il n'y a rien plus proche d'un iris que cet iris lui-même !)

En général, on détermine ce que l'on appelle la *matrice de confusion*. Pour un classifieur à n classes, il s'agit d'une matrice $C_{i,j}$ de taille $n \times n$ où le coefficient $C_{i,j}$ correspond au nombre d'objets dont la classe réelle est i qui ont été reconnus comme un objet de la classe j.

4. Les valeurs pour la moyenne et l'écart-type peuvent être choisies arbitrairement, on souhaite juste qu'elles soient similaires pour chacun des paramètres.

5. En mathématiques, ce résultat est lié à ce que l'on appelle les zones de Voronoï des données servant à l'apprentissage.

Elle est très aisée à construire en Python, il suffit de créer une « matrice » de taille $n \times n$ (une liste de listes dans le cas présent, pour éviter le recours au module `numpy`), de prendre tous les objets de la base de test, et d'incrémenter le coefficient adéquat de la matrice :

```
C = [[0 for j in range(len(classes))] for i in range(len(classes))]
for i in range(len(data_tst)):
    C[target_tst[i]][ppv(data_tst[i], data_ref, target_ref)] += 1
```

Dans le cas des iris, on obtiendra par exemple le résultat suivant [6] :

```
[[29,  1,  0],
 [ 0, 28,  2],
 [ 0,  3, 27]]
```

Chaque iris bien classé est comptabilisé sur la diagonale (un objet de classe i classé comme i). On a donc déjà un résultat remarquable [7] avec un algorithme très élémentaire : sur les 30 iris de la première espèce dans l'ensemble de contrôle, 29 ont été classés correctement. 28 et 27 l'ont été pour les 30 iris des deux autres espèces. Au total, le taux de succès dépasse les 93% !

La matrice nous donne en fait davantage de renseignements : elle a plus de difficulté à faire la différence entre les espèces 2 et 3, car on peut remarquer que 5 des 6 erreurs de classifications ont été des confusions entre ces deux espèces. L'analyse de la matrice de confusion est donc très utile pour déterminer ce qui fonctionne bien dans un classifieur et les difficultés qu'il rencontre (par exemple dans le but de trouver des paramètres supplémentaires permettant de différencier les classes souvent confondues).

2.5 k plus proches voisins

Revenons à notre problème de frontière trop accidentée. Comme nous l'avons dit, cela est du au fait que l'on ne considère que la plus proche donnée parmi les données de la base de test, ce qui donne un caractère trop local à la classification. Pour améliorer les choses, on peut considérer les $k > 1$ plus proches données. La décision se fera ensuite, par exemple, à la majorité parmi les classes de ces k données. C'est la méthode dite des « k plus proches voisins ».

Un classifieur, pour une donnée `elem`, peut fonctionner de la sorte : à tout moment, on conserve une liste nommée `meilleurs` contenant des couples `(typ, d)`, associés aux objets de la base d'entraînement les plus proches de l'élément à classer que l'on ait identifiés, `typ` représentant la classe de l'objet en question et `d` la distance à laquelle il se trouve. On gardera par exemple cette liste triée par ordre croissant de distance.

On s'assure que la longueur de la liste `meilleurs` ne dépasse jamais k. Pour chaque nouvel objet considéré dans la base d'entraînement, plusieurs cas peuvent se présenter :

- si la liste ne contient pas encore k couples, on ajoute le couple `(typ, d)` correspondant à ce nouvel objet à la fin de la liste ;
- si la liste contient déjà k couples et que le dernier d'entre eux correspond à une distance plus grande que la distance entre l'objet à classer et l'objet de la base d'entraînement considéré, on retire le dernier couple de la liste avant d'ajouter le couple `(typ, d)` correspondant au nouvel objet en fin de liste ;
- si la liste contient déjà k couples plus proches de l'élément à classer (il suffit de vérifier le dernier puisqu'ils sont classé !), alors on ignore le nouvel objet de la base d'entraînement.

Dans les deux premiers cas, l'ajout d'un couple en fin de liste a pu rompre l'ordonnancement de la liste. On rétablit l'ordre par une *insertion* (similaire à celle vue en première année dans le tri du même nom), en faisant progresser le couple en question vers la gauche par échanges jusqu'à ce que la liste soit à nouveau triée.

Une fois tous les éléments de la base d'entraînement considérés, on construit la liste des types des k éléments les plus proches identifiés. Il ne reste alors plus qu'à choisir le plus fréquent [8] (on a pris ici un raccourci en utilisant la classe`Counter` du module `collections` pour ne pas alourdir la fonction). Cela donne par exemple :

```
def kppv(elem, data_ref, target_ref, k):
    meilleurs = []
    for i in range(len(data_ref)):
        elem_ref = data_ref[i]
        typ_ref = target_ref[i]
        d = dist(elem, elem_ref)
        if len(meilleurs) < k:
            meilleurs.append((typ_ref, d))
        elif meilleurs[-1][1] > d:
            meilleurs[len(meilleurs)-1] = (typ_ref, d)
        j = len(meilleurs)-1
        while j>0 and meilleurs[j][1] < meilleurs[j-1][1]: # insertion
            meilleurs[j], meilleurs[j-1] = meilleurs[j-1], meilleurs[j]
            j = j-1
    typs = [typ for (typ, d) in meilleurs]
    return Counter(typs).most_common()[0][0]
```

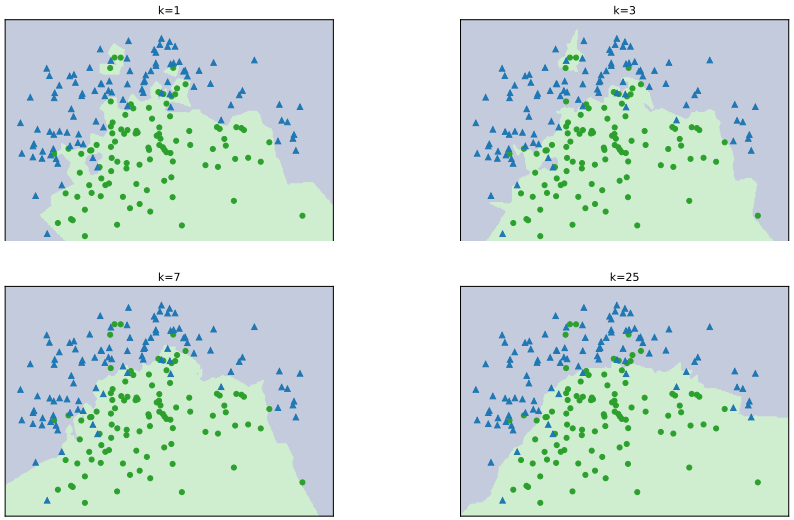
6. Compte tenu de la répartition aléatoire entre le groupe destiné à l'entraînement et celui servant au contrôle, les résultats obtenus pour la matrice de confusion peuvent être légèrement différents.

7. La base des iris est un peu ancienne, et il se trouve qu'il est bien trop simple de classifier correctement les données, et nous l'avons choisie d'abord parce qu'elle permet de se faire une image plus claire du principe de la classification, et non pour réellement tester les performances d'un algorithme.

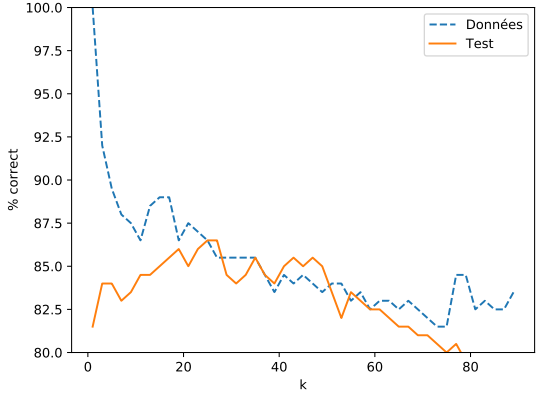
8. En cas d'égalité, il faudra trouver des règles pour faire un choix, mais dans la majorité des situations, cela n'aura guère d'importance.

2.6 Résultats et influence de k

Sur notre exemple à deux classes dans le plan, on peut remarquer que l'augmentation de k lisse bien la frontière entre les deux classes, comme on le souhaitait [9] :



Pour savoir si l'augmentation de k améliore les résultats fournis par le classifieur, on peut tracer l'évolution du pourcentage d'éléments correctement classés en fonction de k. On l'a fait à la fois pour les objets de la base d'entraînement et de la base de contrôle ci-dessous :



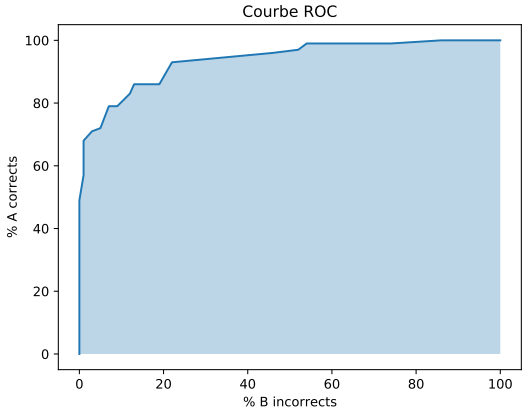
9. Il y a des variations parfois importantes sur les bords, mais comme peu d'objets devraient se trouver dans cette zone, c'est sans importance.

Bien évidemment, la courbe pour la base de contrôle débute à 100% pour $k = 1$, et décroit lorsque l'on augmente k (les quelques « anomalies » isolées et entourées d'élément d'une autre classe ne sont plus classés correctement lorsque $k > 1$). Mais c'est les résultats sur la base de test qui sont intéressant : l'augmentation de k conduit à une augmentation du pourcentage d'objets bien classés. De 81.5% environ pour $k = 1$, on monte à 86.5% pour $k = 25$. La modification à l'algorithme a donc bien l'effet souhaité.

Si l'on continue à augmenter k, les performances cependant commencent à reculer. Ce n'est guère surprenant : plus k est grand, plus les k plus proches voisins forment un ensemble étendu, et on perd progressivement le caractère « local » de la classification.

Aussi, les tests ne servent pas seulement à contrôler le bon fonctionnement de l'algorithme : ils permettent aussi de choisir convenablemnt la valeur de k qui conduira aux meilleures performances possibles pour le classifieur. Il n'existe en effet pas de valeur idéale universelle pour le choix de k, car elle dépend de beaucoup de facteurs, dont le nombre d'éléments dans la base de référence !

Lorsque l'on ne travaille qu'avec deux classes, autre manière d'envisager le choix de k est de tracer une courbe dite « courbe ROC ». On trace alors le taux d'identification correcte d'objets la classe 1 en fonction du taux d'objets de la classe 2 reconnus comme objets de la classe 1 à tort, pour toutes les valeurs possibles du paramètre k. On peut imaginer par exemple le cas d'un test médical : si la classe 1 correspond à une personne malade et la classe 2 à une personne saine, on compare la capacité du test à identifier correctement une personne malade, et à ne pas déclarer malade une personne qui ne l'est pas. On souhaite donc être le plus en haut à gauche du graphe !



Reste ensuite à choisir le k qui répondra le mieux aux besoins. Le même principe s'applique à quantité d'autres tests, car ils nécessitent généralement de comparer une quantité à un « seuil » pour conclure. Selon les situations, il peut être préférable de n'avoir que 50% de détection des personnes malades mais de garder un taux de faux positifs très faible, mais

dans d'autres situations on voudra un tests qui identifie correctement 99% des malades quitte à avoir un grand nombre de faux positifs... De la même façon, dans un système de conduite autonome, on souhaite un taux très élevé d'identification des piétons, même si cela signifie des faux positifs fréquents : mieux vaut ralentir parce qu'un objet lointain pourrait être une personne en train de traverser que de provoquer un accident faute d'un classement correct !

Si l'on applique la méthode des k plus proches voisins à nos iris, on peut voir qu'en choisissant par exemple $k = 5$, les résutats sont améliorés, avec la matrice de confusion suivante :

```
[[30,  0,  0],
 [ 0, 29,  1],
 [ 0,  3, 27]]
```

On a atteint un taux de classements corrects de 95%, avec en outre 100% de réussite sur les objets de la classe 1. Seules les classes 2 et 3 sont encore parfois confondues, mais il est difficile, voire impossible, d'obtenir une classification toujours juste de ces deux classes, même avec d'autres méthodes, tant les caractéristiques des pétales et sépales considérées pour ces deux espèces peuvent parfois être similaires.

2.7 Intérêt de la méthode

On l'a vu, en dépit de sa grande simplicité, la méthode des k plus proches voisins donne des résultats déjà très satisfaisants sur les exemples considérés, et il en sera de même sur des problèmes raisonnablement difficiles (comme la reconnaissance de chiffres manuscrits par exemple). Parmi ses principaux avantages, on peut citer :

- la simplicité de l'idée et de l'algorithme ;
- la facilité d'interprétation des résultats obtenus (ce qui n'est pas toujours le cas, par exemple avec des réseaux neuronaux) ;
- le fait de pouvoir se dispenser d'une phase d'apprentissage ;
- la possibilité d'ajouter des exemples à la base de référence au fil de l'eau ;
- et enfin le nombre très réduit de paramètres à ajuster, à savoir principalement l'entier k (outre un éventuel choix de la distance).

Cela dit, la méthode présente aussi quelques inconvénients, parmi lesquels :

- une grande sensibilité aux bruit, aux « données anormales », ainsi qu'aux données manquantes ;
- un besoin de mise à l'échelle des paramètres pour donner de bons résultats ;
- une incapacité à traiter autre chose que des données numériques ;
- un coût algorithmique élevé (en raison du calcul d'un grand nombre de distances) si les données de références sont nombreuses (même s'il est possible de choisir des structures adéquates et d'utiliser par exemple l'inégalité triangulaire pour réduire le nombre de calculs à effectuer) et si le nombre de paramètres considérés est élevé.

Lorsque l'on est confronté à un problème de classification, il est donc toujours intéressant d'envisager les très nombreuses méthodes de classifications dont on dispose (dont certaines ont été listées en début de chapitre) et de peser les avantages et inconvénients de chacune en fonction du problème auquel on est confronté.

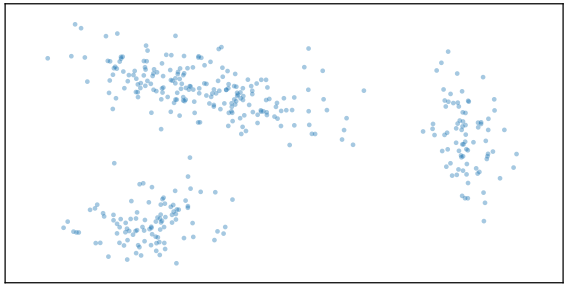
3 Apprentissage non supervisé

1 Introduction

1.1 Buts poursuivis

Supposons à présent que l'on dispose d'un volume de donnnées conséquent, mais que ces données n'aient pas fait l'objet d'un étiquetage (pour reprendre le cas des iris évoqué au chapitre précédent, nous aurions collecté les dimensions des pétales et sépales d'un grand nombre de fleurs, mais que nous n'aurions aucune information concernant l'espèce à laquelle appartiennent chacune de ces fleurs, ni même du *nombre* d'espèces différentes qui se trouvent dans le lot).

Le but de l'apprentissage non supervisé reste de partitionner l'ensemble des données en k groupes (ou classes), k étant ou non choisi à l'avance, de manière à avoir les groupes les plus « homogènes » possibles (avec une première difficulté consistant à définir concrètement en quoi deux données sont semblables ou dissemblables). Par exemple, pour les points dans le plan représentés ci-dessous, on souhaiterait disposer d'un algorithme capable de constituer 3 classes distinctes :



L'objectif est généralement d'analyser les données dont on dispose, et de déterminer si l'on peut mettre en évidence des objets de différentes nature dans les données. En d'autres termes, savoir si les données ont une structuration intéressante. Par exemple, dans le cas de plantes, pour spontanément faire émerger l'existence de plusieurs espèces différentes. Ou, si l'on analyse le trafic réseau en un nœud de communication, de déterminer si plusieurs protocoles différents sont à l'œuvre.

1.2 Similarité et données numériques

Comme dans le cas de la classification supervisée, il est fréquent que chaque donnée soit décrite par une série de p grandeurs numériques, typiquement réelles. On associe donc à chaque donnée un vecteur $x \in \mathbb{R}^p$. Dans le cas des données sur les iris, chaque fleur était associée à un vecteur de $\mathbb{R}^4$.

Dans cette situation, on peut reprendre l'idée d'utiliser une distance, par exemple la distance euclidienne, pour estimer la dissemblance entre deux objets. Et l'homogénéïté d'une partie $\mathcal{S}$ de nos données peut naturellement être lié à la *variance* $\mathbb{V}[\mathcal{S}]$ de ces données $x \in \mathcal{S}$, que l'on peut définir par

$$\mathbb{V}[\mathcal{S}] = \frac{1}{|\mathcal{S}|} \times \sum_{x \in \mathcal{S}_i} \|x - \mu\|^2$$

où μ est le barycentre des $x \in \mathcal{S}$.

Si l'on souhaite partitionner [1] les données que l'on étudie en k ensembles $\mathcal{S}_i$ de manière à ce qu'il soit les plus homogènes possibles, on veut donc minimiser les variances de chacun des ensembles. Plus précisément, afin de considérer également la taille des ensembles, nous allons chercher la partition $\mathcal{P} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}\}$ qui minimise la quantité :

$$\sum_{i=0}^{k-1} |\mathcal{S}_i| \times \mathbb{V}[\mathcal{S}_i] \;=\; \sum_{i=0}^{k-1} \sum_{x \in \mathcal{S}_i} \|x - \mu_i\|^2$$

Les termes $\sum_{x \in \mathcal{S}_i} \|x - \mu_i\|^2$ sont généralement appelées *moments d'inertie* des ensembles $\mathcal{S}_i$. D'après la définition des barycentres, il est possible de montrer que ces moments sont liées aux distances entre points à l'intérieur de ce même ensemble :

$$\sum_{x \in \mathcal{S}_i} \|x - \mu_i\|^2 \;=\; \frac{1}{|\mathcal{S}_i|} \sum_{x,y \in \mathcal{S}_i^2} \|x - y\|^2$$

On cherche donc, de la même façon, à minimiser les distances entre données de chaque ensemble, en cherchant la partition $\mathcal{P} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}\}$ qui minimise la quantité :

$$\sum_{i=0}^{k-1} \frac{1}{|\mathcal{S}_i|} \times \sum_{x,y \in \mathcal{S}_i^2} \|x - y\|^2$$

L'ennui, c'est que la résolution exacte de ce problème est généralement hors de portée de moyens numériques, car le nombre de partitions possibles en k classes est exponentiel en le nombre de données, malgré quelques avancées récentes qui ont apporté des pistes intéressantes. En revanche, il existe de très nombreux algorithmes qui fournissent des solutions *approchées* qui peuvent se révéler bien suffisantes.

1. Par partitionner, on entend que chaque donnée va être associée à un et un seul des ensembles $\mathcal{S}_i$.

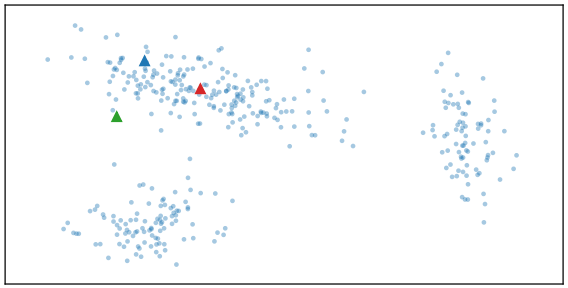
2 Méthode des k-moyennes

2.1 Principe

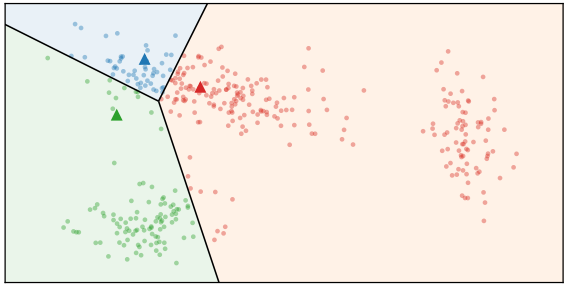
La méthode des k-moyennes est un de ces algorithmes visant à trouver une solution approchée au problème du partitonnement décrit précédemment.

Elle utilise une approche itérative simple, construisant des partitions successives qui diminuent progressivement la quantité que l'on souhaite minimiser. On peut résumer les différentes étapes de l'algorithme, pour une partition en k classes, de la façon suivante, en illustrant chacune de ces étapes sur le nuage de points (dans $\mathbb{R}^2$) présenté en introduction de ce chapitre :

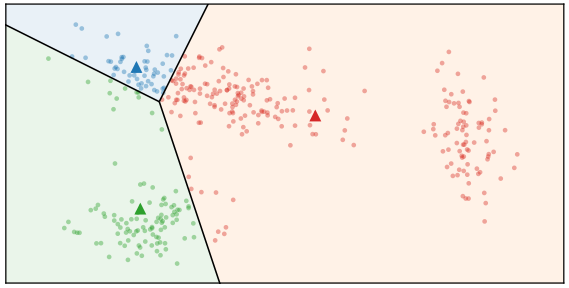
① On choisit k « représentants » $r_i \in \mathbb{R}^p$ (le choix des r_i proprement dit peut être effectué de différentes façons ; il n'est pas nécessaire, pour la suite, que les r_i correspondent à l'une des données, mais on préférera en revanche qu'ils soient tous distincts deux à deux), représentés par les trois triangles ci-dessous :



② On partitionne l'ensemble des données en k sous-ensembles $\mathcal{S}_i$ grâce à l'algorithme du plus proche voisin étudié dans le chapitre précédent, en se servant des r_i comme seul et unique représentant pour chacune des k classes. Cela revient à partitionner l'espace $\mathbb{R}^p$ en k « zones de Voronoï » (dont les frontières sont les hyperplans médians de chaque paire de r_i, les données dans chacune de ces zones constituant un ensemble $\mathcal{S}_i$) :

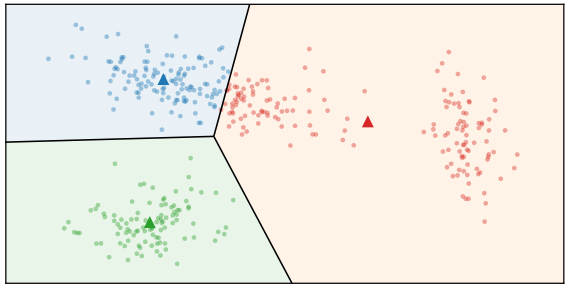


③ On remplace remplace chacun des représentants r_i par le barycentre de l'ensemble des données dans $\mathcal{S}_i$. Cette mise à jour n'est bien évidemment possible que si $\mathcal{S}_i$ n'est pas vide (dans le cas contraire, on peut simplement conserver le r_i existant, ou utiliser d'autres solutions, nous y reviendrons).

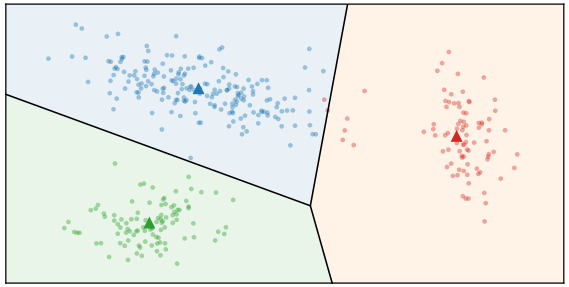


④ Une fois les représentants mis à jour, on reprend à l'étape ② afin de partitionner à nouveau l'ensemble des données, et ainsi de suite.

Au bout d'un certain temps, les classes se stabilisent, et les r_i n'évoluent plus. On arrête alors l'algorithme. Par exemple, pour notre nuage de points, après 4 itérations, on obtient la répartition suivante :



Et après dix itérations, le partitionnement n'évolue pratiquement plus, et se stabilise dans cette situation, qui fait clairement apparaître les trois groupes de points, même si quelques points en bordure droite du groupe « supérieur » ne sont probablement pas classés comme on l'aurait souhaité :



2.2 Convergence de l'algorithme

Il est possible de montrer que cet algorithme converge en un temps fini. Tout d'abord, on remarquera qu'il y a un nombre fini de partitions $\mathcal{P} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}\}$ possibles (certes gigantesque, k^n pour une partition de n données en k classes, ce qui nous a empêché d'espérer pouvoir trouver une solution exacte en les considérant toutes).

Ensuite, on peut remarquer les étapes ② et ③ ont toutes deux pour effet de faire diminuer (au sens large) la quantité $Q(\mathcal{P}, r_i) = \sum_{i=0}^{p-1} \sum_{x \in \mathcal{S}_i} \|x - r_i\|^2$. Dans le cas de l'étape ② parce que l'on construit les $\mathcal{S}_i$ en associant chaque donnée x au r_i dont il est le plus proche, dans le cas de l'étape ③ parce que les r_i nouvellement choisis sont les barycentres des données de $\mathcal{S}_i$.

En outre, pour une partition donnée $\mathcal{P}$, à l'issue de l'étape ③, il n'y a qu'une valeur possible pour la quantité $Q(\mathcal{P}, r_i)$ (les r_i étant déduits des $\mathcal{S}_i$). Il y a donc un nombre fini de valeurs possibles que peut prendre $Q(\mathcal{P}, r_i)$. Comme à chaque itération les valeurs obtenues sont plus petites, on va nécessairement arriver à une situation où $Q(\mathcal{P}, r_i)$ n'évolue plus.

Il est éventuellement possible que le partitionnement $\mathcal{P}$ et les r_i oscillent encore entre plusieurs possibilités [2] donnant une même valeur de $Q(\mathcal{P}, r_i)$, mais comme on souhaite minimiser cette quantité, les solutions sont de toute façon équivalentes.

2.3 Correction du résultat

La méthode des k-moyennes n'est, rappelons-le, qu'une méthode heuristique visant à trouver une partition assez proche de la partition recherchée : la partition $\mathcal{P}$ obtenue par l'algorithme ne garantit pas nécessairement que $Q(\mathcal{P}, r_i)$ soit minimal parmi toutes les partitions possibles de l'ensemble des données. Il est fort possible que l'algorithme se coince dans un minimum « local » et ne parvienne pas à trouver le minimum global, comme c'est fréquemment le cas dans des algorithmes d'optimisation.

Toutefois, quitte à faire plusieurs essais avec des choix différents pour les r_i initiaux, la méthode des k-moyennes donnent généralement une assez bonne approximation [3] de la solution recherchée.

2.4 Implémentation

En Python, on représentera chaque donnée (vecteur de $\mathbb{R}^p$) sous la forme d'une liste à p éléments. L'ensemble de ces listes se trouve dans une liste `data`. Les p représentants r_i sont eux mémorisés dans une liste `reps` contenant k listes de taille p.

2. Sous certaines conditions concernant l'implémentation sur lesquelles nous ne nous appesantirons pas, on peut s'assurer que ce ne sera pas le cas.
3. Que, sous certaines conditions, on peut parvenir à quantifier.

Durant l'étape ② qui construit la partition $\mathcal{P} = \{\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{k-1}\}$, il nous faut déterminer, pour chaque donnée x, le représentant r_i le plus proche, et construire la liste des i correspondants. Pour ce faire, nous pouvons utiliser la fonction ppv du chapitre précédent, et écrire la fonction suivante :

```python
def classif(data, reps):
    """retourne une liste de len(data) entiers entre 0 et len(reps)-1
       qui correspond à la classification 1-ppv de chaque donnée x
       avec les représentants r_i"""
    ids = [i for i in range(len(reps))]
    classes = []
    for x in data:
        classes.append(ppv(x, reps, ids))
    return classes
```

Durant l'étape ③, il nous faut recalculer les nouveaux r_i, en déterminant le barycentre de chaque $\mathcal{S}_i$. On conservera ici les r_i lorsque le $\mathcal{S}_i$ associé est vide. On écrira par exemple :

```python
def centroids(data, classes, reps):
    """retourne une liste des nouveaux représentants r_i
       pour chacune des classes, construite à partir de la
       partition fournie par la liste classes et les données"""

    # On détermine le cardinal nb[i] de S_i
    # et la somme sum_{x in S_i} x des éléments dans sums[i]
    p, k = len(data[0]), len(reps)
    sums, nb = [[0]*p for i in range(k)], [0]*k
    for i in range(len(data)):
        c = classes[i]
        for j in range(p):
            sums[c][j] += data[i][j]
        nb[c] = nb[c]+1

    # On construit les nouveaux représentants à partir des moyennes
    nouv_reps = []
    for i in range(len(reps)):
        if nb[i]>0:
            nouv_reps.append([sums[i][j] / nb[i] for j in range(p)])
        else:
            nouv_reps.append(reps[i]) # S_i vide, r_i conservé

    return nouv_reps
```

L'initialisation des r_i (étape ①) est une question délicate, car la qualité des résultats obtenus peut dépendre des valeurs initiales des r_i. Il existe plusieurs stratégies possibles, la plus simple étant de sélectionner k données aléatoirement parmi l'ensemble des n données. Cela peut par exemple se faire en construisant une liste de k entiers distincts choisis aléatoirement entre 0 et $n-1$, et en extrayant les données correspondantes. Cela s'écrira par exemple :

```python
from random import randint

def choix_representants(data, k):
    # On choisit aléatoirement k entiers distincts dans [0..n-1]
    index = []
    for i in range(k):
        j = randint(0, len(data)-1)
        while j in index:
            j = randint(0, len(data))
        index.append(j)
    # On construit la liste des r_i correspondants
    return [data[j] for j in index]
```

Une autre question délicate est celle de l'arrêt. On pourrait vouloir attendre que les r_i n'évoluent plus du tout, mais sous certaines conditions, cela peut ne jamais arriver. Dans la pratique, on peut simplement s'arrêter lorsqu'ils n'évoluent pratiquement plus (lorsque la distance maximale entre les r_i à une étape et à l'étape suivante est majorée par une petite quantité `maxdist`) ou bien lorsque l'on a effectué un nombre d'itérations `maxiter` choisi à l'avance. L'algorithme des k-moyennes peut alors s'écrire :

```python
def kmoys(data, k, maxdist, maxiter):
    reps = choix_representants(data, k)              # ①
    for _ in range(maxiter):
        classes = classif(data, reps)                # ②
        nouv_reps = centroids(data, classes, reps)   # ③

        # On détermine le déplacement maximal d'un r_i
        m = dist2(reps[0], nouv_reps[0])
        for i in range(1, k):
            m = max(m, dist2(reps[i], nouv_reps[i]))
        # S'il est suffisamment petit, on arrête tout !
        if m**0.5 < maxdist:
            return reps, classif(data, reps)

        # Les barycentres deviennent les nouveaux r_i pour la suite
        reps=nouv_reps                               # ④
```

2.5 Analyse de la méthode

La méthode des k-moyennes est particulièrement simple grâce à l'utilisation de la distance euclidienne. Mais elle ne donne pas toujours d'excellents résultats sur tous les ensembles de données. En particulier, elle tend à identifier

- des clusters plutôt « sphériques », que l'on peut séparer par des hyperplans
- des clusters de dimensions comparables,
- des clusters de cardinaux comparables...

Par ailleurs, le choix de k est très délicat, surtout si l'on ne connaît pas par avance le nombre de classes que l'on cherche à identifier, et il est difficile de juger, à partir des résultats, si le k choisi initialement était bon.

En outre, lors de l'exécution de l'algorithme, certaines classes peuvent se « vider » complètement, aussi le nombre de classes obtenues à l'issu de l'algorithme peut fort bien être strictement inférieur à k ! En général, ce n'est pas ce que l'on souhaite, aussi lorsqu'une classe se vide, on choisit généralement un nouveau r_i. Il existe de nombreuses méthodes pour ce faire. On pourra par exemple choisir une des données à classer au hasard, en privilégiant éventuellement celles se trouvant dans les classes $\mathcal{S}_i$ où la variance est la plus grande.

4 Théorie des jeux

1 Qu'est-ce que la théorie des jeux ?

1.1 De nombreuses « théories »

Le sujet des « jeux » est vaste, et beaucoup de « théories des jeux » ont vu le jour, dans des domaines très différents. Il ne s'agit pas ici de tenter de proposer quoi que ce soit d'exhaustif, mais simplement d'introduire quelques idées sur certains types de jeux bien précis (principalement des jeux d'accessibilité, même si nous irons un peu plus loin que cela) et de voir comment l'informatique peut nous aider à les analyser.

1.2 Ce que l'on entendra par « jeu »

Dans le cadre de ce cours, un *jeu* désignera ce qui peut être décrit par

- un ensemble $\mathcal{S}$ d'« états » (généralement appelés *positions*), qui peut être fini ou infini ;
- un ou plusieurs [1] joueur(s) qui, alternativement ou simultanément, font évoluer l'état par un choix parmi un ensemble de *coups* possibles, définis par les règles du jeu ; ces coups représentent donc des « transitions » entre deux états du jeu ;
- des situations où la règle du jeu attribue un *gain* (numérique, sous la forme de points marqués, ou bien simplement une victoire) à un ou plusieurs joueurs.

Les coups permis, dans un état donné, peuvent dépendre non seulement de l'état du jeu mais également de l'historique, et éventuellement d'une composante aléatoire (dés, pioche, etc.) De même, la condition clôturant le jeu peut dépendre de l'état du jeu (mat, objectif atteint, ...) mais également possiblement de l'historique de la partie.

Nous nous intéresserons plus particulièrement à des jeux d'*accessibilité*, où l'objectif pour chaque joueur se résume essentiellement à atteindre un état particulier du jeu (et empêcher son adversaire de faire de même).

Même avec cette description semble-t-il très large de ce qui peut consituter un jeu est pourtant restrictive et ne couvre pas toutes les possibilités de ce qui peut constituer un jeu : par exemple, on peut envisager d'étudier des jeux où les joueurs ont une action *continue* sur le jeu et où la notion de coup n'a pas de sens.

1. Il est possible d'envisager des « jeux » sans joueur, mais ce sont des problèmes très spécifiques qui n'ont que de très ténus liens avec ce dont nous voulons parler dans ce cours.

1.3 Le vocabulaire de la théorie des jeux

On peut ainsi trouver une grande variété dans les jeux. Nous l'avons dit, le nombre de joueurs est un aspect important. On s'intéressera principalement dans le cadre de ce cours à des jeux à deux joueurs. Au-delà de deux joueurs, les choses se compliquent beaucoup, car peuvent apparaître des questions « politiques » : un joueur peut se trouver dans une situation où un de ses choix n'a aucune conséquence sur son gain propre, mais peut influer sur le gain de deux de ses adversaires. Il devient alors plus difficile de modéliser sa décision, puisqu'elle ne repose plus sur des critères purement objectifs.

Dans un jeu à plusieurs joueurs, ceux-ci peuvent avoir un intérêt commun (jeux collaboratifs, où le gain est commun à tous les joueurs), ou des intérêts opposés. Un cas particulier intéressant est le jeu dit à « *somme nulle* » où le gain d'un joueur est directement opposé à celui de ses adversaires (c'est par exemple le cas d'un jeu à deux joueurs où il y aura un gagnant et un perdant, une situation que nous étudierons plus en détail dans ce cours). Les intérêts peuvent même être plus complexes, voire évoluer au cours de la partie.

Un jeu sera dit *impartial* si les règles que doivent suivre chacun des joueurs sont les mêmes. Il sera qualifié de *partisan* dans le cas contraire. Beaucoup de jeux usuels sont naturellement rangés dans la catégorie des jeux *partisans* car chaque joueur a son pion ou ses pièces, et qu'il ne peut déplacer que le pion ou les pièces qui lui appartiennent. Par exemple, aux échecs, aux dames ou au go, un joueur joue avec les pièces/pierres blanches, l'autre avec les pièces/pierres noires.

Cependant, la frontière pour de tels jeux est quelque peu floue : comme les joueurs jouent tour à tour, il suffit d'ajouter une règle indiquant que chaque joueur doit jouer une pièce ou une pierre de couleur différente de celle jouée précédemment [2] pour que le jeu devienne impartial sans rien changer aux règles. Il existe cependant de très nombreux jeux partisans par nature, où les joueurs suivent des règles radicalement différentes.

Enfin on peut distinguer les jeux à *information complète*, où l'état complet du jeu est connu à tout instant de l'ensemble des joueurs (ce qui est le cas des échecs, des dames, du go, du morpion, etc.) et ceux où les joueurs n'ont pas accès à la totalité de l'information (certaines informations pouvant même être inconnues de la totalité des joueurs), par exemple dans un jeu où les joueurs ont des cartes qu'ils dissimulent aux autres joueurs (ou, en renversant les conventions dans un jeu comme Hanabi, les cartes d'un joueur sont connues de tous sauf du joueur en question).

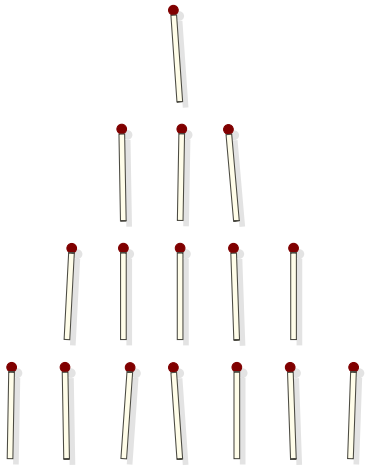
1.4 Quelques exemples de jeux d'accessibilité

Dans la suite, nous nous intéresserons principalement à des jeux d'*accessibilité* à deux joueurs, à information complète, impartiaux ou non, et où les joueurs jouent tour à tour. Les jeux d'accessibilité sont des jeux où, nous l'avons évoqué, on cherche à atteindre un ou plusieurs état(s) spécifique(s) du jeu.

2. En mettant de côté la situation où on passe son tour.

Jeu de Nim

Prenons pour premier exemple le cas du jeu de Nim. Dans sa version classique, il se joue avec des objets, typiquement des allumettes. Celles-ci sont regroupées en plusieurs rangées. Chacun à son tour, les joueurs peuvent retirer un nombre quelconque (non nul) d'allumettes d'une (unique) rangée. Le joueur qui prend la dernière allumette est déclaré vainqueur (de façon équivalente, on peut aussi dire que le joueur qui ne peut pas prendre d'allumettes à sont tour est déclaré perdant). Chaque joueur cherche donc à atteindre l'état du jeu où il ne reste aucune allumette.

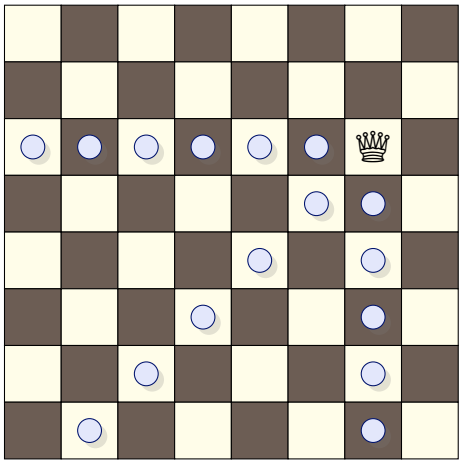


Il semblerait que ce soit le tout premier jeu qui ait été étudié et résolu mathématiquement. Ce jeu connait bien des variantes, et dans le film *L'année dernière à Marienbad*, on y trouve la variante de type « misère » où c'est le joueur qui *prend* la dernière allumette qui perd le jeu (on peut cependant montrer que les stratégies sont cependant quasiment identiques).

Il existe également des variantes où l'on ne distingue pas plusieurs rangées, mais où on limite le nombre d'éléments que l'on peut prendre : dans l'épreuve figurant dans *Fort Boyard*, par exemple, le candidat et son adversaire « maître du jeu » peuvent prendre au choix, à leur tour, 1, 2 ou 3 batonnets (et c'est également celui qui prend le dernier batonnet qui perd la partie).

Jeu de Wythoff

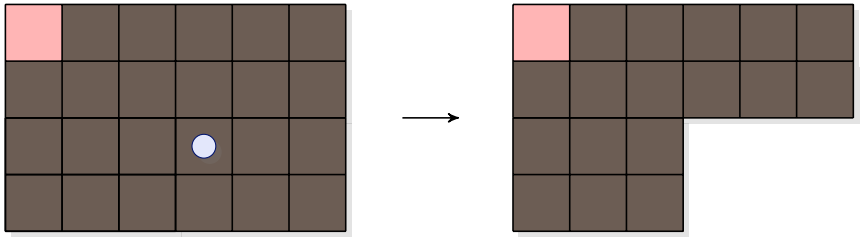
Variante du jeu de Nim, et possiblement le second jeu à avoir été résolu mathématiquement, le jeu de Wythoff se joue avec une pièce sur un plateau semblable à un échiquier. Chacun à son tour, les deux joueurs déplacent la pièce d'autant de cases qu'ils le souhaitent (au minimum d'une case), soit parallèlement à un des bords, soit en diagonale, mais toujours de manière à se rapprocher de la case en bas à gauche, comme illustré ci-après (les points représentant les déplacements autorisés de la pièce).



Dans le jeu de Wythoff, le joueur qui amène la pièce sur la case en bas à gauche est le gagnant (en d'autres termes, celui qui ne peut plus déplacer la pièce perd la partie).

Jeu de Chomp

Enfin, pour clore cette liste d'exemples, le jeu de Chomp, inventé indépendamment par Frederik Schuh en 1952 et David Gale en 1974 (sous une formulation un peu différente, plus grand public, que l'on reprend ici), utilise un ensemble de carrés, que l'on peut imaginer comme des morceaux d'une tablette de chocolat. Chacun à son tour, les joueurs choisissent un carré parmi ceux restant, et « mordent » dans la tablette, ce qui a pour effet de supprimer tous les carrés situés en dessous et à droite du carré choisi, comme illustré ci-dessous.



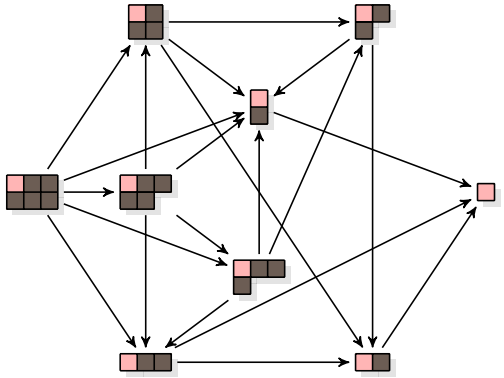
Le joueur qui mord le carré en haut à gauche perd la partie (en d'autres termes, le joueur qui, après avoir joué, ne laisse qu'un seul carré la remporte).

Ce jeu présente un aspect intéressant : malgré sa simplicité et les similitudes avec les jeux précédents, même si l'on sait, pour certaines situations, déterminer quel joueur remportera la partie, on ne sait pas encore efficacement déterminer les meilleurs coups dans une situation donnée.

2 Graphes et modélisation du jeu

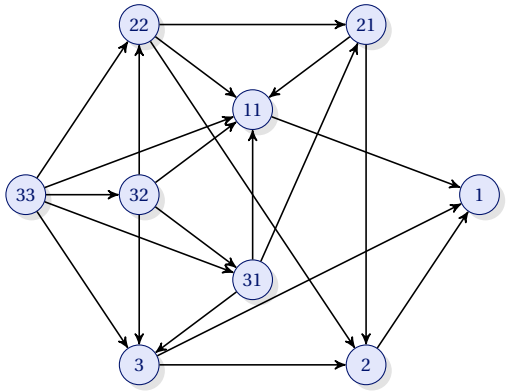
2.1 Arènes

Si l'on s'intéresse au jeu de Chomp, et que l'on part d'une modeste tablette de taille 2×3, l'ensemble des positions pouvant être atteintes lors d'une partie, et les coups les liant, est représenté ci-dessous :



D'un point de vue algorithmique, il s'agit ici d'un graphe, comme nous en avons croisé en première année : les sommets du graphes correspondent aux positions du jeu, les arcs (orientés) sont eux associés aux coups autorisés.

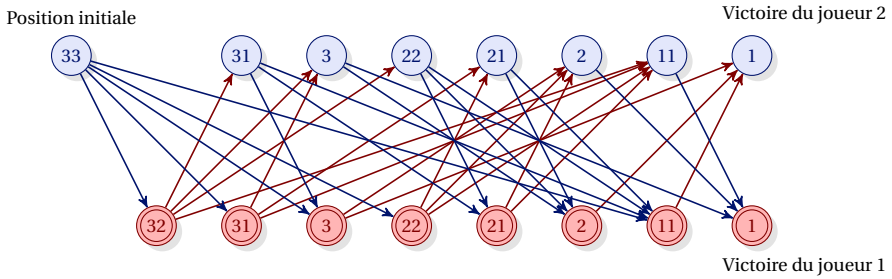
Dans la suite, on ne représentera pas l'état des différentes tablettes, mais simplement des sommets avec un identifiant, par simplicité (les identifiants ici correspondent au nombre de carrés sur chacune des lignes de la position correspondante), comme ci-dessous :



La partie peut être jouée directement sur le graphe : on part du sommet le plus à gauche (33), et chaque joueur, à son tour, choisit un arc menant à un nouveau sommet. Le joueur qui parvient à atteindre le sommet le plus à droite (1) gagne la partie.

Un sommet du graphe précédent ne nous renseigne toutefois pas entièrement sur l'état du jeu : il nous faut connaître, en outre, le joueur dont c'est le tour (on dit généralement en théorie des jeux que le joueur a le *trait*). Si l'on souhaite construire un graphe où chaque sommet correspond à un état possible du jeu, il nous faut donc dupliquer l'ensemble des sommets, les uns lorsque le premier joueur a le trait, les autres lorsque c'est le second.

Seul le premier joueur cependant peut se trouver dans l'état initial puisqu'il est forcé de retirer au moins un carré. On peut également remarquer qu'il est impossible de se trouver dans l'état 32 avec le trait pour le joueur qui a débuté la partie (que l'on désignera par « joueur 1 » dans la suite, par simplicité). Le graphe ressemble donc à celui-ci, après avoir arrangé la position des sommets pour que ceux qui correspondent aux situations où le joueur qui a débuté la partie a le trait se trouvent en haut, et celles où l'autre joueur a le trait, en bas (et représentés doublement cerclés) :



La partie débute ainsi sur le sommet en haut à gauche, et les arcs représentent toujours les coups permis, mais les deux joueurs ont cette fois des objectifs différents : le premier joueur essaie d'atteindre le sommet en bas à droite (une situation dans laquelle le second joueur, dont c'est alors le tour, n'a plus de coup possible), tandis que le second joueur veut atteindre le sommet en haut à droite (c'est le premier joueur, se trouvant dans l'impossibilité de jouer, qui perd).

Le grand nombre d'arcs figurant sur ce graphe n'en facilite pas la lecture, mais puisque dans le jeu les joueurs jouent alternativement, on peut diviser l'ensemble des sommets du graphe en deux groupes : ceux pour lesquels le premier joueur a le trait, et ceux pour lesquels le second joueur à le trait. Les arcs lient uniquement des sommets d'un groupe à des sommets de l'autre groupe. Un tel graphe est qualifié de *biparti*.

L'ensemble constitué
- des états du jeu (sommets)
- des coups possibles (arcs)
- de la position initiale (en haut à gauche sur notre exemple)
- des positions finales (en haut à droite et en bas à droite), en précisant pour chacune qui remporte la partie

constitue ce que l'on appelle en théorie des jeux l'*arène* du jeu. Son étude permet de déterminer si un des deux joueurs, en jouant de manière optimale, peut remporter à coup sûr la victoire, et éventuellement de proposer une stratégie pour y parvenir.

2.2 Implémentation en Python

Pour implémenter un tel jeu en Python, une tâche importante sera de créer une fonction qui, à partir d'un état du jeu (les carré restants et le trait dans le jeu de Chomp) fournit la liste des coups possibles, ou en d'autres termes, la liste des états accessibles par un coup valide.

Dans le cadre du jeu de Chomp, on peut décrire l'état de la tablette par un n-uplet indiquant le nombre de carrés restant sur chaque rangée (ligne). Par exemple, `(3, 2)` indique une tablette à laquelle il reste deux rangées, avec trois carrés sur la première rangée, deux sur la seconde. Si le jeu est parti d'une tablette rectangulaire, les éléments dans chaque n-uplet seront toujours rangés dans un ordre décroissant [3]. O choisit fréquemment d'utiliser en Python des objets de type « `tuple` » pour représenter l'état du jeu car étant immuables, ils peuvent être utilisés comme clés de dictionnaire, ce qui présente beaucoup d'avantages pratiques.

Pour décrire complètement la position du jeu, on ajoute un booléen indiquant si le joueur 1 a le trait : « `(3, 2), True` » correspond donc à une position où c'est au joueur 1 de jouer. La position initiale, sur notre exemple précédent, serait donc « `(3, 3), True` », la position gagnante pour le joueur 1 « `(1,), False` » [4] et la position gagnante pour le joueur 2, « `(1,), True` ».

Une fonction qui, à partir d'une position, retourne la liste des positions que l'on peut atteindre en jouant, peut par exemple s'écrire :

```python
def suiv(pos):
    etat, trait_j1 = pos

    # Si l'on choisit un carré en début de rangée (sauf la première)
    res = [(etat[:i], not trait_j1) for i in range(1, len(etat))]

    # Sinon, pour chaque rangée (première comprise)
    for i in range(len(etat)):
        # on envisage tous les carrés de la rangée excepté le premier
        for j in range(1, etat[i]):
            # et on construit le tuple résultant de ce choix
            n_etat = etat[:i]+tuple(min(j, e) for e in etat[i:])
            res.append((n_etat, not trait_j1))

    return res
```

3. Et toutes les combinaisons respectant cette condition pourront être atteintes, il suffit pour s'en convaincre de voir que l'on peut retirer les carrés un à un.

4. La virgule suivant le 1 est la convention, utilisée par Python, pour faire la différence entre un n-uplet à un seul élément et un simple entier entouré de parenthèses.

Par exemple, les positions accessibles depuis la position de départ « `(3, 3), True` » sont, d'après notre fonction :

```
In []: suiv(((3,3), True))
Out[]:
[((3,), False),
 ((1, 1), False),
 ((2, 2), False),
 ((3, 1), False),
 ((3, 2), False)]
```

Usuellement, cette fonction est la plus utile pour travailler avec un jeu. Mais parfois, on peut vouloir construire explicitement le graphe correspondant au jeu. Pour ce faire, on peut utiliser une exploration de ce graphe pour, par exemple, construire un dictionnaire représentant le graphe : les clés représenteront toutes les positions accessibles, les valeurs associées les positions qui correspondent à un coup possible. Par exemple, une exploration en profondeur [5], prenant en argument une fonction `suiv` construisant les coups possibles et un objet `init` correspondant à la position initiale, et construisant un tel dictionnaire peut s'écrire :

```python
def graphe(suiv, init):
    g = {} # Le dictionnaire qui contiendra le graphe

    def explore(pos):     # Exploration en profondeur
        if pos not in g:
            S = suiv(pos)
            g[pos] = S
            for s in S:
                explore(s)

    explore(init) # On lance l'exploration depuis init

    return g
```

Si on appelle g le graphe retourné par la fonction précédente, `g[pos]` contiendra directement les coups possibles depuis la position pos. L'un des avantages de construire un tel graphe est que l'on a dorénavant accès à la l'ensemble des positions qu'il est possible de rencontrer au cours d'une partie (en particulier, `g.keys()` fournit un itérable qui contient l'intégralité des clés du dictionnaire g, donc des positions que l'on peut atteindre).

Bien évidemment, une telle fonction ne peut être utilisée que si le nombre de positions est raisonnable. Aux échecs, on estime à $4{,}5 \times 10^{46}$ le nombre de positions pouvant être atteintes dans une partie, aussi si la fonction `suiv` peut parfaitement être écrite dans le

5. On pourrait tout aussi bien utiliser une exploration en largeur, cela n'a pas d'importance.

cadre des échecs, le graphe ne peut évidemment pas être reconstitué dans la mémoire d'un ordinateur (et l'exploration ne pourrait pas non plus être réalisée compte tenu du temps nécessaire !)

Dans le cadre du jeu de Chomp partant d'une tablette de taille 2×3, on peut vérifier que l'on retrouve bien les 16 positions précédemment illustrées sur notre graphe biparti :

```
In []: g = graphe(suiv, ((3, 3), True))

In []: g.keys()
Out[]: dict_keys([((3, 3), True), ((3,), False), ((1,), True),
((2,), True), ((1,), False), ((1, 1), False), ((2, 2), False),
((1, 1), True), ((2, 1), True), ((2,), False), ((3, 1), False),
((3,), True), ((3, 2), False), ((2, 2), True), ((2, 1), False),
((3, 1), True)])
```

2.3 Stratégies

En théorie des jeux, une *stratégie*, pour un joueur, est une méthode indiquant, à tout moment de la partie, comment choisir un coup parmi l'ensemble des coups possibles.

En général, le choix est fait à partir de la position actuelle. Une stratégie est donc une application $\sigma : \mathcal{S}' \subseteq \mathcal{S} \to \mathcal{S}$ qui à une position $s \in \mathcal{S}'$ associe une position $\sigma(s)$ telle que $(s, \sigma(s))$ soit un arc dans le graphe, autrement dit un coup valide.

On remarque ici que la stratégie peut n'être définie que pour un sous-ensemble $\mathcal{S}'$ des sommets du graphe : en effet, on peut se dispenser de traiter le cas de sommets qui ne seront pas à considérer lors de la partie (cela inclue tous les sommets pour lesquels le trait est à l'adversaire, mais également des sommets que l'on n'atteindra pas en raison de choix effectués par la stratégie).

Il peut arriver que la stratégie, pour effectuer un choix, ait besoin non seulement de la position actuelle, mais de l'ensemble des positions de la partie depuis le début. Cela n'a pas d'intérêt pour des jeux à information complète tels que ceux que l'on étudie, donc nous laisserons cette possibilité de côté [6].

Dans un jeu avec un gagnant et un perdant, une stratégie pour un joueur est dite *gagnante* si le joueur en question est assuré du gain de la partie s'il applique cette stratégie, *quels que soient les coups de son adversaire*. Bien entendu, dans un tel jeu, il ne peut y avoir de stratégie gagnante pour les deux joueurs. Une des questions les plus importante en théorie des jeux est : quel joueur a une stratégie gagnante ?

Certains jeux permettent, en plus d'une victoire de l'un ou l'autre des joueurs, de terminer sur une partie nulle. La question est alors de savoir si l'un des deux joueurs a une stratégie gagnante, ou si, lorsque les deux joueurs jouent de façon optimale, la partie est nécessairement toujours nulle (ce qui est par exemple le cas du tic-tac-toe).

Un jeu est dit *résolu* si l'on est en mesure de répondre à ces questions. De nombreux jeux ont été résolus, dont le jeu de Nim, le jeu de Withoff, ou encore les dames anglaises [7] qui se terminent en nul si les deux joueurs jouent parfaitement. Les dames, le reversi et les échecs résistent encore [8], même si l'on pense que la partie devrait se finir sur une nulle dans le cas des dames, possiblement aussi dans le cadre d'Othello [9]. Quant aux échecs, on pense que le joueur qui commence est probablement celui qui a une stratégie gagnante.

Pour certains jeux comme le Go, dans lesquels il y a une notion de « points » (prises et territoire en go) pour désigner le gagnant, il est possible d'attribuer des points au début de la partie à un joueur pour compenser l'avantage ou l'inconvénient de débuter. Actuellement au go, on attribue entre 6.5 et 7.5 points (*komi*) au joueur qui ne commence pas pour rendre la partie plus égale. La valeur idéale n'est évidemment pas connue, et la résolution du jeu dépend de la valeur de cette compensation (il est par exemple possible que le joueur 1 ait une stratégie gagnante avec un komi de 6.5 points, mais que le joueur 2 ait une stratégie gagnante pour un komi de 7.5 points) !

La résolution est dite *ultra-faible* lorsque l'on peut répondre à cette question sans pour autant pouvoir construire de stratégie qui assure d'une victoire (ou d'un nul). C'est par exemple le cas du jeu de Chomp lorsque la position initiale est une tablette rectangulaire.

Raisonnons par l'absurde et supposons que le joueur 2 dispose d'une stratégie gagnante. Supposons à présent que le joueur 1 débute la partie en ne retirant que le carré en bas à droite. À partir de là, par hypothèse, le joueur 2 a une série de coups qui l'assure de la victoire. Cependant, on remarque aisément que le premier coup que va jouer le joueur 2 aurait pu être joué par le joueur 1 dès son premier coup ! Le joueur 1 peut donc appliquer la stratégie du joueur 2 et s'assurer la victoire, ce qui est en contradiction avec les hypothèses.

Dans le jeu de Chomp où la position initiale est une tablette rectangulaire, on a donc la certitude que le joueur 1 remporte la partie s'il joue de façon optimal, même si l'on n'est pas en mesure de produire une stratégie gagnante. L'argument utilisé ici pour résoudre le jeu est appelé *vol de stratégie*.

Dans d'autres cas, il est possible de construire une stratégie gagnante. Considérons par exemple le jeu de Nim où la position initiale contient deux rangées avec le même nombre d'allumettes [10]. La stratégie dite *miroir* assure une victoire au joueur 2. Le principe est simple : à chaque fois que le premier joueur retire p alumettes d'une rangée, le second

6. En outre, on pourrait se ramener au cas où seule la position courante compte en incluant l'intégralité de l'historique dans la définition de la position. En fait, aux échecs ou au go, c'est même théoriquement nécessaire, car la règle mentionne des conditions sur le fait de revenir à un état du plateau déjà rencontré plus tôt dans la partie, et les coups valides (ou les conditions de victoire) dépendent donc des états précédents du plateau et non du seul état actuel.

7. Le jeu de dames où les dames n'effectuent pas de déplacement long.

8. Il est possible que l'on ne puisse jamais résoudre certains d'entre eux.

9. Du moins sur un plateau de taille 8×8, sur un plateau de taille 6×6, le jeu est résolu, et le joueur qui ne commence pas gagne la partie s'il joue de façon optimale.

10. Il existe une stratégie gagnante au jeu de Nim dans le cas général, basé sur les représentations binaires du nombre d'éléments dans chaque rangée. Le joueur qui a une stratégie gagnante dépend de la position initiale.

joueur a simplement à retirer p alumettes d'une autre rangée. Après le coup du joueur 2, il y a donc nécessairement toujours le même nombre d'alumettes sur les deux rangées. Si ce nombre est non nul, le joueur 1 ne peut pas prendre la dernière alumette sur son coup suivant et remporter la partie. Si ce nombre est nul, alors le joueur 2 vient de remporter la partie en prenant la dernière allumette.

Heureusement ou malheureusement, cette stratégie ne permet pas de résoudre la plupart des jeux. Par exemple aux échecs, on ne peut pas appliquer cette stratégie, car la capture d'une pièce par le premier joueur va possiblement rompre la symétrie (le joueur 2 n'ayant plus la pièce lui permettant d'effectuer le coup symétrique à celui du joueur 1).

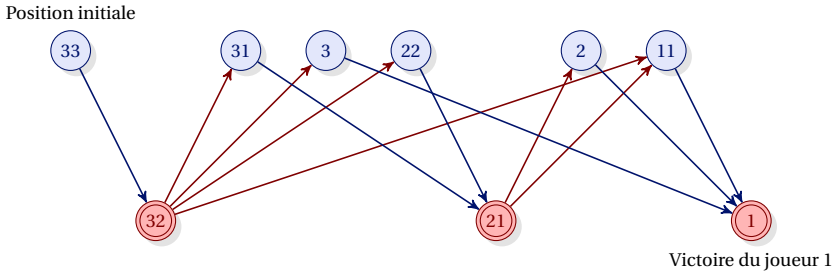
2.4 Attracteurs

S'il n'est pas possible de déterminer directement une stratégie gagnante, on peut, si le graphe du jeu n'est pas trop grand, déterminer l'ensemble des *positions gagnantes* d'un jeu. Une position est dite gagnante pour le joueur 1 s'il existe une stratégie gagnante pour le jeu lorsqu'il débute à cette position. On qualifiera parfois d'*attracteur* l'ensemble des positions gagnantes pour un joueur.

Il est relativement aisé de construire récursivement ces positions gagnantes en partant des fins de parties. Revenons par exemple sur notre exemple du jeu de chomp.

- la position « `(1,), False` » est l'objectif du joueur 1, donc c'est naturellement une position gagnante pour le joueur 1 ;
- les positions « `(2,), True` », « `(3,), True` » et « `(1, 1), True` » sont également des positions gagnantes pour le joueur 1 : en effet, c'est à lui de jouer et il peut choisir, parmi les coups valides, le coup qui l'amène à la position « `(1,), False` » et lui donne la victoire ;
- plus subtilement, la position « `(2, 1), False` » est égalemnt une position gagnante pour le joueur 1 : bien que ce soit au joueur 2 de jouer, il n'a que deux coups possibles, qui le conduisent aux positions « `(2,), True` » et « `(1, 1), True` », qui sont toutes deux des positions gagnantes pour le joueur 1 ;
- cela fait de « `(2, 2), True` » et « `(3, 1), True` » des positions gagnantes pour le joueur 1, puisqu'il peut jouer le coup conduisant à la position gagnante « `(2, 1), False` » ;
- par conséquent « `(3, 2), False` » est une position gagnante pour le joueur 1, car les quatres coups possibles pour le joueur 2 conduisent *tous* à une position gagnante pour le joueur 1 ;
- et enfin, la position initiale « `(3, 3), True` » est une position gagnante pour le joueur 1 car, parmi les coups qui lui sont proposés, il peut choisir le coup menant à la position gagnante « `(3, 2), False` ».

La position initiale étant une position gagnante pour le joueur 1, on peut donc en conclure que le joueur 1 a une stratégie gagnante qui lui assure la victoire, quoi que fasse le joueur 2. En effet, quels que soient ses choix, le joueur 1 peut s'assurer que toute la partie se déroule dans des positions gagnantes pour lui. Le graphe se résume alors à ceci [11] (pour chaque sommet correspondant à une position où le trait est au joueur 1, il n'y a qu'un seul coup indiqué, celui suggéré par la stratégie établie précédemment) :



On peut donc construire les positions gagnantes pour un joueur de la façon suivante :

- les positions qui lui donnent immédiatement la victoire sont, évidemment, des positions gagnantes ;
- toutes les positions pour lesquelles il a le trait et pour lesquelles il existe *au moins un coup* menant à une position gagnante sont des positions gagnantes
- toutes les positions pour lesquelles son adversaire a le trait et pour lesquelles *tous les coups possibles* mènent à une position gagnante sont des positions gagnantes

Pour écrire un programme Python déterminant les positions gagnantes, on regroupera celles que l'on identifie dans un dictionnaire dont elles sont les clé (les valeurs associées n'ont pas d'importance, on ne se sert ici d'un dictionnaire que pour effectuer rapidement des tests d'appartenance avec `in`). Pour savoir si une position est gagnante, on utilisera la fonction suivante, prenant en argument le graphe du jeu, la position que l'on étudie, et un dictionnaire des positions gagnantes connues :

```python
def est_gagnante_j1(g, pos, gagnantes_j1):
    trait_j1 = pos[1]

    if trait_j1: # Au moins un coup vers une position gagnante ?
        for p in g[pos]:
            if p in gagnantes_j1:
                return True
        return False
    else: # Tous les coups mènent à des position gagnantes pour J1 ?
        for p in g[pos]:
            if p not in gagnantes_j1:
                return False
        return True
```

11. On pourra aisément vérifier que toutes les positions restantes dans le graphe sont des positions gagnantes pour le joueur 2.

La fonction retourne **True** si on peut affirmer que la position fournie est une position gagnante, et **False** dans le cas contraire, *ce qui ne signifie pas que la position n'est pas gagnante, mais que l'on n'a pas encore les moyens de l'affirmer*. Incidemment, on remarquera que la fonction retourne bien **True** pour la position finale gagnante du joueur 1, quel que soit le contenu du dictionnaire `gagnantes` !

Notons que le langage Python fournit des fonction `any` et `all` qui retournent **True** si au moins un (`any`) ou la totalité (`all`) des booléens fournis en argument sont égaux **True**, et permettent de simplifier l'écriture de ce type de fonctions, tout en les rendant également plus aisées à comprendre :

```
def est_gagnante_j1(g, pos, gagnantes_j1):
    if pos[1]:
        return any(p in gagnantes_j1 for p in g[pos])
    else:
        return all(p in gagnantes_j1 for p in g[pos])
```

Pour construire l'attracteur, on peut alors écrire :

```
def attracteur_j1(g):
    gagnantes_j1 = {}

    fini = False
    while not fini:
        fini = True
        for pos in g.keys():
            if (pos not in gagnantes_j1
                    and est_gagnante_j1(g, pos, gagnantes_j1)):
                fini = False
                gagnantes_j1[pos] = True # (Valeur sans importance)

    return list(gagnantes_j1.keys())
```

Le fonctionnement est simple : à chaque itération, on étudie toutes les positions qui n'ont pas encore été déclarées gagnantes. Si certaines d'entre elles peuvent être déterminées gagnantes grâce à la fonction `est_gagnante`, on les ajoute au dictionnaire [12], et on poursuit la recherche. Si une itération ne permet pas de trouver de nouvelles positions gagnantes pour le joueur 1, le booléen `Fini` restera à **True** et la fonction va s'arrêter.

On peut garantir l'arrêt de la fonction précédente grâce à un variant de boucle : le nombre de positions dans le dictionnaire `gagnantes` est un entier qui croît stritement à chaque itération, et ne peut dépasser le nombre (fini) de sommets du graphe, aussi ne peut-il pas y avoir un nombre infini d'itérations !

La fonction nous fournit bien les sommets gagnants pour le joueur 1 :

```
In []: attracteur_j1(g)
Out[]: [((1,), False), ((1, 1), True), ((3,), True),
 ((2,), True), ((2, 1), False), ((3, 1), True),
 ((2, 2), True), ((3, 2), False), ((3, 3), True)]
```

Il s'agit d'une implémentation naïve de l'algorithme : à chaque étape, on réexamine *toutes* les positions. S'il y a n positions, le nombre d'itérations peut être de l'ordre de n, chaque itération examinera n positions, et chaque vérification peut avoir une complexité en $O(n)$, en fonction du nombre de coups possibles depuis la position considérée. On a donc ici une complexité cubique ($O(n^3)$), élevée (surtout que n peut être gigantesque).

Il est bien évidemment possible d'améliorer les choses, en identifiant mieux les positions dont la situation a pu changer (ce qui peut se faire en regardant les positions qui permettent de mener à une position qui vient d'entrer dans l'ensemble des positions gagnantes) et en dénombrant, pour chaque position, le nombre de coups ne menant pas à une position déjà dans l'ensemble des positions gagnantes, pour accélerer les tests. La complexité peut être réduite à une complexité linéaire en le nombre d'arcs dans le graphe, au prix d'un fonction plus complexe. On ne s'en préoccupera pas ici, seul le principe général nous intéresse.

Si le graphe est dépourvu de cycles [13], comme c'est le cas ici, alors toutes les cases qui ne figurent pas dans la liste retournée par notre fonction sont des positions gagnantes pour le joueur 2. En effet, toute position qui n'est pas retournée est, d'après l'algorithme mis en œuvre :

- soit une victoire pour le joueur 2 ;
- soit une position où le joueur 2 a le trait et dont au moins un coup conduit à une position qui n'est pas dans l'attracteur
- soit une position où le joueur 1 a le trait et où *tous* les coups conduisent à des positions qui ne sont pas dans l'attracteur.

Si le jeu se trouve dans une telle postion, le joueur 2 dispose donc d'une stratégie qui empêche le joueur 1 de se placer dans l'attracteur. Si la partie est finie (pas de cycles), elle se terminera donc nécessairement en une victoire du joueur 2 (puisque les positions victorieuses du joueur 1 se trouvent dans l'attracteur).

12. La valeur **True** associée n'a aucune importance, elle ne sert pas dans l'algorithme. Il existe une autre structure, les `set`, qui sont essentiellement des dictionnaires avec des clés sans valeur associée pour ce genre d'usage, mais cette structure n'est pas au programme.

13. Si le graphe contient des cycles, la partie peut durer indéfiniment si aucun des deux joueurs n'a d'intérêt à briser le cycle, et il faudra sans doute une règle pour déterminer un gagnant dans ce genre de situation, ou convenir d'une nullité.

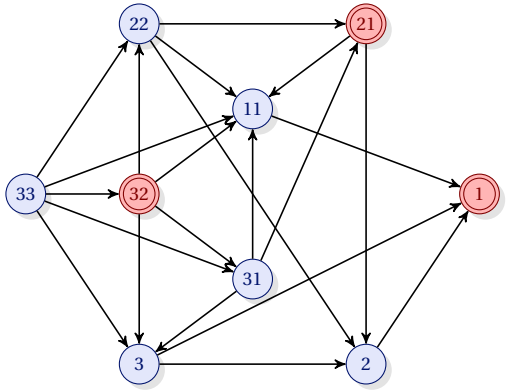
2.5 Noyau

Dans le cas très spécifique de jeux impartiaux tels que ceux présentés jusqu'ici, on peut remarquer une forte symétrie entre les situations des deux joueurs : si une position est gagnante pour un joueur lorsqu'il a le trait, la situation similaire sera perdante pour ce même joueur si son adversaire à le trait (et inversement).

Il n'est donc pas forcément nécessaire de faire entrer le trait dans le graphe, et on peut directement raisonner sur le graphe des configurations.

Dans un graphe donné [14], un sous-ensemble $\mathcal{S}'$ de sommets est dit *stable* si tout sommet de $\mathcal{S}'$ n'a aucun successeur dans $\mathcal{S}'$. Un sous-ensemble $\mathcal{S}'$ de sommets est dit *absorbant* si tout sommet n'appartenant pas à $\mathcal{S}'$ possède au moins un successeur dans $\mathcal{S}'$. Un sous-ensemble $\mathcal{S}'$ de sommets est un *noyau* s'il est à la fois stable et absorbant.

Il est possible de montrer qu'un graphe orienté sans cycle possède un unique noyau. Par exemple, pour le graphe associé aux configurations du jeu de Chomp débutant sur une tablette 2×3, le noyau correspond aux sommets doublement cerclés ci-dessous :

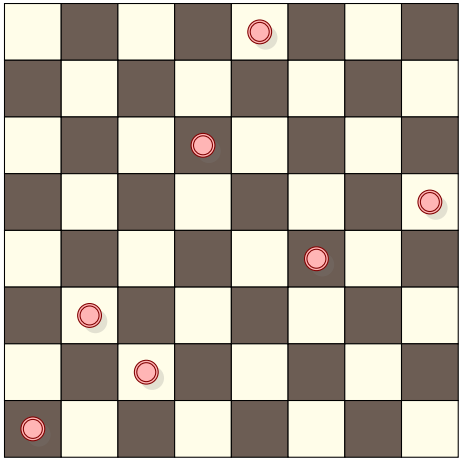


La construction du noyau peut se faire itérativement, de manière très semblable à la détermination d'un attracteur. Il est alors aisé de déduire l'attracteur pour le joueur 1 à partir de ce noyau : il s'agit des sommets du noyau pour lesquels son adversaire a le trait, et les sommets qui ne sont pas dans le noyau pour lesquels il a le trait [15].

La stratégie, pour le joueur 1, consiste donc à systématiquement jouer des coups qui amènent à l'intérieur du noyau. Comme le noyau est stable, les coups du joueur 2 sortiront nécessairement du noyau. Et comme le noyau est absorbant, le joueur 1 disposera toujours d'un coup pour y revenir. Le joueur 1 dispose donc d'une stratégie gagnante si et seulement si l'état initial du jeu est hors du noyau. Dans le cas contraire, c'est le joueur 2 qui dispose d'une stratégie gagnante.

14. Les propriétés définies ici et la notion de noyau ne figurent pas au programme.
15. Et inversement pour construire l'attracteur pour le joueur 2.

Dans le cas du jeu de Withoff sur un échiquier de taille 8×8, le noyau est ainsi représenté ci-dessous : dès qu'un joueur peut amener la pièce dans le noyau, il dispose d'une stratégie gagnante et remportera la partie s'il joue de façon optimale, quels que soient les coups de son adversaire.



3 Algorithme min-max

3.1 Un jeu de collecte

Il n'est fréquemment pas possible de construire l'attracteur complet d'un jeu, mais on souhaiterait cependant pouvoir construire une stratégie à partir d'une position donnée. Il existe plusieurs approches pour ce faire, dont l'algorithme dit « min-max ».

Pour illustrer son principe, considérons un jeu utilisant une grille de taille $n \times n$ contenant des valeurs, par exemple les entiers de 1 à n^2 comme sur la grille ci-dessous :

3	6	2
7	9	1
8	5	4

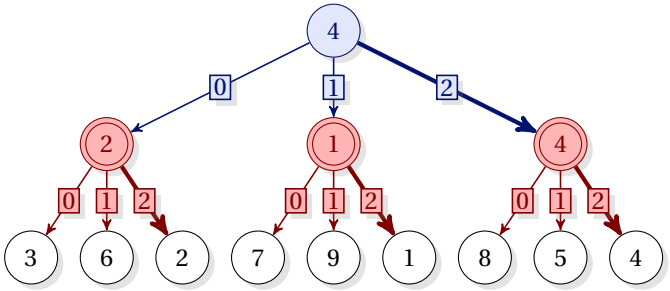
Le jeu se joue à deux joueurs, dans un premier temps avec les règles suivantes :

- le joueur 1 choisit une ligne i (entre 0 et $n-1$) ;
- le joueur 2, connaissant le choix du joueur 1, choisit une colonne j (toujours entre 0 et n) ;
- le joueur 1 marque les points $c_{i,j}$ indiqués dans la case indiquée (ligne i, colonne j) et le joueur 2 marque $a - c_{i,j}$.

La valeur de a peut être choisie pour équilibrer le jeu, nous y reviendrons. Mais elle n'a pas d'importance. Le joueur 1 va jouer de manière à avoir le plus grand $c_{i,j}$ possible, le joueur 2 au contraire cherchera à le minimiser.

Par exemple, si l'on considère la grille nous ayant servi d'exemple, le joueur 1 pourrait être tenté de choisir la seconde ligne ($i = 1$) pour tenter de marquer 7 ou 9 points, mais on comprendra que le joueur 2 choisirait alors la troisième colonne ($j = 2$), et le joueur 1 ne marquerait que $c_{1,2} = 1$ point. La bonne stratégie consiste, pour le joueur 1, à choisir la *troisième* ligne ($i = 2$) de sorte que le joueur 2 sera contraint de lui céder au moins 4 points.

Pour comprendre comment on peut parvenir automatiquement à ce raisonnement, nous allons construire l'ensemble des déroulements possibles de la partie sous une forme arborescente :



Les « flèches » de ce graphe indiquent les choix possible (le numéro de ligne ou de colonne), pour le joueur 1 en haut, puis pour le joueur 2 en bas. Sur la dernière ligne, on retrouve le score du joueur 1 à l'issue de la partie.

Lorsque c'est au joueur 2 de jouer, il va nécessairement essayer de minimiser le score du joueur 2, puisque cela aura pour conséquence de maximiser le sien. On peut prévoir le choix qu'il fera, et déterminer à l'avance le score final de la partie (en supposant qu'il joue de façon appropriée) en déterminant le *minimum* parmi les scores finaux qui peuvent être atteints après le choix de la ligne par le joueur 1. Ces minimas ont été reportés sur la seconde ligne du graphe.

Dans un second temps, en considérant que le joueur 1 va effectuer un choix qui va maximiser son score, comme il peut attendre du joueur 2 qu'il joue de manière optimale, il va opter pour la ligne qui correspond au *maximum* parmi les valeurs déterminées à l'étape précédente.

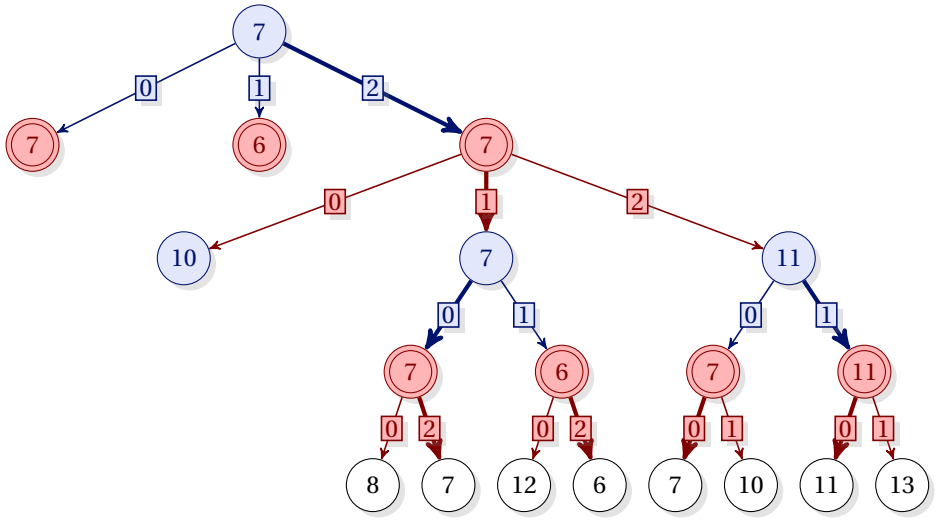
Les choix optimaux des deux joueurs dans toutes les situations ont été marqués par des traits plus épais dans le graphe ci-dessus. On voit que le calcul du minimum ou du maximum (selon le joueur qui a le trait) permet de déterminer à l'avance le résultat de la partie (si les deux joueurs jouent de façon optimale) et également la stratégie, pour chacun des deux joueurs, pour y arriver. Si l'un des deux joueurs ne suit pas cette stratégie, alors il offre à son adversaire une possibilité de marquer davantage de points.

3.2 Tours successifs

Que se passe-t-il si l'on complique le jeu en proposant *deux* tours au lieu d'un (après la première collecte, le joueur 1 choisit à nouveau une ligne i', puis le joueur 2 choisit à nouveau une colonne j', et le joueur 1 ajoute $c_{i',j'}$ à son score), avec la condition supplémentaire que les deux joueurs ne peuvent pas faire un choix qu'ils ont déjà effectué [16].

Il est intéressant de voir que les joueurs en peuvent pas agir sur un tour sans prévoir le tour suivant. Supposons en effet que le joueur 1 commence par choisir la ligne $i = 2$. Si, comme précédemment, le joueur 2 choisit la colonne $j = 2$, le joueur 1 pourra ensuite choisir la ligne $i' = 1$ et marquer $4 + 7 = 11$ points au total. Mais si le joueur 2 avait choisi, au premier tour, la colonne $j = 1$, le joueur 1 choisira la ligne $i' = 0$ au second tour et ne marquera que $5 + 2 = 7$ points.

Pour déterminer le score de la partie si les joueurs jouent de façon optimale et les stratégies correspondantes, on peut à nouveau construire un graphe arborescent des différentes parties. Comme il est de grande taille (36 branches!), on ne le représente que partiellement ci-dessous (même si l'arbre entier a été nécessaire pour déterminer les valeurs représentées ici). On y trouve la confirmation que 7 est bien le score sur lequel devrait se clore la partie.



La construction est la même que précédemment : les fins de parties (en bas) ont un score bien déterminé, et on détermine les autres valeurs en remontant. Lorsque le trait appartient au joueur 1, celui choisit la plus grande valeur parmi celles immédiatement en-dessous, lorsqu'il appartient au joueur 2, c'est la même chose, mais en choisissant la plus petite valeur.

16. Sinon, leur stratégie optimale serait simplement de refaire la même chose qu'au premier tour.

3.3 Implémentation

Pour déterminer le score d'une partie idéale en Python, on commencera par écrire une fonction prenant, comme souvent, l'état du jeu et renvoyant la liste des coups possibles. On décrira l'état du jeu par une liste contenant les choix des joueurs depuis le début de la partie. Par exemple, si le premier joueur a choisi $i = 2$ et le second $j = 1$ et que l'on se trouve au début du second tour, l'état du jeu sera décrit par `[2, 1]`. La fonction retournant la liste des coups possibles peut par exemple s'écrire :

```
def coups_possibles(etat, n):
    possibles = []
    for k in range(n):
        if len(etat)<2 or etat[len(etat)-2]!=k:
            possibles.append(k)
    return possibles
```

Elle prend en argument, outre l'état, l'entier `n` désignant le nombre de lignes/colonnes. Elle examine toutes les possibilités `k` entre `0` et `n-1` ; si on se trouve au second tour (la liste contient au moins deux éléments), elle vérifie que la possibilité n'a pas été utilisée par ce même joueur (le coup joué au premier tour se trouvant à la position `len(etat)-2` dans la liste `etat`).

Ainsi, par exemple, si l'état du jeu est décrit par la liste `[2, 1]`, la fonction indique que seuls les coups $i' = 0$ et $i' = 1$ sont des coups valides ($i' = 2$ constituant une répétition du choix du premier tour) :

```
In []: coups_possibles([2, 1], 3)
Out[]: [0, 1]
```

Puis il ne « reste » qu'à déterminer le score de la partie avec la démarche « min-max » que l'on vient de présenter. Pour cela, on écrit une fonction prenant l'état courant de la partie (`etat`) et la grille décrivant le contenu des cases (`M`). Elle devra renvoyer le score final de la partie, si celle-ci est passé par l'état indiqué en paramètre, en supposant que les deux joueurs jouent de manière idéale.

Le cas le plus simple est celui où la partie est terminée, c'est-à-dire lorsque `etat` contient 4 valeurs, i, j, i' et j'. Il suffit alors de déterminer le score final en utilisant `M`.

Dans le cas contraire, il reste des coups à jouer. On envisage donc tous les coups possibles avec la fonction précédente, et pour chaque coup `k`, on détermine l'état du jeu si le coup est joué (en construisant un nouvel état [17] `etat+[k]`, consistant en la concaténation de l'état courant et du coup supplémentaire k) et, grâce à un appel récursif, le score final de la partie en partant de cet état.

17. Il serait sans doute plus pertinent d'utiliser append, mais il faudrait penser à retirer k avec un pop après l'appel récursif, et avant d'envisager un autre coup.

Il ne reste alors qu'à chercher le minimum parmi ces possibilités (le joueur 2 a le trait) ou le maximum (le joueur 1 a le trait) pour déterminer le score final de la partie si elle se trouve, à un moment donné, dans l'état décrit par `etat`. Le joueur qui dispose du trait est aisément déduit de `etat` grâce à la parité de sa longueur. Cela donne par exemple :

```
def minmax(etat, M):
    if len(etat) == 4:
        i1, j1, i2, j2 = etat
        return M[i1][j1] + M[i2][j2]

    # On envisage les coups possibles
    # et, par récursion, le résultat obtenu par min-max
    L = []
    for k in coups_possibles(etat, len(M)):
        L.append((k, minmax(etat+[k], M)))

    # On détermine le coup optimal
    meilleur, meilleur_score = L[0]
    for k in range(1, len(L)):
        # J1 a le trait -> cherche le maximum
        if len(etat)%2 == 0 and L[k][1] > meilleur_score:
            meilleur, meilleur_score = L[k]
        # J2 a le trait -> cherche le minimum
        if len(etat)%2 == 1 and L[k][1] < meilleur_score:
            meilleur, meilleur_score = L[k]

    return meilleur_score
```

On peut alors confirmer que le score d'une partie idéale sera bien 7 (l'état initial du jeu, avant qu'aucun coup ne soit joué, est naturellement décrit par la liste vide `[]`) :

```
In []: minmax([], [[3, 6, 2], [7, 9, 1], [8, 5, 4]])
Out[]: 7
```

Compte tenu de ce résultat, le bon choix pour a, dans le cas précédent, serait probablement 3.5 : si les deux joueurs jouent de façon idéale, cela se terminerait sur une partie nulle. Et si un joueur joue mieux que son adversaire, il peut espérer l'emporter.

On peut envisager d'augmenter la taille de la grille, et de multiplier les tours. Cela étant dit, le calcul du score d'une partie « idéale » deviendra rapidement prohibitif. La mémoïzation ne permettrait pas directement de nous aider, car tous les états visités, par construction, seront différents. Toutefois, si on remarque que certains états (les états `[0, 0, 1, 2]` et `[1, 2, 0, 0]` par exemple) sont équivalents, on doit pouvoir réduire le nombre de branches à effectivement explorer.

Il existe également des techniques (dites techniques d'*élagage* permettant d'éviter d'explorer certaines parties du graphe. Si l'on sait par avant qu'un coup est meilleur que tout ce qui pourrait se trouver dans un partie donnée du graphe, alors il est inutile de l'explorer intégralement.

Lorsque l'algorithme détermine le score d'une partie idéale à partir d'un état donné, il peut également mémoriser les coups joués qui y conduisent. Ces coups (et notamment le premier d'entre eux) fournit une stratégie pour le joueur dont c'est le tour. C'est ce genre d'approche (dans des versions un peu plus élaborées pour des raisons d'efficacité) qui est utilisée dans les « intelligences artificielles » qui sont capables de jouer.

3.4 Retour sur les jeux avec un gagnant

Si l'on revient à nos jeu d'accessibilité (ou de façon générale aux jeux avec un gagnant et un perdant), il n'y a pas de notion de score qui nous permette d'effectuer la même chose. Mais il est aisé d'en construire un : il suffit de dire que le joueur 1 a un score de $+v$ (v étant strictement positif) s'il remporte la partie, $-v$ s'il la perd. Si le nul est possible, on choisira un score de 0.

Fréquemment, on choisi $v = +\infty$ en théorie des jeux, mais on peut tout aussi bien choisir les valeurs $+1$ et -1 (manipuler ∞ peut présenter quelques difficultés pratiques d'implémentation), cela n'a pas d'importance.

Le reste de la démarche est la même : le joueur 1 continue à souhaiter obtenir le score maximal (une victoire ($+v$) si c'est possible, un score nul à défaut (0) et une défaite uniquement s'il ne peut pas faire autrement ($-v$). Pour le joueur 2, c'est l'inverse, il recherche le minimum (une défaite du joueur 1 si c'est possible ($-v$), une nulle à défaut (0), et une victoire du joueur 1 uniquement s'il ne peut pas faire autrement ($+v$)).

3.5 Heuristiques

Malheureusement, le nombre de possibilités est souvent bien trop grand pour que l'on puisse explorer l'arbre des parties possibles jusqu'à la fin de celles-ci. Pour éviter un temps de calcul trop long, on va être amené à limiter le nombre de coups successifs envisagés. L'ennui, c'est que lorsque l'on atteint cette limite, on ne peut pas dire avec certitude si la partie se terminera sur une victoire du joueur 1, une victoire du joueur 2 (ou éventuellement une partie nulle).

Dans cette situation, on se sert d'une *heuristique* : une fonction qui examine l'état courant de la partie, et retourne une valeur dans $]-v, v[$ qui estime à quel point on semble proche d'une victoire du joueur 1 (la valeur renvoyée sera proche de v), d'une victoire du joueur 2 (la valeur renvoyée sera proche de $-v$), ou que la partie est encore indécidée (la valeur renvoyée sera proche de 0).

Les meilleures heuristiques permettront d'obtenir de meilleures stratégies lorsque l'on est encore loin de la fin de la partie, et elles ne sont pas du tout évidentes à établir. Aux échecs, par exemple, elles sont basées généralement sur le « matériel » (les pièces) encore présentes sur le plateau. On considère généralement qu'un pion vaut 1 point, les cavaliers 3 points, les fous 2.5 points (3 points si les deux fous sont encore présents), les tours 5 points, les reines 9.5 points. L'heuristique utilisée consiste alors généralement en la différence de valeur entre le matériel du joueur 1 et celui du joueur 2, souvent corrigée en utilisant la mobilité de chacune des pièces et quelques autres paramètres (on choisira évidemment un v assez grand pour que l'heuristique demeure dans l'intervalle $]-v, v[$!)