

## Cours 7 & 8

# Algorithmes, Arbres, Graphes et Intelligence Artificielle

*Sébastien Combéfis, Quentin Lurkin*



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Objectifs

- Découverte de l'**algorithmique**
  - Définition et principe
  - Programmation récursive
- Le type abstrait de données **arbre**
  - Type abstrait de données
  - Arbre, propriétés et implémentation récursive
  - Backtracking

# Objectifs

- Techniques de **recherche de solution** en intelligence artificielle
  - Espace d'états, état du jeu et coups
  - Recherche non informée
  - Recherche informée
- **Librairies et framework** d'IA en Python
  - Librairie Simple IA
  - Framework easyAI

# Algorithme

## Juhan's Day Algorithm

Kiss my wife + kids

Eat 1 green meal

Design 1 thing

Sketch (10m)

Walk twice (20m each)

Listen to a story (20m)

Read a story (20m)

Make a story (20m)

# Algorithme

- Un **algorithme** est un ensemble d'opérations à effectuer

*Décrit un processus qu'il est possible d'exécuter*

- Nom provenant de **al-Khwārizmī**

*Mathématicien, astronome, géographe et savant Persé*

- Plusieurs **applications**

- Calcul
- Traitement de données
- Raisonnement automatique

# Description d'algorithme

- Méthode effective pour calculer une fonction

*Décrise dans un langage formel bien défini*

- Plusieurs façons de décrire un algorithme

- Langue naturelle
- Pseudo-code
- Langage de programmation
- Formalisme mathématique

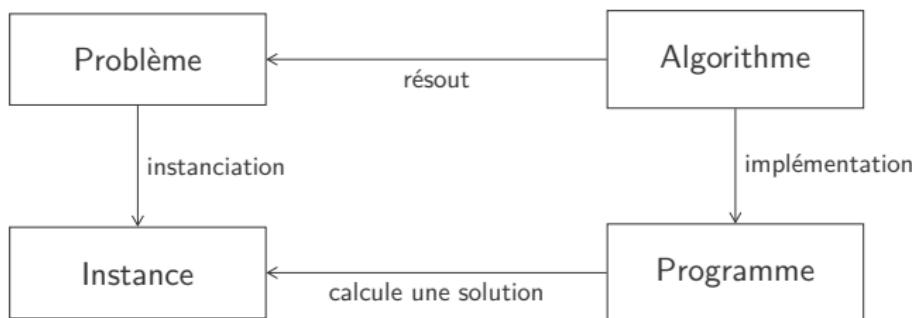
# Problème

- Un algorithme a pour but de résoudre un **problème**

*Qui est exprimé sous la forme d'une fonction à calculer*

- **Implémentation** dans un langage de programmation

*Permet de résoudre concrètement des instances du problème*



# Spécification d'un problème (1)

- **Quatre éléments** à identifier pour spécifier un problème
  - **Entrées** : *données à fournir qui sont nécessaires au problème*
  - **Sorties** : *résultat produit qui est solution du problème*
  - **Effet de bord** : *modification de l'environnement et des paramètres*
  - **Situations exceptionnelles** où une exception est levée

```
1 def factorial(n):
2     """Calcule la factorielle d'un nombre naturel.
3
4     Input: n, un nombre naturel.
5     Output: n!, la factorielle de n.
6     Raise: ArithmeticError lorsque n < 0
7     """
8     pass
```

# Spécification d'un problème (2)

- **Quatre éléments** à identifier pour spécifier un problème
  - **Précondition** : *conditions qui doivent être satisfaites sur l'environnement et les paramètres avant de pouvoir faire l'appel*
  - **Sorties** : *conditions qui sont satisfaites sur l'environnement, la valeur renvoyée et les paramètres après l'appel, si les préconditions étaient satisfaites*
  - **Situations exceptionnelles** où une exception est levée

```
1 def factorial(n):
2     """Calcule la factorielle d'un nombre naturel.
3
4     Pre: n > 0.
5     Post: La valeur renvoyée contient la factorielle de n.
6     Raise: ArithmeticError lorsque n < 0
7     """
8     pass
```

# Implémentation

- Plusieurs **implémentations** possibles pour un algorithme  
*Langages et choix d'implémentation différents*
- Plusieurs **algorithmes** possibles pour un problème  
*Approches et structures de données différentes*

# Exemple : Recherche dans une liste

- Étant donné une liste, **rechercher** si un élément en fait partie

*Attention, la liste peut être vide*

- **Spécification** du problème

```
1 def contains(data, elem):
2     """Recherche un élément donné dans une liste de données.
3
4     Pre: -
5     Post: La valeur renvoyée est
6         True si elem apparaît au moins une fois dans la liste
7         et False sinon.
8     """
9     pass
```

# Algorithme en langue naturelle

- 1 Si la liste est **vide**, on renvoie directement False et l'algorithme se termine
- 2 Pour chaque élément de la liste
  - 1 Si l'élément parcouru est **égal à celui recherché**, on renvoie directement True et l'algorithme se termine
  - 3 L'élément recherché n'est **pas dans la liste**, on renvoie False

# Algorithme en pseudo-code

- Le **pseudo-code** est une façon de décrire un algorithme  
*De manière indépendante de tout langage de programmation*
- Pas de réelle **convention** pour le pseudo-code  
*Les opérations sont de haut niveau*

---

**Algorithm 1:** Recherche si un élément se trouve dans une liste

---

```
if  $n = 0$  then
    ↘ return false
foreach  $e \in L$  do
    if  $e = elem$  then
        ↘ return true
return false
```

---

# Optimisation de l'algorithme

- Possibilité d'**optimisation** d'un algorithme
  - Diminution du nombre d'opérations à effectuer
  - Simplification de l'algorithme
- Le **cas  $n = 0$**  ne doit pas être traité séparément

---

**Algorithm 2:** Recherche si un élément se trouve dans une liste

---

```
foreach e ∈ L do
    if e = elem then
        return true
return false
```

---

# Algorithme en langage de programmation

- Implémentation effective dans un **langage de programmation**

*Utilisation des caractéristiques spécifiques du langage*

- Peut être fait par **traduction** du pseudo-code

*Traduction des constructions de haut niveau dans le langage*

```
1 def contains(data, elem):
2     for e in data:
3         if e == elem:
4             return True
5     return False
```

# Optimisation du programme

- Possibilité d'**optimisation** d'un programme
  - Utilisation de constructions spécifiques au langage
  - Exploitation de la librairie standard
- Python possède un **opérateur in** pour tester l'appartenance

```
1 def contains(data, elem):  
2     return elem in data
```

# Récurseion



714

Strange Loops, Or Tangled Hierarchies

FIGURE 142. Print Gallery, by M. C. Escher (lithograph, 1956).

program now. On a low (machine language) level, the program looks like any other program; on a high (chunked) level, qualities such as "will", "inclusion", "creativity", and "consciousness" can emerge.

The important idea is that this "vortex" of self is responsible for the tangledness of the mental processes. People have said to me on occasion, "This staff with self-reference and so on is very confusing and cryptic, but do you really think there is anything serious to it?" I have always responded, "Yes, it is serious, and it is at the core of AI, and it certainly is, I think it will eventually turn out to be at the core of AI, and the reason all attempts to understand how human minds work, that is why Gödel is so deeply woven into the fabric of my book."

## An Escher Vortex Where All Levels Cross

A strikingly beautiful, and yet at the same time disturbingly grotesque, illustration of the cyclonic "eye" of a Tangled Hierarchy is given to us by Escher in his *Print Gallery* (Fig. 142). What we see is a picture gallery where a young man is standing, looking at a picture of a ship in the harbor of a small town, perhaps a Maltese town, to guess from the architecture, with its little turrets, occasional cupolas, and flat stone roofs, upon one of which sits a boy, relaxing in the heat, while two floors below him a woman—perhaps his mother—gazes out of the window from her apartment which sits directly above a picture gallery where a young man is standing, looking at a picture of a ship in the harbor of a small town, perhaps a Maltese town—What? We are back on the same level as we began, though all logic dictates that we cannot be. Let us draw a diagram of what we see (Fig. 143).

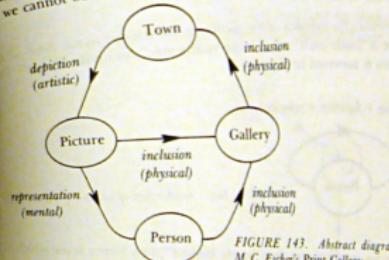


FIGURE 143. Abstract diagram of M.C. Escher's Print Gallery.

What this diagram shows is three kinds of "in-ness". The gallery is *physically* in the town ("inclusion"); the town is *artistically* in the picture ("depiction"); the picture is *mentally* in the person ("representation"). Now while this diagram may seem satisfying, in fact it is arbitrary, for the number of levels shown is quite arbitrary. Look below at another way of representing the top half alone (Fig. 144).

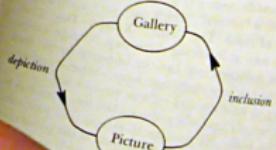


FIGURE 144. A collapsed version of the previous figure.

Hierarchies

715

FIGURE 142. Print Gal...

# Récursion

- Processus de **récursion** découpe en sous-problèmes
  - Dont la structure est la même que le problème original
  - Plus simples à résoudre
- **Décompositions successives** du problème original

*Jusqu'à avoir un sous-problème qui se résout directement*
- Classe des **algorithmes récursifs**

*Algorithme qui exploite la récursion pour résoudre le problème*

# Collecte de fonds (1)

- Politicien doit trouver \$1000 pour sa campagne

*Collecte de \$1 auprès de 1000 supporters*

- Algorithme itératif (boucle)

*On demande successivement \$1 aux 1000 supporters*

```
1 def collect1000():
2     for i in range(1000):
3         # collect $1 from person i
```

# Collecte de fonds (2)

- **Sous-traiter** la recherche d'argent à des intermédiaires

*Trouver 10 personnes qui vont chercher \$100*

- Algorithme **récursif** (récursion)

*On délègue à des intermédiaires qui vont eux-mêmes déléguer...*

```
1 def collect(n):
2     if n == 1:
3         # donne $1 au supérieur
4     else:
5         for i in range(n): # délégation à n personnes
6             collect(n // 10)
7         # donne l'argent récoltée au supérieur
```

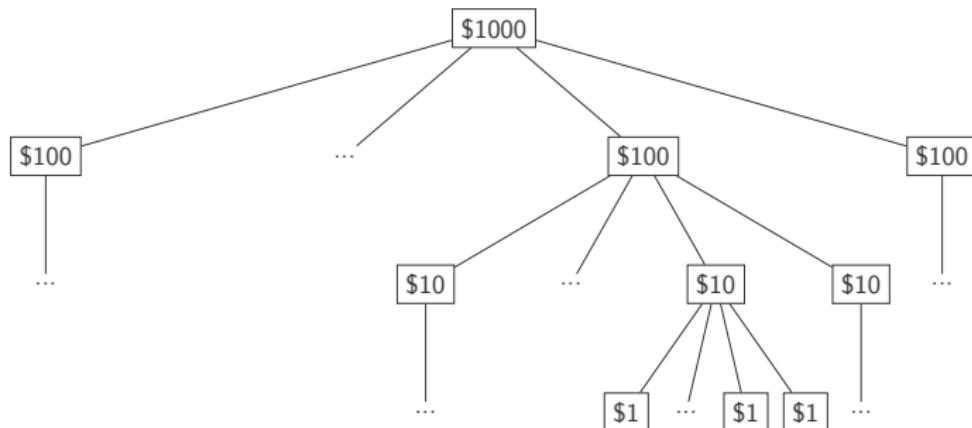
# Concepts

- Stratégie de **diviser pour régner** (*divide-and-conquer*)

*Décomposition du problème original en sous-problèmes*

- **Cas de base** simple et cas récursif à décomposer

*Représentation graphique avec un arbre de solution*



# Caractérisation

- Suivre l'approche de **diviser pour régner**

*Plusieurs instances du même problème plus simples à résoudre*

- Problème **candidat** à une solution récursive

- 1 On peut **décomposer** le problème original en *instances plus simples* du même problème
- 2 Les sous-problèmes doivent finir par *devenir suffisamment simples* que pour être **résolus directement**
- 3 On peut **combiner** les solutions des sous-problèmes pour *produire la solution* du problème original

# Terminaison

- Un processus récursif peut **ne pas se terminer**

*Équivalent des boucles infinies pour les itérations*

- Le **cas de base** doit toujours finir par être atteint
- Collecte de fonds avec **un délégué** pour rechercher les \$1000

*Le processus ne se termine pas, cas de base jamais atteint*

```
1 def collect(n):
2     if n == 1:
3         # donne $1 au supérieur
4     else:
5         collect(n) # délégation à une personne
6         # donne l'argent récoltée au supérieur
```

# Fonction récursive

- Résolution du **problème original et des sous-problèmes générés**

*Les résolutions peuvent se faire avec la même fonction*

- Les **paramètres** permettent d'identifier les sous-problèmes

*Ces paramètres sont utilisés pour gérer les cas de base et récursif*

# Exemple : Factorielle (1)

- Factorielle d'un nombre naturel

$$n! = n \times (n - 1) \times \dots \times 1 \text{ et } 0! \text{ par convention}$$

- Deux visions possibles pour calculer cette fonction

*Vision itérative et vision récursive*

fact(0) =	= 1
fact(1) =	1 = 1
fact(2) =	2 x 1 = 2
fact(3) =	3 x 2 x 1 = 6

fact(0) = 1
fact(1) = 1 x fact(0)
fact(2) = 2 x fact(1)
fact(3) = 3 x fact(2)

# Exemple : Factorielle (2)

## ■ Récursion sur $n$

Cas de base  $0! = 1$

*Factorielle de 0 vaut 1 par convention*

Cas récursif  $n! = n \cdot (n - 1)!$

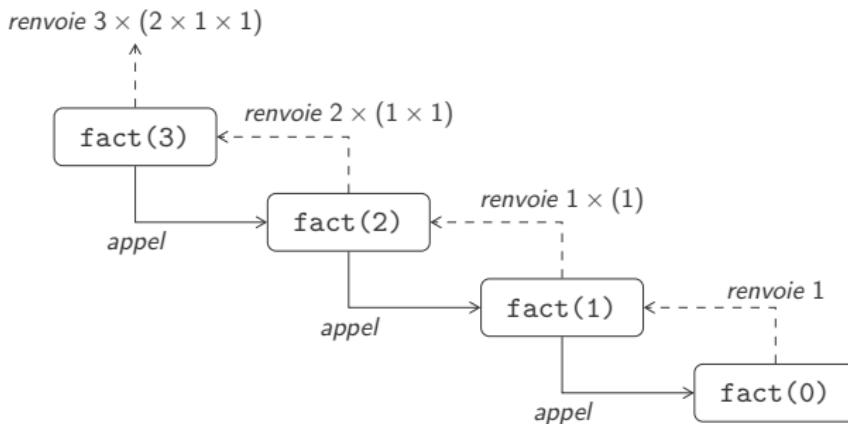
*Factorielle de  $n$  se calcule à partir de celle de  $n - 1$*

```
1 def fact(n):
2     if n == 0:
3         return 1
4     return n * fact(n - 1)
```

# Exemple : Factorielle (3)

- La **fonction fact** est appelée plusieurs fois

*Une récursion produit également une boucle*



# Exemple : Fibonacci

## ■ Récursion sur $n$

Cas de base  $F_1 = 1$  et  $F_2 = 1$

*1<sup>er</sup> et 2<sup>e</sup> nombres de Fibonacci en cas de base*

Cas récursif  $F_n = F_{n-1} + F_{n-2}$

*n<sup>e</sup> nombre de Fibonacci dépend des deux précédents*

```
1 def fibo(n):
2     if n == 1 or n == 2:
3         return 1
4     return fibo(n - 1) + fibo(n - 2)
```

# Exemple : Recherche dans une liste

- Récursion sur la taille de la liste

Cas de base La liste est **vide** ou contient un élément

*On peut rechercher immédiatement l'élément recherché*

Cas récursif Recherche dans la liste sans le premier élément

*La recherche se fait sur une liste plus courte*

```
1 def contains(data, value):
2     if len(data) == 0:
3         return False
4     if data[0] == value:
5         return True
6     return contains(data[1:], value)
```

# Exemple : Tri d'une liste par fusion

- Récursion sur la taille de la liste

Cas de base La liste est **vide** ou contient un élément

*Une telle liste est déjà triée*

Cas récursif Tri séparé de deux listes (on coupe l'originale)

*Ensuite, fusion des deux listes triées*

```
1 def sort(data):
2     def merge(l1, l2):
3         # ...
4         n = len(data)
5         if n <= 1:
6             return data
7         return merge(sort(data[:n//2]), sort(data[n//2:])))
```



Arbre

# Type abstrait de données

- Type abstrait de données (TAD) spécifie mathématiquement
  - Un ensemble de données
  - Les opérations qu'on peut effectuer
- Correspond à un cahier des charges
- Plusieurs implémentations possibles pour un même TAD

*Implémentation du CDC par une structure de données*

*Se différencient par la complexité calculatoire et spatiale*

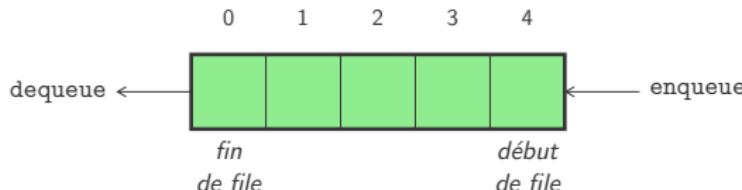
# File

- Séquence de type **First-in First-out (FIFO)**

*Le premier élément qui a été ajouté sera le premier à sortir*

- **Opérations** possibles

size	donne la taille de la file
isEmpty	teste si la file est vide
front	récupère l'élément en début de file
enqueue	ajoute un élément en fin de file
dequeue	retire l'élément en début de file



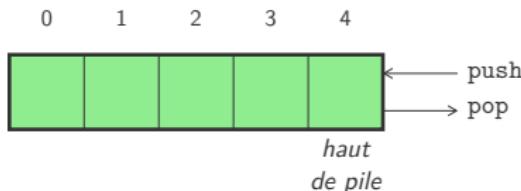
# Pile

- Séquence de type **Last-in First-out (LIFO)**

*Le dernier élément qui a été ajouté sera le premier à sortir*

- **Opérations** possibles

<code>size</code>	donne la taille de la pile
<code>isEmpty</code>	teste si la pile est vide
<code>top</code>	récupère l'élément en haut de la pile
<code>push</code>	ajoute un élément en haut de la pile
<code>pop</code>	retire l'élément en haut de la pile



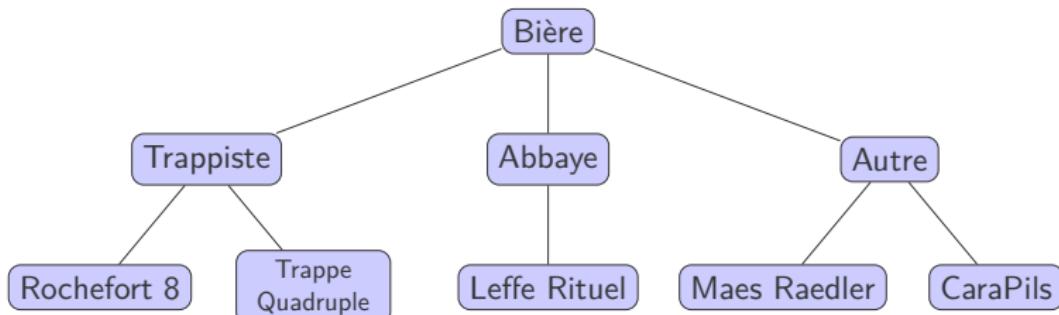
# Arbre

- Éléments d'un **arbre** organisés de manière hiérarchique

*Un arbre est un ensemble de nœuds (qui contiennent les valeurs)*

- Chaque nœud possède un **parent** et zéro ou plusieurs **enfants**

*Sauf la racine de l'arbre qui n'a pas de parent*



# Définition récursive

- **Deux cas** possibles pour définir un arbre
  - Un arbre **vide** (sans enfants)
  - Un nœud avec un élément et une **liste de sous-arbres**
- **Opérations** possibles

size        donne la taille de l'arbre

value        récupère la valeur stockée à la racine de l'arbre

children    récupère la liste des sous-arbres enfants de la racine

addChild    ajoute un sous-arbre comme enfant à la racine

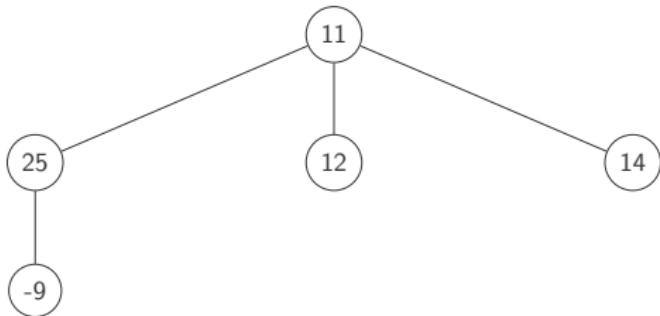
# Classe Tree (1)

- Classe Tree possède **deux variables d'instance**

*Une valeur (racine) et une liste de sous-arbres (enfants)*

```
1 import copy
2
3 class Tree:
4     def __init__(self, value, children=[]):
5         self.__value = value
6         self.__children = copy.deepcopy(children)
7
8     @property
9     def value(self):
10        return self.__value
11
12    @property
13    def children(self):
14        return copy.deepcopy(self.__children)
15
16    def addChild(self, tree):
17        self.__children.append(tree)
18
19    # ...
```

## Classe Tree (2)



```
1 t1 = Tree(-9)
2 t2 = Tree(25, [t1])
3 t3 = Tree(12)
4 t4 = Tree(14)
5
6 t = Tree(11, [t2, t3, t4])
```

# Taille d'un arbre

- Calcul de la **taille d'un arbre** de manière récursive

$1 + \text{somme des tailles des sous-arbres enfants}$

Cas de base Aucun enfant

Cas récursif Appel récursif pour chaque enfant

```
1 # ...
2
3 @property
4 def size(self):
5     result = 1
6     for child in self.__children:
7         result += child.size
8     return result
9
10 # ...
```

# Redéfinition de l'opérateur []

- Définition de l'**opérateur []** pour les objets de type Tree

*Il faut redéfinir la méthode `__getitem__(self, index)`*

```
1 # ...
2
3 def __getitem__(self, index):
4     return self.__children[index]
5
6 # ...
7
8 t = Tree(78, [Tree(14), Tree(9)])
9 for i in range(len(t.children)):
10    print(t[i].value)
```

```
14
9
```

# Représentation textuelle d'un arbre (1)

## ■ Représentation textuelle de manière récursive

*Valeur concaténée avec les représentations de chaque sous-arbre*

Cas de base Aucun enfant

Cas récursif Appel récursif pour chaque enfant

## ■ Utilisation d'une fonction auxiliaire pour gérer le niveau

```
1 # ...
2
3     def __str__(self):
4         def _str(tree, level):
5             result = '[{}]\n'.format(tree.__value__)
6             for child in tree.children:
7                 result += '{}|--{}'.format(' ' * level, _str(
8                     child, level + 1))
9             return result
10        return _str(self, 0)
```

# Représentation textuelle d'un arbre (2)

```
1 c1 = Tree(25, [Tree(-9)])
2 c2 = Tree(12)
3 c3 = Tree(14)
4
5 t = Tree(11, [c1, c2, c3])
6 print(t)
7
8 t[2].addChild(Tree(8))
9 print(t)
```

```
[11]
|--[25]
|  |--[-9]
|--[12]
|--[14]
```

```
[11]
|--[25]
|  |--[-9]
|--[12]
|--[14]
|  |--[8]
```

# Backtracking

- Récursion beaucoup utilisée en **intelligence artificielle**  
*Recherche choix optimal dans un ensemble de possibilités*
- Faire une **tentative** de séquences de choix
  - Possibilité de faire marche arrière par rapport à un choix
  - Exploration de nouvelles décisions
- La récursion permet de faire facilement du **backtracking**

# Backtracking : Labyrinthe

```
1 laby = [
2     "#####E####",
3     "#          E",
4     "# # #####",
5     "# #       #",
6     "# # # ####",
7     "#####      #",
8     "#       # ####",
9     "# # # #   #",
10    "# #       #",
11    "##### ######",
12 ]
13
14 def set(laby, l, c, char):
15     laby[l] = laby[l][:c] + char + laby[l][c+1:]
16
17 def printLaby(laby):
18     for line in laby:
19         print(line)
```

# Backtracking : Labyrinthe

```
1 directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
2
3 # (l, c) is the starting point
4 # return True if an exit is found
5 def find(laby, l, c):
6     # Did we find the exit?
7     if laby[l][c] == 'E':
8         return True
9
10    # Are we in a wall or in our way?
11    if laby[l][c] in '#*':
12        return False
13
14    # Trace our way
15    set(laby, l, c, '*')
16
17    # Search all directions
18    for direction in directions:
19        if find(laby, l+direction[0], c+direction[1]):
20            return True
21
22    # No exit found, backtrack
23    set(laby, l, c, ' ')
24    return False
```

# Lookahead

- **Explorer** un maximum de coups possibles à l'avance

*Sélectionner le coup qui mène à la meilleure situation*

- **Pas toujours possible** d'explorer tous les coups

*Trouver le moins pire étant donné une contrainte temporelle*

- **Deux notions** clés

- L'**état du jeu** représente la situation de ses joueurs
- Un **coup** fait la transition entre deux états

# Jeu de Nim

- Le **jeu de Nim** se joue à deux joueurs
  - Le joueur choisit une rangée et retire autant de pièces qu'il veut
  - Le joueur qui retire la dernière pièce a gagné

Rangée 1



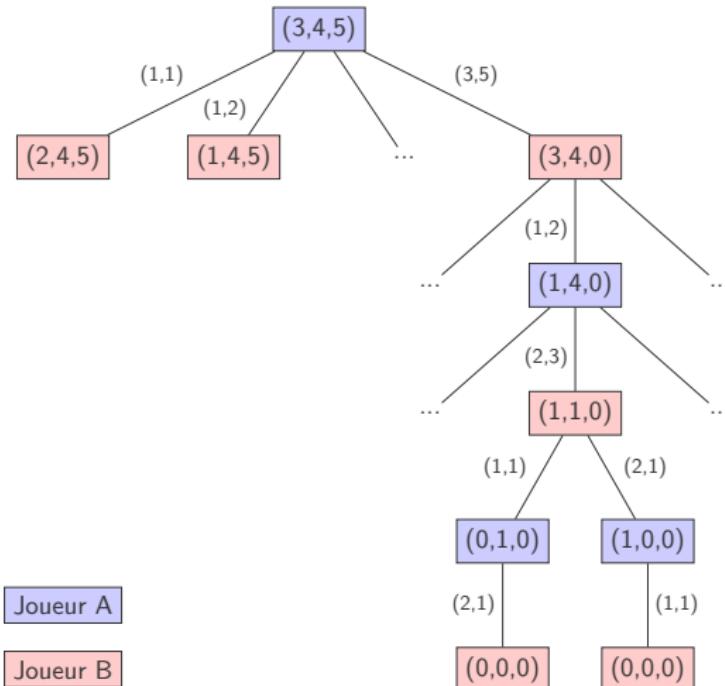
Rangée 2



Rangée 3



# Arbre complet du jeu



# Trouver le meilleur coup (1)

## ■ Deux **fonctions utilitaires**

- Tester si le jeu est **fini** (il ne reste plus de pièces)
- Générer la liste des **coups possibles**

```
1 def isgameover(state):
2     for n in state:
3         if n > 0:
4             return False
5     return True
6
7 def getmoves(state):
8     moves = []
9     for i in range(len(state)):
10        moves += [(i, n) for n in range(1, state[i] + 1)]
11
12 return moves
```

# Trouver le meilleur coup (2)

- Deux **fonctions récursives** (récursion mutuelle)
  - Tester si un état est **mauvais** (mène à une perte du jeu)
  - Trouver un **bon coup** (qui ne mène pas à un mauvais état)

```
1 def isbadposition(state):
2     if isgameover(state):
3         return True
4     return findgoodmove(state) is None
5
6 def findgoodmove(state):
7     for move in getmoves(state):
8         nextstate = tuple(state[i] - move[1] if i == move[0] else
9                           state[i] for i in range(len(state)))
10        if isbadposition(nextstate):
11            return move
12    return None
```

# Problème de recherche



# État

- L'**état** d'un système le décrit à un instant donné

*Typiquement décrit par un ensemble de variables avec leur valeur*

- **Modification de l'état** selon le type d'environnement

*Environnement de type discret ou continu*

- Au départ, le système est dans un **état initial**

# Action

- Une **action** est effectuée sur l'environnement  
*Modification de l'état de l'environnement suite à l'action*
- Ensemble d'**actions possibles** pour chaque état  
*Des actions peuvent être indisponibles dans certains états*
- Définition d'une action par une **fonction successeur**

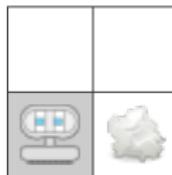


# Robot nettoyeur (1)

- Robot dans une pièce qui doit la nettoyer

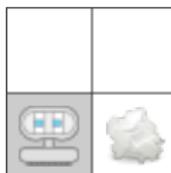
*Démarre de sa base et doit y retourner après avoir tout inspecté*

- Description de l'**état** composée de deux éléments
  - Contenu de chaque case (robot, déchet)
  - Direction courante du robot



# Robot nettoyeur (2)

- Quatre actions possibles
  - move **avance** d'une case dans la direction courante
  - left **tourne** sur lui-même de 90 degrés vers la gauche
  - right **tourne** sur lui-même de 90 degrés vers la droite
  - clean **nettoie** la case sur laquelle il se trouve
- L'**objectif** est d'avoir nettoyé toutes les cases
  - Et d'éventuellement être revenu à la case de départ*



# Arbre d'exécution

- L'arbre d'exécution reprend toutes les exécutions possibles

*Cet arbre peut éventuellement être infini*

- État dans les nœuds et actions sur les arêtes

*Nombre maximum de fils correspond au nombre d'actions*

- Un chemin dans l'arbre représente une exécution donnée

*L'arbre représente donc bien toutes les exécutions*

# Espace d'états

- Représentation compacte sous forme d'un **graphe**

*On ne duplique plus les états égaux*

- **Espace d'états** complètement défini par

- l'état initial
- et la fonction successeur

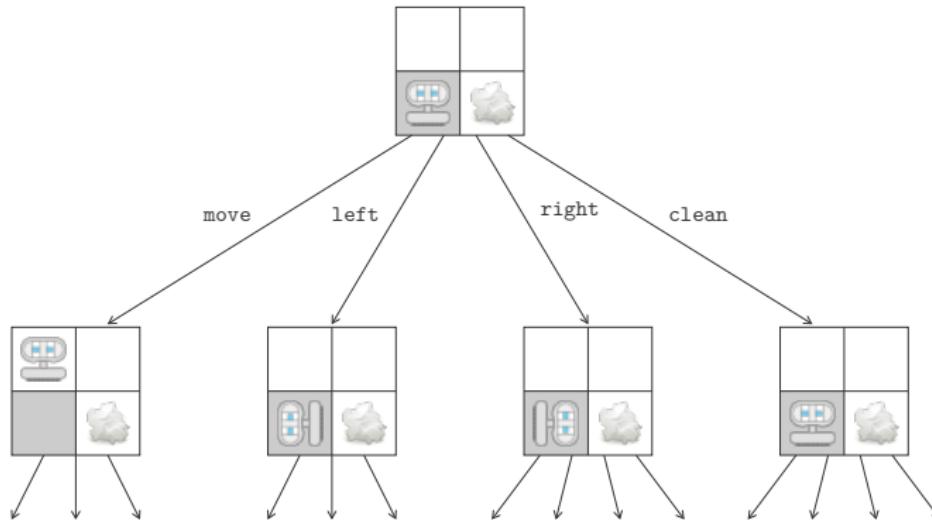
- Un **chemin** dans le graphe représente une exécution donnée

*Une boucle indique une exécution infinie possible*

# Robot nettoyeur (3)

- Ensemble d'actions possibles dépend de l'état

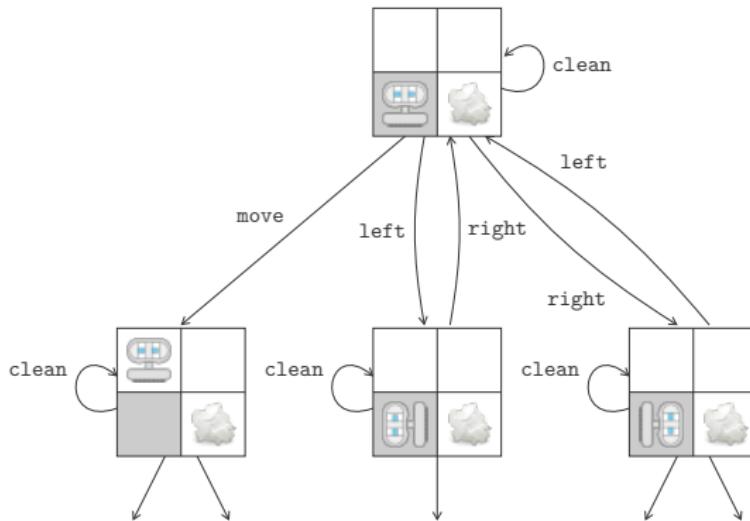
*Le robot ne peut pas avancer une fois au bord du terrain*



# Robot nettoyeur (4)

- Les **états équivalents** de l'arbre d'exécution sont fusionnés

*Forme beaucoup plus compacte avec 32 états en tout*



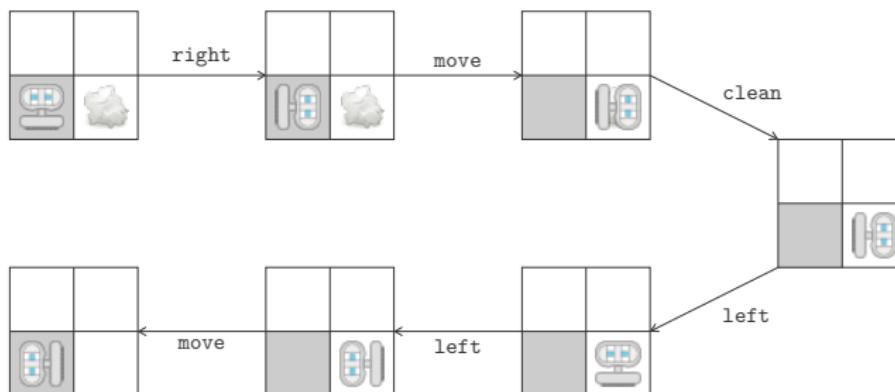
# Cout et objectif

- Possibilité d'ajouter un **cout** pour les actions  
*Cout pour effectuer une action qui mène d'un état à un autre*
- **Minimiser cout** du chemin d'exécution de la solution  
*Différence entre solution et solution optimale*
- L'**objectif** est l'ensemble des états à atteindre  
*Une solution est un chemin de l'état initial à un objectif*

# Robot nettoyeur (5)

- La **solution optimale** pour le robot fait six actions

*Cette solution n'est pas unique*



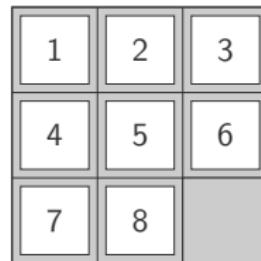
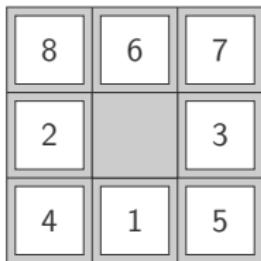
# Exemple : 8-puzzle

- Plateau de jeu  $3 \times 3$  avec 8 blocs numérotés de 1 à 8

*Le dernier bloc est le blanc (ou le trou)*

- Les actions consistent à déplacer un bloc numéroté vers le trou

*Le but est de réordonner tous les blocs numérotés*



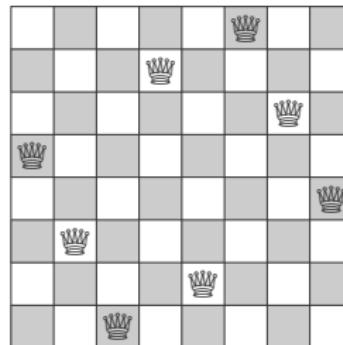
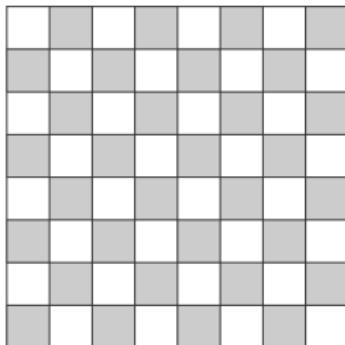
# Exemple : 8-queens

- Plateau de jeu  $8 \times 8$  de type échecs

*Les règles du jeu pour la reine s'appliquent*

- Les actions consistent à **placer une reine** sur une case libre

*Le but est de placer 8 reines sans qu'aucune soit en danger*



# Algorithme de recherche



# Mesure de performance

- Quatre critères pour évaluer les **performances d'un algorithme**
  - **Complétude** : l'algorithme trouve-t-il toujours une solution ?
  - **Optimalité** : la solution trouvée est-elle la meilleure ?
  - **Complexité temporelle** : temps pour trouver la solution
  - **Complexité spatiale** : mémoire pour trouver la solution
- Selon la situation, certains critères seront **ignorés**

*Parfois, l'algorithme boucle et ne termine donc jamais*

# Recherche non informée

- Recherche **non informée** ou recherche aveugle
  - Exploration complète de l'espace d'états*
- Recherche basée uniquement sur la **définition du problème**
  - Génération d'états avec la fonction successeur
  - Test de si un état fait partie de l'objectif ou non
- Distinction entre algorithmes selon l'**ordre d'exploration**

*Peut conduire à atteindre plus ou moins vite une solution*

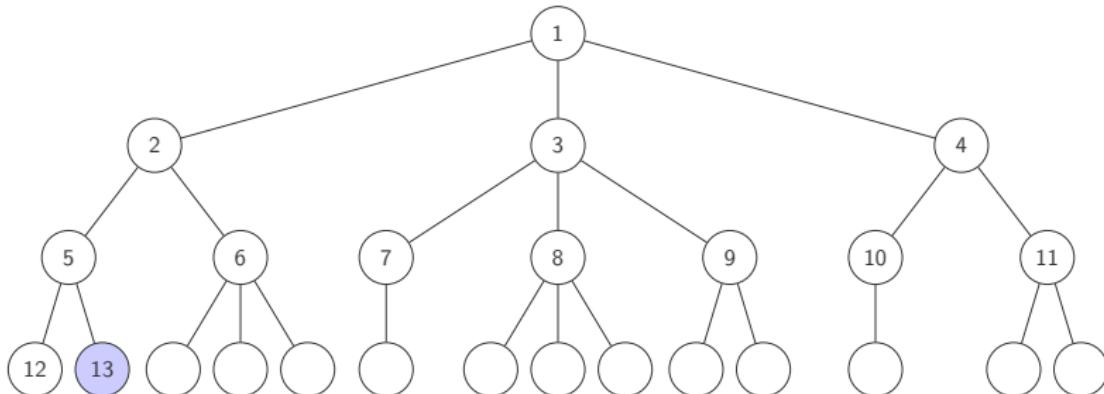
# Breadth-First Search

- Exploration successives des successeurs **en largeur**

*D'abord les successeurs d'un nœuds avant leurs successeurs...*

- Exploration de l'arbre d'exécution **par niveaux**

*Algorithme complet, mais pas forcément optimal*



# Breadth-First Search (2)

- Complexité temporelle de l'algorithme en  $\mathcal{O}(b^{d+1})$ 
  - $b$  facteur de branchement (nombre maximum de fils)
  - $d$  la profondeur d'une solution (de la moins profonde)

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b)$$

- Même complexité spatiale

*Peut très vite être problématique pour les grosses instances*

- Optimal lorsque tous les couts sont identiques

*Exploration du nœud non exploré le moins profond*

# Uniform-Cost Search

- Exploration via l'action qui a le **cout le plus faible**

*Permet d'explorer d'abord le chemin de cout total minimal*

- **Complétude et optimalité** assurées si  $c(q, a) > \varepsilon$  pour  $\varepsilon > 0$

*Noeuds parcourus en ordre croissant du cout du chemin associé*

- Complexité **temporelle et spatiale** en  $\mathcal{O}(b^{1+\lfloor C^*/\varepsilon \rfloor})$

*Avec  $C^*$  le cout optimal et  $\varepsilon > 0$  (souvent plus grand que  $b^d$ )*

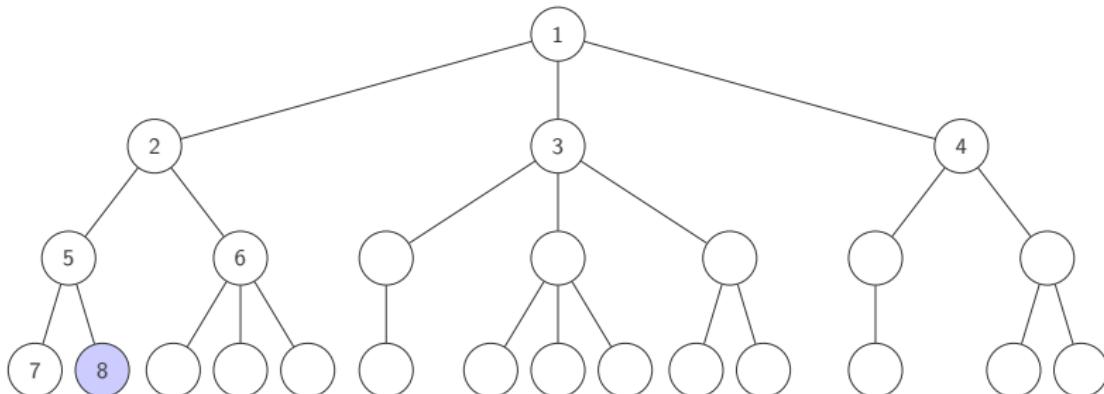
# Depth-First Search (1)

- Exploration d'abord **en profondeur**

*D'abord explorer le nœud non exploré le plus profond*

- Descente jusqu'à une **feuille de l'arbre**

*Pas complet (peut être coincé dans une boucle), ni optimal*



# Depth-First Search (2)

- **Complexité temporelle** de l'algorithme en  $\mathcal{O}(b^m)$ 
  - $b$  facteur de branchement (nombre maximum de fils)
  - $m$  la profondeur maximale dans l'arbre
- $$b + b^2 + b^3 + \dots + b^m$$
- **Complexité spatiale** de l'algorithme en  $\mathcal{O}(bm + 1)$ 

*La variante recherche backtracking ne nécessite que  $\mathcal{O}(m)$*

# Depth-Limited Search

- Ajout d'une **profondeur maximale** d'exploration  $\ell$ 
  - Pas complet si  $\ell < d$  (solution hors de portée)
  - Pas optimal si  $\ell > d$  (peut rater la solution la moins profonde)

- **Complexité** temporelle en  $\mathcal{O}(b^\ell)$  et spatiale en  $\mathcal{O}(b\ell)$

*Cas particulier de Depth-First Search avec  $\ell = \infty$*

- Deux **types d'échecs** différents
  - **Failure** lorsque pas de solutions
  - **Cutoff** lorsque pas de solutions dans la limite  $\ell$

# Iterative Deepening Depth-First Search

- Augmentation progressive de la profondeur maximale

*Depth-Limited Search avec successivement  $\ell = 0, \ell = 1\dots$*

- Solution optimale trouvée lorsque  $\ell = d$

*Algorithme complet et optimal*

- Complexité temporelle en  $\mathcal{O}(b^d)$  et spatiale en  $\mathcal{O}(bd)$

---

### Algorithm 3: Iterative-Deepening-Search

---

```
Function IDS(problem)
    for depth  $\leftarrow 0$  to  $\infty$  do
        result  $\leftarrow$  DLS(problem, depth)
        if result  $\neq$  cutoff then
            return result
```

---

# Bidirectional Search

- Effectuer **deux recherches** en parallèle  
*Une depuis l'état initial et une depuis le(s) objectif(s)*
- Nécessite que la **fonction prédecesseur** soit disponible  
*Facile si les actions sont réversibles*
- Solution trouvée lorsque les deux **recherches se croisent**

# Recherche informée

- Recherche **informée** utilise des connaissances spécifiques

*Beaucoup plus efficace que les recherches non informées*

- Stratégie générale de type **Best-First Search**

*Choix du meilleur nœud à explorer à chaque étape*

- **Fonction d'évaluation**  $f(n)$

*Choix du nœud à explorer avec la plus faible valeur*

- **Fonction heuristique**  $h(n)$

*Cout estimé du chemin le moins cher vers l'objectif*

# Greedy Best-First Search

- Choix du nœud **le plus proche** de l'objectif

*En utilisant  $f(n) = h(n)$*

- L'**heuristique** est choisie en fonction du problème

*Souvent une mesure de distance vers l'objectif*

- **Similaire** à Depth-First Search (exploration d'un chemin)

*Pas complet et pas optimal*

- **Complexité** temporelle en  $\mathcal{O}(b^m)$  et spatiale en  $\mathcal{O}(b^m)$

*Réduit en fonction du problème et de la qualité de l'heuristique*

# $A^*$ Search

- $A^*$  (prononcé « A-star ») combine deux fonctions
  - $g(n)$  donne le cout d'avoir atteint  $n$
  - $h(n)$  heuristique du cout pour atteindre l'objectif depuis  $n$
- Fonction d'évaluation  $f(n) = g(n) + h(n)$ 

*Cout estimé pour atteindre l'objectif en passant par n*
- Complet et optimal si  $h(n)$  est **admissible**

*$h(n)$  ne surestime jamais le cout pour atteindre l'objectif*

# Adversarial Search

- Recherche de solution pour des jeux avec **deux adversaires**

*Deux joueurs appelés MIN et MAX (joue en premier)*

- Définition comme un **problème de recherche**

- **État initial** : position sur le plateau et joueur qui commence
- **Fonction successeur** : liste de paires (*move, state*)
- **Test terminal** : teste si le jeu est terminé (état terminaux)
- **Fonction d'utilité** : donne une valeur aux états terminaux

- **Arbre du jeu** défini par l'état initial et les mouvements légaux

# Stratégie optimale

- MAX veut **atteindre** un état gagnant (terminal)

*Tout en sachant que MIN a son mot à dire*

- Proposition d'une **stratégie** pour MAX qui définit
  - Le mouvement dans l'état initial
  - Le mouvement suite à chaque mouvement possible de MIN
  - ...
- Jouer **le meilleur coup possible** à chaque tour

*En supposant que le joueur en face suit une stratégie parfaite*

# Caractéristiques des jeux (1)

- Jeu de **somme nulle** lorsque la somme des gains vaut 0  
*Le gain de l'un correspond obligatoirement à une perte de l'autre*
- Jeu avec **information parfaite** pour les joueurs  
*Toute l'information du plateau est accessible aux deux joueurs*
- Pas de **chance** impliquée dans le jeu
- **Valeur** d'un état terminal  
*Évaluation de la situation pour un joueur donné*

# Caractéristiques des jeux (2)

	Déterministe	Chance
<b>Information parfaite</b>	Échecs Dames Go Othello	Backgammon Monopoly
<b>Information imparfaite</b>	Stratego	Bridge Poker Scrabble Nuclear War

# Valeur minimax

- Fonction *MinimaxValue* associe une valeur à chaque nœud  $n$

*Définition récursive de cette fonction*

- Hypothèse que les deux joueurs *jouent parfaitement*

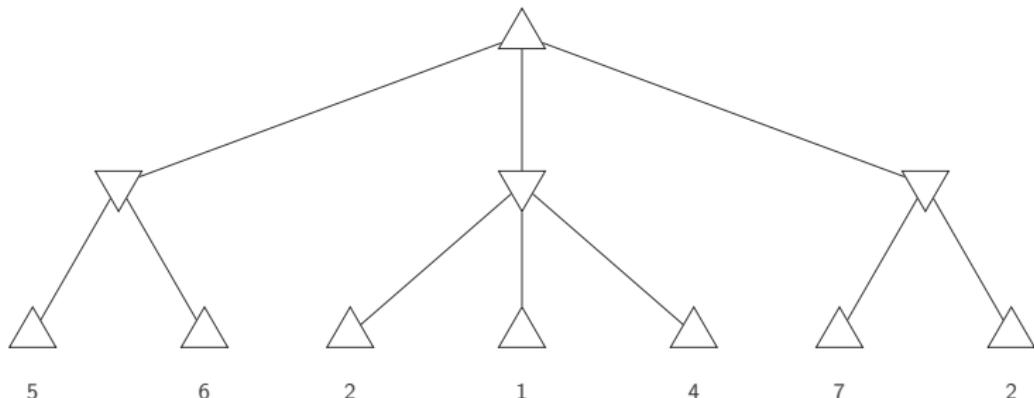
- MAX préfère aller vers une situation de plus grande valeur
- et MIN de plus petite valeur

$$\text{MinimaxValue}(n) = \begin{cases} \text{Utility}(n) & \text{si } n \text{ est un nœud terminal} \\ \max_{s \in \text{Successors}(n)} \text{MinimaxValue}(s) & \text{si } n \text{ est un nœud MAX} \\ \min_{s \in \text{Successors}(n)} \text{MinimaxValue}(s) & \text{si } n \text{ est un nœud MIN} \end{cases}$$

# Algorithme minimax

- Arbre du jeu avec  $\Delta$  pour MAX et  $\nabla$  pour MIN

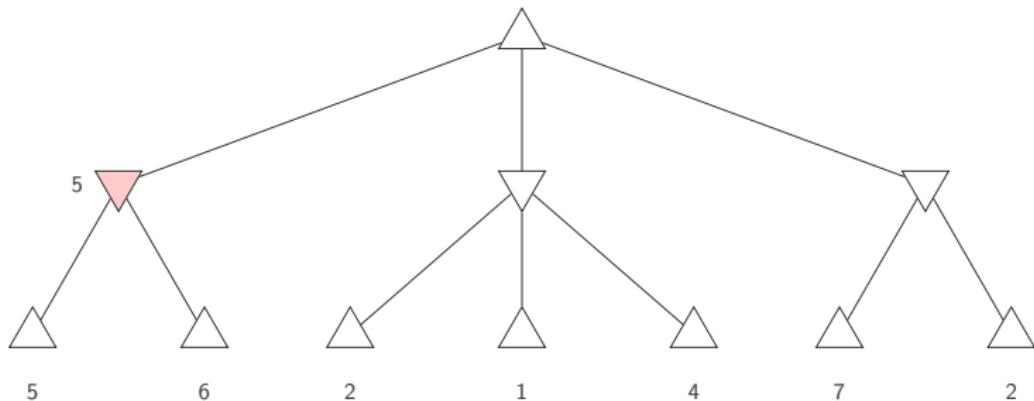
*MAX choisit toujours le coup qui maximise la valeur minimax*



# Algorithme minimax

- Arbre du jeu avec  $\Delta$  pour MAX et  $\nabla$  pour MIN

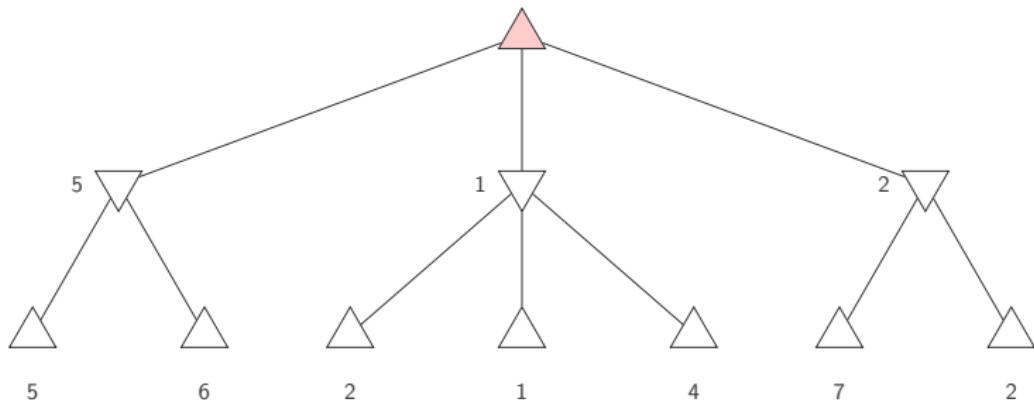
*MAX choisit toujours le coup qui maximise la valeur minimax*



# Algorithme minimax

- Arbre du jeu avec  $\Delta$  pour MAX et  $\nabla$  pour MIN

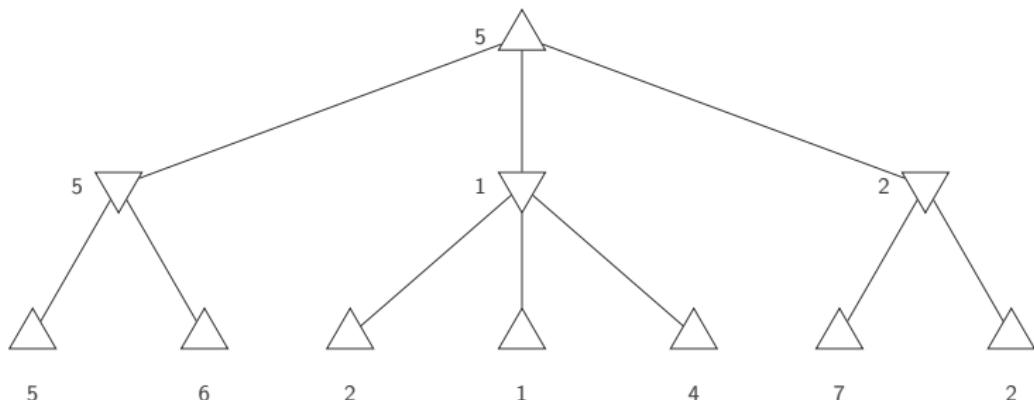
*MAX choisit toujours le coup qui maximise la valeur minimax*



# Algorithme minimax

- Arbre du jeu avec  $\Delta$  pour MAX et  $\nabla$  pour MIN

*MAX choisit toujours le coup qui maximise la valeur minimax*

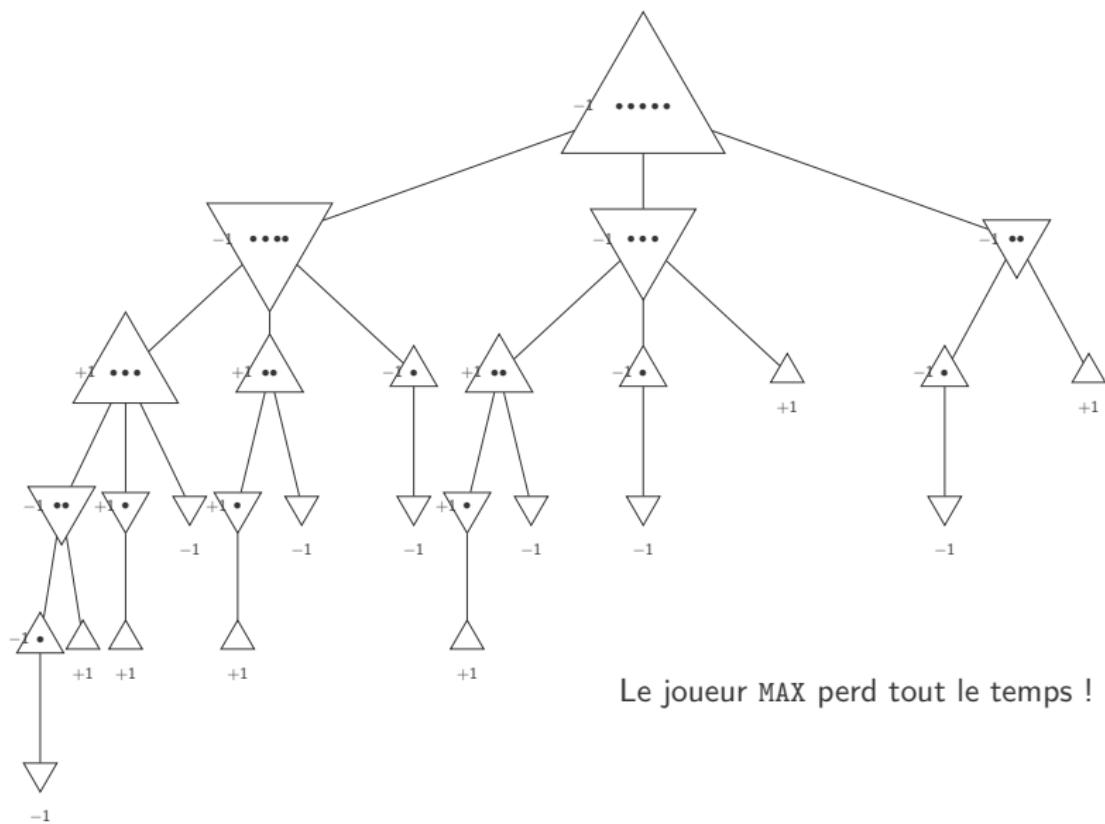


# Jeu des bâtonnets (1)

- À chaque tour, le joueur **retire** 1, 2 ou 3 bâtonnets  
*Le joueur qui retire le dernier bâtonnet a perdu*



## Jeu des bâtonnets (2)



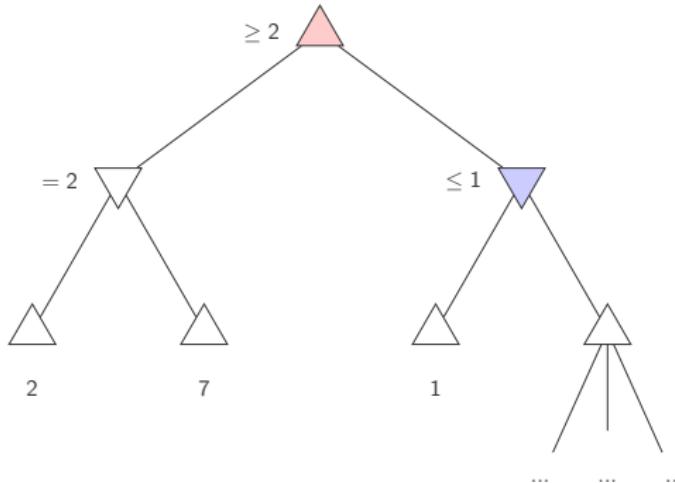
Le joueur MAX perd tout le temps !

# Alpha-Beta pruning (1)

- Éviter d'explorer un sous-arbre lorsque ce n'est pas nécessaire  
*On ne sait pas faire mieux que la valeur minimax actuelle*
- Deux situations de simplification possibles
  - $\alpha$  plus grande borne inférieure pour MAX
  - $\beta$  plus petite borne supérieure pour MIN
- Mémorisation des bornes durant l'exploration de l'arbre

# Alpha-Beta pruning (2)

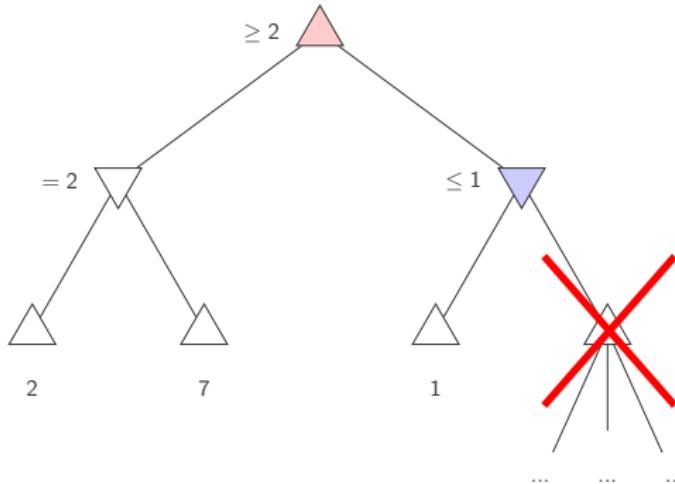
- On peut **raisonner** sur la situation actuelle de l'exploration
  - La racine sera  $\geq 2$  puisque c'est un MAX
  - Le fils droit de la racine sera  $\leq 1$  car c'est un MIN



Exemple animé : <http://alphabeta.alekskamko.com/>

# Alpha-Beta pruning (2)

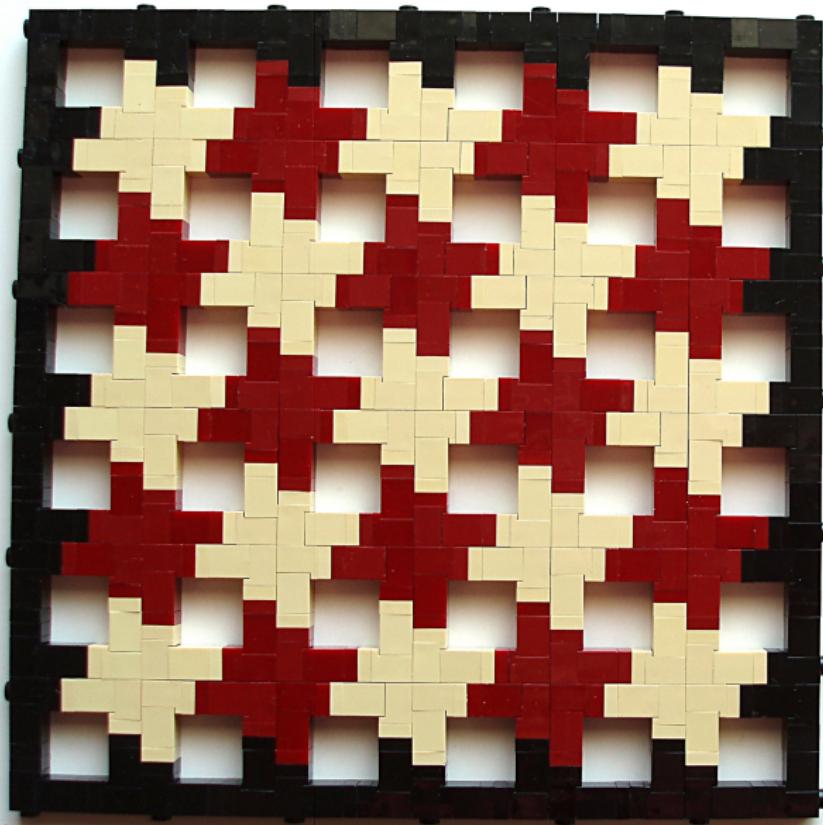
- On peut **raisonner** sur la situation actuelle de l'exploration
  - La racine sera  $\geq 2$  puisque c'est un MAX
  - Le fils droit de la racine sera  $\leq 1$  car c'est un MIN



Exemple animé : <http://alphabeta.alekskamko.com/>

# Alpha-Beta pruning (3)

- Mémorisation de **deux bornes** par nœud
  - $\alpha(n)$  plus grande valeur trouvée actuellement ( $-\infty$  au départ)
  - $\beta(n)$  plus petite valeur trouvée actuellement ( $+\infty$  au départ)
- Deux situations de **pruning**
  - **Beta cutoff** sur nœud MAX  $n$   
si  $\alpha(n) \geq \beta(i)$  pour  $i$  ancêtre MIN de  $n$
  - **Alpha cutoff** sur nœud MIN  $n$   
si  $\beta(n) \leq \alpha(i)$  pour  $i$  ancêtre MAX de  $n$



Framework

# Simple AI (1)

- Librairie d'implémentation de plusieurs **algorithmes d'IA**
  - Algorithmes de recherche
  - Apprentissage automatique (machine learning)
- Librairie **open-source** disponible sur GitHub

*<https://github.com/simpleai-team/simpleai>*

# Simple AI (2)

- Bugs avec la librairie lorsqu'on utilise **Python 3**

*Utilisation d'un **fork** : <https://github.com/combefis/simpleai>*

- Algorithme de **recherche d'une librairie** par Python

- Recherche dans le même dossier
- Recherche dans les dossiers de la variable PYTHONPATH
- Recherche dans le dossier système

```
$ echo $PYTHONPATH  
$ export PYTHONPATH=/Users/combefis/Documents/Projects/simpleai  
$ echo $PYTHONPATH  
/Users/combefis/Documents/Projects/simpleai
```

# Définition d'un problème (1)

- Création d'une nouvelle classe de **type SearchProblem**

*Représentation du problème de recherche*

- Plusieurs **méthodes à définir** dans la classe
  - `actions` définit les actions possibles dans un état
  - `result` calcule le nouvel état suite à une action effectuée
  - `is_goal` teste si un état correspond à l'objectif

# Définition d'un problème (2)

- Exemple d'un problème de recherche d'un mot

*Les états sont des mots, les actions sont le choix d'une lettre*

```
1 from simpleai.search import SearchProblem
2
3 GOAL = 'HELLO'
4
5 class HelloProblem(SearchProblem):
6     def actions(self, state):
7         if len(state) < len(GOAL):
8             return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
9         return []
10
11     def result(self, state, action):
12         return state + action
13
14     def is_goal(self, state):
15         return state == GOAL
```

# Exécution de la recherche

- Création d'une **instance du problème**

*Instance de la nouvelle classe définie pour le problème*

- Choix d'un **algorithme de recherche** et lancement de celle-ci

*breadth\_first, depth\_first, greedy...*

```
1 from simpleai.search import breadth_first
2
3 problem = HelloProblem(initial_state=' ')
4 result = breadth_first(problem)
5 print(result.state) # État atteint
6 print(result.path()) # Chemin d'exécution pour atteindre l'objectif
```

```
HELLO
[('None', ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'),
 , ('O', 'HELLO')]
```

# Heuristique (1)

- Évaluation de la distance avec l'objectif à atteindre

*On compte le nombre de lettres différentes et manquantes*

- Ajout d'une méthode **heuristic** dans la classe du problème

*Va diriger le choix des actions vers la solution*

```
1 from simpleai.search import SearchProblem
2
3 GOAL = 'HELLO'
4
5 class HelloProblem(SearchProblem):
6     ...
7     def heuristic(self, state):
8         wrong = sum([1 if state[i] != GOAL[i] else 0 for i in range
9                     (len(state))])
10        missing = len(GOAL) - len(state)
11        return wrong + missing
```

# Heuristique (2)

- À chaque état, c'est la **meilleure lettre** qui est choisie

*Et au vu de l'heuristique, c'est à chaque fois la bonne*

- Importance du **choix de l'heuristique**
- Mesure du **temps d'exécution** pour résoudre le problème

*MacBookPro 2.9 GHz Intel Core i7, 8 Go 1600 MHz DDR3*

---

Breadth-First Search	Greedy Best-First Search
96.65512 s	0.00274 s

---

- Framework d'IA pour des **jeux à deux joueurs**  
*Negamax avec alpha-beta pruning et table de transposition*
- Framework **open-source** disponible sur GitHub  
*<https://github.com/Zulko/easyAI>*

# Définition d'un problème (1)

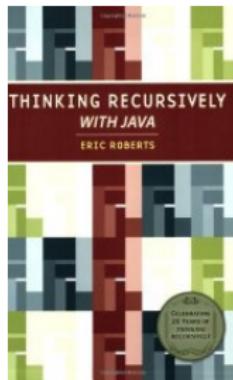
```
1  from easyAI import TwoPlayersGame, Human_Player, AI_Player, Negamax
2
3  class SimpleNim(TwoPlayersGame):
4      def __init__(self, players):
5          self.players = players
6          self.nplayer = 1
7          self.__sticks = 5
8
9      def possible_moves(self):
10         return [str(v) for v in (1, 2, 3) if v <= self.__sticks]
11
12     def make_move(self, move):
13         self.__sticks -= int(move)
14
15     def win(self):
16         return self.__sticks <= 0
17
18     def is_over(self):
19         return self.win()
20
21     def show(self):
22         print('{} sticks left in the pile'.format(self.__sticks))
23
24     def scoring(self):
25         return 1 if self.win() else 0
```

## Définition d'un problème (2)

```
1 ai = Negamax(13)
2 game = SimpleNim([Human_Player(), AI_Player(ai)])
3 history = game.play()
```

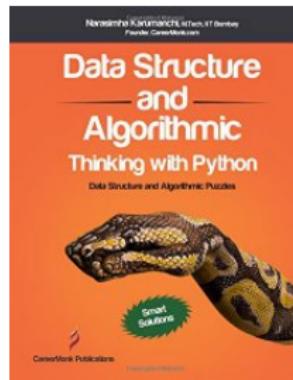
```
5 sticks left in the pile
Player 1 what do you play ? 3
Move #1: player 1 plays 3 :
2 sticks left in the pile
Move #2: player 2 plays 1 :
1 sticks left in the pile
Player 1 what do you play ? 1
Move #3: player 1 plays 1 :
0 sticks left in the pile
```

# Livres de référence



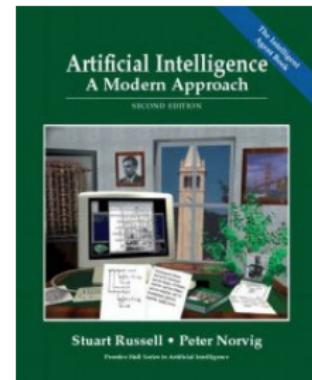
ISBN

978-0-471-70146-0



ISBN

978-8-192-10759-2



ISBN

978-0-130-80302-3

# Crédits

- Photos des livres depuis Amazon
- <https://www.flickr.com/photos/juhansonin/8331686714>
- <https://www.flickr.com/photos/gadl/279433682>
- <https://www.flickr.com/photos/127497725@N02/16695848708>
- [https://fr.wikipedia.org/wiki/Fichier:Kempelen\\_chess1.jpg](https://fr.wikipedia.org/wiki/Fichier:Kempelen_chess1.jpg)
- <http://static-numista.com/catalogue/photos/belgique/g2442.jpg>
- <https://www.flickr.com/photos/gsfc/6800387182>
- <https://openclipart.org/detail/168755/cartoon-robot>
- <https://www.flickr.com/photos/grahamvphoto/17479235311>
- <https://www.flickr.com/photos/eilonwy77/9185293834>