ECAM
Brussels Engineering School

PI2T Développement informatique

Séance 7

Algorithmique II : Techniques de recherche en intelligence artificielle

Sébastien Combéfis, Quentin Lurkin

14 mars 2016

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


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

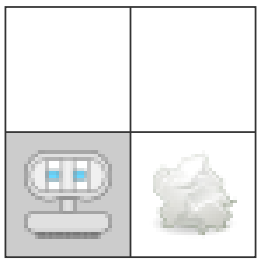
- Définition d'une action par une fonction successeur

Robot nettoyeur (1)

- Robot dans une pièce qui doit la nettoyer

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

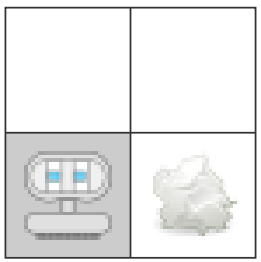
- Description de l'état composée de deux éléments
  - Contenu de chaque case (robot, déchet)
  - Direction courante du robot



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

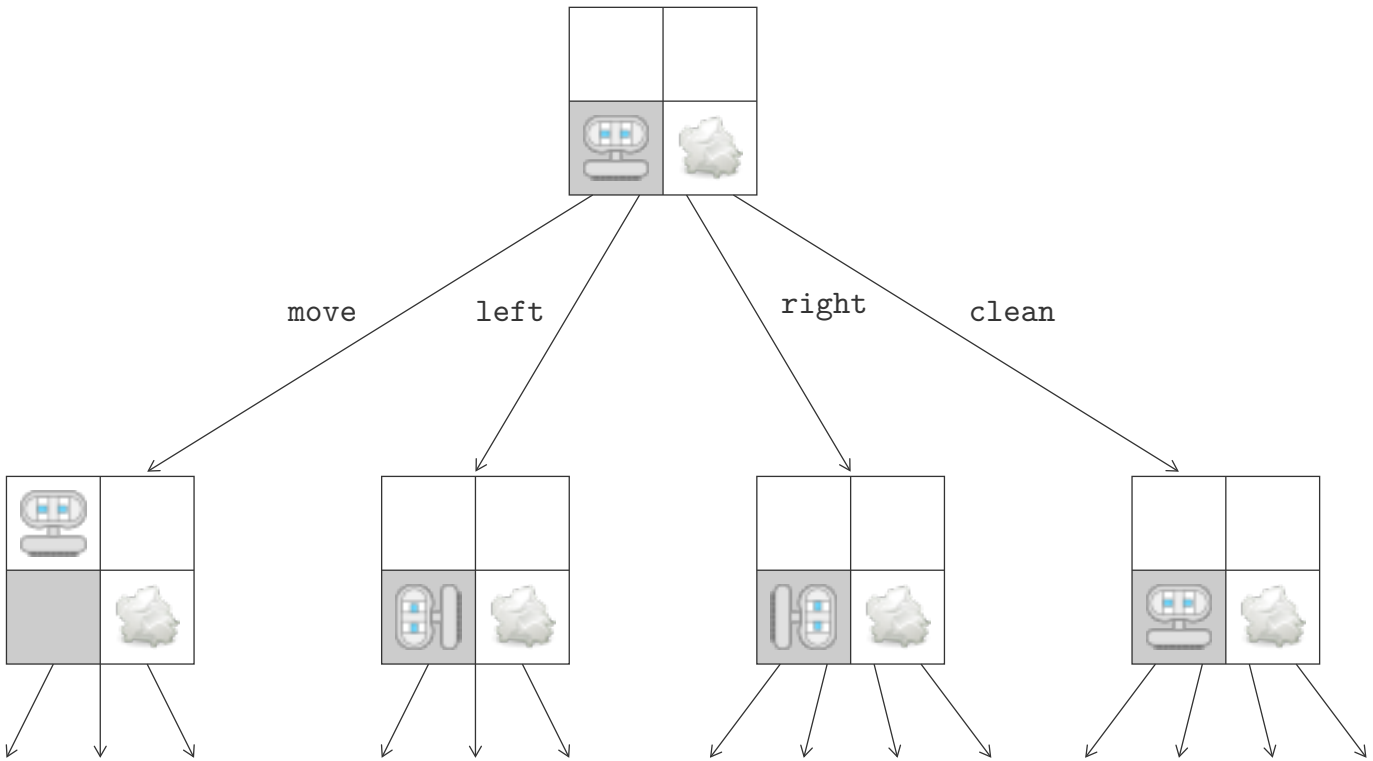
- Un chemin dans le graphe représente une exécution donnée

  Une boucle indique une exécution infinie possible

Robot nettoyeur (3)

- Ensemble d'actions possibles dépend de l'état

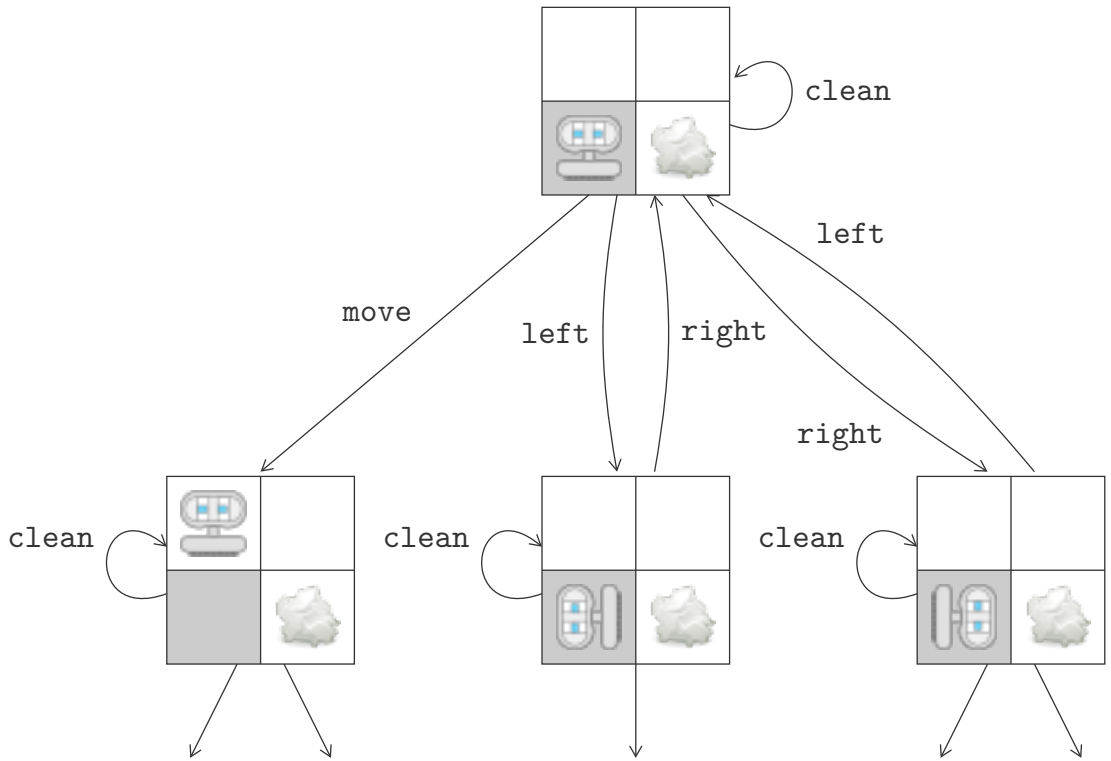
  Le robot ne peut pas avancer une fois au bord du terrain



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

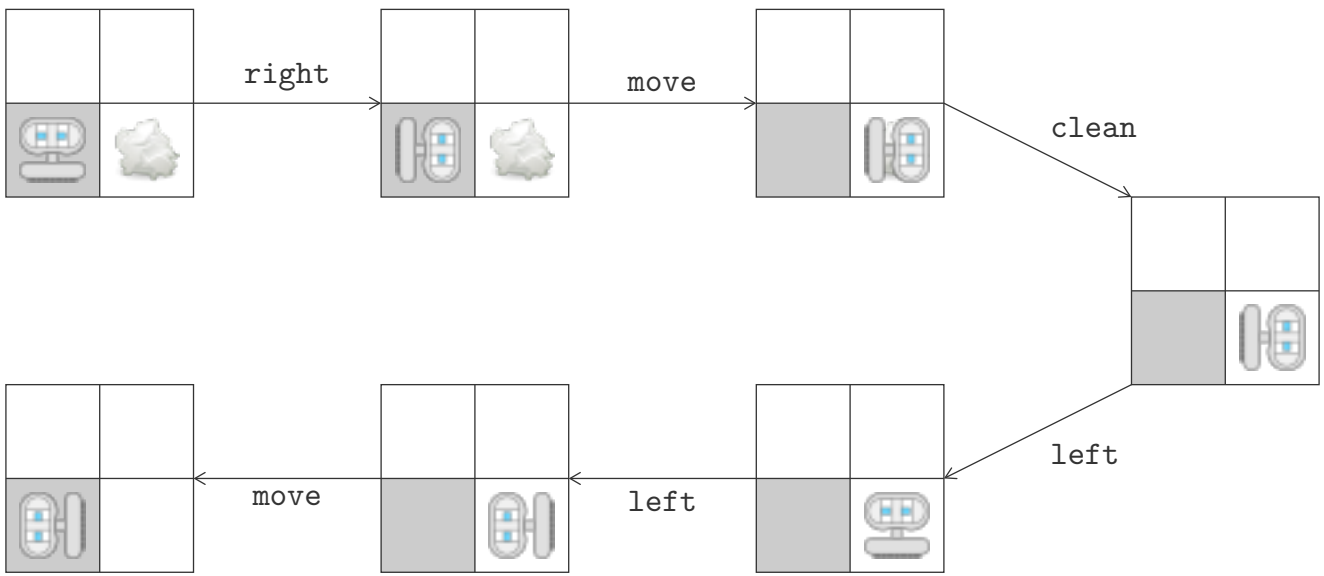
- L'objectif est l'ensemble des états à atteindre

  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



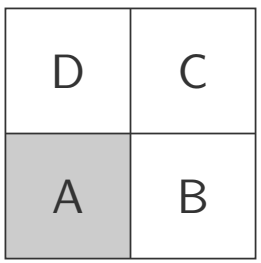
Formalisation

- Formalisation mathématique d'un problème de recherche
  - Ensembles des états et des actions : Q et A
  - État initial : $q_0 \in Q$
  - Fonction successeur : $succ : Q \times A \to Q$
  - Fonction de cout : $c : Q \times A \to \mathbb{R}$
  - Objectif : $G \subseteq Q$
- Permet de raisonner sur les problèmes de recherche

  Raisonnement abstrait applicable sur des instances

Formalisation du robot nettoyeur (1)

- Définition formelle de l'exemple du robot nettoyeur
  - $Q = \{A, B, C, D\} \times \{N, S, W, E\} \times 2^{\{A,B,C,D\}}$ et $A = \{move, left, right, clean\}$
  - $q_0 = (A, N, \{B\})$
  - $succ(q, a) = \ldots \quad (\forall q \in Q, a \in A)$
  - $c(q, a) = 1 \quad (\forall q \in Q, a \in A)$
  - $G = \{(A, N, \emptyset), (A, S, \emptyset), (A, W, \emptyset), (A, E, \emptyset)\}$



Formalisation du robot nettoyeur (2)

- Définition de la fonction successeur

Au maximum 4 *actions pour chacun des* 32 *états différents*

(q_s, a)	q_e
$((A, N, \{B\}), move)$	$(D, N, \{B\})$
$((A, N, \{B\}), left)$	$(A, W, \{B\})$
$((A, N, \{B\}), right)$	$(A, E, \{B\})$
$((A, N, \{B\}), clean)$	$(A, N, \{B\})$
$((A, S, \{B\}), left)$	$(A, E, \{B\})$
$((A, S, \{B\}), right)$	$(A, W, \{B\})$
$((A, S, \{B\}), clean)$	$(A, S, \{B\})$

(q_s, a)	q_e
$((A, W, \{B\}), left)$	$(A, S, \{B\})$
$((A, W, \{B\}), right)$	$(A, N, \{B\})$
$((A, W, \{B\}), clean)$	
$((A, E, \{B\}), move)$	$(B, E, \{B\})$
$((A, E, \{B\}), left)$	$(A, N, \{B\})$
$((A, E, \{B\}), right)$	$(A, S, \{B\})$
$((A, E, \{B\}), clean)$	$(A, E, \{B\})$

...etc.

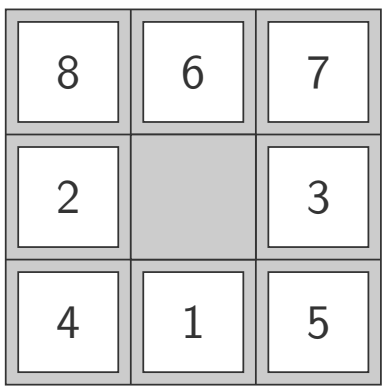
Exemple : *8-puzzle*

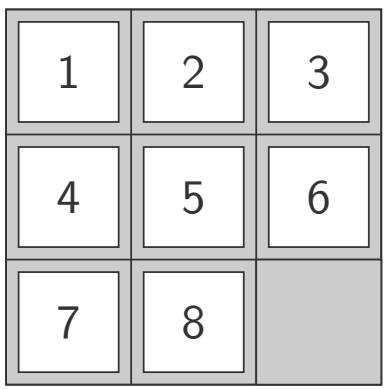
- Plateau de jeu 3×3 avec 8 blocs numérotés de 1 à 8

  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





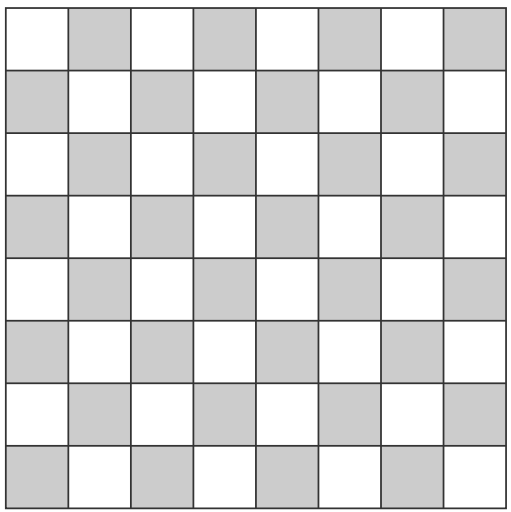
Exemple : *8-queens*

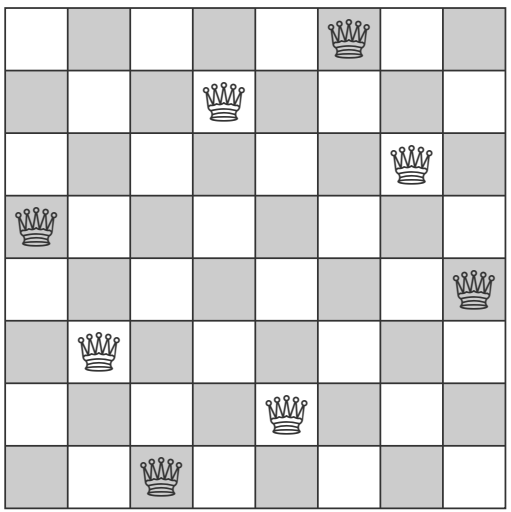
- Plateau de jeu 8×8 de type échecs

  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

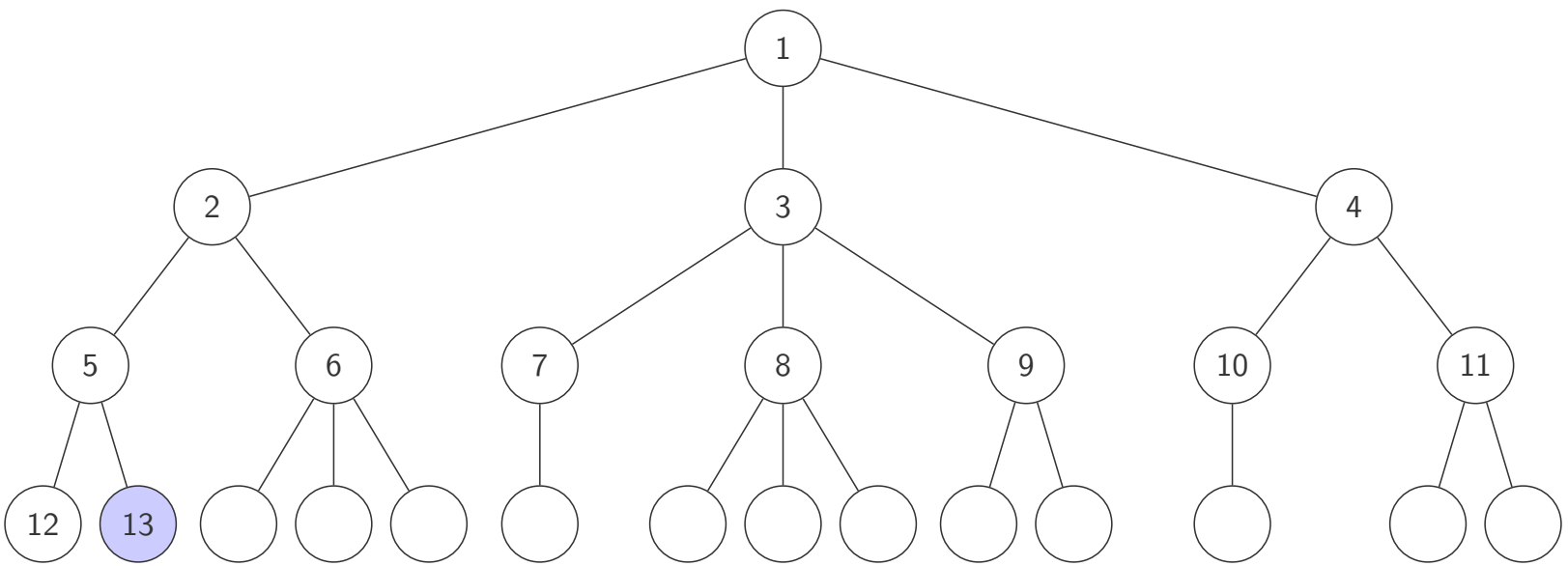
Breadth-First Search (1)

- 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)

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

- 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 $\varepsilon > 0$

  Nœuds 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)

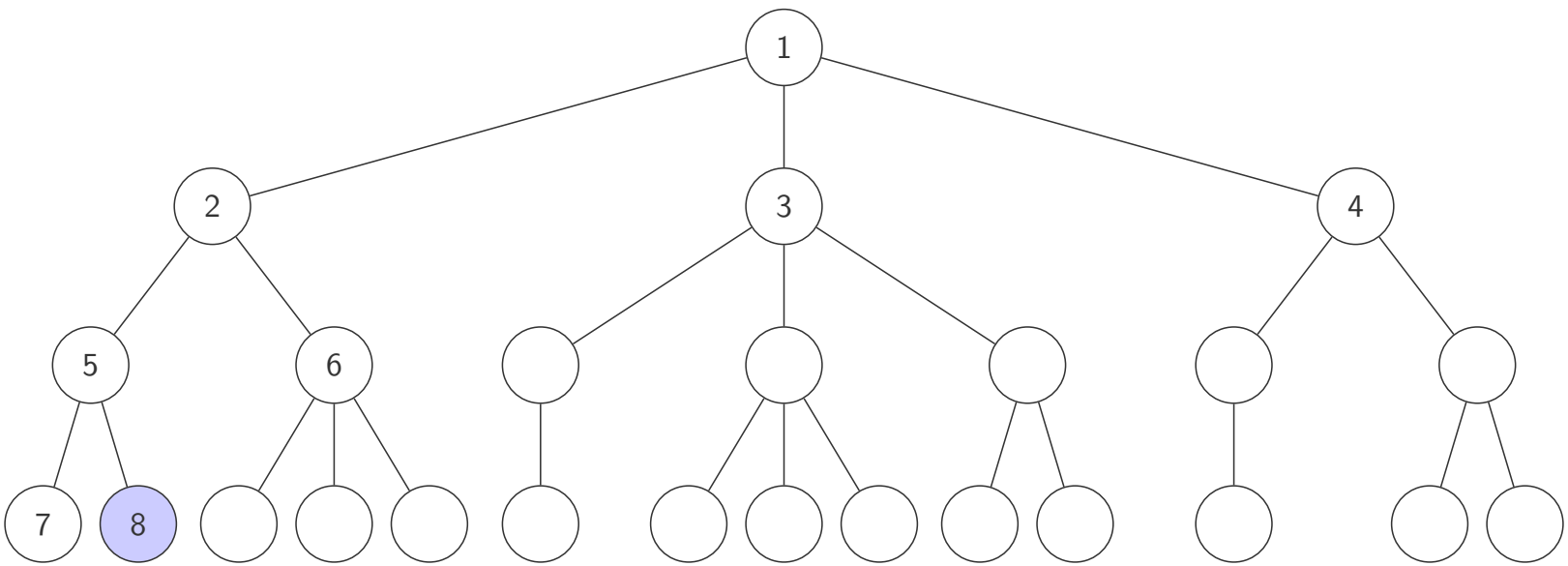
Depth-First Search (1)

- 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

  $$b + b^2 + b^3 + ... + b^m$$

- 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 ℓ
  - 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 ℓ

Iterative Deepening Depth-First Search

- Augmentation progressive de la profondeur maximale

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

- Solution optimale trouvée lorsque $\ell = d$

  Algorithme complet et optimal

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

Algorithm 1: Iterative-Deepening-Search

```
Function IDS(problem)
    for depth ← 0 to ∞ do
        result ← DLS(problem, depth)
        if result ≠ 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 prédécesseur 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**
Information parfaite	Échecs Dames Go Othello	Backgammon Monopoly
Information imparfaite	Stratego	Bridge Poker Scrabble Nuclear War

Valeur minimax

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

  Définition récursive de cette fonction

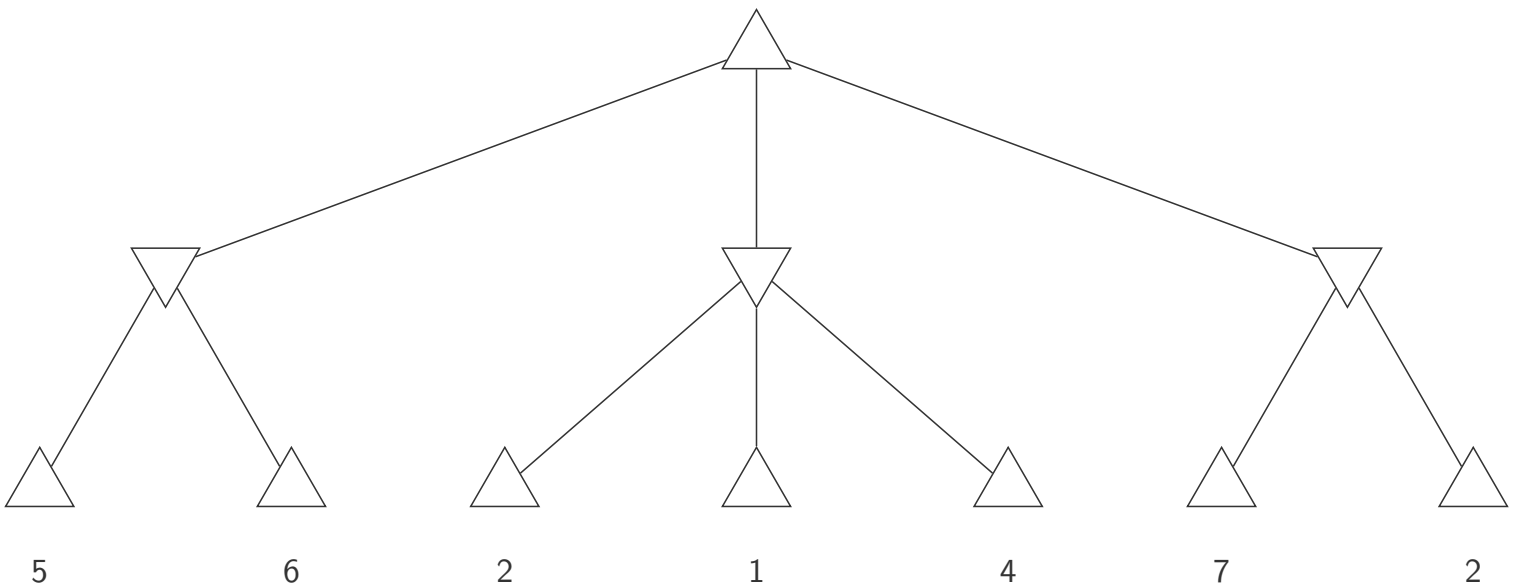
- Hypothèse que les deux joueurs jouent parfaitement
  - `MAX` préfère aller vers une situation de plus grande valeur
  - et `MIN` de plus petite valeur

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

Algorithme minimax

- Arbre du jeu avec $\triangle$ pour `MAX` et $\triangledown$ pour `MIN`

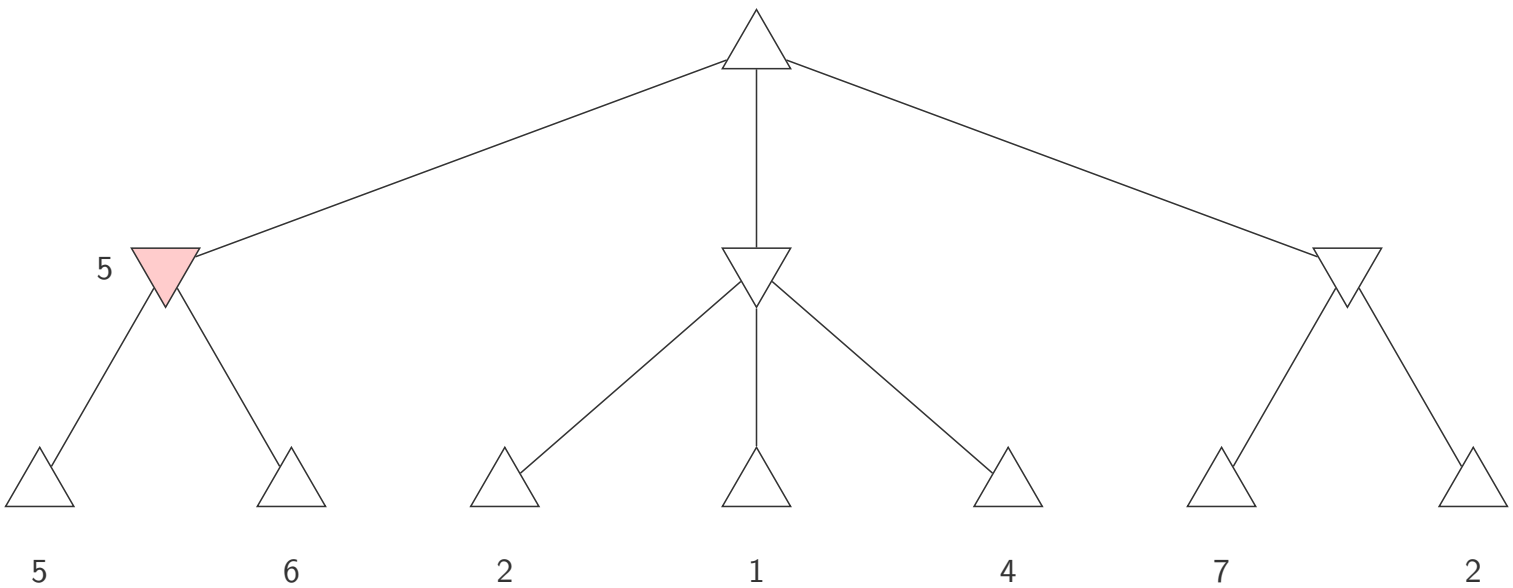
  `MAX` choisit toujours le coup qui maximise la valeur minimax



Algorithme minimax

- Arbre du jeu avec $\triangle$ pour `MAX` et $\triangledown$ pour `MIN`

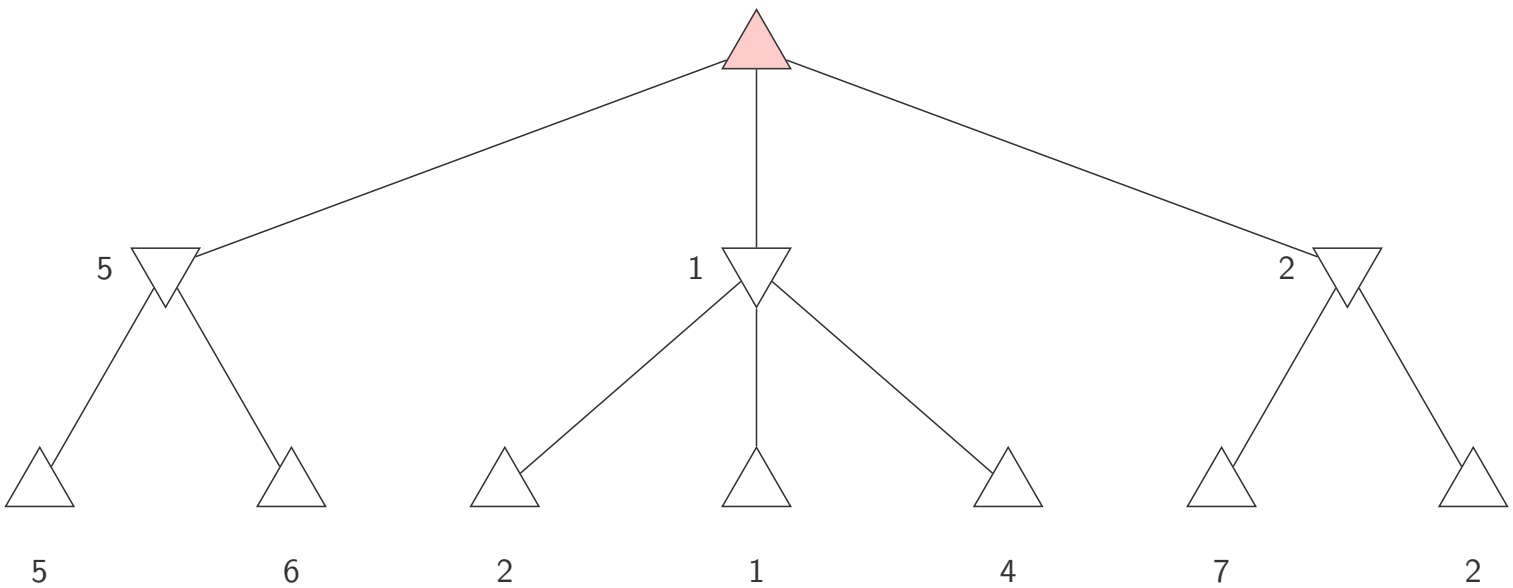
  `MAX` choisit toujours le coup qui maximise la valeur minimax



Algorithme minimax

- Arbre du jeu avec $\triangle$ pour `MAX` et $\triangledown$ pour `MIN`

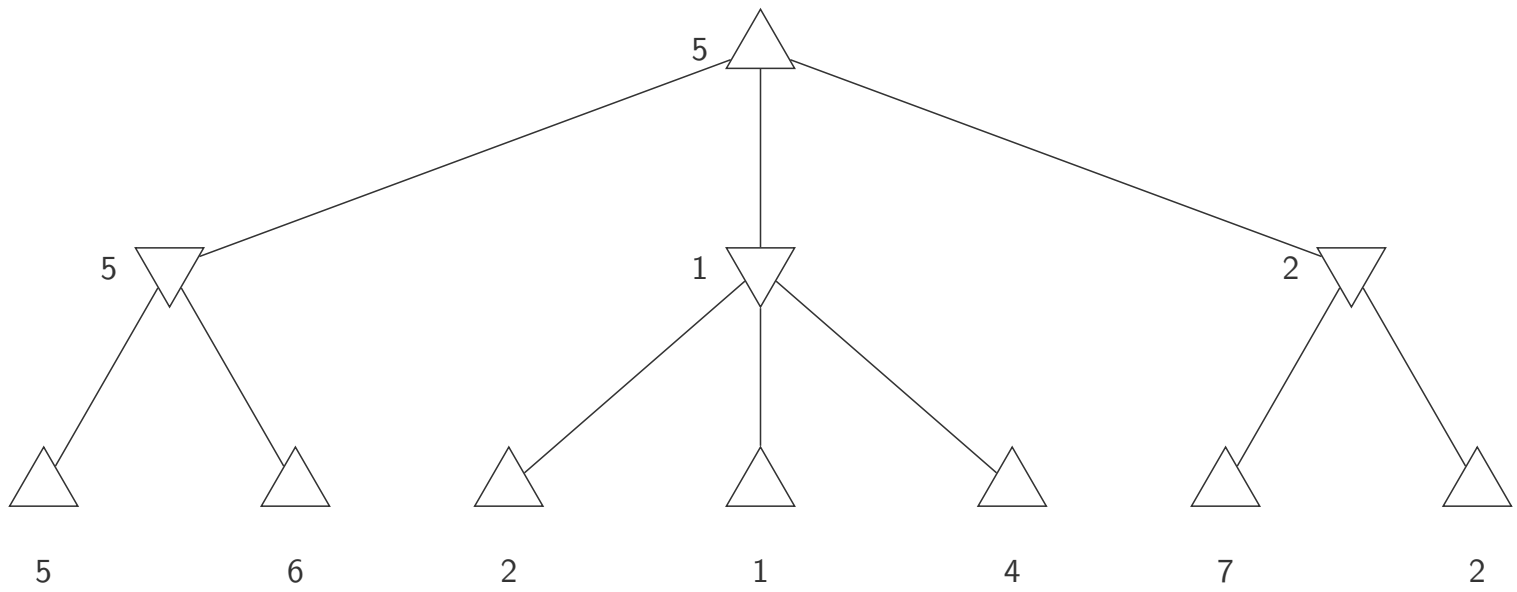
  `MAX` choisit toujours le coup qui maximise la valeur minimax



Algorithme minimax

- Arbre du jeu avec $\triangle$ pour `MAX` et $\triangledown$ pour `MIN`

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

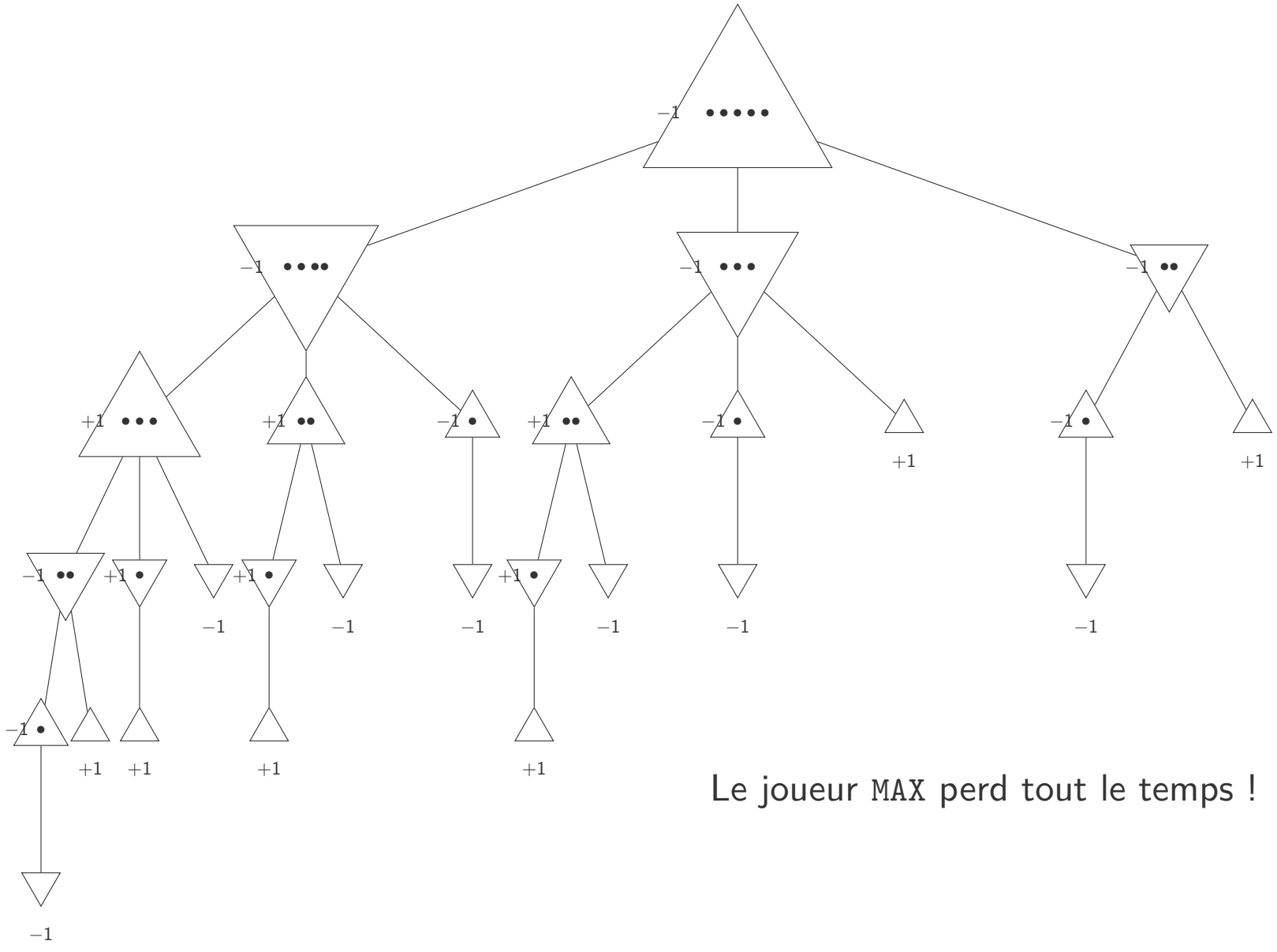


Jeu des bâtonnets (1)

- 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

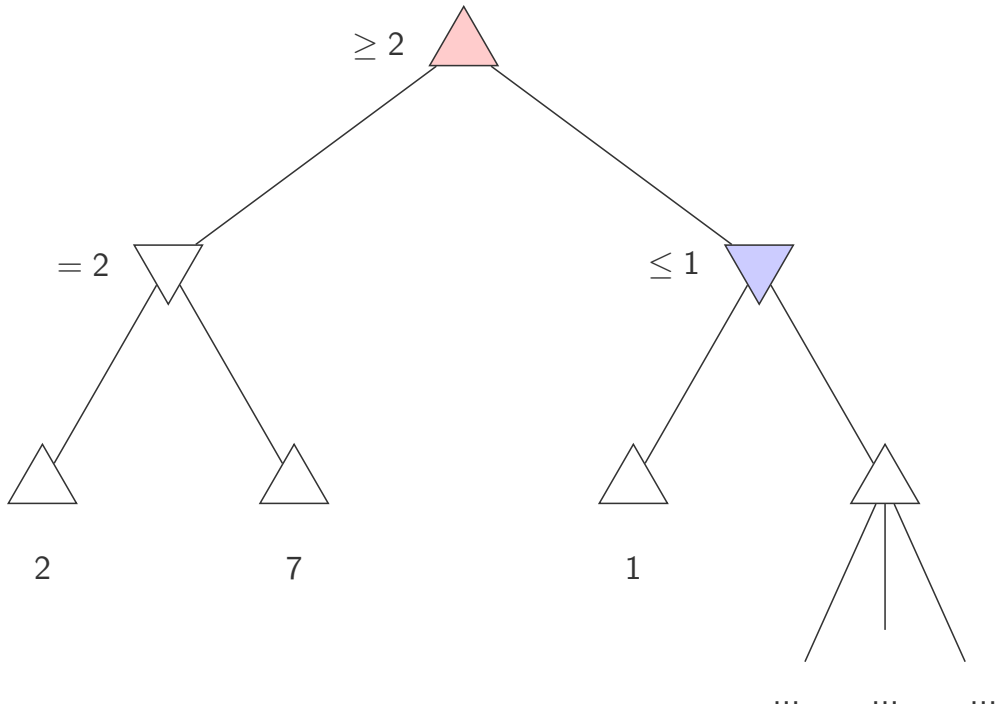
  On ne sait pas faire mieux que la valeur minimax actuelle

- Deux situations de simplification possibles
  - α plus grande borne inférieure pour `MAX`
  - β plus petite borne supérieure pour `MIN`

- Mémorisation des bornes durant l'exploration de l'arbre

Alpha-Beta pruning (2)

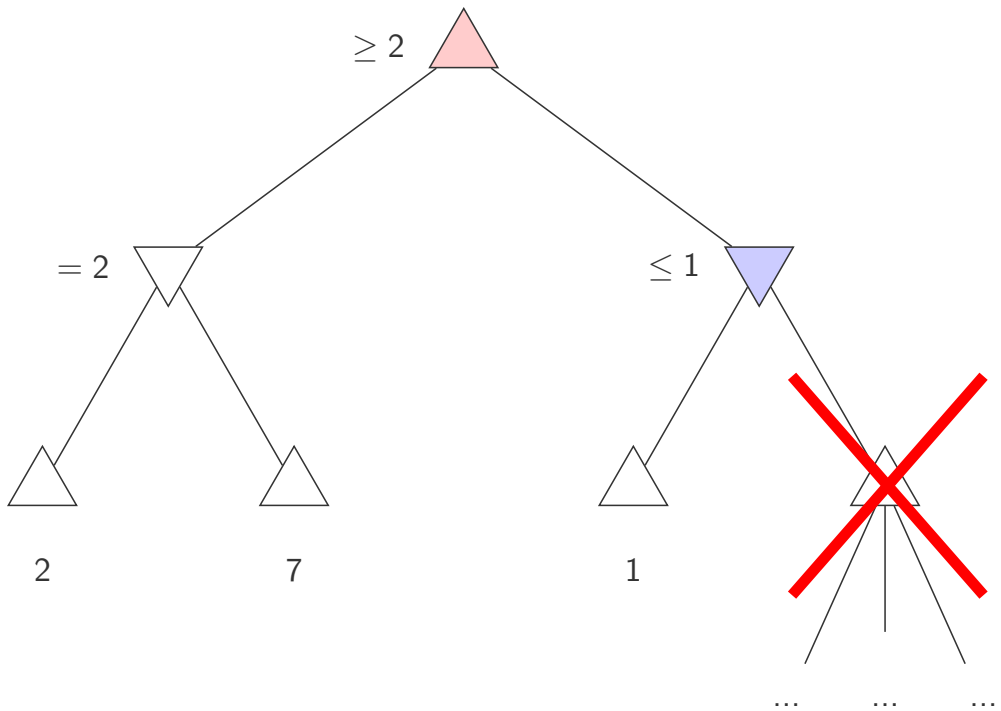
- On peut raisonner sur la situation actuelle de l'exploration
  - La racine sera ≥ 2 puisque c'est un `MAX`
  - Le fils droit de la racine sera ≤ 1 car c'est un `MIN`



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

Alpha-Beta pruning (2)

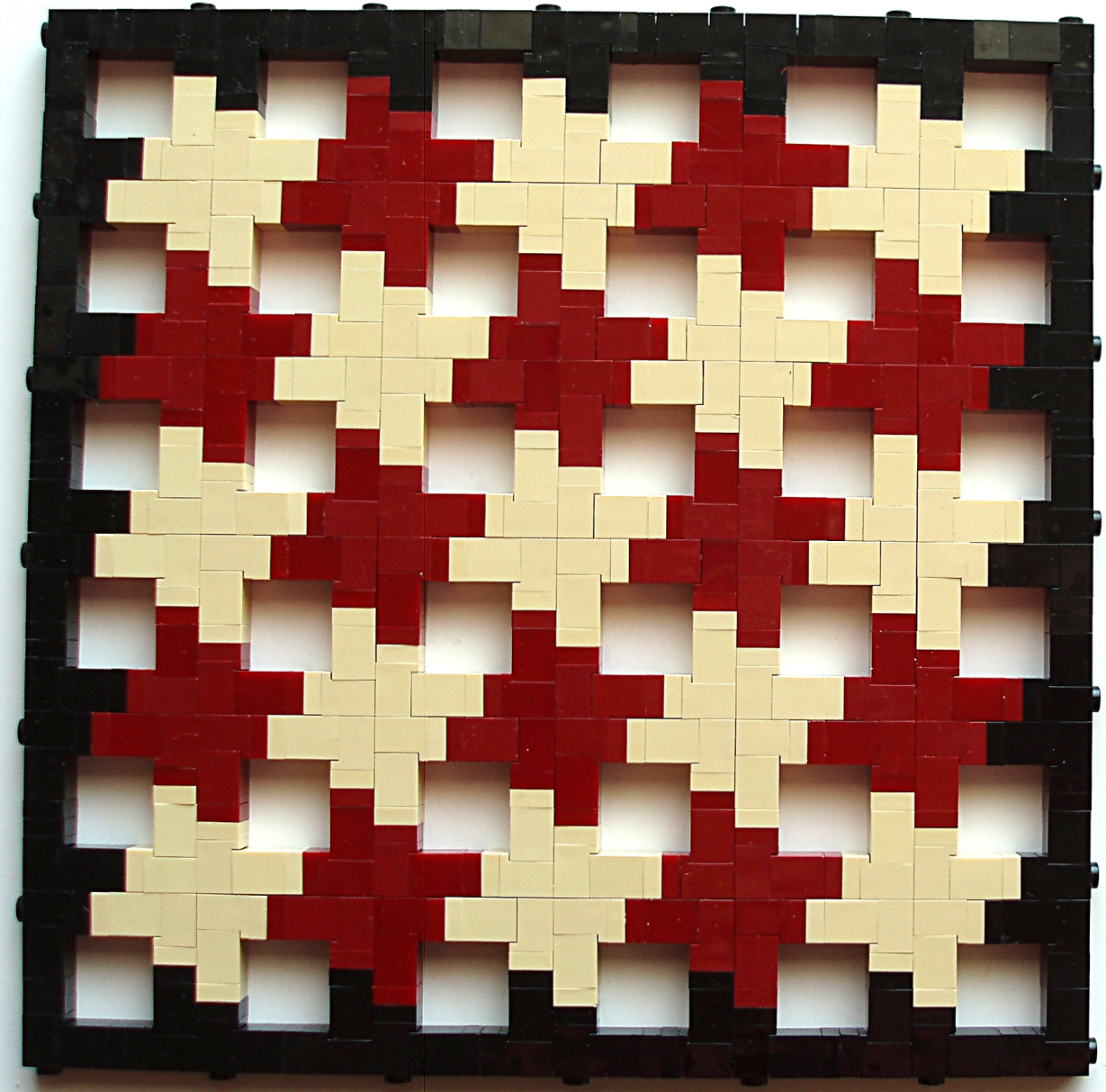
- On peut raisonner sur la situation actuelle de l'exploration
  - La racine sera ≥ 2 puisque c'est un `MAX`
  - Le fils droit de la racine sera ≤ 1 car c'est un `MIN`

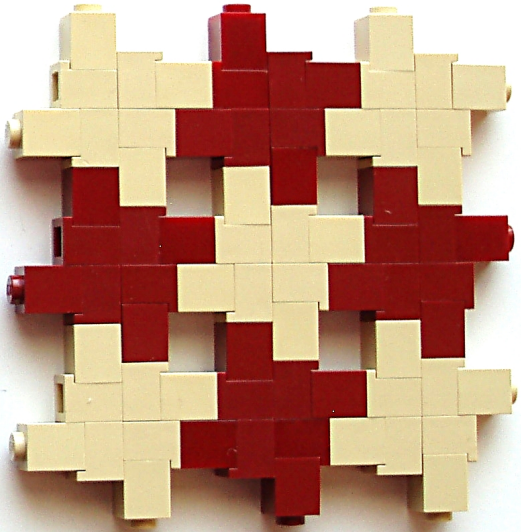


Exemple animé : http://alphabeta.alekskamko.com/

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
           (len(state))])
9          missing = len(GOAL) - len(state)
10         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

easyAI

- 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)

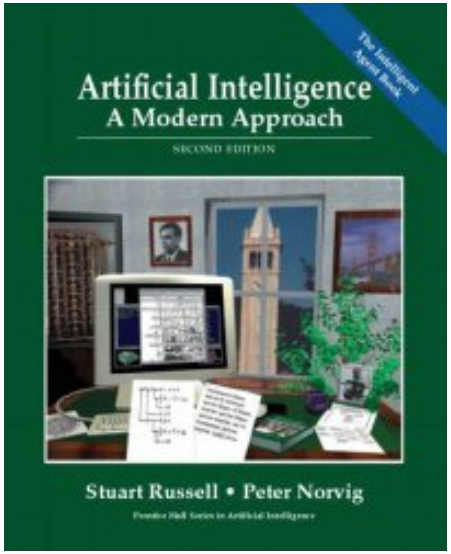
```
1  from easyAI import TwoPlayersGame, Human_Player, AI_Player, Negamax
2
3  class SimpleNim(TwoPlayersGame):
4      def __init__(self, players):
5          self.players = players
6          self.nplayer = 1
7          self.__sticks = 5
8
9      def possible_moves(self):
10         return [str(v) 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

978-0-130-80302-3

Crédits

- Photos des livres depuis Amazon
- 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