ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Résumé

2I1AE1: Artificial Intelligence

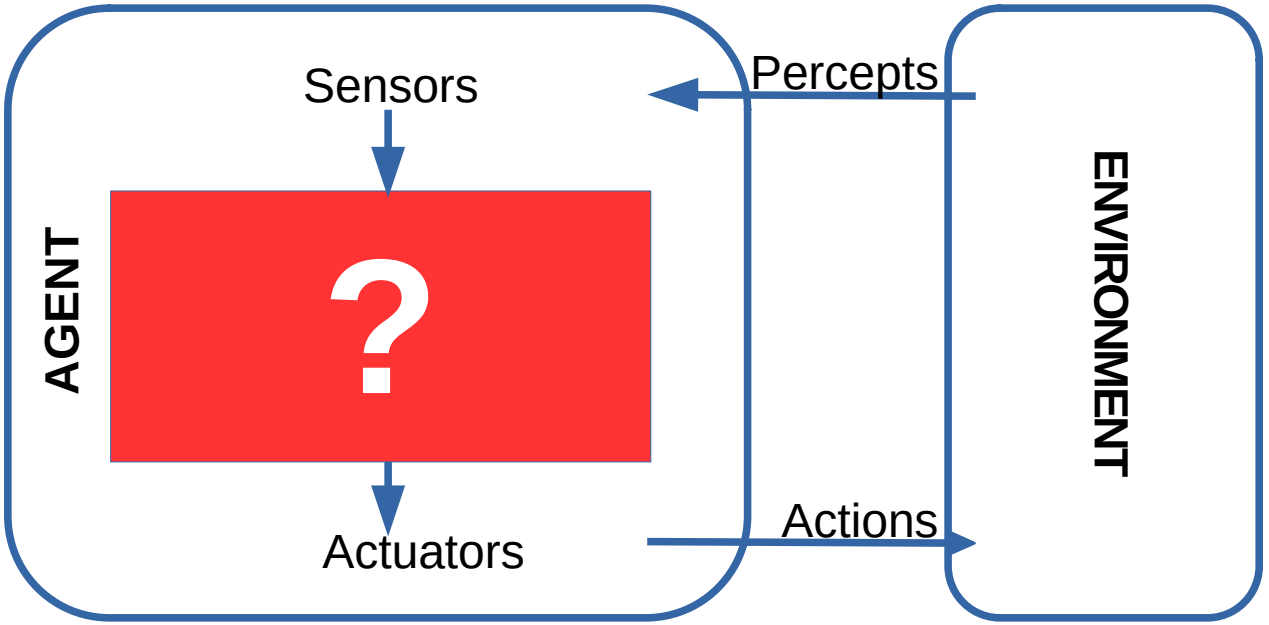
Régis Clouard, ENSICAEN - GREYC

Intelligence artificielle

- IA : mécaniser l'intelligence
- Deux visions
  - **IA Forte**
    - Conscience de soi, expression de vrais sentiments, compréhension de ses propres sentiments.
    - Introspection : ce qui importe, c'est la façon de résoudre les problèmes en se référant à l'humain.
  - **IA Faible**
    - Approche d'ingénierie visant à construire des algorithmes capables de résoudre des problèmes avec une approche non conventionnelle conférant des capacités d'adaptabilité : **agent rationnel**
    - Ce qui importe, c'est la solution quelle que soit la façon d'y parvenir. La méthode de résolution peut être stupide (cf. chambre chinoise de John Searle).
- Ce cours porte sur l'**IA faible** avec un point de vue sur la **recherche de solutions dans un espace d'états**.

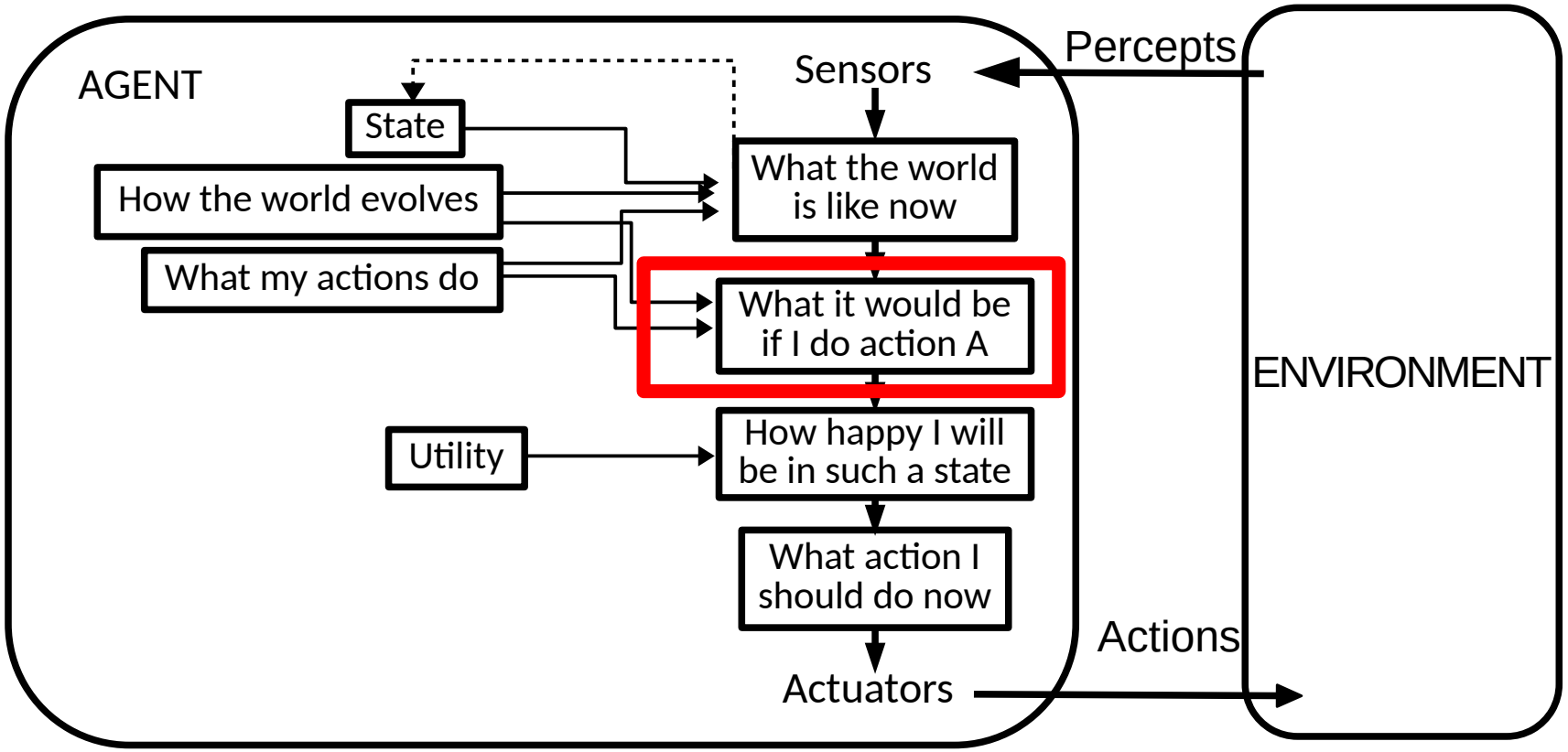
Agent autonome

- Capacités
  - de connaître son environnement
  - d'agir sur l'environnement



Agent intelligent rationnel

- **Intelligent** : raisonne par anticipation des effets de ses actions.
- **Rationnel** : utilise une mesure de performance pour évaluer la qualité de ses actions.



Contexte d'évolution d'un agent : PEAS

- Le système PEAS est utilisé pour catégoriser les agents
  - P : mesure de performance
  - E : environnement
  - A : effecteurs
  - S : capteurs

Agent Rationnel (sans apprentissage)

- Discrétisation du temps : à chaque top d'horloge une décision.

```
function INTELLIGENT-AGENT(percept, goal) returns an action
   static: state, the agent's memory of the world state

   state ← UPDATE-STATE-FROM-PERCEPTS(state, percept)
   action ← CHOOSE-BEST-ACTION(state)
   state ← UPDATE-STATE-FROM-ACTION(state, action)
   return action
```

Agent Rationnel (avec apprentissage)

- Discrétisation du temps : à chaque top d'horloge une décision.

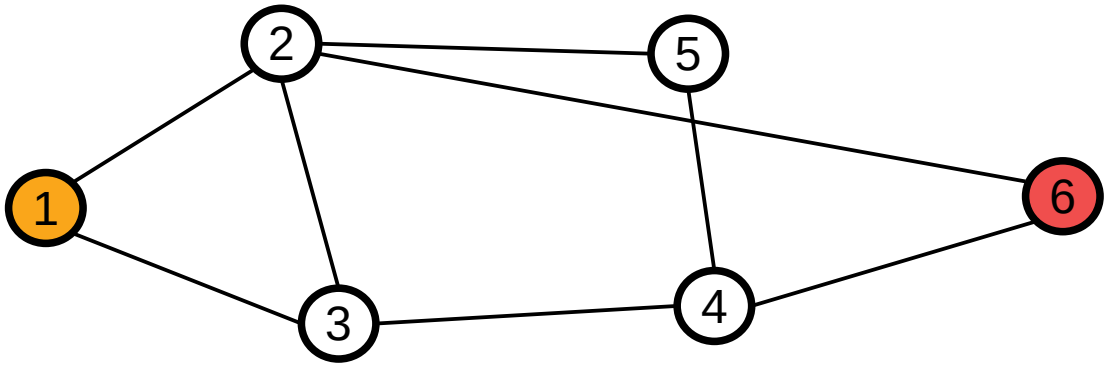
```
function INTELLIGENT-AGENT(percept, goal) returns an action
   static: state, the agent's memory of the world state

   state ← UPDATE-STATE-FROM-PERCEPTS(state, percept)
   nextState, reward ← DO-ACTION(state, action)
   UPDATE_FROM_TRANSITION(state, action, nextState, reward)
   state ← UPDATE-STATE-FROM-ACTION(state, action)
   return action
```

Une famille de problèmes : Recherche dans un espace d’états

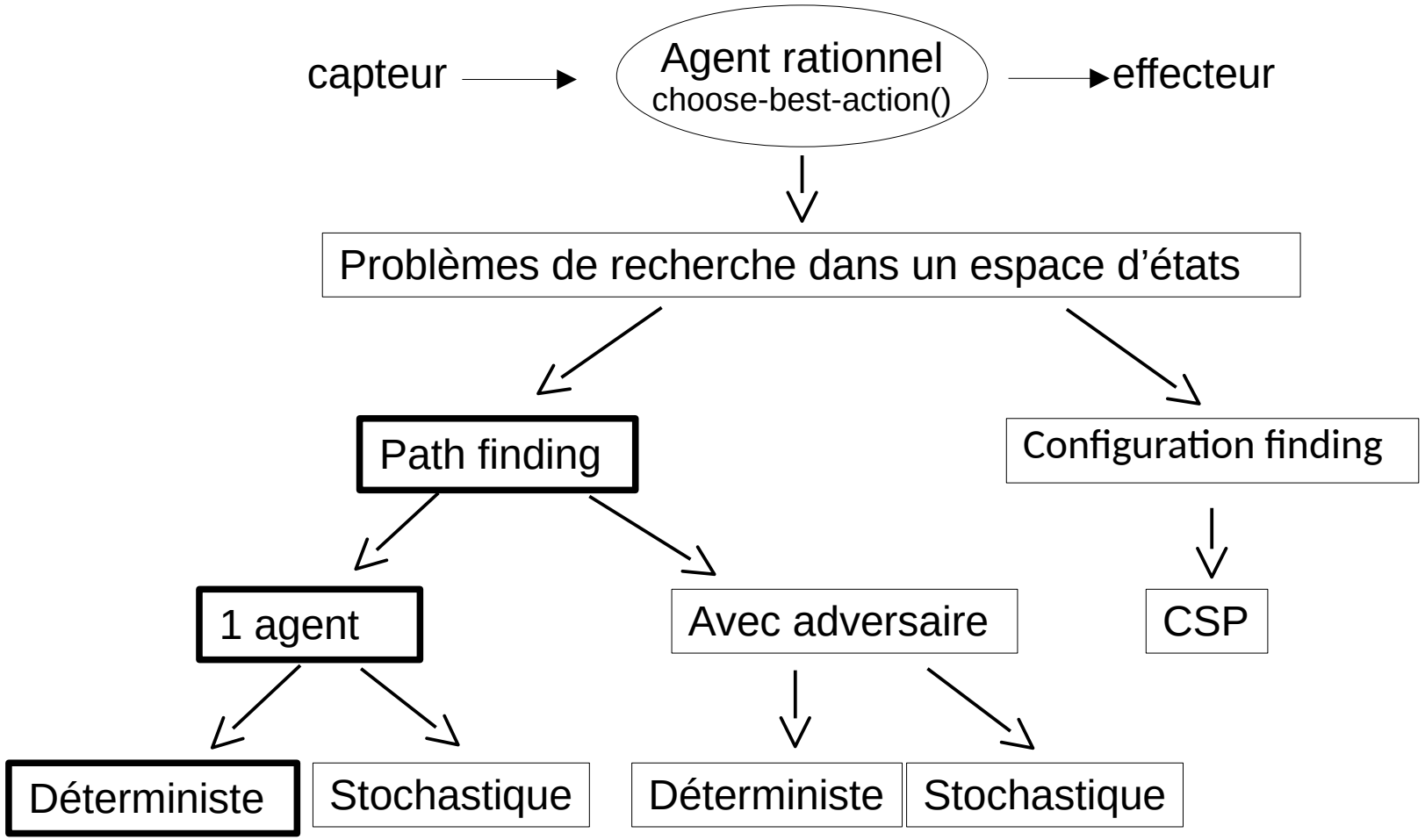
Recherche de solution dans un espace d'état

- Formulation des problèmes
  - **État** : un instantané de l'environnement (obtenu par les capteurs)
  - **État de départ** : en général donné
  - **Actions** : passer d'un état à un autre, ie prévoir l'état de l'environnement suivant en fonction de l'action appliquée (conséquence de l'action).
  - **Test de but** : identifier les états buts



- Deux types de problème :
  - Recherche du **chemin** entre l'état initial et un état but.
  - Recherche de la **configuration** d'un état final.

Classes d'algorithme de recherche



Algorithme de recherche général

- Algorithme général : parcours d'une liste faite avec les états jusqu'à trouver un état solution.

```
function GRAPH-SEARCH(problem) returns solution
  nodes ← MAKE-LIST(MAKE-NODE(INITIAL-STATE[problem]))
  closed-list ← MAKE-SET(MAKE-NODE(INITIAL-STATE[problem])
  LOOP
     IF EMPTY(nodes) THEN return failure
     node ← REMOVE-FRONT(nodes)
     IF IS-GOAL(problem, STATE[node]) THEN return the solution
     neighbors ← GET-SUCCESSORS(node, problem)
     FOR neighbor in neighbors DO
        IF STATE[neighbor] is not in closed-list THEN
          closed-list.add(STATE[neighbor])
          nodes ← ADD-IN-LIST(neighbor, nodes)
end
```

Algorithmes de recherche

Force brute : ne néssite aucune connaissance sur la résolution du problème

Method	Completness	Optimality	Time complexity	Space complexity	Agenda	f(n)
Breadth-first	Yes	Yes	$O(b^d)$	$O(b^d)$	FIFO	
Depth-first	No	No	$O(b^d)$	$O(bm)$	LIFO	
IDS	Yes	Yes	$O(b^d)$	$O(b.(d+1))$	LIFO	
Uniform cost	Yes	Yes	$O(b^d)$*	$O(b^{d+1})$*	PQUEUE	f(n)=g(n)

Meilleur d'abord : utilise la connaissance sur la résolution exprimée sous la forme d'heuristique

Method	Completness	Optimality	Time complexity	Space complexity	Agenda	f(n)
Greedy	No	No	$O(b^m)$*	$O(b^m)$*	PQUEUE	f(n)=h(n)
A*	Yes	Yes	$O(b^d)$*	$O(b^{d+1})$*	PQUEUE	f(n)=g(n)+h(n)
IDA*	Yes	Yes	$O(kb^d)$*	$O(bd)$	PQUEUE	f(n)=g(n)+h(n)

Note : la liste des nœuds visités

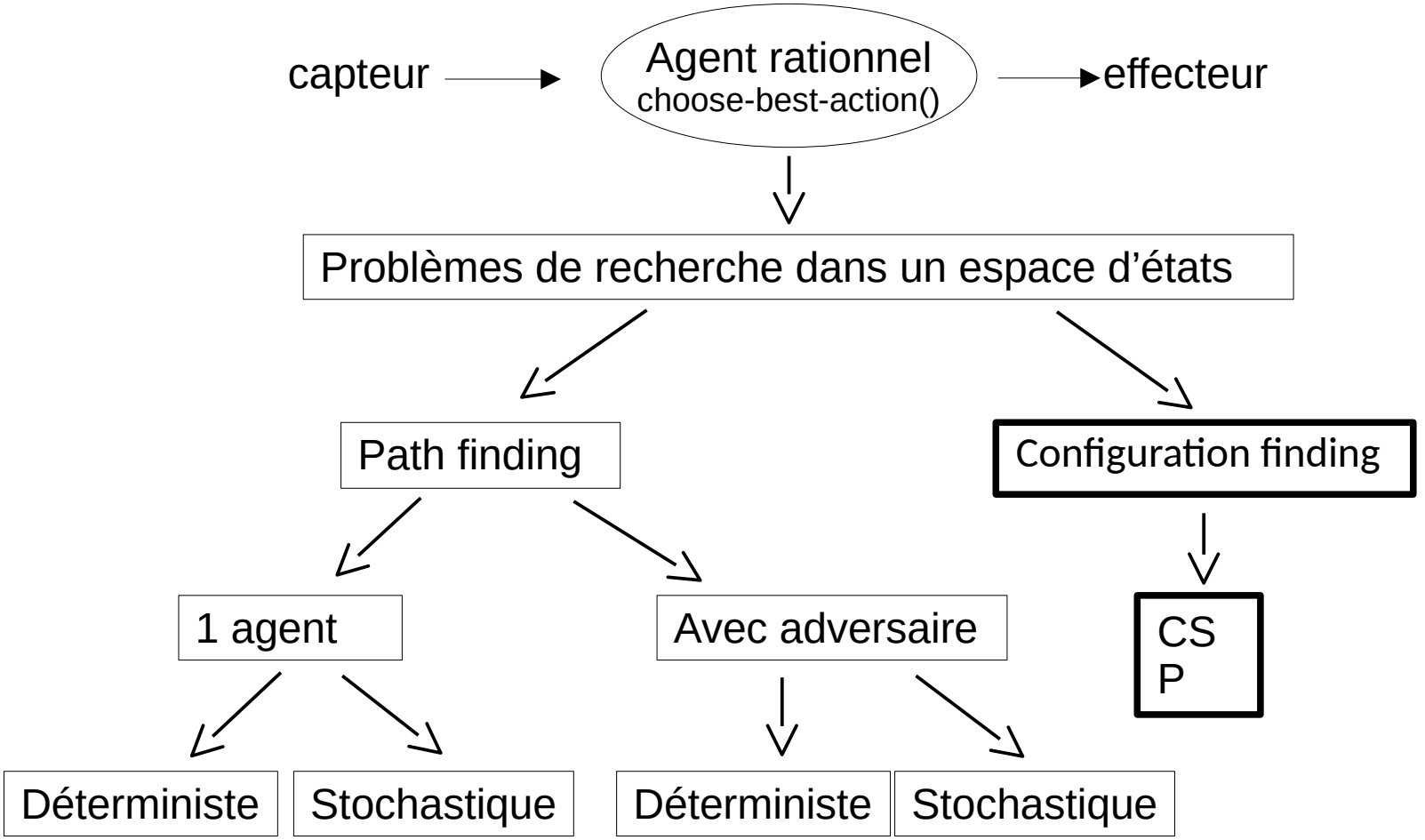
- Permet d'améliorer le temps de recherche. Mais rend tous ces algorithmes exponentiels en espace : $O(b^d)$.
- Tous les problèmes n'ont pas forcément besoin de cette *closed_list* (p. ex. N-reines où l'on met la reine dans la colonne la plus à gauche non pourvue).
- Ne fait pas partie de la définition des algorithmes.

Heuristique pour A* et IDA*

- Admissible : $0 \leq h(s) < h^*(s)$
- Créer les heuristiques
  - Forme relaxée du problème
  - Précalcul de solutions relaxées par programmation dynamique

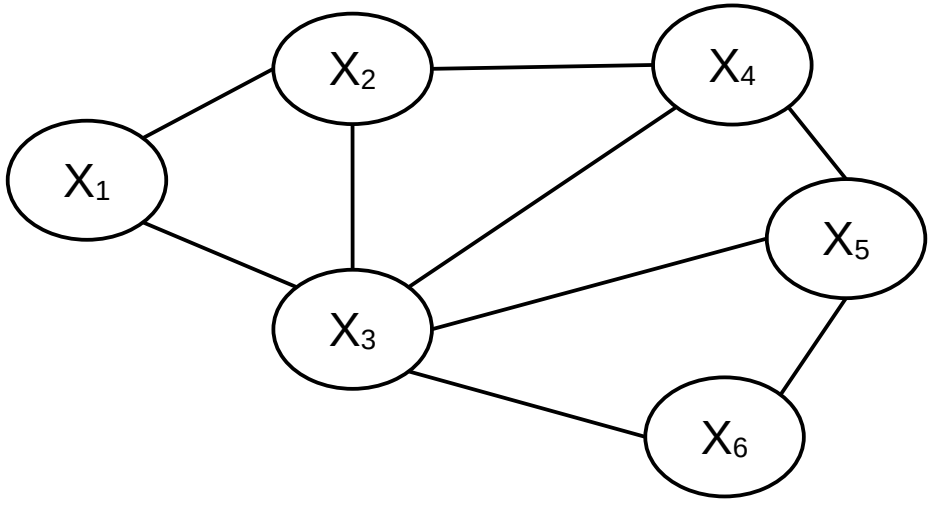
Cas particulier des problèmes de **recherche de configuration**

Problème de Satisfaction de Contraintes



Formulation d'un problème

- Nouvelle formulation :
  - État : Affectation partielle ou complète d'un ensemble de variables X. Chaque variable X_i à son domaine de définition D_i
    Les variables sont régies par un ensemble de contraintes C_i
  - État initial : $X_i = \{\}$
  - Test de but : tous les X_i ont une valeur et le système vérifie les contraintes
  - Action : affecter une valeur à une variable de telle manière qu'elle ne viole pas les contraintes
  - Type : recherche d'une configuration
- Représentation sous la forme d'un graphe de contraintes.



Algorithmes CSP

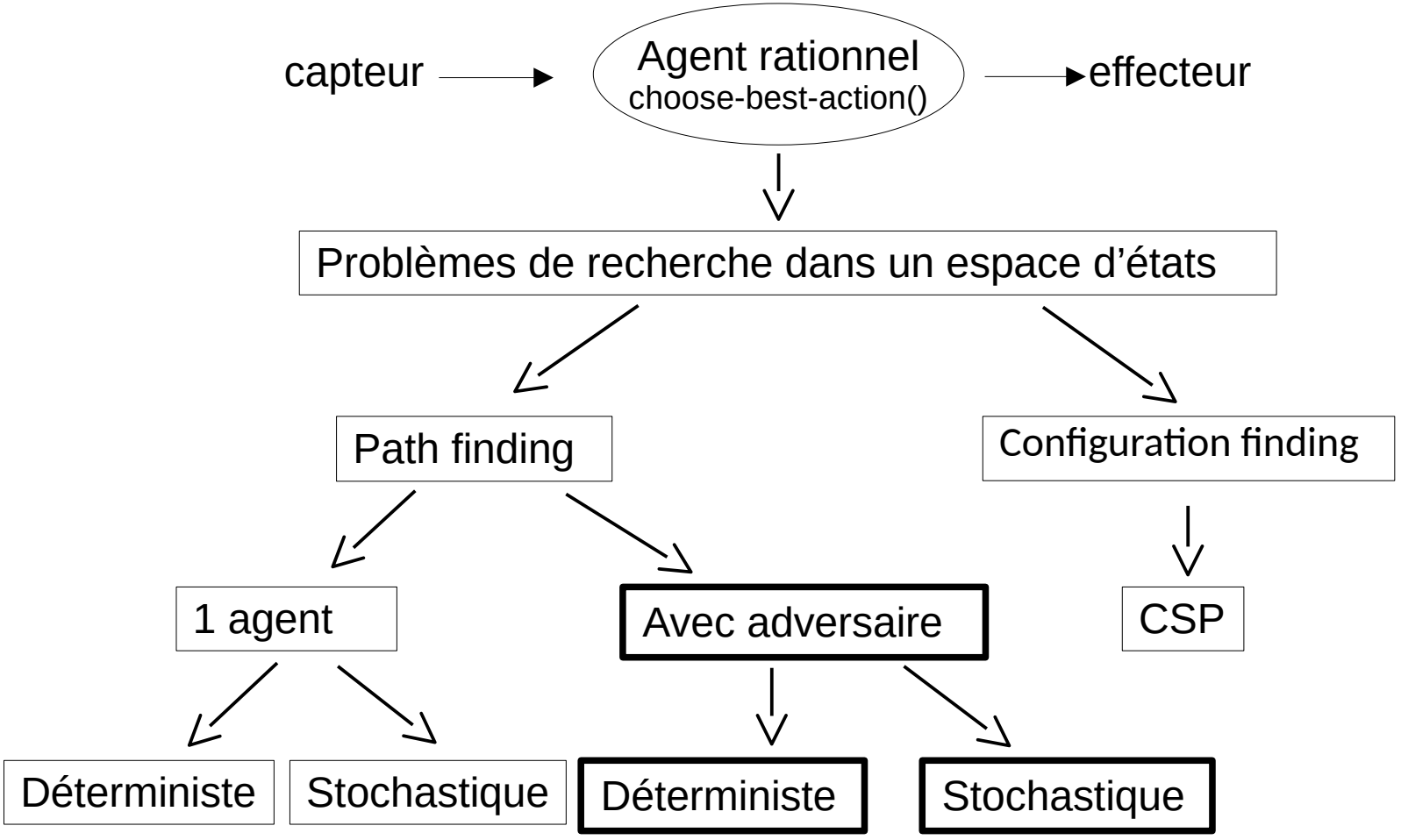
- Deux algorithmes
  - **Forward Checking** : après chaque affectation d'une variable, propager les contraintes pour restreindre les valeurs possibles des variables liées dans le graphe de contraintes. Permet d'élaguer l'arbre de recherche a posteriori.
  - **Arc consistency**
    - Ne garder que des arcs consistants : X→Y est consistant si pour chaque valeur de X, il reste des valeurs dans Y après application des contraintes.
    - **Preprocessing** : avant affectation, vérifier la consistance de tous les arcs dirigés. Plus long mais élague mieux l'arbre de recherche a priori.
    - **Maintaining Arc Consistency** : après affectation d'une variable X, revérifier tous les arcs Y→X

Heuristiques CSP

- Heuristiques générales et indépendantes du problème.
- Choix de la variable :
  - La plus contrainte
  - Celle qui apparaît dans le plus de contraintes
- Choix de la valeur dans la domaine de la variable choisie :
  - Celle qui retire le moins de valeurs aux autres après application des contraintes.

Cas particulier des problèmes de recherche de chemin en présence **d'adversaires**

Théorie des jeux

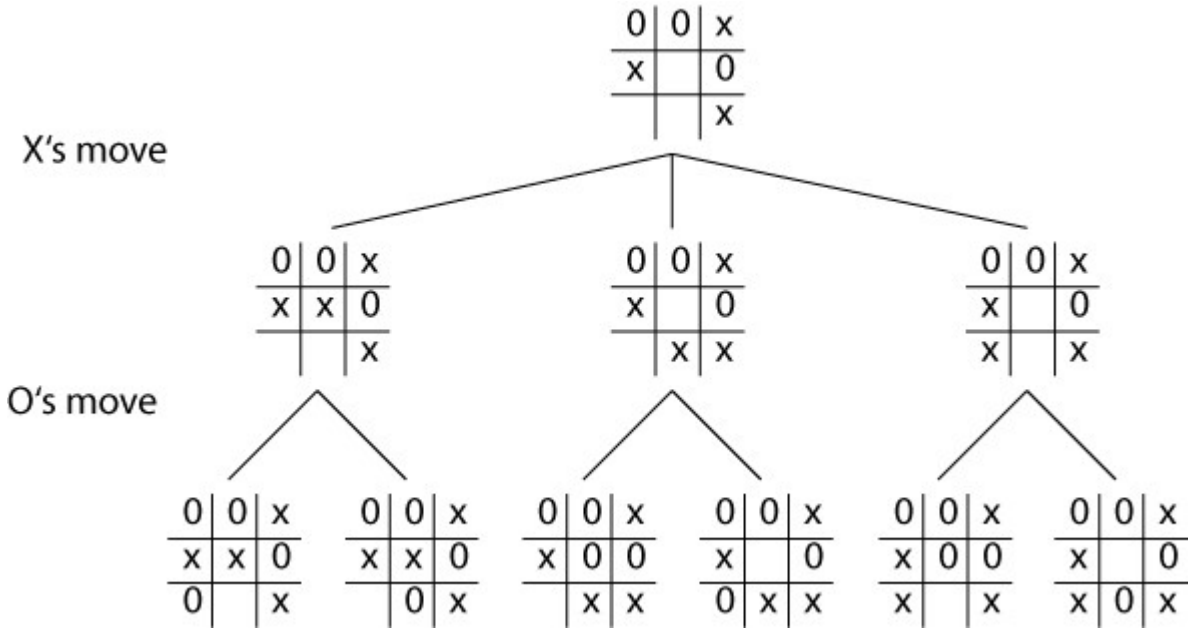


Problème avec adversaire

- Formulation du problème
  - **États** S : Configuration **+ joueur p ϵ P = {1, …, n})**
  - **État initial** S_0 : position initiale + **premier joueur**
  - **Actions** A : déplacements légaux
  - **Test de but** : S → {Vrai, Faux}
  - **Fonction d'utilité** : $S \times P \rightarrow \Re$
  - **Type**: chemin

Problème avec adversaire

- Minimax
  - Hypothèse 1 : deux joueurs, jeu information parfaite, somme nulle, déterministe, asynchrone
  - Hypothèse 2 : les deux joueurs sont rationnels.
  - Arbre de jeu : MAX choisit le coup lui donnant une valeur d'utilité maximale et MIN celle qui lui retourne une valeur d'utilité minimale.



- Inapplicable en réalité : gigantisme de l'arbre de jeu

Réduire la complexité : Solution 1

- Alpha-Bêta
  - Même hypothèse que précédemment. Le but est d'élaguer l'arbre de recherche pour réduire le temps de parcours de l'arbre de recherche.
  - Cela donne le même résultat que précédemment (pas d'approximation).
  - Permet de descendre une profondeur supplémentaire dans l'arbre pour le même temps de recherche.
- Amélioration : Iterative Deepening Alpha-Beta
  - Le nombre de coupures dépend du parcours : commencer par les branches qui donnent la valeur maximale pour MAX et minimale pour MIN.
  - Comment anticiper : version itérative de Alpha-Bêta
    - Ordonner l'arbre pour une itération en utilisant les valeurs remontées à l'itération précédente.

Réduire la complexité : Solution 2

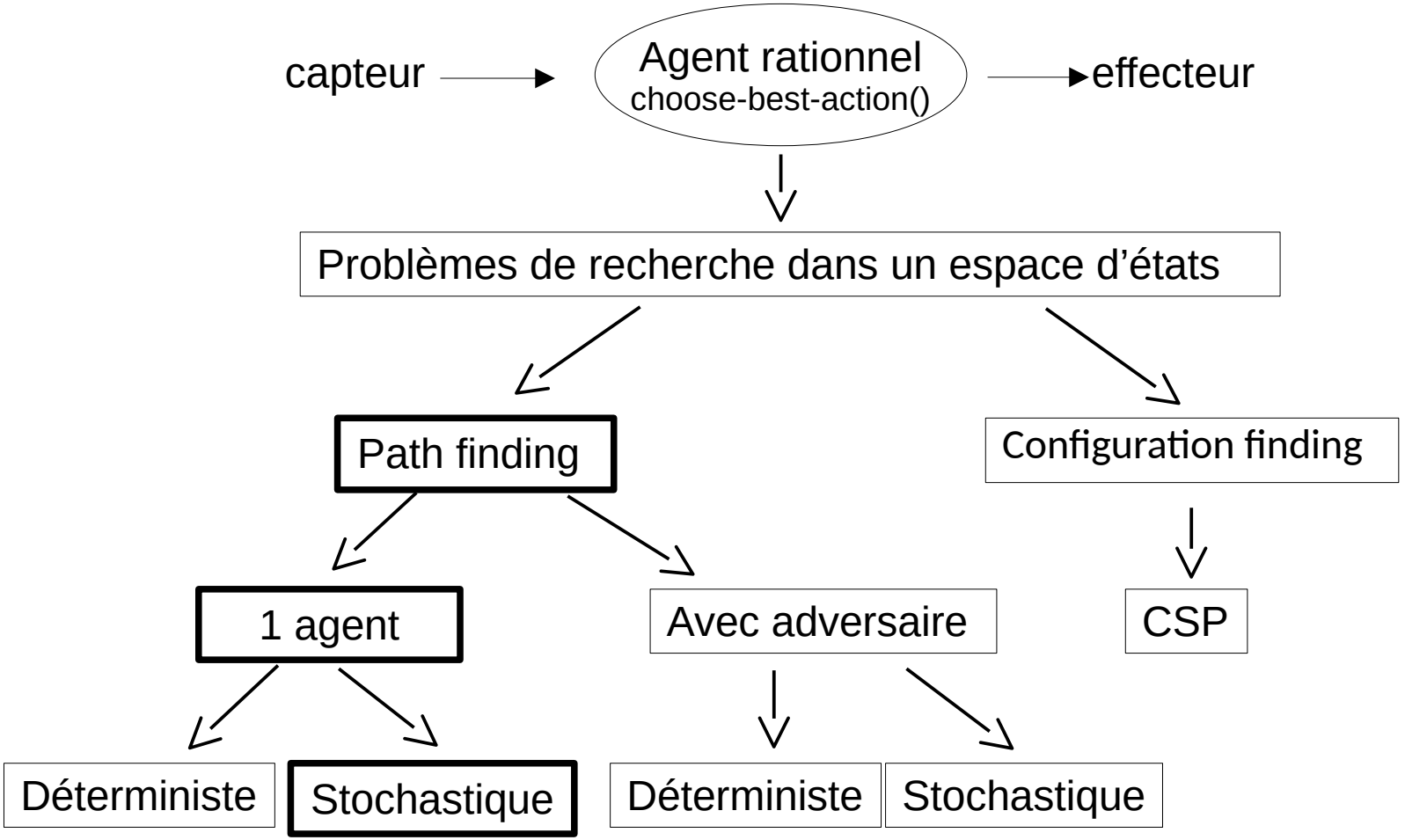
- Coupure : H-minimax / H-alpha-beta
  - Limiter la profondeur de l'arbre
  - Remplacer la fonction d'utilité par une fonction d'évaluation capable d'évaluer la valeur de confort de l'état.
    - Fonction d'utilité de la forme : $\Sigma_i w_i . f_i(s, a)$
  - Effet d'horizon
    - Explorer un peu plus profondément quand une feuille n'est pas « tranquille »

Variantes

- Problème stochastique
  - **Expectiminimax** : on introduit un niveau chance et on prend non plus le MAX ou le MIN mais l'espérance de gain, selon les probabilités de chaque action.
- Somme non nulle
  - Utiliser un vecteur valeurs d'utilité.
- Information incomplète
  - Se ramener à un problème à information complète, par exemple Monte Carlo

Cas particulier des problèmes de recherche de chemin dans un cadre **stochastique**

Processus de Décision Markoviens



Processus de Décision Markoviens : MDP

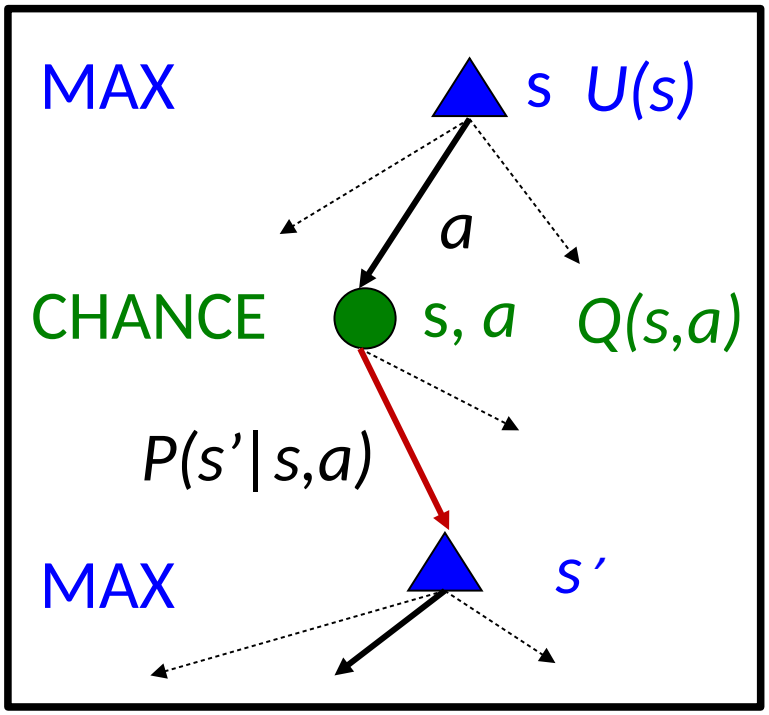
- Formulation du problème
  - **État: S** un ensemble de configuration
  - **État initial**: s_0
  - **Actions :** A(s) ensemble d'actions possibles pour l'état *s*
  - **Test de but :** final states
  - **Type** : recherche de chemins
- Nouveautés : contexte stochastique
  - P(s|s', a) : probabilité de passer en s' à partir de s par a.
  - Récompense R(s) que reçoit l'agent en arrivant dans l'état s.

Politique vs Plan

- Modèle
  - On cherche une politique
  - Différence plan / politique :
    - Plan : une séquence d'actions. Ne peut être remise en cause.
    - Politique : à chaque état, une table π(s) qui donne la meilleure prochaine action. L'agent devient alors un **simple agent réflexe** si la table π(s) est connue.

MDP

- Résolution
  - Hypothèse : P(s|s', a) et R(s) sont connus.
  - Le but est de trouver la politique π.
  - Pour cela : calculer la valeur d'utilité optimale en chaque état.
    - U(s) = R(s) + récompense dans le futur
    - $U^*(s) = R(s) + \gamma \max_{a \in A(s)} \Sigma_{s' \in S} P(s'|s, a) U^*(s')$
- Algorithme programmation dynamique
  - Au début tous les $U^*(s)$ à 0.
  - Jusqu'à idem potence (ie $U^*(s)' - U^*(s)) < \varepsilon$
    - $U^*(s) = R(s) + \gamma \max_{a \in A(s)} \Sigma_{s' \in S} P(s'|s, a) U^*(s')$
  - Extraction de la politique
- Politique :
  - $\pi^*(s) = \arg\max_a [\gamma \Sigma_{s' \in S} P(s'|s, a) U^*(s')$



Apprentissage dans un cadre de recherche de solution dans un espace d’états

Apprentissage par renforcement

Apprentissage par renforcement

- Hypothèse : cette fois on ne connaît pas P(s|s', a) ni R(s), mais on veut connaître la politique.
- On va donc l'apprendre par des essais.
- Algorithme Q-Learning : apprendre les Q(s,a)
  - $Q(s,a) = R(s) + \gamma \sum_{s' \in S} P(s'|s, a)\, U(s')$
  - $U(s) = \text{argmax}_a\, Q^*(s,a)$
- Utilisation de la différence temporelle avec un taux d'apprentissage α.
  - $Q(s,a) \leftarrow Q(s,a) + \alpha$ (difference)
  - $\text{difference} = [R(s) + \gamma \max_{a'} Q(s',a')] - Q(s,a)$

Problème de l'exploration-exploitation

- Exploitation : choisir l'action issue de la politique courante → n'apprend pas grand chose
- Exploration : choisir une actions au hasard→ comportement aléatoire
- Une solution triviale : epsilon-greedy
  - 1-ε fois : $\pi^*(s) = \arg\max_a Q^*(s,a)$
  - ε fois : $\pi^*(s) = \text{random}(A(s))$
- Une solution plus adaptative
  - fonction d'exploration : forcer l'essai d'actions non encore essayées.

Généralisation

- Généralisation
  - Facteur limitant : la taille du tableau Q[s,a]
    - Occupation mémoire et temps d'apprentissage prohibitifs
- Ne plus considérer les Q[s,a] mais une signature des Q[s,a] :
  - Q[s,a] n'est plus tableau mais une fonction Q(s,a) qui utilise un tableau de poids et une fonction de calcul d'une valeur.
    - $Q(s,a)= \Sigma_i w_i f_i(s,a)$
    - $w_i \leftarrow w_i + \alpha[\text{difference}] f_i(s,a)$
    - $\text{difference} = [R(s,a) + \gamma \max_{a'} Q(s',a')] - Q(s,a)$

Objectifs du cours

- Formulation de problèmes
- Complexité algorithmique : la perfection est inatteignable
- Heuristiques pour trouver une très bonne solution
- Les algorithmes ne doivent pas être modifiés pour les adapter aux problèmes. Cela passe par :
  - `get_successors()`
  - `add_in_list()`
  - Coût d'une action
  - Fonction d'utilité : f(s)
    - $\Sigma_i w_i . f_i(s, a)$
  - Contraintes
  - R(s)
  - P(s'|s, a)
  - γ
  - α
- Passage à l'échelle : ingénierie