

# Synthèse Artificial Inteligence LINFO1115

Jacques HOGGE

May 22, 2023

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Turing test . . . . .	6
<b>2</b>	<b>2.1-2.3</b>	<b>7</b>
2.1	Agents and environnements . . . . .	7
2.2	Concept of Rationality . . . . .	7
2.3	Nature et environnements . . . . .	8
<b>3</b>	<b>Solving probleme by search</b>	<b>10</b>
3.1	Structure d'un agent . . . . .	10
3.2	Problem solving Agent . . . . .	10
3.2.1	Definition Probleme . . . . .	10
3.2.2	Solution . . . . .	11
3.2.3	State space . . . . .	11
3.2.4	Search Tree . . . . .	11
3.2.5	State and Nodes . . . . .	12
3.2.6	Repeated States . . . . .	12
3.3	Searching for Solutions . . . . .	12
3.3.1	Stratégies . . . . .	13
3.4	Uninformed Search . . . . .	13
3.4.1	Breadth-First Search(BFS) . . . . .	13
3.4.2	Uniform-Cost search (UCS) . . . . .	15
3.4.3	Depth-first Search (DFS) . . . . .	15
3.4.4	Depth-Limited Search . . . . .	15
3.4.5	Iterative Deepening . . . . .	15
3.4.6	Bidirectional Search . . . . .	17
3.5	Informed Search . . . . .	17
3.5.1	Heuristic Fonction . . . . .	17
3.5.2	Triangle inequality . . . . .	18
3.5.3	Monotonicity . . . . .	18
3.5.4	Greedy Best-First Search(GBFS) . . . . .	18
3.5.5	A* . . . . .	19
3.5.6	Memory-bounded heuristic search . . . . .	21
3.6	Heuristic en pratique . . . . .	22
<b>4</b>	<b>Local Search</b>	<b>23</b>
4.1	Optimisation probleme . . . . .	23
4.2	Neighbourhood . . . . .	23
4.3	Heuristic et Metaheuristics . . . . .	24
4.4	Hill Climb Algo . . . . .	24
4.4.1	Varaintes . . . . .	25
4.5	Random Walk . . . . .	25
4.6	Simulated annealing . . . . .	25
4.7	Local Beam Search . . . . .	26
4.8	Genetic Algorithms (GAs) . . . . .	26
4.9	Tabu search Metaheuristics . . . . .	28

4.10	Intensification vs. Diversification . . . . .	28
4.11	Other Local Search . . . . .	28
<b>5</b>	<b>Constraint Satisfaction Problem (CSP)</b>	<b>29</b>
5.1	Constraint Propagation . . . . .	30
5.1.1	Arc-consistency algorithm . . . . .	30
5.2	Backtracking Search for CSPs . . . . .	31
5.2.1	Incremental formulation . . . . .	31
5.2.2	Backtracking search . . . . .	31
5.2.3	Variable ordering . . . . .	33
5.2.4	Interleaving search and inference . . . . .	33
5.2.5	Intelligent Backtracking . . . . .	33
5.3	Local Search for CSP . . . . .	33
5.3.1	Complete state formulation . . . . .	33
5.3.2	Min Conflict . . . . .	34
5.4	Structure of the problems . . . . .	34
5.4.1	Independent subproblems . . . . .	34
5.4.2	Tree-structure Subproblems . . . . .	34
<b>6</b>	<b>Adversarial Search and Games</b>	<b>35</b>
6.1	Game . . . . .	35
6.2	MinMax Algo . . . . .	35
6.3	Alpha-Beta pruning . . . . .	37
6.3.1	Move ordering . . . . .	39
6.4	Imperfect Decisions . . . . .	39
6.4.1	Evaluation Function . . . . .	39
6.4.2	Cutting off the search . . . . .	40
6.4.3	Forward pruning . . . . .	40
6.4.4	Search VS. Lookup . . . . .	40
6.5	Monte Carlo Tree Search . . . . .	40
6.6	Stochastic Games . . . . .	41
6.6.1	Expectminimax . . . . .	41
<b>7</b>	<b>Logical Agent</b>	<b>43</b>
7.1	Knowledge Based Agents . . . . .	43
7.1.1	Operation KBA . . . . .	43
7.1.2	Knowledge representation . . . . .	43
7.2	Exemple : the Wumpus World . . . . .	44
7.3	Logique . . . . .	46
7.3.1	Implication (entailment) . . . . .	47
7.3.2	Monotonicité . . . . .	48
7.4	Proposition logique . . . . .	49
7.4.1	Syntax . . . . .	49
7.4.2	Sémantique . . . . .	50
7.4.3	Inference . . . . .	50
7.5	Propositional Theorem Proving . . . . .	51
7.5.1	Preuve inference . . . . .	51

<b>8 First-Order Logic</b>	<b>54</b>
8.1 Propositional logic . . . . .	54
8.2 Syntax and semantics of FOL . . . . .	54
8.2.1 Complex sentences . . . . .	56
8.2.2 Quantifiers . . . . .	56
8.3 Using FOL . . . . .	56
8.3.1 Ex Wumpus . . . . .	56
8.4 Complement on FOL . . . . .	56
<b>9 Inference in First-Order Logic</b>	<b>57</b>
9.1 Propositional vs FOL inference . . . . .	57
9.1.1 Reduction to Propositional Logic . . . . .	57
9.2 Unification . . . . .	59
9.2.1 Most General Unifier . . . . .	59
9.2.2 Standardize apart . . . . .	59
9.2.3 Generalized Modus Ponens . . . . .	59
9.3 Forward Chaining . . . . .	61
9.3.1 First-order definite clauses . . . . .	61
9.3.2 Analyse . . . . .	61
9.4 Backward chaining . . . . .	61
9.4.1 Logic programming: Prolog . . . . .	62
9.5 Résolutions . . . . .	62
9.5.1 Skolemization . . . . .	62
9.5.2 Resolution inference rule . . . . .	63
9.5.3 Introducing answers . . . . .	63
<b>10 Automated Planning</b>	<b>64</b>
10.1 Definition of classical planning . . . . .	64
10.1.1 Représentation du State . . . . .	65
10.1.2 Représentation des Actions . . . . .	65
10.1.3 Représentation des objectifs . . . . .	65
10.2 Algorithms for classical planning . . . . .	67
10.2.1 Complexité . . . . .	67
10.2.2 Forward and backward search . . . . .	67
10.2.3 Forward state-space search . . . . .	67
10.2.4 Backward state-space Search . . . . .	67
10.2.5 Planning as Boolean satisfiability . . . . .	67
10.3 Heuristics for planning . . . . .	68
10.3.1 Relaxed problem . . . . .	68
10.3.2 Set covering Problem . . . . .	68
10.3.3 Restricted precondition . . . . .	68
10.3.4 Ignore-delete-list heuristic . . . . .	68
10.3.5 Domain-independent pruning . . . . .	68
<b>11 Learning from examples</b>	<b>68</b>
11.1 Feedback . . . . .	68
11.2 Supervised Learning . . . . .	69
11.2.1 Classification VS. Regression . . . . .	69

11.2.2 Ockam's Razor . . . . .	69
11.3 Learning Decision Trees . . . . .	69
11.4 Decision tree from exemple . . . . .	70
11.5 The Decision-Tree-Learning Algorithm . . . . .	71
11.6 Entropy . . . . .	72
11.7 Leaning Curves . . . . .	72
11.8 Ensemble learning . . . . .	72
11.8.1 Bagging . . . . .	73
11.8.2 Random Forest . . . . .	73
11.8.3 Bootsing . . . . .	73

	<b>Human</b>	<b>Rationality</b>
<b>Thinking</b>	Systems that think like humans Cognitive modeling approach	Systems that think rationally The “Law of Thought” approach
<b>Acting</b>	Systems that act like humans The Turing Test approach	Systems that act rationally The rational agent approach

## 1 Introduction

Plusieurs définitions pour IA, pas encore acceptées

C'est un "Umbrella Term" (représente un concept), c'est multidisciplinaire car souvent modifié.

Une intelligence artificielle est un système qui peut :

- penser comme un humain ?
- agir comme un humain ?
- ...

### 1.1 Turing test

une IA est "réussie" si un humain pense que il parle avec un humain

## 2 2.1-2.3

### 2.1 Agents and environnements

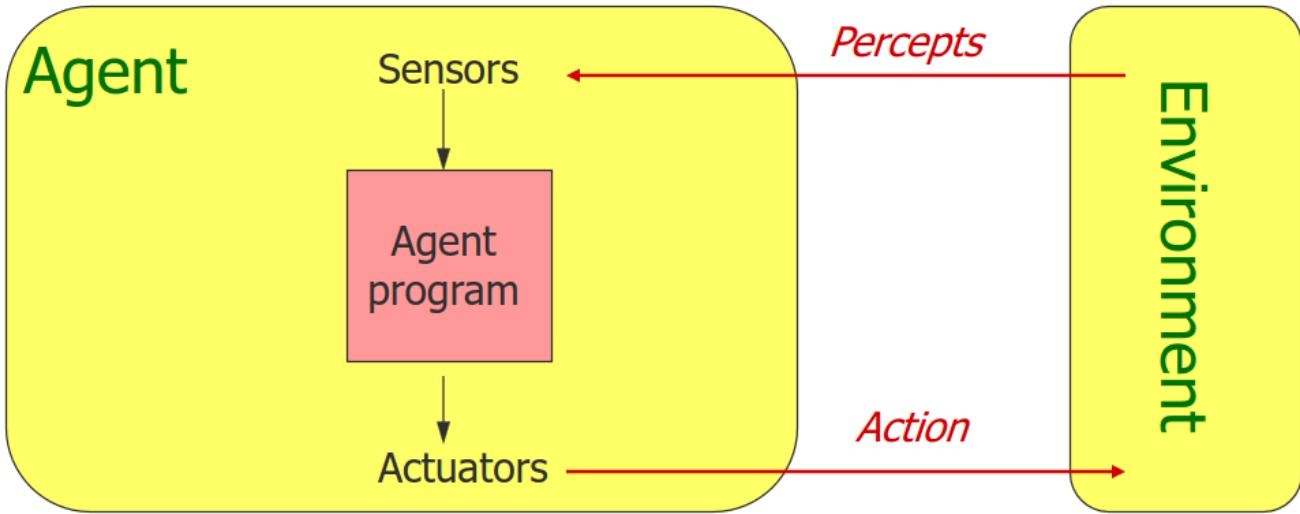


Figure 1: Agent

- **Agent** : Entité qui interagi avec son environnement, perception avec des sensors et action avec des actuators
- **Percept** : Perception de l'agent à un instant  $t$
- **Percept sequence** : Historique complet des percept d'un agent

### 2.2 Concept of Rationality

**Rationalité** : "Faire la bonne action", l'action qui amène au meilleur résultat. Mais un ordinateur ne sait pas ce qu'est la bonne chose à faire. On va donc avoir besoin d'une **Performance measure** qui va calculer un score si l'action est bonne ou pas.

**Agent rationnel** : Pour chaque séquence d'action possible, un agent rationnel doit sélectionner une action qui maximise la **Performance measure**.

La rationalité d'une action dépend de :

1. Une **Performance measure** qui dépend des critères de succès
2. La connaissance préalable de l'environnement par l'agent
3. Les actions que l'agent peut faire
4. La séquence de perception actuelle de l'agent

Attention, **Rationality  $\neq$  Omniscience**

**Agent omniscient** : sait le réel résultat de ses actions

**Agent autonome** : apprend ce qu'il peut faire pour compenser ce qu'il ne sait pas faire. (machine learning)

## 2.3 Nature et environnements

Environnement de travail d'un agent est spécifique à (PEAS) Performance, Environment, Actuators, Sensors.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint, angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry
Taxi driver	Safe, fast, legal, max profit	Roads, other traffic, customer, pedestrians	Accelerator, display, brake, signal	Cameras, sonar, speedometer, FPS,...

Figure 2: Exemple PEAS

Les propriétés sont les suivantes:

**Fully observable vs. Partially observable** : Toutes les informations pertinentes sont accessibles (échecs vs poker)

**Single agent vs. Multiagent** : il y a t'il plus que 1 agent, si oui c'est une compétition ou coopération ? (crossword puzzle vs chess)

**Deterministic vs. Stochastic** : (Chess vs.Poker)

- Le prochain état (state) de l'environnement dépend entièrement de l'action de l'agent.
- L'environnement pour l'agent est moins déterministe car il ne dispose pas de toutes les informations sur l'environnement.(Il ne peut pas déterminer avec précision l'état suivant à cause du hasard).

**Episodic vs. Sequential** : L'expérience de l'agent est divisée en épisodes atomiques indépendants les uns des autres.

**Static vs. Dynamic** : L'environnement peut changer quand l'agent est entrain de réfléchir.

**Discrete vs. Continuous** : Le nombre différent de state est fini et l'ensemble des actions est discret (chess vs. taxi driving)

**Known vs. Unknown** : le résultat de chaque action est donné. Ce n'est pas la même chose que d'être totalement observable, par exemple les jeux vidéo, on connaît toutes les informations via

l'écran mais on ne sait pas ce que fait le bouton.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle Chess with a clock	Fully	Single	Deterministic	Sequential	Static	Discrete
	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker Backgammon	Partially	Multi	Stochastic	Sequential	Static	Discrete
	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving Medical diagnosis	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis Part-picking robot	Fully	Single	Deterministic	Episodic	Semi	Continuous
	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller Interactive English tutor	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

### 3 Solving probleme by search

#### 3.1 Structure d'un agent

Un agent est composé de 2 partie :

- **Architecture** : c'est les composants de l'ordinateur sur le quelle l'agent tourne
- **Programme** : Se sont les fonctions qui Map les perceptions en actions

Exemple d'un programme d'agent:

- Table de conduite de l'agent (**table driven**)
- une table essor : Tous les states atteignable depuis le state actuelle
- **Path** : Séquence de states connecté par une sequence d'action
- **Operators** : Manière de modifier un state avec une action
- **Goals Test** : Fonction qui teste si un state est le résultat final
- **Step Cost** : Cout numérique de passer du state  $s$  avec l'action  $a$  pour passer au state  $s'$
- **Path Cost** : Fonction qui donne un cout a chaque path
- **Solution** : Séquence d'action de l'initial state jusqu'au goal state
- **Optimal Solution** : Solution avec le path avec le plus petit cout parmi toutes les solutions
- **Node** : Data structure qui constitue les graph et arbre de recherche
- **Frontier** : Ensemble de Node générée au quelle les ancêtre on été Goal-tested (visited)
- *C'étais long mais bon fallait le faire*

#### 3.2 Problem solving Agent

##### 3.2.1 Definition Probleme

Un problème peut être définie avec :

1. States ou Initial State
2. Une description des actions possible et valide par l'agent
3. Une description de quoi chaque action fait (transition model)

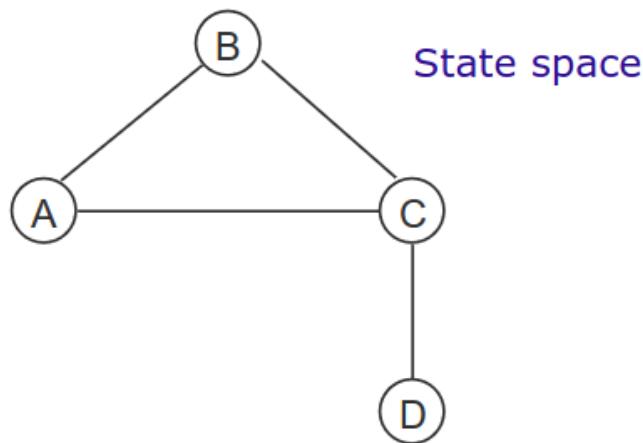
4. Un Goal test
5. Un Path cost qui utilise un step cost

### 3.2.2 Solution

Une solution est une séquence d'action qui commence du state initial et qui termine à un des goal state. La solution est optimal si elle a la plus petite path cost parmis toutes les solutions

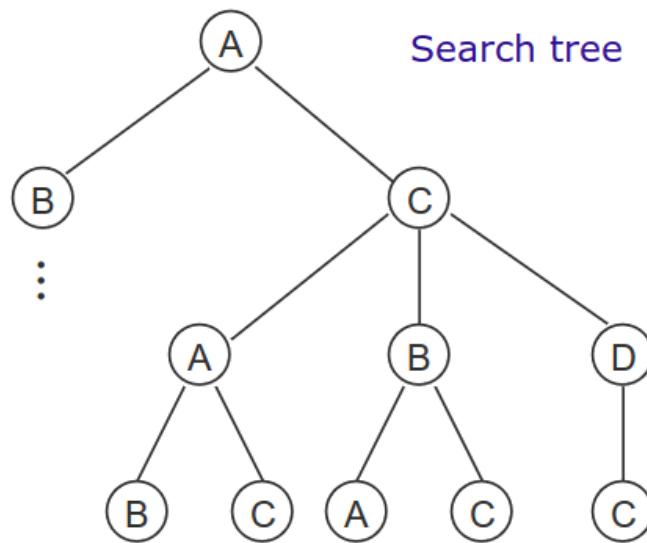
### 3.2.3 State space

Ensemble qui représente le problème avec les actions, les couts, ... (graph représentation). Possibilité d'un ensemble infini de states.



### 3.2.4 Search Tree

Représentation du problème sous la forme d'arbre, avec les path entre les states et les goals. multiple path. Et un seul path d'un node à la racine.



### 3.2.5 State and Nodes

- **State** représentation d'un configuration physique qui represente un state intermédiaire.
- **Node** data structure qui constitue le search tree

il peut y avoir plusieurs nodes avec le même state

### 3.2.6 Repeated States

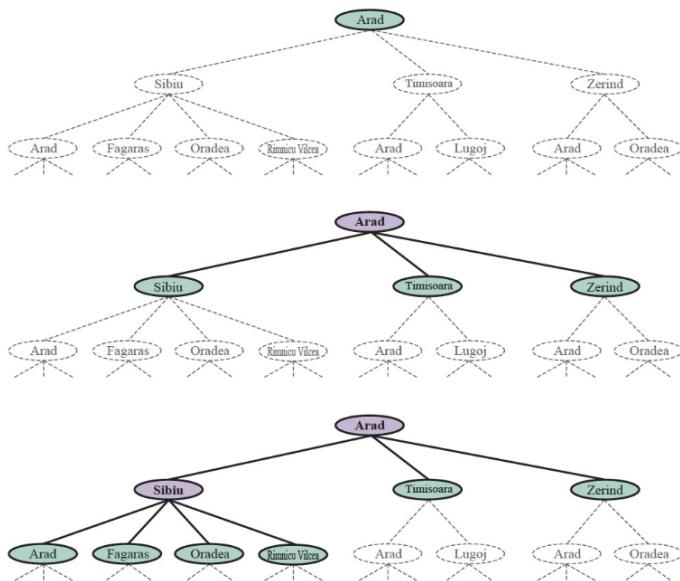
On va éviter de visiter des Nodes qui on déjà été visité avant, pour cela, on va utiliser la représentation en arbre (search tree) et on ne pas va *Expand* les nodes déjà visité.

## 3.3 Searching for Solutions

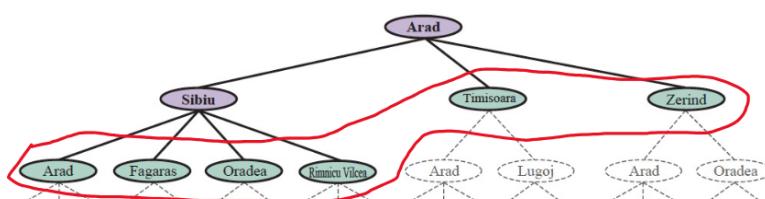
Trouver une solution est le fait de traverser un state space (graph ou tree) et du initial state au goal state avec un ensemble d'actions valide.

**Expanding** : appliquer chaque action légal au state actuel.

## Example



**Frontier** : Ensemble des node non-Expanded au quelle les ancêtres on déjà été visité. La frontière peut être représenté grâce a une Data structure Queue (lol zizi), On peut utiliser un Priority, LIFO, FIFO, ...



### 3.3.1 Stratégies

Il y a 2 types de recherches:

1. **uninformed search** où la seul information de l'agent est "suis-je le but final ?".
2. **informed search** où l'agent a les informations d'avant.

On peut évaluer ces recherche avec 4 critères:

- **Completeness** : l'agent trouve un solution s'il en existe au moins une.
- **Time complexity** : Souvent en terme du nombre de nodes générés/Expanded
- **Space complexity** : Nombre de node en mémoire
- **Optimality** : Trouve solution avec le cout minimal

On va utiliser 3 variables :

- **b** : Facteur de branchement maximal du Search tree
- **d** : Profondeur de la solution la moins couteuse
- **m** : Profondeur max de l'arbre (peut etre  $\infty$ )

## 3.4 Uninformed Search

	<b>BFS</b>	<b>UCS</b>	<b>DFS</b>	<b>DLS</b>	<b>IDS</b>	<b>BDS</b>
	level by level	cheap path	last depth	limited depth	iterative depth	bi direct
<b>Complete</b>	if <b>b</b> finite	if <b>sp</b> > 0	NO	if <b>l</b> ≥ <b>d</b>	YES	YES
<b>Optimal</b>	if <b>sc</b> = 1	YES	NO	if <b>l</b> == <b>d</b>	if <b>+= 1</b>	if <b>sc</b> = 1
<b>Time</b>	$b^d$	$b^{1+C/\epsilon}$	$b^m$	$b^l$	$b^d$	$b^{d/2}$
<b>Space</b>	$b^d$	$b^{1+C/\epsilon}$	$bm$	$bl$	$bd$	$b^{d/2}$
<b>Frontier</b>	FIFO	Priority	LIFO	LIFO	LIFO	FIFO

### 3.4.1 Breadth-First Search(BFS)

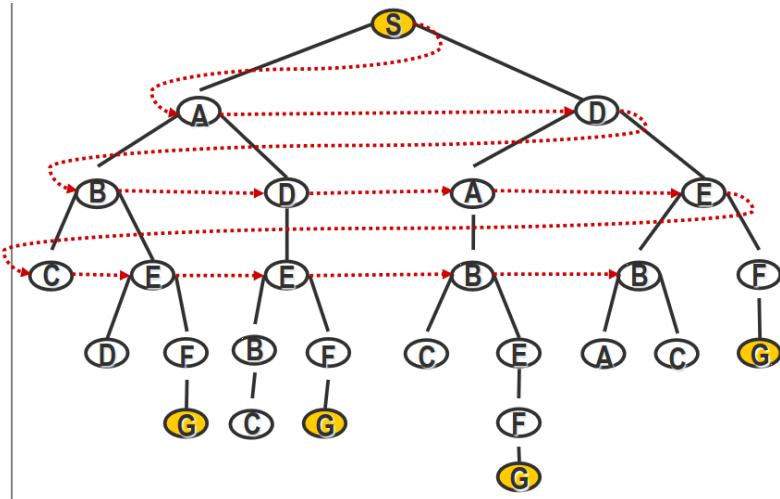
Le but est de vérifier par niveau de l'arbre. Problème la mémoire est vite remplie. On commence du root node et ensuite on passe au sub-nodes qui sont dans une FIFO Queue qui représente la frontière.

Il est **complet** si **b** est fini, il est **Optimal** si le cout part étapes est de 1 (souvent il n'est pas optimal)

La complexité temporelle et de  $\mathcal{O}(b^d)$ <sup>1</sup> et la complexité spatial est de  $\mathcal{O}(b^d)$

---

<sup>1</sup> $1 + b + b^2 + \dots + b^d + b(b^d - 1) = \mathcal{O}(b^{d+2}) = \mathcal{O}(b^d)$



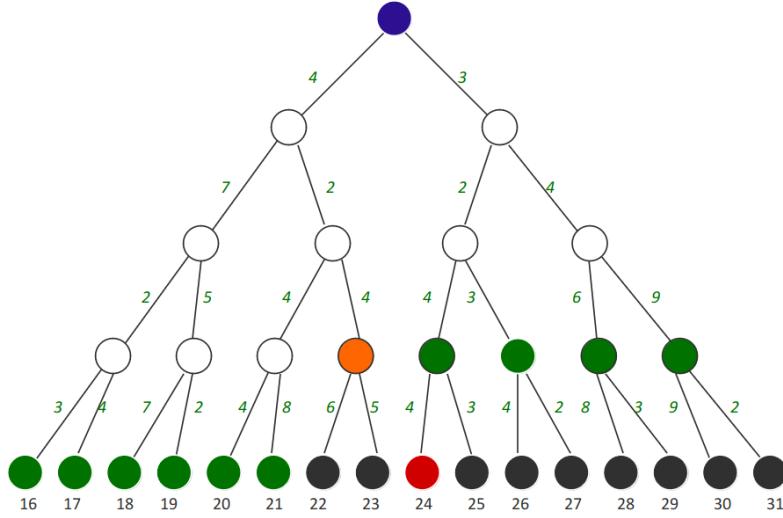
```

function BEST-FIRST-SEARCH(problem,f) returns a solution node or failure
  node  $\leftarrow$  NODE(STATE=problem.INITIAL)
  frontier  $\leftarrow$  a priority queue ordered by f, with node as an element
  reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node
  while not Is-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.Is-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s  $\leftarrow$  child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s]  $\leftarrow$  child
        add child to frontier
  return failure

```

### 3.4.2 Uniform-Cost search (UCS)

Comme le BFS sauf que le Node avec le cout le plus petit est exploré en premier. Il y a un cout pour passer d'un node à l'autre. La frontière est implémenté avec un Priority Queue.

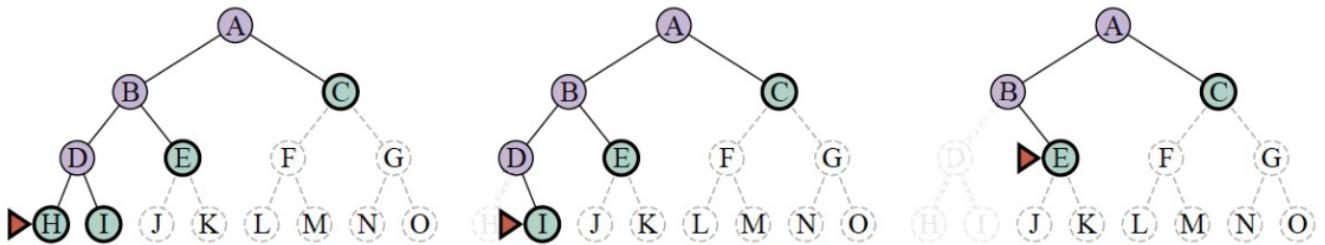


L'algo est **Complet** si le cout entre chaque step est strictement positif ( $\geq \epsilon$ ), **Time et Space complexity** =  $\mathcal{O}(b^{C^*/\epsilon})$  où  $C^*$  est le cout optimal de la solution, et il est **Optimal** car les node sont expanded par ordre croissant grâce à la Priority Queue.

### 3.4.3 Depth-first Search (DFS)

Concrètement, on visite toutes une branche de l'arbre jusqu'à arriver à la *Leaf*, si on ne trouve pas le goal on passe à la prochaine, etc. La Frontière est implémenté avec un Queue LIFO.

Cette algo n'est pas **Complete** car il peut tomber dans une boucle infinie, la **Complexity time** est de  $\mathcal{O}(b^m)$  et la **Space complexity** est de  $\mathcal{O}(mb)$ , **Il n'est pas optimal**



### 3.4.4 Depth-Limited Search

Pareil de DFS sauf que la on met une limite de profondeur(depth)  $l$ .

### 3.4.5 Iterative Deepening

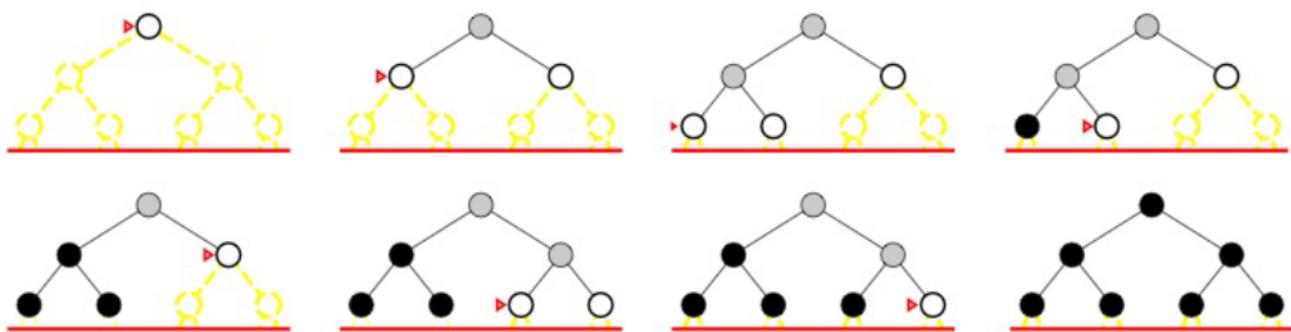
Pareil que DLS mais si  $l < d$  alors on ne trouveras jamais de solutions, donc on va incrémenter petit à petit  $l$

Plusieurs nodes sont visité plusieurs fois, mais ce n'est pas très grave car il n'y en a pas beaucoup.

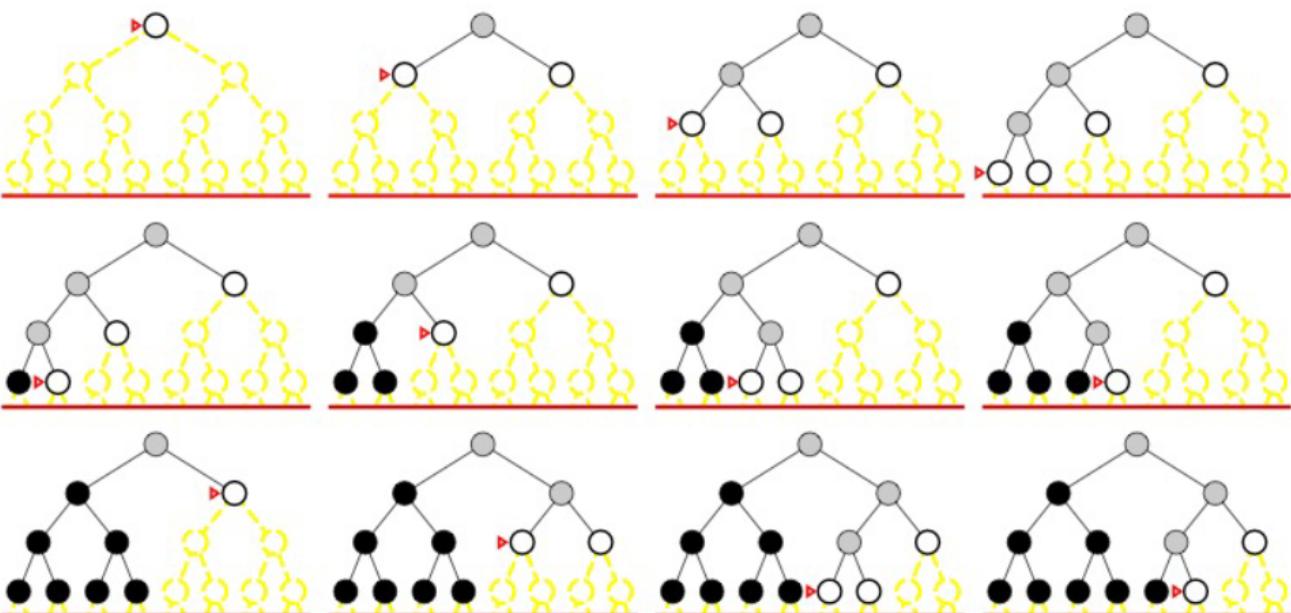
$l = 1$



$l = 2$



$l = 3$



Cet algo est **Complete** et **Optimal** si on incrémente de 1 à 1

la **Complexité temporel** est  $\mathcal{O}(b^d)$  et la **Space complexity** est  $\mathcal{O}(bd)$

### 3.4.6 Bidirectional Search

On commence du root jusqu'au Goal et aussi du Goal jusqu'au root et on stoppe quand les 2 s'intersecte. Il y quelque difficulté.

- Prédécesseurs du goal doivent être générés (pas toujours possible)
- Search doit être coordonnée entre les 2 recherches
- Problème si plusieurs Goal
- Tous les nodes doivent rester en mémoire

## 3.5 Informed Search

En utilisant des connaissances spécifiques au problème, trouver et/ou déduire des informations sur les States futurs et les paths futures.

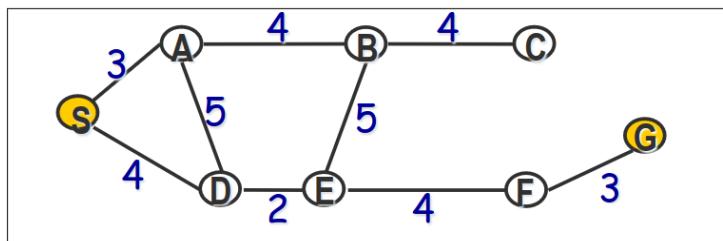
BFS est un algo de recherche où les nodes sont sélectionnés pour être *Expand* basé sur **une fonction d'évaluation**  $f(n)$ , cela donne un cout et on choisit le node avec le plus petit cout avec une Priority Queue.

Attention on calcule une évaluation et non pas la distance exacte

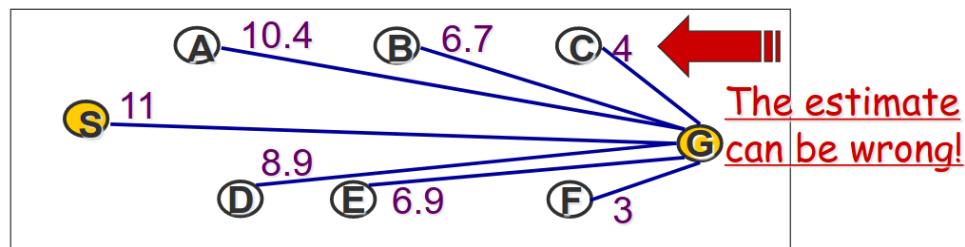
### 3.5.1 Heuristic Fonction

Noté  $h(n)$  est une fonction qui estime le cout d'un node vers le goal. c'est une approximation car on ne connaît pas le cout exacte.

#### Finding a route on a road map



Define  $h(n)$  = the straight-line distance from n to G



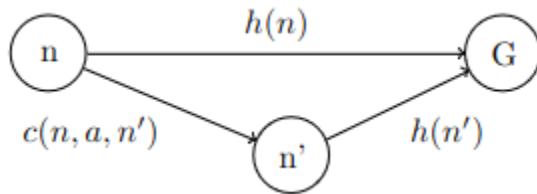
Il y a 3 types de fonction Heuristique :

- **Optimistic** : Admissible car elle pense que le cout pour résoudre le problème est inférieur au cout réel
- **Admissible** : Ne surestime jamais le cout pour atteindre le goal. Le cout qu'il estime est plus grand que le plus petit cout possible par rapport au node actuel.  $h(n) = 0$  si c'est un Goal state.
- **Consistent** : L'estimation de un node  $n$  au goal est plus petit que le cout réel de  $n$  a un nouveau node  $n'$  avec l'action  $a$  plus l'estimation de  $n'$  au goal

$$h(n) \leq c(n, a, n') + h(n') \quad (1)$$

un consistent heuristi est admissible

### 3.5.2 Triangle inequality



Chaque coté du triangle ne peut pas être plus long que la somme des autres cotés.

- **$h(n)$**  : Estime le cout entre  $n$  et  $G$
- **$C(n,a,n')$**  : cout pour aller en  $n'$  depuis  $n$  avec l'action  $a$
- **$h(n')$**  : cout estimé entre  $n'$  et  $G$

### 3.5.3 Monotonicity

Si  $h(n)$  est consistant alors  $f(n)$  le long d'un path quelconque n'est pas décroissant. Supposons que  $n'$  est un successeur de  $n$  alors :

$$\begin{aligned}
 g(n') &= g(n) + c(n, a, n') \\
 f(n') &= g(n') + h(n') \\
 &= g(n) + c(n, a, n') + h(n') \\
 c(n, a, n') + h(n') &\geq h(n) \\
 g(n) + c(n, a, n') + h(n') &\geq g(n) + h(n) \\
 &\geq f(n) \\
 f(n') &\geq f(n)
 \end{aligned}$$

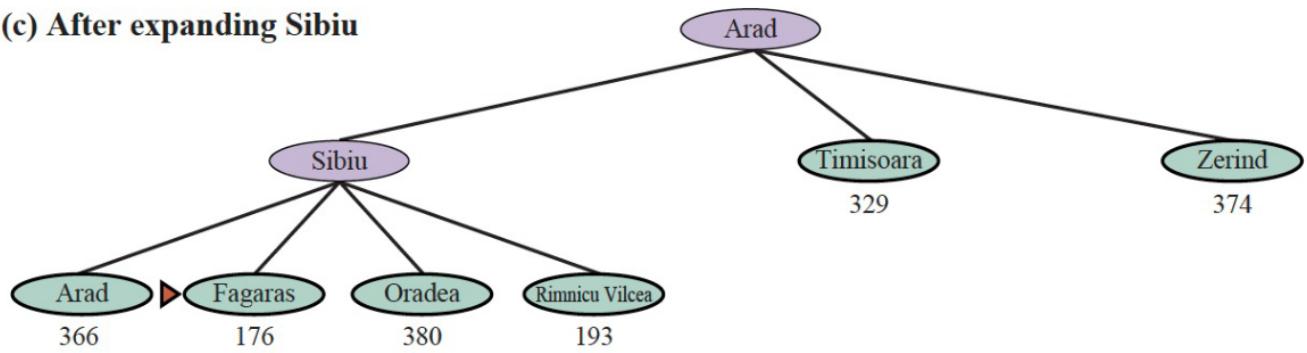
### 3.5.4 Greedy Best-First Search(GBFS)

Tente d'étendre le node qui est le plus proche du Goal.

Attention il n'est pas optimal et pas complet car il peut loop

Sa **Time complexity** et **Space complexity** est  $\mathcal{O}(b^m)$

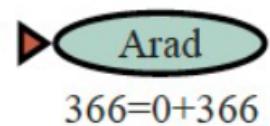
(c) After expanding Sibiu



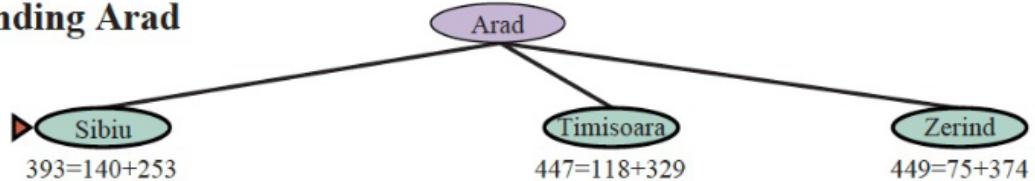
### 3.5.5 A\*

Fusion de GBFS et Uniform cost. Minimise le cout total estimé de la solution

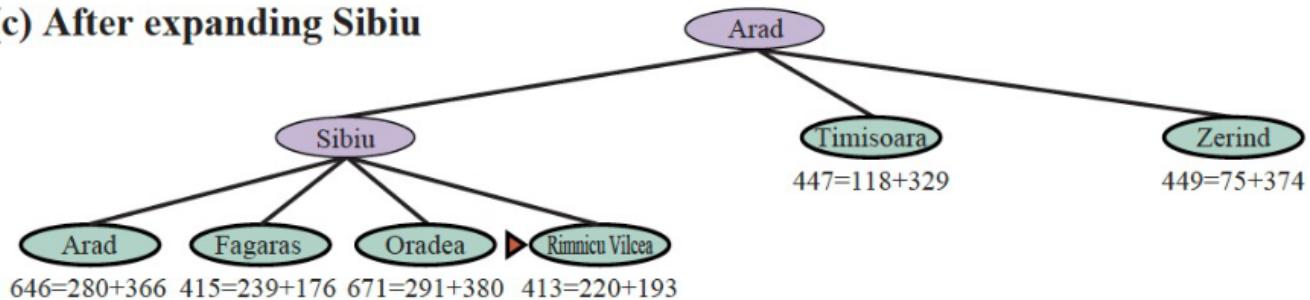
(a) The initial state



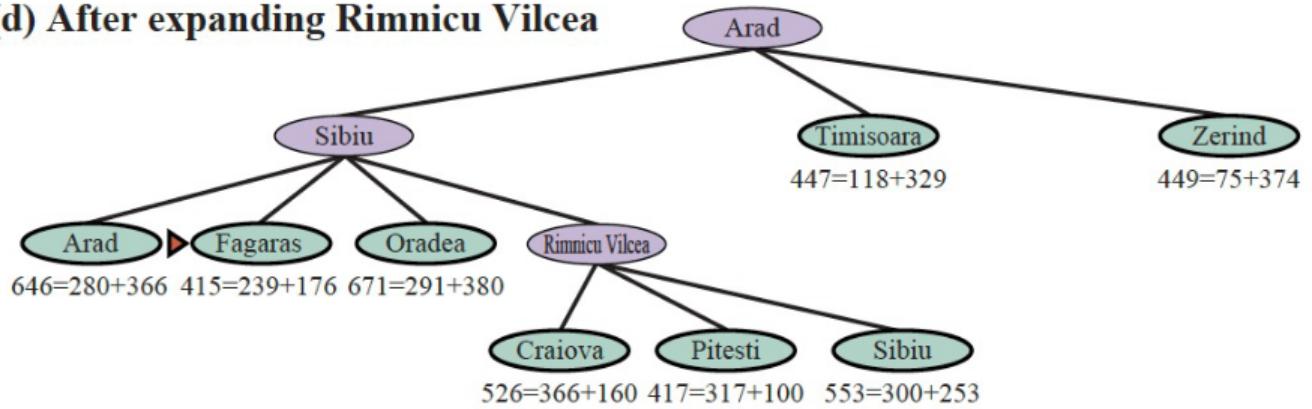
(b) After expanding Arad



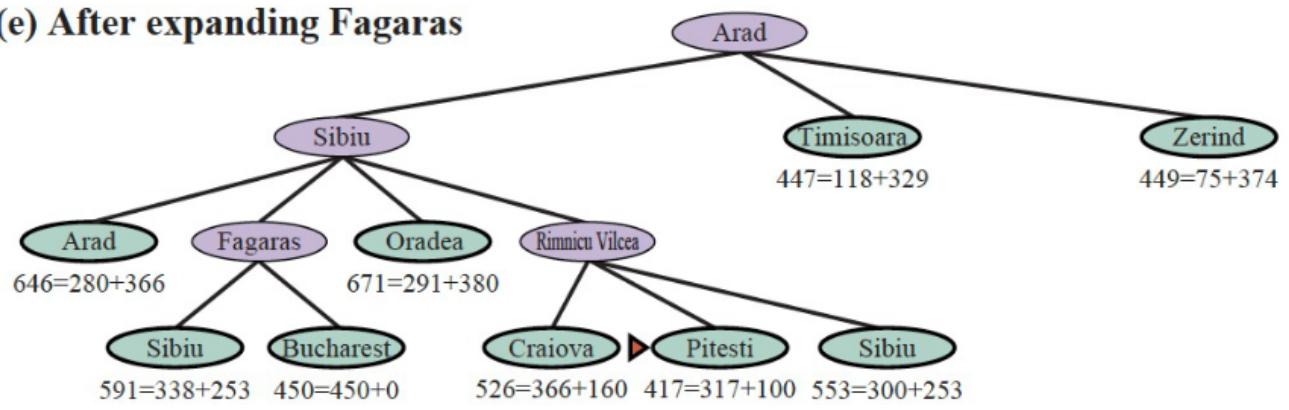
(c) After expanding Sibiu



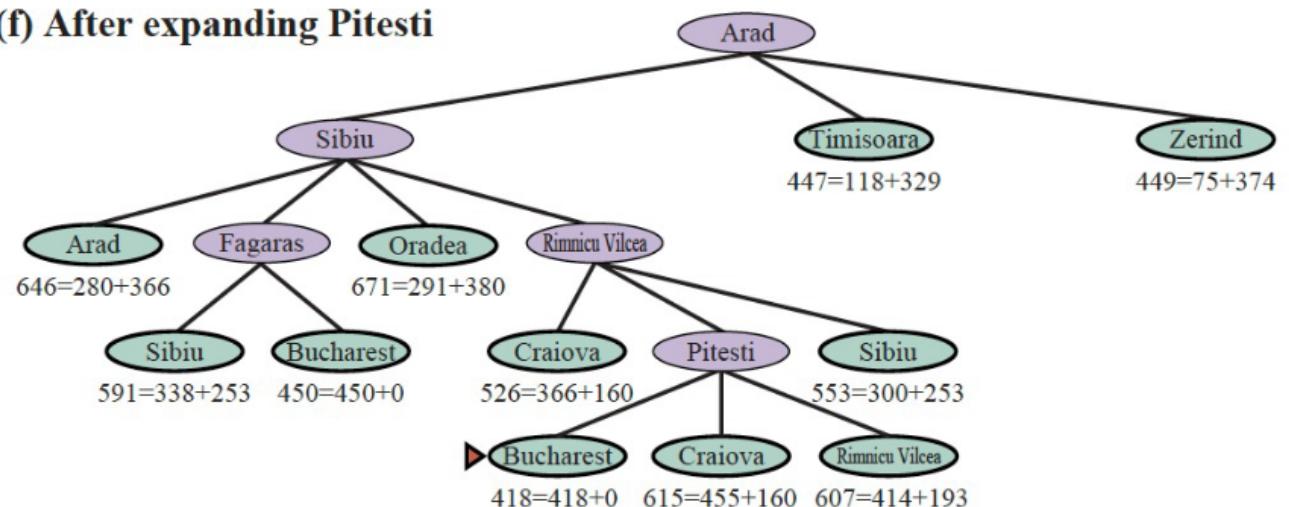
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



A\* utilise la distance et additionne la distance actuelle vers le goal.  $g(n)$  le cout de la solution vers le node n et  $h(n)$  la fonction heuristique

$$f(n) = g(n) + h(n) \quad (2)$$

$h(n)$  ne doit pas surestimer le cout vers le goal

**Time et Space complexity** =  $\mathcal{O}(b^d)$  un bon algo mais prend trop de mémoire.

L'algo est **Complete** si le cout ne peut pas être négatif, il est **optimal** si la fonction heuristique est admissible (tree) ou consistent(graph)

Propriété de A\* :

- Avec  $h(n)$  consistent, la séquence de nodes étendues par  $A^*$  est pas d'ordre décroissance  $def(n)$
- $A^*$  est optimal si  $h(n)$  est consistent
- $A^*$  étend tous les nodes avec  $f(n) \leq C^*$  ( $C^*$  est le cout optimal)
- $A^*$  étend au moins un node du contour de  $C^*$  avant de le trouver
- $A^*$  étend aucun node avec  $f(n) > C^*$

Pour la preuve que  $A^*$  est optimal → Voir slides

### 3.5.6 Memory-bounded heuristic search

On tente ici de réduire la mémoire que on a besoin, et on va utiliser les fonctions heuristiques pour nous aider.

**Iterative-deepening A\* (IDA\*)**  $A^*$  avec une limite de depth  $l$  qui est incrémenté itérativement.

IDA\* expend seulement les nodes avec  $f\text{-cost}() \leq l$  que les nodes non expand a la dernière itérations.

Ce n'est pas efficace quand le nombre des  $f\text{-cost}()$  sont élevés.

Propriétés :

- **Complet**
- **Time complexity** : exponentiel
- **Space complexity** : Linéaire
- **Optimal et  $h()$  consistent**
- **Efficace si absence de monotonie**

**Recursive best-first search (RBFS)** Pareil que DFS mais ne va pas au bout de la branche, mais utilise  $f_{limit}$  pour garder la trace de la valeur f-value du meilleur chemin alternatif a partir de l'ancêtre du node actuel.

- **Time complexity** : Dépend de  $h(n)$
- **Space complexity** :  $\mathcal{O}(bd)$  si  $h(n)$  est admissible

Utilise trop de mémoire

**SMA\*** Run avec  $A^*$  tant que la mémoire est pas pleine. Quand elle est plaine, on doit supprimer un node pour le remplacer, on va donc retirer le pire node (avec la plus grande valeur f-value)

Propriétés :

- **Complet** si assez de mémoire pour le path de la solutions la plus courte

- **Optimal** si assez de mémoire pour enregistrer le path de la meilleur solutions
- **Time complexity** : Pareil que A\* si assez de mémoire pour le tree
- **Space complexity** : utilise ce qui est disponible

### 3.6 Heuristic en pratique

On doit utiliser un heuristique **Admissible** et **Consistent**.

Pour comparer 2 heuristique, il faut comparer :

- le nombre de nodes générées
- le facteur de branchage  $b^*$

## 4 Local Search

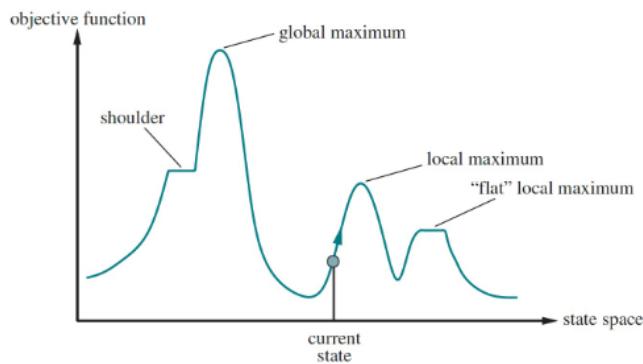
On ne s'occupe pas du path dans cette partie, notre but est d'avoir le Goal

- Utilise peu de mémoire
- Peut trouver des solutions dans une infinité de recherche
- Trouve des solutions raisonnables (pas optimal)

On améliore itérativement la solution actuelle. La prochaine solution est dans le voisinage (**neighborhood**) de la solution actuelle.

### 4.1 Optimisation problème

On a une **Fonction objective** que on peut maximiser ou minimiser. On veut trouver le Max/Min Global (par ex: distance, nb de queens sur le plateau ...). Le problème sont les Max/Min locaux et on peut rester coincé dessus.



### 4.2 Neighbourhood

#### Size

Si on décide de choisir un grand *voisinage*, on va avoir des paths plus petits pour la solution, mais on va avoir besoin de plus de temps pour parcourir toutes les possibilités. Il faut trouver un juste milieu entre longueur des paths et temps d'explorations

#### Connectivity

Pour chaque solution, il y a un path à une solution optimale. Cela a 2 avantages :

- Pas besoin de recommencer la stratégie (on ne bloque jamais)
- Nécessaire pour la propriété de convergence

#### Contraintes

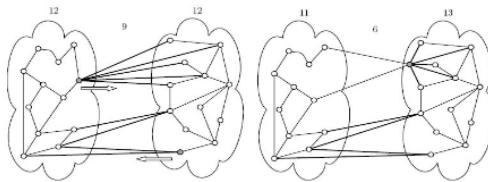
Il y a 2 approches possibles pour combiner faisabilité avec requièremment optimal

- Approche 1
  - Maintenir la faisabilité à tout moment
  - Explorer seulement les solutions faisables

- Approche 2
  - Ne pas maintenir la faisabilité à tout moment ; assouplir un sous-ensemble de contraintes
  - Explorer un plus grand *Search Space*
  - Conduire à chercher des solutions faisable et de qualités

Un exemple car c'est un peu trop théorique : Graph partitioning S

- Balanced property not maintained
  - $N'(V_1, V_2) = \{ (V_1 \setminus \{a\}, V_2 \cup \{a\}) \mid a \in V_1 \}$   
 $\cup \{ (V_1 \cup \{a\}, V_2 \setminus \{a\}) \mid a \in V_1 \}$



- Objective function combines optimality and feasibility:  
e.g.
  - $f(V_1, V_2) = \alpha \# \{(v, w) \in E \mid v \in V_1 \wedge w \in V_2\} + \beta (\#V_1 - n/2)^2$

## 4.3 Heuristic et Metaheuristics

**Heuristics** Se concentre sur choisir la prochaine solution et conduire la recherche avec des optimum local basé sur l'information locale (solutions actuelle et le neighbourhood). utilise peu de mémoire

**Metaheuristics** Tente d'échapper les optimum locaux pour trouver le global optimum basé sur la collecte d'information avec les exécutions faites avant. Prend plus de mémoire car learning derrière.

**Systematic heuristics** Exploration (partiellement possible) du voisinage pour déterminer solution suivante.

## 4.4 Hill Climb Algo

```

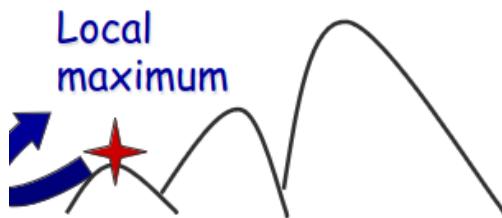
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current  $\leftarrow$  problem.INITIAL
  while true do
    neighbor  $\leftarrow$  a highest-valued successor state of current
    if VALUE(neighbor)  $\leq$  VALUE(current) then return current
    current  $\leftarrow$  neighbor
  
```

On choisit un état de départ, et on avance dans la direction des valeurs qui augmentent et on stop l'itération quand on ne peut plus aller plus loin. Cette algo va rapidement à la meilleure solution mais à plusieurs problèmes :

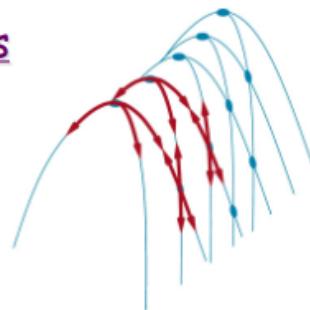
- S'arrête si c'est un Max/Min local
- Si on a un plateau, l'algorithme se stop (on peut mettre une limite d'itération si on tombe dessus)
- Si on a une séquence de Max/Min locaux alors on a du mal à naviguer

Hill climb dépend énormément de la forme du state space et vu que il ne fera jamais de down-hill, si il tombe sur un maxLocal il reste bloqué

### Foothills:



### Ridges



#### 4.4.1 Variantes

Il existe plusieurs variantes :

- **Stochastic hill climbing** : Choisis en mode random parmi les mouvements qui monte
- **First-choice hill climbing** : On choisit le premier bon successeur que l'on trouve, pratique si le nombre de successeur est large
- **Random restart** : On commence de plusieurs points random

## 4.5 Random Walk

On sélectionne en mode random un élément du neighbourhood et on décide si on doit l'accepter comme la prochaine solution. Il y a plusieurs approches possibles :

- Amélioration aléatoire
- Heuristique de Metropolis : accepter de dégrader la solution

## 4.6 Simulated annealing

Algorithme qui combine Hill climb, un scheduler et un metropolis step.

- Toujour avance en montant (uphill) si possible
- Parfois aller en descendant (downhill)

On commence avec beaucoup de random et on reduit graduellement l'intensité du random dans l'algo (Analogie avec la métalurgie qui chauffe du fer et le laisse refroidir lentement)

Opérations :

1. Toujours aller uphill si possible
2. parfois (random) aller downhill (quand la température est haute faut laisser redescendre)
3. C'est assurée d'être optimal si réduction lente.

## 4.7 Local Beam Search

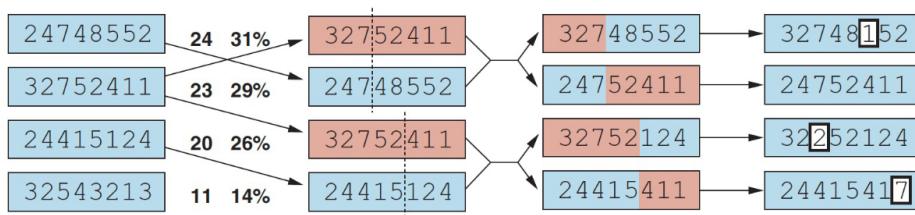
Mélange de hillclimb et annealing en gardant un seul state en mémoire. L'idée est de :

1. garder  $k$  states en mémoire
2. chaque step, générer tous les successeurs des  $k$  states
3. On stop si un state est un goal
4. Sinon, on sélectionne les  $k$  meilleur successeurs de la liste complète de tous les successeurs

L'algo souffre d'un manque de diversité car les states converges rapidement vers la régions

## 4.8 Genetic Algorithms (GAs)

Basée sur la théorie de l'évolution (woké en panique). On commence avec  $k$  supposition de début (populations), chaque individus de la population a un string de longueur fixe qui représente son gène. On produit la prochaine génération en **reproduisant** les individus de la populations en ajoutants quelque random mutations. Chaque individus possède un score de fitness

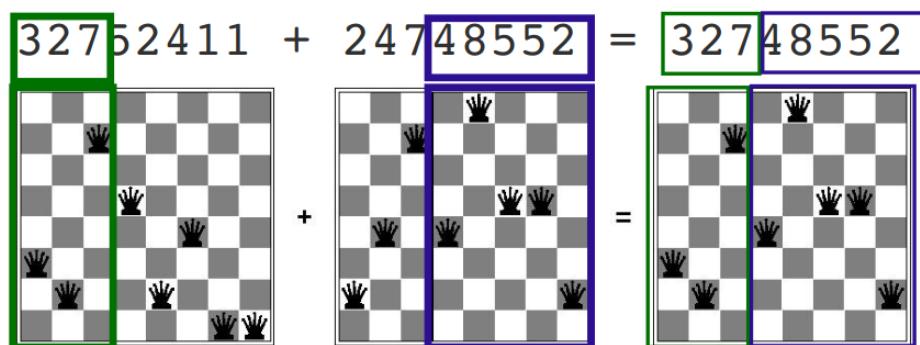
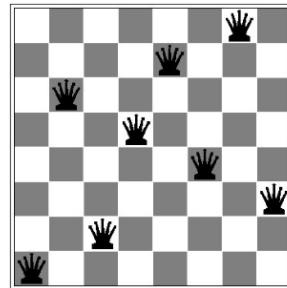


Exemple 8-queens:

Les mutations randoms peuvent amener à une bonne situation et permet d'explorer de nouvelle partie du search space

un problème avec cet algo est que le crossover est pas applicable à tous les problèmes.

16257483



- Individuals with a high fitness have a higher chance to contribute to the next generation (reproduction)

```

function GENETIC-ALGORITHM(population,fitness) returns an individual
repeat
    weights  $\leftarrow$  WEIGHTED-BY(population,fitness)
    population2  $\leftarrow$  empty list
    for i = 1 to SIZE(population) do
        parent1, parent2  $\leftarrow$  WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child  $\leftarrow$  REPRODUCE(parent1, parent2)
        if (small random probability) then child  $\leftarrow$  MUTATE(child)
        add child to population2
    population  $\leftarrow$  population2
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
    n  $\leftarrow$  LENGTH(parent1)
    c  $\leftarrow$  random number from 1 to n
    return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))

```

## 4.9 Tabu search Metaheuristics

Sélectionne le meilleur voisin **qui a pas encore été visité**. Un problème est que c'est difficile de garder la trace de tous les nodes déjà visités. Une solution est de garder en mémoire un suffixe de la séquence de node visité.

## 4.10 Intensification vs. Diversification

	Intensification	Diversification
<b>Goal</b>	Accroître la recherche dans les domaines prometteurs	Explorer de nouveaux endroits
<b>Risk</b>	Convergence prématuée (Max/Min Local)	Convergence vers l'optimal peut prendre du temps
<b>Mean</b>	Favorise les bonnes solutions	Choix probabiliste des solutions

## 4.11 Other Local Search

- Variable Neighborhood Search
- Guided Local Search
- Adaptive Local Search
- Ant Colony Optimization
- Statistic Local Search <sup>2</sup>

---

<sup>2</sup>voir slides

## 5 Constraint Satisfaction Problem (CSP)

Un CSP est composé de 3 composant :

- **X** : ensemble des variables  $X_1, X_2, \dots, X_n$
- **D** : ensemble des domaines  $D_1, D_2, \dots, D_n$ , un pour chaque variables
- **C** : ensemble des contraintes qui spécifie les combinaison de valeur possible

Les éléments de **C** consiste en une paire  $\langle Scope, rel \rangle$  ou *scope* est un tuple de toutes les variables impliqué et *rel* leurs relations

Une solution consiste en une affectation qui ne viole aucune contrainte

Exemple avec les 8-queens car trop dur a comprendre : On veut placer 8 reines sur un échquier de sorte a ce que elle ne puisse pas se chevaucher.

On a donc nos variables  $X_i$  qui sont les positions (colone) de la reine dans la ligne i ( $1 \leq i \leq 8$ )<sup>3</sup>

Notre domaine de  $X_i$  est alors  $\{1, 2, 3, 4, 5, 6, 7, 8\}$

Nos contraintes sont que

- 2 reines ne peuvent pas être sur la même colonne ( $X_i \neq X_j$ )
- 2 reines ne peuvent pas être sur la même diagonale ( $X_i - X_j \neq i - j \wedge X_i - X_j \neq j - i$ )

On a donc 8 variables avec 84 contraintes (je vais pas les lister va voir les slides)

On peut représenter cela sous la forme d'un graph où les nœuds représentent les variables et les arêtes les contraintes

Il y a différent type de CSP qui dépend du domaine :

- Discret et domaine fini
- Discret et domaine infini
- Continus et domaine infini

Il y a aussi différentes types de contrainte :

- Contrainte Unary : sur seulement une variable (SA ≠ Green cf exemple Color mapping Australie slide)
- Contrainte binaire : Sur 2 variables ( $x + y \leq 12$ )
- Contrainte High-order : Sur plusieurs variables (peuvent être transformé en binaire avec des variables additionnel)

Ici on considère seulement les CSP avec contrainte binaire

---

<sup>3</sup>On sait que il ne peut y avoir que 1 seule reine par ligne et colonne car sinon elles se chevauchent, on est donc obligé d'utiliser toutes les lignes et colonnes par une seule reine

## 5.1 Constraint Propagation

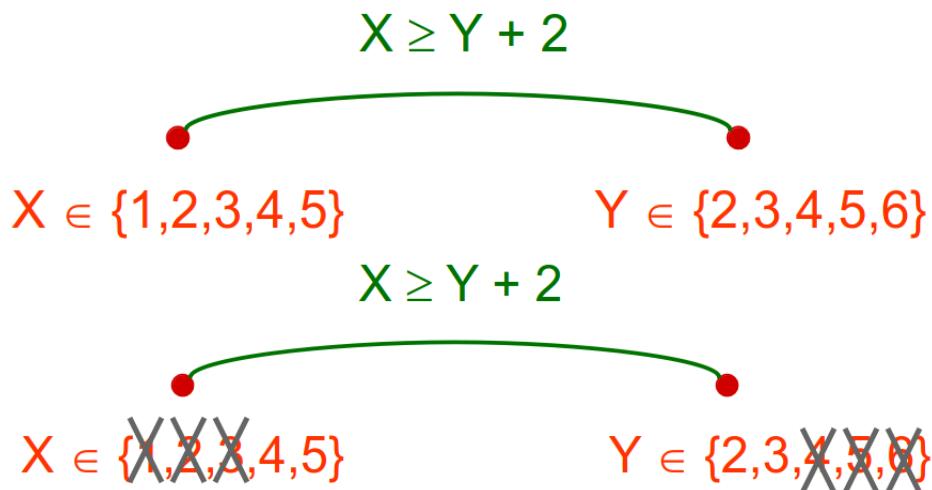
Quand une variable est assignée, on propage ça aux autres contraintes et on réduit le search space<sup>4</sup>  
Le but est de réduire le plus possible le search space et trouver un CSP équivalent à l'original mais avec un domaine plus petit

Il y a 2 méthodes pour réduire le domaine en considérant les contraintes comme **Locale**

- **Arc consistency:** on considère des contraintes entre 2 variables

$X_i$  est un arc consistant de  $X_j$  si pour toutes les valeurs dans  $D_j$  il y a une valeur dans  $D_i$  qui satisfait la contrainte binaire  $(X_i, X_j)$

$$X, Y \in \{0, 1, \dots, 9\}, X = Y^2 \rightarrow X \in \{0, 1, 2, 3\} \wedge Y \in \{0, 1, 4, 9\} \quad (3)$$



- **Path Consistency:** on considère des contraintes entre  $n$  variables

**Bound consistency:** Une forme plus faible de *Arc consistency*. On considère seulement les limites du domaine, donc on retire toujours les valeurs sur les limites et jamais au milieu

**Global consistency:** Pour les contraintes spéciales, c'est géré avec un "ad hoc specialized method". pour un nombre arbitraire de variable

### 5.1.1 Arc-consistency algorithm

Avec une complexité de  $\mathcal{O}(n^2 d^3)$

un peu d'explication :

1. on commence un queue avec un arc initial
2. temps que il y a  $(X_i, X_j)$  dans la queue
3. si *REVISITE*
4. alors pour tous les valeurs voisines de  $X_i$  sans compter  $X_j$ , on ajoute  $(X_k, X_i)$  dans la queue

---

<sup>4</sup>Voir exemple slide 22-27 Chap 5

```

function AC-3(csp) returns false if an inconsistency is found and true otherwise
  queue  $\leftarrow$  a queue of arcs, initially all the arcs in csp

  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{POP}(\text{queue})$ 
    if REVISE(csp,  $X_i$ ,  $X_j$ ) then
      if size of  $D_i = 0$  then return false
      for each  $X_k$  in  $X_i.\text{NEIGHBORS} - \{X_j\}$  do
        add  $(X_k, X_i)$  to queue
    return true

function REVISE(csp,  $X_i$ ,  $X_j$ ) returns true iff we revise the domain of  $X_i$ 
  revised  $\leftarrow$  false
  for each  $x$  in  $D_i$  do
    if no value  $y$  in  $D_j$  allows  $(x,y)$  to satisfy the constraint between  $X_i$  and  $X_j$  then
      delete  $x$  from  $D_i$ 
      revised  $\leftarrow$  true
  return revised

```

*REVISE* supprime toutes les valeurs dans  $X_i$  tels que elles ne satisfont pas les contrainte entre  $X_i$  et  $X_j$ .

## 5.2 Backtracking Search for CSPs

### 5.2.1 Incremental formulation

- le **Init state** est une affectation vide
- La **fonction de succession** assigne une valeur a une variable non assignée si aucune contrainte est violé
- Le **Goal test** verifie si l'affectation actuelle est complete
- Le **Path cost** Valeur constante pour chaque etapes qui indique le coup/profondeur du path

L'arbre de recherche a une profondeur de  $n$  (nombre de variables) et le nombre de state possible est  $\mathcal{O}(d^n)$  avec  $d$  la taille du domaine.

La taille de l'arbre est de  $\mathcal{O}(n!d^n)$  en raison que le premier niveau est  $n.d$  , le deuxieme  $(n-1).d$  etc.

CSP seach algo doit seulement considerer une seule variables pour le successeur a chaque node car avec la **commutativité** quand on assigne une valeur a une variable, nous obtenons la même affectation partielle quel que soit l'ordre

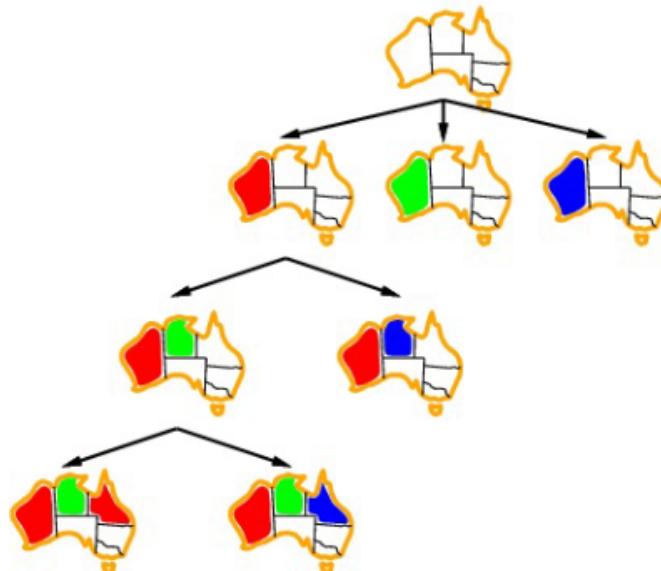
### 5.2.2 Backtracking search

c'est une recherche par DFS qui choisit une valeur pour une variables et qui reviens en arrière quand une variable plus de valeur qui ne viole pas les contrainte disponibles

# Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
    return BACKTRACK(csp, {})

function BACKTRACK(csp, assignment) returns a solution or failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp, assignment)
    for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences  $\leftarrow$  INFERENCE(csp, var, assignment)
            if inferences  $\neq$  failure then
                add inferences to csp
                result  $\leftarrow$  BACKTRACK(csp, assignment)
                if result  $\neq$  failure then return result
                remove inferences from csp
                remove {var = value} from assignment
    return failure
```



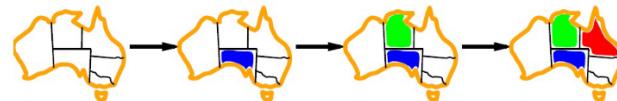
### 5.2.3 Variable ordering

On va optimiser le fait de choisir des variables et des valeurs le tous afin de réduire la recherche

- Minimum Remaining Value (MRV) : on choisit la variable avec le moins de valeur légale
- Degree heuristic : on choisit la variable qui entraîne le plus grand nombre de contraintes (réduit le facteur de branchement)
  - Minimum Remaining Value



- Most constrained variable



### 5.2.4 Interleaving search and inference

On veut réduire le *Search Space* pour être plus efficace. Lorsque nous choisissons une valeur pour une variable, nous avons une toute nouvelle possibilité de déduire de nouvelles réductions de domaine sur les variables voisines.

**Forward checking** : Quand  $X$  est attribué à  $v$ :

- On regarde à chaque variables  $T$  non assignées qui sont connectées à  $X$
- Retirer du domaine  $Y$  la valeur incompatible avec la valeur choisie pour  $X$ .

### 5.2.5 Intelligent Backtracking

Au lieu de faire un bon en arrière quand on arrive bloqué, on back up à la variable responsable du problème (calculable avec forward checking). On garde un **conflict set** qui garde pour chaque valeur la liste des assignments qui sont en conflit avec la valeur.

## 5.3 Local Search for CSP

### 5.3.1 Complete state formulation

- **Initial state** : une valeur à chaque variables
- **Succesor function** : change la valeur d'une variable
- **Goal test** : check si l'assignement actuelle est consistant et complet
- **Path cost** : Valeur constante par étapes

Quand on choisit une valeur pour les variables, on choisit avec le minimum de conflit possible

```

function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up

  current  $\leftarrow$  an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var  $\leftarrow$  a randomly chosen conflicted variable from csp.VARIABLES
    value  $\leftarrow$  the value v for var that minimizes CONFLICTS(csp, var, v, current)
    set var = value in current
  return failure

```

### 5.3.2 Min Conflict

## 5.4 Structure of the problems

### 5.4.1 Independent subproblems

On divise le CSP en sous-problème indépendant entre eux pour ainsi former  $k$  CSP différents. On peut les résoudres séparément et la solution est donc l'unions des différentes solutions.

Avec ça, le complexité temporelle est de  $\mathcal{O}(d^{n/k}k)$

### 5.4.2 Tree-structure Subproblems

Si le tree est *directed arc consistency* il peut etre résolué en  $\mathcal{O}(nd^2)$

Pour le transformer en tree :

- : retire des nodes :
  - choisis un sous-ensemble  $S$  de taille  $c$
  - pour chaque assignement possible aux variables de  $S$  qui s'attisfaisent toutes les contraintes dans  $S$ ,
    - \* Retire dans le domaine, n'importe quelle variables qui sont incohérente avec l'assignement de  $S$ ,
    - \* Si CSP a une solution, on la retourne avec assignement  $S$
- Grouper les nodes:
  - chaque variable apparait au moins une fois dans un sous-problème
  - 2 variables connectées par une contrainte doivent etres ensembles dans au moins un sous-problème
  - si une variables apparaît dans 2 sous-problèmes, alors elle doit etre dans tous les sous-problèmes sur le path pour connecté tous les sous-problème.

# 6 Adversarial Search and Games

## 6.1 Game

Un environnement multiagent peut etre vu comme un jeu et donc un problème résolut par recherche.

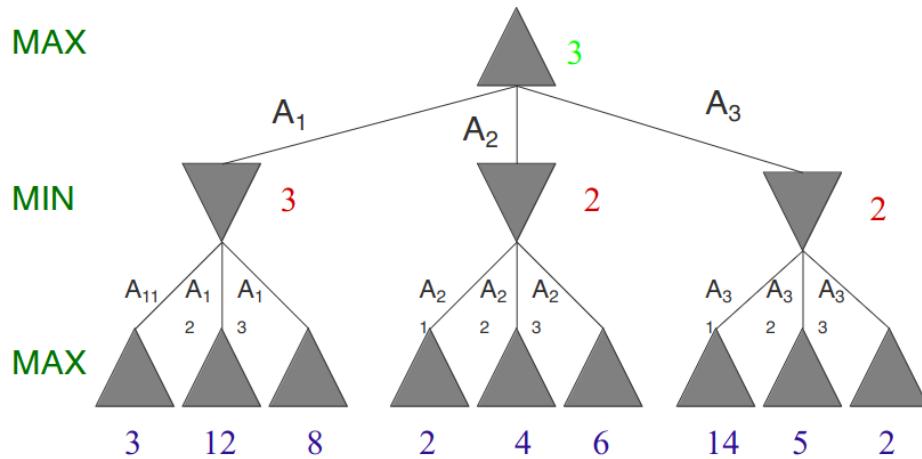
On va parler de **Zero-sum** qui veut dire que si un mouvement avantage un joueur alors il n'avantages pas l'autre joueur. le jeux possède un seul gagnant et un seul perdant, pas de neutre ou 2 gagnants ou 2 perdants.

Un jeux est défini par :

- **Initial state** :  $S_0$  positions de début du jeu
- **To-Move(s)** : Le joueur qui doit jouer au state  $s$
- **Actions(s)** : ensemble des mouvement légaux qu'il a fait en state  $s$
- **Result(s,a)** : modèle de transition, state final après actions  $a$  sur le state  $s$
- **Is-Terminal(s)** : test terminal du state  $s$ , si la partie est finie
- **Utility(s,p)** : valeur numérique du résultat du jeux en au state  $s$  pour le player  $p$

*Initial states, Actions(s) et Results(s,a) definisse le State space search et le tree du jeux.*

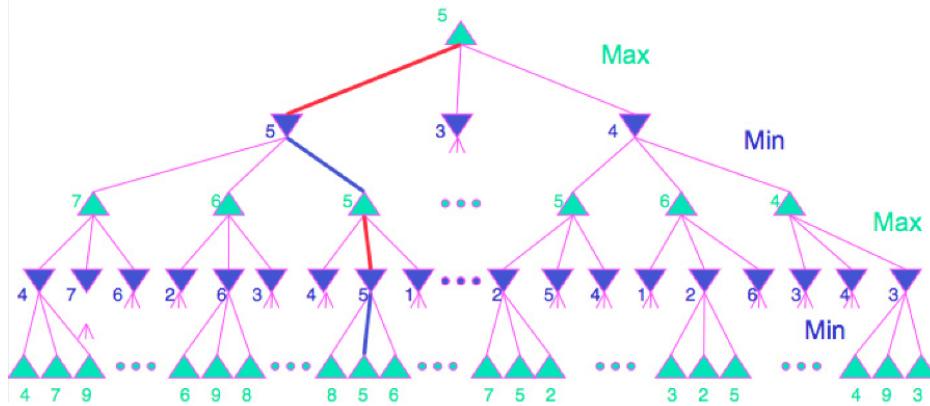
## 6.2 MinMax Algo



Optimal pour les jeux déterministe et a 2 joueurs.

Chaque niveau représente à un joueur de jouer en fonction du state, un joueur (MAX) veut maximiser la valeur le deuxième joueur (MIN) lui veut minimiser. Si un des joueurs ne joue pas de maniére optimale alors l'autre joueur va encore mieux en profitant et en ressortir un meilleur score.

MinMax génère le tree entièrement et applique une fonction d'utilité sur les derniers state (les leafs de l'arbre). Ensuite on remonte l'arbre en choisissant le MIN ou le MAX value en fonction du niveau de l'arbre (tours du joueurs). Le node final au dessus représente le meilleur mouvement possible




---

```

function MINIMAX-SEARCH(game, state) returns an action
    player  $\leftarrow$  game.TO-MOVE(state)
    value, move  $\leftarrow$  MAX-VALUE(game, state)
    return move

```

---

```

function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move  $\leftarrow -\infty$ 
    for each a in game.ACTIONS(state) do
        v2, a2  $\leftarrow$  MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move  $\leftarrow$  v2, a
    return v, move

```

---

```

function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v, move  $\leftarrow +\infty$ 
    for each a in game.ACTIONS(state) do
        v2, a2  $\leftarrow$  MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move  $\leftarrow$  v2, a
    return v, move

```

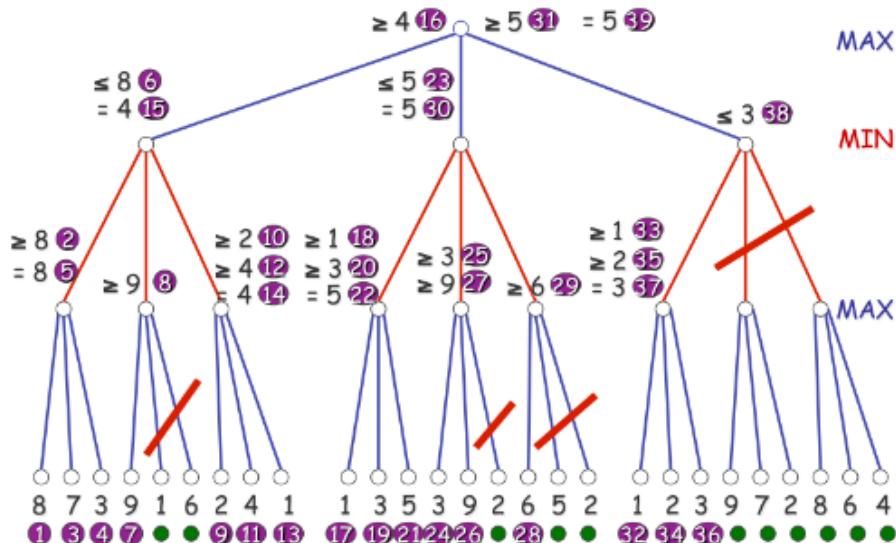
Il est possible de modifier l'algorithme pour avoir plus de 2 joueurs, il suffit de garder un vecteur à chaque node ou les éléments du vecteur sont les utilités de chaque joueur. Chaque joueur va choisir le mouvement qui maximise l'utilité

Propriétés :

- Basé sur DFS (récursif)
- Complete si l'arbre est fini
- Time complexity :  $\mathcal{O}(b^m)$
- Space complexity :  $\mathcal{O}(bm)$

### 6.3 Alpha-Beta pruning

On cherche à "élaguer" l'arbre afin de retirer les branches au quelle nous sommes sur que il n'y a pas de meilleures valeurs et donc inutile. Il n'affecte pas le résultat final



```

function ALPHA-BETA-SEARCH(game, state) returns an action
  player  $\leftarrow$  game.TO-MOVE(state)
  value, move  $\leftarrow$  MAX-VALUE(game, state,  $-\infty$ ,  $+\infty$ )
  return move



---


function MAX-VALUE(game, state,  $\alpha$ ,  $\beta$ ) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v  $\leftarrow -\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MIN-VALUE(game, game.RESULT(state, a),  $\alpha$ ,  $\beta$ )
    if v2  $>$  v then
      v, move  $\leftarrow$  v2, a
       $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
    if v  $\geq \beta$  then return v, move
  return v, move



---


function MIN-VALUE(game, state,  $\alpha$ ,  $\beta$ ) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v  $\leftarrow +\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MAX-VALUE(game, game.RESULT(state, a),  $\alpha$ ,  $\beta$ )
    if v2  $< v$  then
      v, move  $\leftarrow$  v2, a
       $\beta \leftarrow \text{MIN}(\beta, v)$ 
    if v  $\leq \alpha$  then return v, move
  return v, move

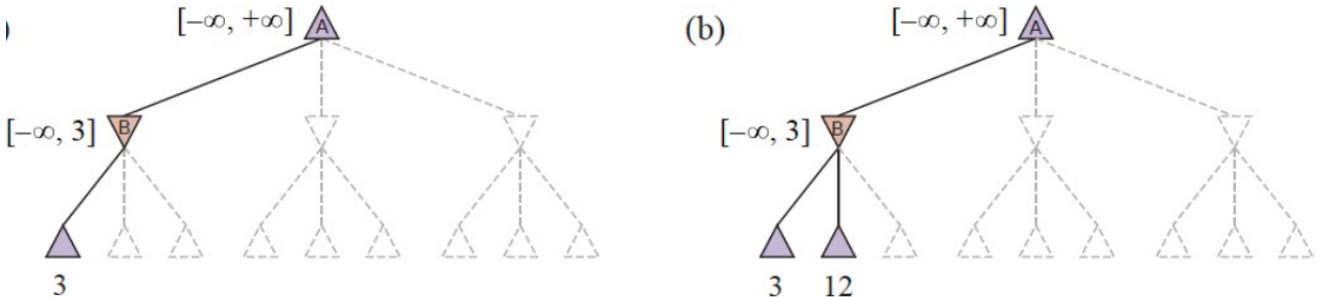
```

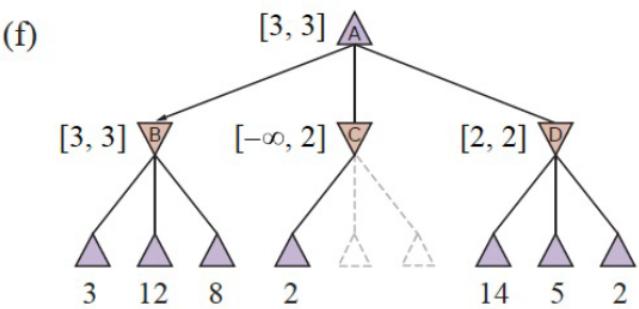
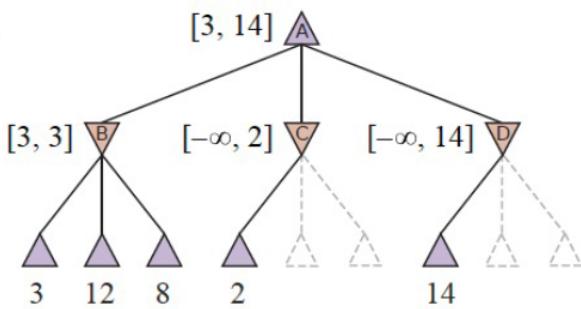
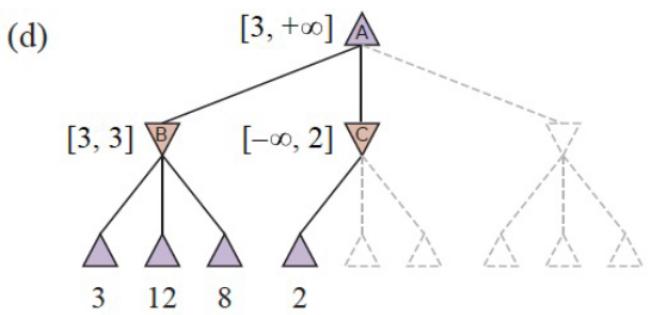
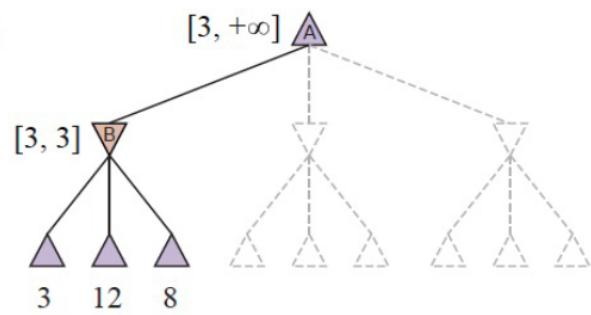
$\alpha$  : Valeur du meilleur (plus grande) choix trouvé d'ici la depuis les path visité pour MAX

$\beta$  : Valeur du meilleur (plus petit) choix trouvé d'ici la depuis les path visité pour MIN

Si les sucesseur de l'arbre sont idéalement visité, la complexité temporelle est alors de  $\mathcal{O}(b^{d/2})$  et si visité random  $\mathcal{O}(b^{3d/4})$

Voila un autre exemple de prunning car un peu dure a comprendre parfois :





### 6.3.1 Move ordering

La disposition des nodes est idéal si le meilleur node est le plus à gauche. On peut tenter de réarranger les nodes afin d'avoir le meilleur le plus à gauche, et trouver le **Killer move**

## 6.4 Imperfect Decisions

En pratique il est souvent impossible de faire une research complete car trop grand. Ce que on fait, on fait une search dans seulement un partie du graph. Cela requière une fonction cutoff-test qui indique quand stopper la génération du graph. Mais donc on ne peut pas savoir la valeur de l'utilité des derniers node juste avant le cutoff, on va donc devoir l'estimer avec une fonction heuristique.

### 6.4.1 Evaluation Function

- **H-Minimax(s,d) =**

$$\begin{aligned}
 & \text{Eval}(s) && \text{if Is-cutoff}(s,d) \\
 & \max_{a \in \text{Actions}(s)} \text{H-Minimax}(\text{Result}(s,a), d+1) && \text{if To-Move}(s) = \text{MAX} \\
 & \min_{a \in \text{Actions}(s)} \text{H-Minimax}(\text{Result}(s,a), d+1) && \text{if To-Move}(s) = \text{MIN}
 \end{aligned}$$

La fonction doit être :

- Consistent avec la fonction d'utilité

- équilibré entre précisions et temps de calcul
- Dois refléter la réel chance de gagné
- Fonction linéaire avec du poid sont souvent

#### 6.4.2 Cutting off the search

On doit un peu changer le code de alpha-Beta-Search qui doit appeler Eval quand il y a un cutoff if game.Is-Cutoff(state, depth) then return EVAL(state, player), null

La fonction d'évaluation doit être appliquée uniquement à la position quiescentes, c'est-à-dire stables. (qui ne risquent pas de connaître de fortes variations de valeur dans un avenir proche).

Il y a l'**effet d'horizon** : Nous pouvons retarder les catastrophes, mais nous ne pouvons pas les prévenir. Cela peut arriver à cause d'une coupure car l'horizon n'est pas assez profond. Si un agent est confronté à un mouvement qui cause de graves dommages et qui est inévitable à la fin, il essaiera de l'éviter le plus longtemps possible. l'éviter le plus longtemps possible. Au bout du compte, il perdra autant, voire plus.

#### 6.4.3 Forward pruning

Quelques moves à certains nodes sont "élagué" direct sans considérations directe, il coupe les moves "mauvais" mais il possible que plus tard se soit le move qui nous fait gagner. Avec le **Beam search**, on considère un ensemble de  $n$  mouvement plutôt que tous les mouvements possibles, on réduit de beaucoup de temps de calcule "inutile"

#### 6.4.4 Search VS. Lookup

Beaucoup utilise des table lookup (table avec les actions à faire en fonction de la situation) pour le début de la partie. Ca sert à rien de faire un arbre de millions de nodes pour une personne qui a 2/3 de chance de faire E4 au début.

Les lookup tables sont donc utilisé en début et fin de game car souvent la même choses avec des pattern et des solutions déjà connue.

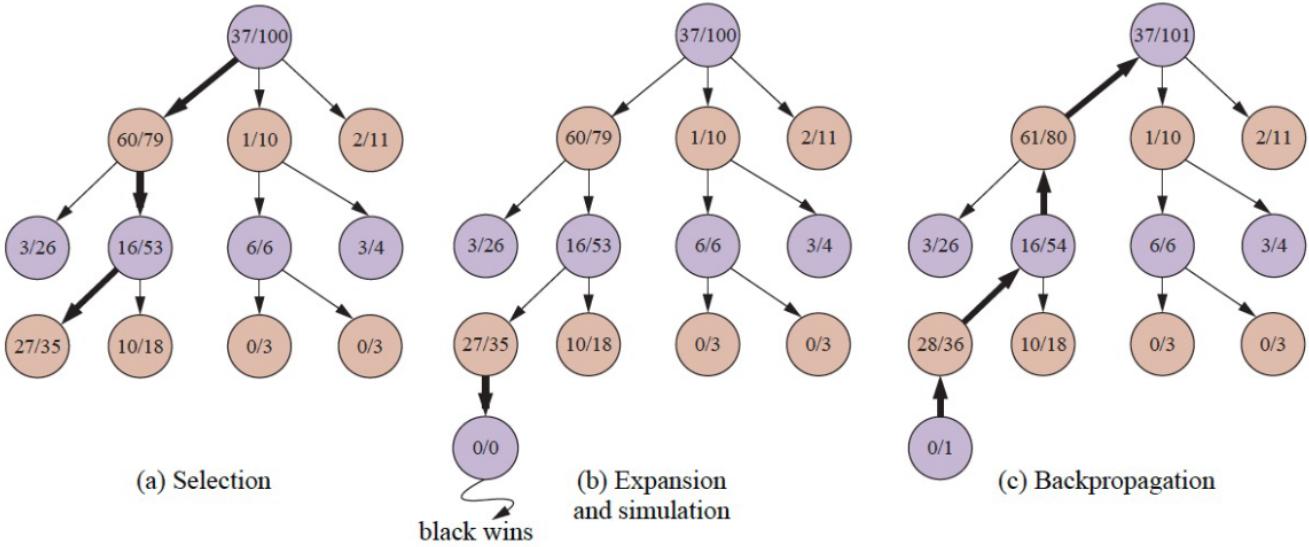
### 6.5 Monte Carlo Tree Search

Fonction d'utilité estimée comme l'utilité moyenne (par exemple, pourcentage de victoire) sur des simulations (appelées playout) de jeux complets. On a déjà tester avant et on sait le pourcentage de victoire avec ces moves.

Jouer avec des coups légaux aléatoires de la part des deux joueurs

On a un search tree qui augmente à chaque itérations

- **Selection** : Commence du root et va à une leaf en utilisant une sélection de stratégies
- **Expansion** : Ajoute un nouvel enfant au node
- **Simulations** : jouer (sans enregistrer les moves)
- **Back-propagation** : update les parents dans le tree



```

function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree  $\leftarrow$  NODE(state)
  while IS-TIME-REMAINING() do
    leaf  $\leftarrow$  SELECT(tree)
    child  $\leftarrow$  EXPAND(leaf)
    result  $\leftarrow$  SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts

```

## 6.6 Stochastic Games

Ce sont des jeux avec un degré d'improbabilité parmi les éléments aléatoires. Cela requiert une chance pour chaque nœud en plus des Min Max

### 6.6.1 Expectminimax

Cet algo ajoute de la chance aux nœuds en plus de MAX MIN. Complexité de  $\mathcal{O}(b^m n^m)$  où  $n$  est le nombre de chance d'événement distincte

**Expectminimax(s) =**

$$\begin{aligned}
 & \text{Utility (s)} && \text{if Is-Terminal(s)} \\
 & \max_{a \in \text{Actions}(s)} \text{Expectminimax}(\text{Result}(s,a)) && \text{if To-Move(s) = MAX} \\
 & \min_{a \in \text{Actions}(s)} \text{Expectminimax}(\text{Result}(s,a)) && \text{if To-Move(s) = MIN} \\
 & \sum_{r \in \text{ACTIONS}(s)} P(r) * \text{Expectminimax}(\text{RESULT}(s,r)) && \text{if To-Move(s) = CHANCE}
 \end{aligned}$$

On peut pruner dans le même genre que alpha beta

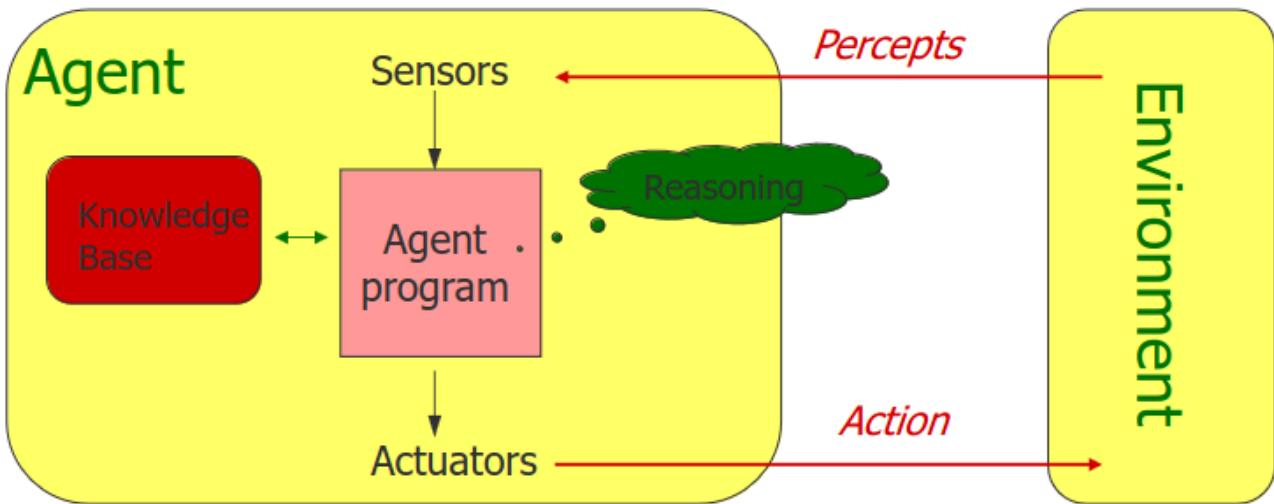
# 7 Logical Agent

## 7.1 Knowledge Based Agents

On peut raisonner avec de la connaissance pour savoir quelle action faire. Il y a 2 types de connaissances :

1. **Logique classique** : Vrai ou fausse
2. **Autre logique** : connaissance incertaine

L'agent peut etre représenté de la sorte :



Le composant central d'un KBA est un ensemble de phrase (KB).

**Sentences** : phrase qui est une affirmations sur le monde

### 7.1.1 Operation KBA

1. **TELL** : Ajoute une nouvelle phrase
2. **ASK** : Demande se qui est connue
3. **MAKE-PERCEPT-SENTENCE** : Construit une phrase affirmant que l'agent a perçu le percept donné à un moment donné.
4. **MAKE-ACTION-QUERY** : Construit une phrase qui ASK quelle action devrait etre faite maintenant
5. **MAKE-ACTION-SENTENCE** : Construit une phrase affirmant que l'action choisie a été exécutée

### 7.1.2 Knowledge representation

Le language dans laquelle la phrase est représenté. Il y a 2 approches

```

function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action  $\leftarrow$  ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t  $\leftarrow$  t + 1
  return action

```

Update state

Result of action

- **Declarative approach** : Nous ne disons à l'agent que ce qu'il a besoin de savoir. Les phrases sont dites une à une (par TELL) jusqu'à ce que l'agent sache comment opérer dans son environnement.
- **Procedural approach** : Nous codons les comportements souhaités directement sous forme de programme.

## 7.2 Exemple : the Wumpus World

Une grille avec un créature qui mange les agents. Il y a des puits dans des cases (à éviter) et le but est de trouver le trésor sans se faire manger ni tomber dans les puits.<sup>5</sup>

- Pas entièrement observable (perception local)
- Déterministe
- Pas épisodique (séquentielle au niveau des actions préformées)
- Statique (Monstre et puits ne bouge pas)
- Discret
- Un seul agent

---

<sup>5</sup>pages 209-210-211 du livre

4	sss		≈
3	w	sss ≈	≈
2	sss		≈
1	S	≈	≈
	1 2 3 4		

World state

4			
3			
2	OK		
1	OK	OK	
	1 2 3 4		

Agent view

Avant de faire une action, l'agent doit "réfléchir" à propos de :

- analyser le state actuelle
- Eviter les dangers (wumpus et puits)
- Trouver le trésor
- Garder une version de l'environement (map, propriété des case ...)

4	sss		≈
3	w	sss ≈	≈
2	sss		≈
1	S	≈	≈
	1 2 3 4		

4			
3			
2	OK	p?	
1	OK	OK	p?
	1 2 3 4		

**p** - Pit  
**w** - Wumpus  
**OK** - safe square

			$\approx$	
4			$\approx$	
3				$\approx$
2			$\approx$	
1	S	$\approx$		$\approx$
	1	2	3	4

4				
3	w!			
2	OK		OK	
1	OK	OK		p!
	1	2	3	4

*The conclusions are guaranteed to be correct !*

			$\approx$	
4			$\approx$	
3				$\approx$
2			$\approx$	
1	S	$\approx$		$\approx$
	1	2	3	4

4				
3	w!		p?	
2	OK	OK	OK	
1	OK	OK	p!	
	1	2	3	4

### 7.3 Logique

La logique correspond à de la syntax et de la sémantique

- **Syntax** : Spécifie toutes les phrases bien formé ( $x + y = 4$  OK  $x2y+$  = KO)
- **Sémantique** : Définir la vérité des phrases par rapport à chaque monde possible
  - $x = 1, y = 3$
  - $x = 2, y = 2$
  - ...

- **Interprétation** : abstraction mathématique d'un monde possible, en donnant une interprétation une phrase est vrai ou fausse.
- **Modele de phrase** :  $\alpha$  est un interprétation ou la phrase est vraie.  
"m est un model de  $\alpha$ "  $\rightarrow \alpha$  est vrai dans l'interprétation de m
- $M(\alpha)$  : ensemble des modeles de  $\alpha$
- $KB \models \alpha$  :  $\alpha$  est vrai si KB est vrai
- $KB \vdash \alpha$  : On peut prouver  $\alpha$  avec KB

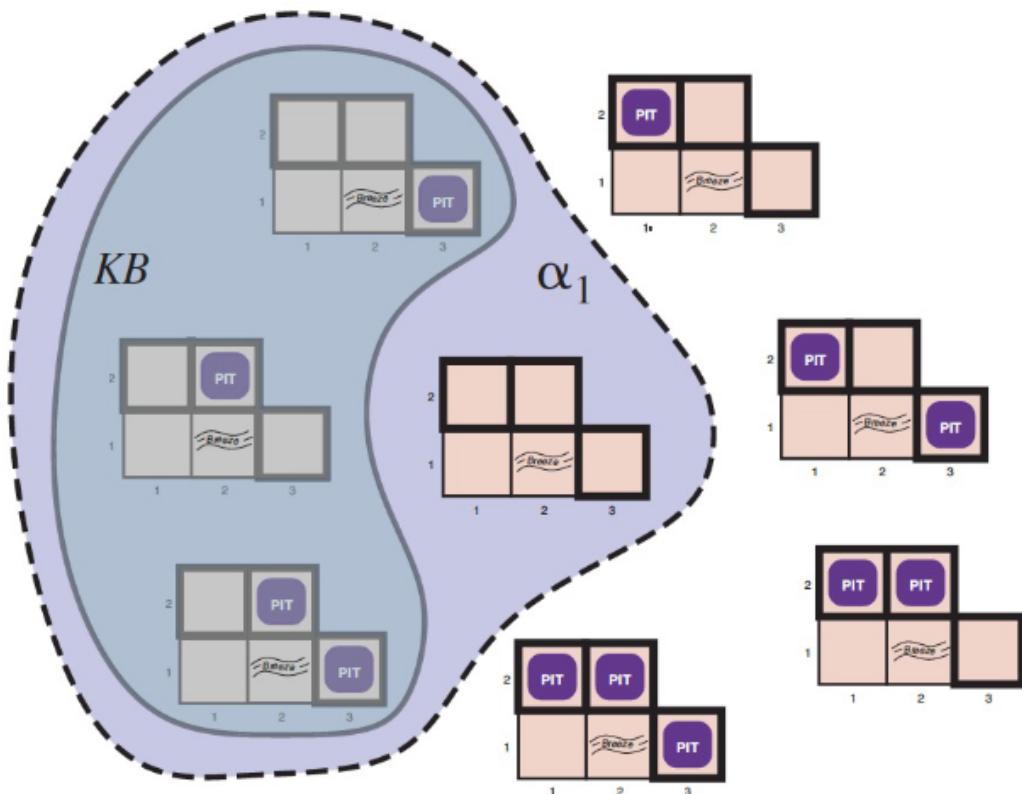
### 7.3.1 Implication (entailment)

Implication entre la phrase  $\alpha$  et  $\beta$ ,  $\beta$  est un conséquence logique de  $\alpha$

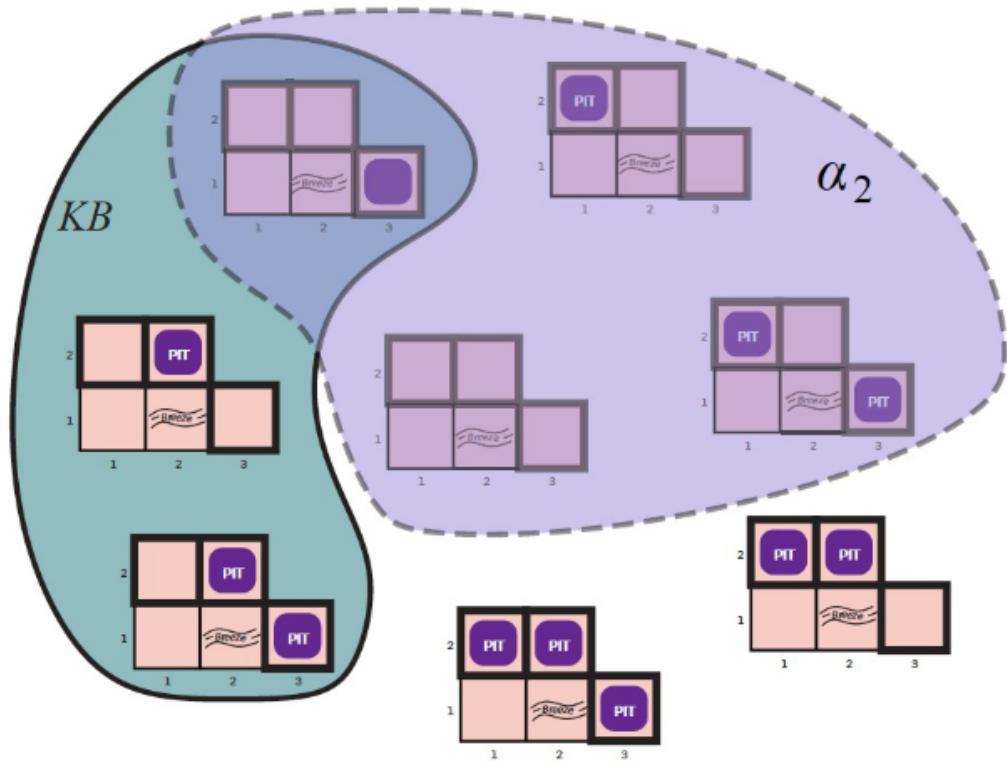
$$\alpha \models \beta \text{ IFF } M(\alpha) \subset M(\beta) \quad (4)$$

$\beta$  est vrai dans chaque model de  $\alpha$  (ex :  $x = 0 \models xy = 0$ )

Dans le Wumpus game apres avoir rien détecter en [1,1] et bougé a droite et sentit la brize en [2,1] (signe d'un puit a coté de cette case). Toutes les interprétations possible pour KB en assumant seulement des pit est le suivant :



- $KB = \text{Regles de wumpus} + \text{observations}$
- $\alpha_1 = "[1,2] \text{ est safe}"$
- $KB \models \alpha_1$ , prouvé par models checking



- $KB = \text{Règles de wumpus} + \text{observations}$
- $\alpha_2 = "[2,2] \text{ est safe}"$
- $KB \not\models \alpha_2$

On voit que 2 modèles de  $KB$  ne sont pas d'accord du fait que [2,2] est sage

si  $KB \models \alpha_1$  on peut conclure que  $\alpha_1$  est vrai

Verifier  $KB \models \alpha$  en utilisant le Model checking

1. énumération des interprétations
2. Trouver les interprétations qui sont un modèles de  $\alpha$
3. Check que  $\beta$  est vrai dans ces modèles  $\alpha \vdash \beta$

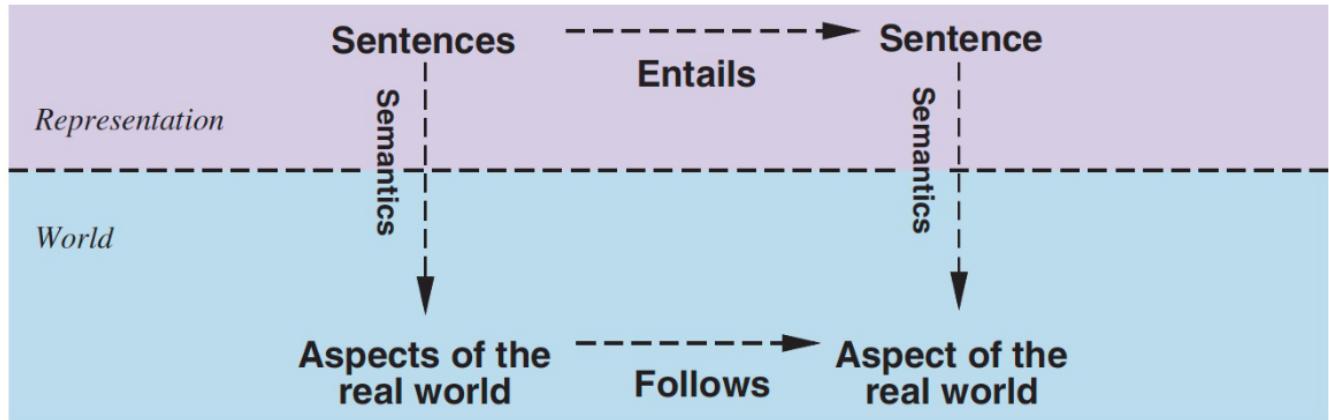
$$\begin{array}{c} \alpha \models \beta \\ \Downarrow \\ \text{Completeness} \qquad \text{Soundness} \\ \Updownarrow \\ \alpha \vdash \beta \end{array}$$

### 7.3.2 Monotonicité

Si  $KB \models \alpha$  alors  $KB \wedge \beta \models \alpha$

L'ensemble des formules impliquées ne peut que augmenter au fur et à mesure que des infos sont ajoutées au KB

Si une formule est impliquée par un sous ensemble de KB, il est alors aussi impliquée par KB



## 7.4 Proposition logique

### 7.4.1 Syntax

- True, False
- Symbols (P, Q, A, ...)
- Connecteur logiques :
  - Négations :  $\neg$
  - Conjonctions :  $\wedge$
  - Disjonctions :  $\vee$
  - Implication :  $\Rightarrow$
  - Equivalence :  $\Leftrightarrow$
  - parenthèse (,)

$$\begin{aligned}
 Sentence &\rightarrow \text{AtomicSentence} \mid \text{ComplexSentence} \\
 \text{AtomicSentence} &\rightarrow \text{True} \mid \text{False} \mid P \mid Q \mid R \mid \dots \\
 \text{ComplexSentence} &\rightarrow (\text{Sentence}) \\
 &\quad \mid \neg \text{Sentence} \\
 &\quad \mid \text{Sentence} \wedge \text{Sentence} \\
 &\quad \mid \text{Sentence} \vee \text{Sentence} \\
 &\quad \mid \text{Sentence} \Rightarrow \text{Sentence} \\
 &\quad \mid \text{Sentence} \Leftrightarrow \text{Sentence}
 \end{aligned}$$

OPERATOR PRECEDENCE :  $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

### 7.4.2 Sémantique

On fait ça avec un table de vérité

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

exemple Wumpus : Un carré est venteux si et seulement si un carré voisin est un puits

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

- Initial conditions
  - $R_1: \neg P_{1,1}$  no pit in [1,1]
- Rules of Breezes (for a few example squares)
  - $R_2: B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
  - $R_3: B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,1} \vee P_{3,1})$
- Percepts
  - $R_4: \neg B_{1,1}$
  - $R_5: B_{2,1}$
- We know:  $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$

### 7.4.3 Inference

Le but est de décider si  $KB \models \alpha$

On veut savoir si  $P_{1,2}$  n'est pas un puit :  $\neg P_{1,2}$

- Imaginons nous avons ses infos :  $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}, P_{3,1}$
- on a donc  $2^n$  interprétation possible ( $n = \text{nombre de symbol}$ ) donc ici  $2^7$

Il faut donc enumérée sur chaque interprétation et check si  $\alpha = P_{1,2}$  est True dans chaque models de KB

Complexité temporelle de  $\mathcal{O}(2^n)$  et spatiale de  $\mathcal{O}(n)$

```

function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
             $\alpha$ , the query, a sentence in propositional logic
  symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ 
  return TT-CHECK-ALL(KB,  $\alpha$ , symbols, { })

function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)
    else return true // when KB is false, always return true
  else
     $P \leftarrow$  FIRST(symbols)
    rest  $\leftarrow$  REST(symbols)
    return (TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{true}$ })
           and
           TT-CHECK-ALL(KB,  $\alpha$ , rest, model  $\cup$  { $P = \text{false}$ }))

```

## 7.5 Propositional Theorem Proving

- Equivalence logique :  $\alpha \equiv \beta$  ssi  $\alpha \models \beta$  et  $\beta \models \alpha$
- Valide (tautologie) : Toutes les interprétations sont valide (True)
- Satifaisant : True dans certaine interprétation mais pas toutes
- Unsatisfait : True pour aucune des interprétations.

$\alpha \models \beta$  IF  $(\alpha \wedge \neg\beta)$  est insatisfait

### 7.5.1 Preuve inference

On veut déduire de nouvelle formules qui sont des conséquence logique de KB

Donc si on veut que  $KB \models \alpha$  alors on va déduire que  $KB \rightarrow F_1$  et déduit une nouvelle formule et donc avec  $(KB \rightarrow F_1) \rightarrow F_2$  on peut déduire une nouvelle formule, etc pour ensuite déduire  $\alpha$   $\alpha$  et  $\beta$  sont toutes les 2 des formules

- Modus Ponens : 
$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$
- And-elimination : 
$$\frac{\alpha \wedge \beta}{\alpha}$$
- Two sens de Morgan : 
$$\frac{\neg(\alpha \wedge \beta)}{\neg\alpha \vee \neg\beta} \Leftrightarrow \frac{\neg\alpha \vee \neg\beta}{\neg(\alpha \wedge \beta)}$$

### Proof as a search

On a notre KB initial et on a une fonction successeur qui retourne un nouveau KB avec une nouvelle formule. Notre Goal test vérifie si KB contient la phrase "final"/de l'objectif.

### Résolution

$$\frac{l_1 \vee \dots \vee l_k \quad , \quad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n} \text{ where } l_i = \neg m_j$$


---

$$\begin{array}{c}
 \text{AstudyCS} \vee \neg \text{AlikeBier} \\
 \text{AlikeBier} \vee \text{AlikeWine}
 \end{array}
 \quad \vdash \quad \text{AstudyCS} \vee \text{AlikeWine}$$

Un règle : Si j'ai 2 disjonctions et que dans une j'ai une proposition logique négative ( $\neg$ ) et que j'ai la même dans l'autre mais positive, je peux déduire que je peux prendre les autres propositions logique et les mettre ensemble ( $\wedge$ ).

J'ai besoin d'avoir 2 formules sous la forme de disjonction

Preuve complète et sound.

Le soucis est que je dois représenter mon KB sous la forme de disjonction. Mais cela peut être fait par un CNF ( $A_1 \wedge \dots \wedge A_n$ ) (n'importe quelle formule peut être transformée en CNF (déjà prouvé))

Il faut ensuite on peut appliquer cet algo qui prouve que  $KB \models \alpha$  iff  $(KB \wedge \neg \alpha)$  unsatisfiable

**function** PL-RESOLUTION( $KB, \alpha$ ) **returns** true or false

**inputs:**  $KB$ , the knowledge base, a sentence in propositional logic  
 $\alpha$ , the query, a sentence in propositional logic

$clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg \alpha$   
 $new \leftarrow \{\}$

**while** true **do**

**for each** pair of clauses  $C_i, C_j$  **in**  $clauses$  **do**

$resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )

**if**  $resolvents$  contains the empty clause **then return** true

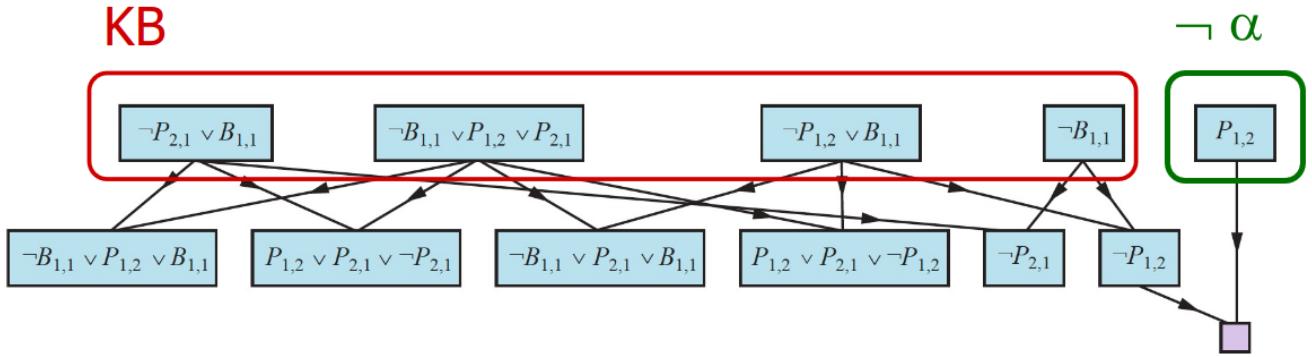
$new \leftarrow new \cup resolvents$

**if**  $new \subseteq clauses$  **then return** false

$clauses \leftarrow clauses \cup new$

Avec notre KB on va déduire de nouvelle clause (formule sous forme disjonctive)  $C_1$ , mais si il y a une clause spéciale qui est vide alors false alors l'entiereté du KB est insatisfait .

Prenons un exemple et un Xanax :



On veut prouver que il n'y a pas de puit en [1,2] ( $\neg P_{1,2}$ )

KB est :  $(B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \vee \neg B_{1,1})$

$\alpha$  est  $\neg P_{1,2}$

On va donc ajouter  $P_{1,2}$  et le transformer en CNF et on fait la résolution

Si on utilise  $(\neg P_{1,2} \vee B_{1,1})$  et  $(\neg B_{1,1})$  et on va avoir le formule  $(\neg P_{1,2})$

autre ex :

Si on utilise  $(\neg P_{2,1} \vee B_{1,1})$  et  $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$  et on va avoir le formule  $(\neg B_{1,1} \vee P_{1,2} \vee B_{1,1})$

Maintenant je peux de nouveau faire une résolutions sur  $\neg P_{1,2}$  et  $P_{1,2}$  et j'obtient un empty donc False

et voila (donner moi un deuxième Xanax)

# 8 First-Order Logic

## 8.1 Propositional logic

Une proposition logique est :

- **Déclarative** : les relations entre les variables sont décrites par des phrases
- **Expressive** : peut présenter des informations partielles en utilisant la disjonction
- **Compositionel** : Si  $A = \text{"il pleut"}$  et  $B = \text{"j'aime la biere"}$  alors  $A \wedge B = \text{"il pleut et j'aime la biere"}$
- **Manque d'expressivité** pour décrire l'environnement de manière concise (ex : ne peut pas dire "les puits provoquent des brises dans les cases adjacentes")

First-Order Logic (FOL) assume que le monde contient :

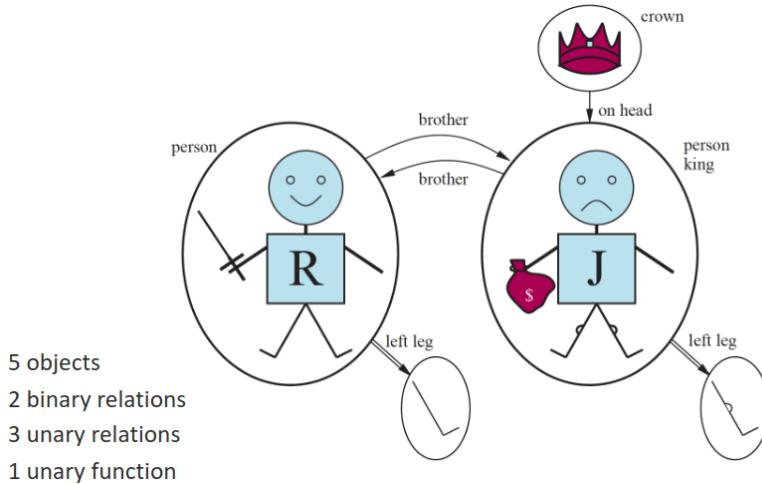
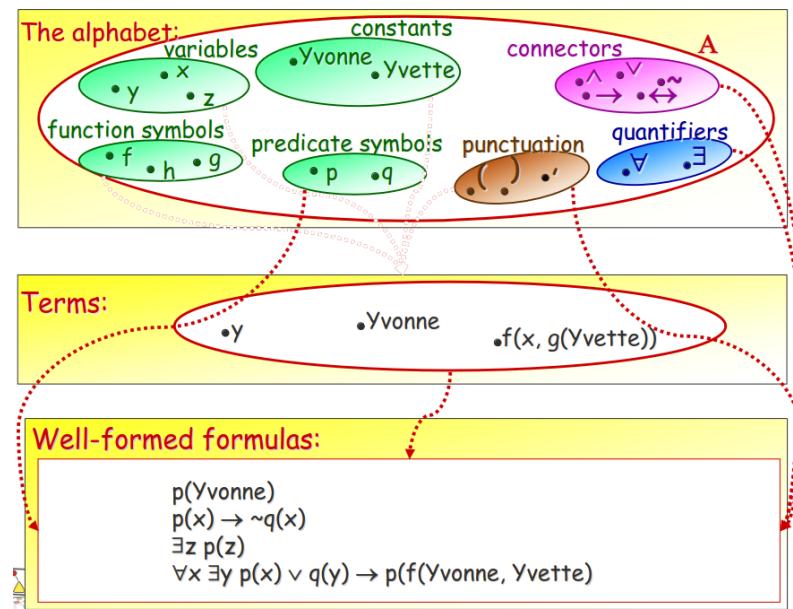
- Objects : personne, voiture, maison, ...
- Relation :
  - Unary : propriété avec les objets
  - N-ary : Relations entre les objets
  - Ex : la relation de fraternité est l'ensemble ( $\langle \text{Bob}, \text{Richard} \rangle, \langle (\text{Richard}, \text{Bob}) \rangle$ )
- Fonctions : father\_of, successeur, ...

## 8.2 Syntax and semantics of FOL

$$\begin{array}{lcl} \text{Sentence} & \rightarrow & \text{AtomicSentence} \mid \text{ComplexSentence} \\ \text{AtomicSentence} & \rightarrow & \text{Predicate} \mid \text{Predicate}(\text{Term}, \dots) \mid \text{Term} = \text{Term} \\ \text{ComplexSentence} & \rightarrow & (\text{Sentence}) \\ & | & \neg \text{Sentence} \\ & | & \text{Sentence} \wedge \text{Sentence} \\ & | & \text{Sentence} \vee \text{Sentence} \\ & | & \text{Sentence} \Rightarrow \text{Sentence} \\ & | & \text{Sentence} \Leftrightarrow \text{Sentence} \\ & | & \text{Quantifier Variable}, \dots \text{ Sentence} \\ \\ \text{Term} & \rightarrow & \text{Function}(\text{Term}, \dots) \\ & | & \text{Constant} \\ & | & \text{Variable} \\ \\ \text{Quantifier} & \rightarrow & \forall \mid \exists \\ \text{Constant} & \rightarrow & A \mid X_1 \mid \text{John} \mid \dots \\ \text{Variable} & \rightarrow & a \mid x \mid s \mid \dots \\ \text{Predicate} & \rightarrow & \text{True} \mid \text{False} \mid \text{After} \mid \text{Loves} \mid \text{Raining} \mid \dots \\ \text{Function} & \rightarrow & \text{Mother} \mid \text{LeftLeg} \mid \dots \\ \\ \text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow \end{array}$$

- **Alphabet** : Variables, constantes; fonctions, ...

- **Termes** : Combinaison de l'alphabet
- **Formule bien formé** : Construite à partir de symboles de prédicats avec des termes comme arguments et à partir de connecteurs, quantificateurs et ponctuations (selon les règles des connecteurs)



quelques exemples :

- Brother(Bob,Richard)
- Married(Father(Richard), Mother(Bob))
- King(Richard)  $\wedge$  King(Bob)
- $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$
- $\exists x \text{ Crown}(x) \Rightarrow \text{OnHead}(x)$
- $\forall s \text{ Breeazy}(s) \Leftrightarrow \exists r \text{ Adjacent}(r, s) \wedge \text{Pit}(r)$

## Sémantique

Le domaine de l'interprétation est l'ensemble des objets

**Constants** : L'interprétation identifie l'object dans le vrai monde.

**Symbol de prédicts** : L'interprétation spécifie les relation particulière dans le domaine. Peut etre définie implicitement ou explicitement a travers l'ensemble des tuples d'objets qui satisfait la relations

**Fonctions de symboles** : Identifie l'object référé par un tuple d'objets. Peut etre définie implicitement ou explicitement au travers d'une table.

**Interprétation** :

- D le domaine
- Une fonction (totale) qui map les constantes a D
- Une fonction (totale) qui maps les fonctions de symboles as des fonctions ( $D \rightarrow D$ )
- Une fonction (totale) qui maps les symoblrs prédit a des prédictions ( $D \rightarrow \text{Booleans}$ )

Voir exemple slidesq p18 chap 8

### 8.2.1 Complex sentences

Elle sont construite en combinant des phrase atomoque en utilisant des connecteurs logique ( $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ ) et des parentheses.

- $\neg \text{Brother}(\text{LeftLeg}(\text{Richard}), \text{Bob})$
- $\text{Brother}(\text{Richard}, \text{Bob}) \wedge \text{Brother}(\text{Bob}, \text{Richard})$
- ...

### 8.2.2 Quantifiers

Peut etre utiliser pour exprimer des propriétés de collections d'objet

- Plus besion d'enuméré chaque objet
- $\forall$  dans une conjonction (quantifier universel) ex:  $\forall x \text{ Human}(x) \Rightarrow \text{Mortal}(x)$
- $\exists$  dans une disjonction (quantifier existentiel) ex :  $\exists x \text{ Bird}(x) \wedge \text{Can\_Fly}(x)$

Attention  $\forall x \forall y$  est la meme chose que  $\forall y \forall x$  pareil pour  $\exists$

Mais  $\exists x \forall y$  n'est pas la meme chose que  $\forall y \exists x$

$\forall x P$  est pareil que  $\neg \exists x \neg P$ , pareil pour  $\exists x P$  est pareil que  $\neg \forall x \neg P$

## 8.3 Using FOL

FOL peut etre utilisé pour modeliser nombre naturel, ensemble de sous ensemble, listes, ...

### 8.3.1 Ex Wumpus

## 8.4 Complement on FOL

Voir slides 41-52

- One's mother is one's parent who is a female
  - $\forall c, m \text{ Mother}(m, c) \Leftrightarrow \text{Parent}(m, c) \wedge \text{Female}(m)$
- A grandparent is a parent of one's parent
  - $\forall c, g \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c)$
- A theorem : siblinghood is symmetric
  - $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x)$

## 9 Inference in First-Order Logic

On va voir comment les algo font pour répondre une question FOL

### 9.1 Propositional vs FOL inference

**Universal Instantiation (UI)** Pour n'importe quelle phrase  $\alpha$ , variables  $v$ , et *ground term*  $g$  (terme sans variables)

$$\frac{\forall v \alpha}{\text{subst}(\{v/g\}, \alpha)} \quad (5)$$

$v$  peut être remplacé par n'importe quelle instance

$\text{subst}(\omega, \alpha)$  applique la substitution  $\omega$  sur  $\alpha$

ex :

$\forall x \text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$  Avec on peut déduire que  $\text{King}(\text{Bob}) \wedge \text{Greedy}(\text{Bob}) \Rightarrow \text{Evil}(\text{Bob})$

**Existential instantiation** Pour n'importe quelle phrase  $\alpha$ , variables  $v$ , et nouveau symbole constant  $k$

$$\frac{\exists \alpha}{\text{subst}(\{v/k\}, \alpha)} \quad (6)$$

$v$  peut être remplacé avec un nouveau symbole

ex :

$\exists x \text{mother}(\text{bob}, x)$  on peut déduire  $\text{mother}(\text{Bob}, \text{ANewMother})$

$\text{ANewMother}$  est un constante de Skolem, équivalence déductive (pas d'équivalence logique)

Avec ces deux instantiations, on peut réduire FOL à des symboles de propositions.

#### 9.1.1 Reduction to Propositional Logic

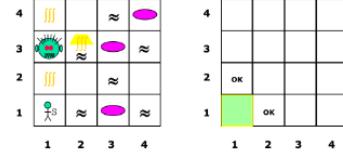
Le problème est que il y a une infinité de *ground terms* mais grâce à Herbrand, on sait que :

*If a sentence  $\alpha$  is entailed by a FOL KB, it's entailed by a **finite** subset of the propositionalized KB*

## More precise axioms than with propositional logic

Sensors: Stench, Breeze, Glitter, Bump, Scream

- Percept has five values
- Time is important
- A typical sentence
  - Percept ([Stench, Breeze, Glitter, None, None], 7)
- The actions are terms
  - Turn(right), Turn(left), Forward, Shoot, Grab, Release
- Computing best action with a query
  - $\exists a: \text{BestAction}(a, 7)$



- Connecting percepts to actions

$$\forall s, b, u, c, t \text{ Percept}([s, b, \text{Glitter}, u, c], t) \Rightarrow \text{Action}(\text{Grab}, t)$$

- requires many rules

- Can be simplified by intermediate predicates

$$\forall b, g, u, c, t \text{ Percept}([\text{Stench}, b, g, u, c], t) \Rightarrow \text{Stench}(t)$$

$$\forall b, g, u, c, t \text{ Percept}([\text{None}, b, g, u, c], t) \Rightarrow \neg \text{Stench}(t)$$

$$\forall s, g, u, c, t \text{ Percept}([s, \text{Breeze}, g, u, c], t) \Rightarrow \text{Breeze}(t)$$

$$\forall s, g, u, c, t \text{ Percept}([s, \text{None}, g, u, c], t) \Rightarrow \neg \text{Breeze}(t)$$

...

$$\forall t \text{ Glitter}(t) \Rightarrow \text{BestAction}(\text{Grab}, t)$$

- Defining squares with [x,y] reference instead of atomic names

- Adjacency between two squares

$$\forall x, y, a, b \text{ Adjacent}([x, y], [a, b]) \Leftrightarrow$$

$$(x=a \wedge y=b-1) \vee (x=a \wedge y=b+1)$$

$$\vee (x=a-1 \wedge y=b) \vee (x=a+1 \wedge y=b)$$

- Modeling the breeze of pits

$$\forall s \text{ Breezy}(s) \Leftrightarrow \exists r \text{ Adjacent}(r, s) \wedge \text{Pit}(r)$$

- ...

Application  $n = 0$  à  $\infty$ :

- créer un KB propositionnelle en l'instanciant avec un terme de profondeur  $n$
- regarder si  $\alpha$  est entaillé par le KB

Si  $\alpha$  est pas entaillé l'algorithme loop à l'infini

**Turing** : Entaillement pour FOL est semidécidable

## 9.2 Unification

C'est une substitution qui fait que des expressions logiques différentes semblent identiques

ex :

$\text{Parent(Bob, }x\text{)} \text{ et } \text{Parent(y, }z\text{)}$

- $\{y/\text{Bob}, x/z\} : \text{Parent(Bob, }z\text{)}$
- $\{y/\text{Bob}, x/\text{Lucia}, z/\text{Lucia}\} : \text{Parent(Bob, Lucia)}$

L'algorithme  $\text{Unify}(p, q)$  prend 2 phrases atomiques  $p$  et  $q$  et retourne une substitution qui fait que  $p$  et  $q$  sont identiques.

$\text{Unify}(p, q) = \emptyset$  où  $\text{Subst}(\emptyset, p) = \text{Subst}(\emptyset, q)$

$\emptyset$  est l'unificateur des 2 phrases. Il y en a possiblement plus d'un

### 9.2.1 Most General Unifier

MGU pour les KOG, fait que "la substitution qui s'engage le moins sur les variables contraignantes"

ex :

$\text{Unify}(\text{Parent(Bob, }x\text{)}, \text{Parent(y, }z\text{)}) = \{y/\text{Bob}, x/z\}$

MGU est unique

### 9.2.2 Standardize apart

Si 2 phrases partagent des variables alors on peut les unifier, pour éviter les conflits de noms, on renomme une des variables.

$\text{Unify}(\text{Parent(Bob, }x\text{)}, \text{Parent(y, }z\text{)}) = \text{Unify}(\text{Parent(Bob, }w\text{)}, \text{Parent(q, }r\text{)})$

### 9.2.3 Generalized Modus Ponens

avec  $\text{Subst}(\emptyset, p_i) = \text{Subst}(\emptyset, p'_i)$  pour tous les  $i$

$$\frac{p'_1, p'_2, \dots, p'_n, (p_1 \wedge, p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{Subst}(\emptyset, q)} \quad (7)$$

ex :

```

function UNIFY( $x, y, \theta$ ) returns a substitution to make  $x$  and  $y$  identical
  inputs:  $x$ , a variable, constant, list, or compound
            $y$ , a variable, constant, list, or compound
            $\theta$ , the substitution built up so far
  if  $\theta = \text{failure}$  then return failure
  else if  $x = y$  then return  $\theta$ 
  else if VARIABLE?( $x$ ) then return UNIFY-VAR( $x, y, \theta$ )
  else if VARIABLE?( $y$ ) then return UNIFY-VAR( $y, x, \theta$ )
  else if COMPOUND?( $x$ ) and COMPOUND?( $y$ ) then
    return UNIFY(ARGS[ $x$ ], ARGS[ $y$ ], UNIFY(OP[ $x$ ], OP[ $y$ ],  $\theta$ ))
  else if LIST?( $x$ ) and LIST?( $y$ ) then
    return UNIFY(REST[ $x$ ], REST[ $y$ ], UNIFY(FIRST[ $x$ ], FIRST[ $y$ ],  $\theta$ ))
  else return failure

function UNIFY-VAR( $var, x, \theta$ ) returns a substitution
  inputs:  $var$ , a variable
            $x$ , any expression
            $\theta$ , the substitution built up so far
  if  $\{var/val\} \in \theta$  then return UNIFY( $val, x, \theta$ )
  else if  $\{x/val\} \in \theta$  then return UNIFY( $var, val, \theta$ )
  else if OCCUR-CHECK?( $var, x$ ) then return failure
  else return add  $\{var/x\}$  to  $\theta$ 

```

$$\frac{King(Bob), Greedy(y), (King(x) \wedge Greedy(y) \Rightarrow Evil(x))}{Evil(Bob)} \quad (8)$$

Avec  $\emptyset = \{x/Bob, y/Bob\}$

## 9.3 Forward Chaining

$\exists$

### 9.3.1 First-order definite clauses

- Trouver toutes les règles dont les prémisses sont satisfaites
- Ajouter leurs conclusions au faits connus
- Représenter l'opération jusqu'à ce qu'une réponse soit apportée à la question ou qu'aucun fait nouveau n'est ajouté

```

function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
           α, the query, an atomic sentence

  while true do
    new  $\leftarrow \{\}$       // The set of new sentences inferred on each iteration
    for each rule in KB do
       $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(rule)$ 
      for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$ 
        for some  $p'_1, \dots, p'_n$  in KB
         $q' \leftarrow \text{SUBST}(\theta, q)$ 
        if  $q'$  does not unify with some sentence already in KB or new then
          add  $q'$  to new
           $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
          if  $\phi$  is not failure then return  $\phi$ 
        if new =  $\{\}$  then return false
        add new to KB
  
```

### 9.3.2 Analyse

**Soundness** : Il ne dérive que les phrases qui sont impliquées, parce que Generalize Modus Ponens le fait.

**Completeness** : Il répond à toutes les requêtes dont les réponses sont impliquées par le KB, mais peut ne pas se terminer en raison de la semi-décidabilité de l'implication avec des clauses définies.

## 9.4 Backward chaining

Commence avec les prémisses de l'objectif

- Chaque prémisses doit être supporté par le KB
- Commencer avec la premières hypothèses et chercher soutien du KB

La fonction est un algo récursive tel que une depth-first.

```

function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
  inputs: KB, a knowledge base
    goals, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution { }
  local variables: ans, a set of substitutions, initially empty
  if goals is empty then return {θ}
  q' ← SUBST(θ, FIRST(goals))
  for each r in KB where STANDARDIZE-APART(r) = (p1 ∧ … ∧ pn ⇒ q)
    and θ' ← UNIFY(q, q') succeeds
    ans ← FOL-BC-ASK(KB, [p1, …, pn | REST(goals)], COMPOSE(θ, θ')) ∪ ans
  return ans

```

#### 9.4.1 Logic programming: Prolog

Algorithm = Logic + Control

TODO

### 9.5 Résolutions

Chaque phrase d'un FOL peut etre convertit en CNF (conjuctive Normal Form)

1. **Eliminer Implications** :  $p \Rightarrow q$  devient  $\neg p \vee q$
2. **Bouger les  $\neg$  qui sont devant** :
  - $\neg(p \vee q)$  devient  $\neg p \wedge \neg q$
  - $\neg(p \wedge q)$  devient  $\neg p \vee \neg q$
  - $\neg\forall x, p$  devient  $\exists x \neg p$
  - $\neg\exists x, p$  devient  $\forall x \neg p$
  - $\neg\neg p$  devient  $p$
3. **standardiser les variables** :  $(\forall x P(x)) \vee (\exists x Q(x))$  devient  $(\forall x P(x)) \vee (\exists y Q(y))$
4. **Bouger les quantifiers à gauche** :  $p \vee \forall x q$  devient  $\forall x p \vee q$
5. **Skolemization** : Voir prochaine sections
6. **De Morgan**

#### 9.5.1 Skolemization

Objectif est de:

- retirer tous les quantifiers existants
- Replacer les variables par des toutes nouvelles constantes qui n'existe pas de le KB

ex :  $\exists x Q(x)$  devient  $Q(A)$  avec où *A* est unique

Pour les phrases plus complexe, on utilise des fonctions de symbole (Skolem functions) pour indiquer des valeur spécifique.

ex :  $\forall Animal(x, A)$  devient  $\forall Animal(x, F(x))$

### 9.5.2 Resolution inference rule

Pour  $p_i$  and  $q_i$  où  $UNIFY(p_j, \neg q_k) = \emptyset$

$$\frac{p_1 \vee \dots p_j \dots \vee p_m, q_1 \vee \dots p_k \dots \vee q_n}{SUBST(\emptyset, (p_1 \vee \dots p_{j-1} \vee p_{j+1} \dots \vee p_m \vee q_1 \vee \dots q_{k-1} \vee q_{k+1} \dots \vee q_n))} \quad (9)$$

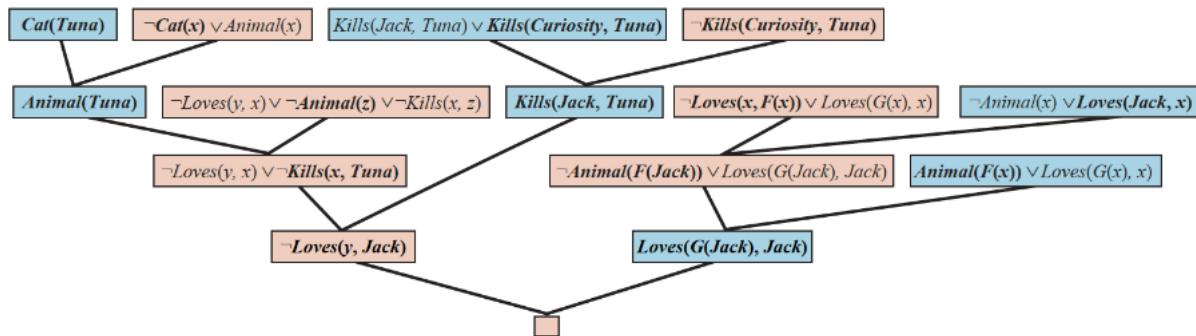
exemple plus concret :

$$\frac{\neg \text{loves}(u,v) \vee \neg \text{Kills}(u,v)}{\text{Animal}(F(x)) \vee \neg \text{Kills}(G(x),x)}$$

With  $\theta = \{ u/G(x), v/x \}$

$\alpha$  est un conséquence logique de KB si  $\text{KB} \models \alpha$  IFF  $(\text{KB} \wedge \neg\alpha)$  insatisfait

On transforme juste  $\neg\alpha$  en CNF et montre que  $KB \wedge \alpha$  insatisfait car apres application des règles de résolutions.



### 9.5.3 Introducing answers

- Resolution : non constructive proof
- If goal is **Kills(Curiosity,Tuna)**  
a yes/no answer is OK
- If goal is  $\exists x \text{ Kills}(x, \text{Tuna})$   
a yes/no answer is not very helpful ...
- Possible to get an answer (substitution) that makes the goal to fail
  - Use the goal  $\exists x \text{ Kills}(x, \text{Tuna}) \wedge \text{Answer}(x)$
  - Stop with the clause **Answer( $\alpha$ )**
  - $\alpha$  is the answer

## 10 Automated Planning

### 10.1 Definition of classical planning

Trouver une séquence d'action pour accomplir l'objectif, c'est une combinaison de différentes technique de IA:

- Seaching (Uninformed et informed)
- CSP
- Positional logic
  - Modélisation
  - Inference (SAT)
- FOL
  - modelisation

L'environnement est :

- Entierement observable
- déterministe
- fini
- statique
- discret

C'est similaire a un *Search problem* :

- Input :

- Ensemble de state
- Action (State - $\in$  State)
- Initial state
- Goal (1 ou un ensemble de state)
- Output
  - Séquence d'actions
- Difficulté :
  - Taille de l'espace de recherche
  - Méthode de recherche
  - Introduire décomposition du problème

### 10.1.1 Représentation du State

C'est une conjonction de (Grand et fonctionne) atoms ( $\{A, B\} = A \wedge B$ )

$$At(Plane1, Melbourne) \wedge In(Plane1, Bob) \quad (10)$$

**Closes-World Assumption** : si pas d'information sur  $p(a)$  alors  $p(a) = False$

### 10.1.2 Représentation des Actions

Spécifiés avec une **signature** (noms et paramètres), **Préconditions** (conjunction of literals), **Ef-fets** (conjunction of literals)

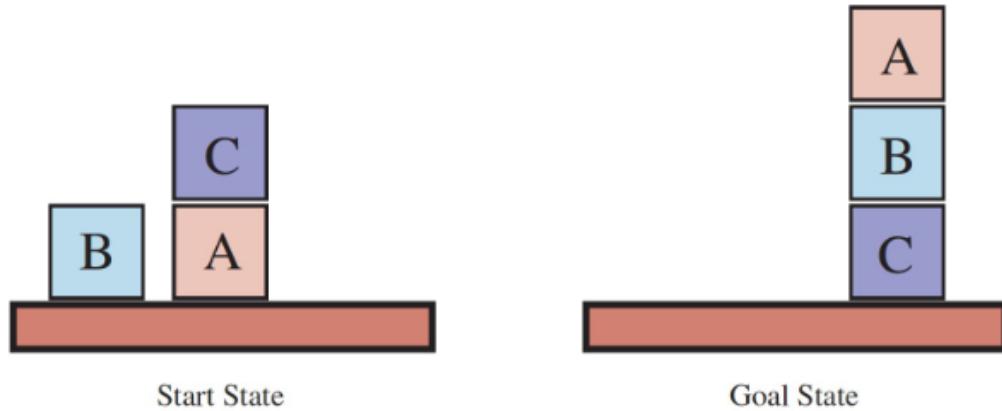
Une action est **Applicable** dans une state  $s$  si et seulement si le préconditions de  $a$  est une conséquence logique de  $s$  si et seulement si  $s$  satisfait les préconditions de  $a$

### 10.1.3 Représentation des objectifs

Un objectif = une state partiellement spécifié

C'est comme une précondition : une conjonction de littéraux (+ ou -)

Un state satisfait un Goal si il contient tous les atoms du goals



$\text{On}(C, \text{Table})$   
 $\text{On}(A, \text{Table})$   
 $\text{On}(B, \text{Table})$

$\text{On}(A, B)$   
 $\text{On}(B, C)$

$\text{Init}(\text{On}(A, \text{Table}) \wedge \text{On}(B, \text{Table}) \wedge \text{On}(C, A) \wedge \text{Block}(A) \wedge \text{Block}(B) \wedge \text{Block}(C) \wedge \text{Clear}(B) \wedge \text{Clear}(C) \wedge \text{Clear}(\text{Table}))$   
 $\text{Goal}(\text{On}(A, B) \wedge \text{On}(B, C))$   
 $\text{Action}(\text{Move}(b, x, y),$   
 PRECOND:  $\text{On}(b, x) \wedge \text{Clear}(b) \wedge \text{Clear}(y) \wedge \text{Block}(b) \wedge \text{Block}(y) \wedge (b \neq x) \wedge (b \neq y) \wedge (x \neq y),$   
 EFFECT:  $\text{On}(b, y) \wedge \text{Clear}(x) \wedge \neg \text{On}(b, x) \wedge \neg \text{Clear}(y))$   
 $\text{Action}(\text{MoveToTable}(b, x),$   
 PRECOND:  $\text{On}(b, x) \wedge \text{Clear}(b) \wedge \text{Block}(b) \wedge \text{Block}(x),$   
 EFFECT:  $\text{On}(b, \text{Table}) \wedge \text{Clear}(x) \wedge \neg \text{On}(b, x))$

## 10.2 Algorithms for classical planning

### 10.2.1 Complexité

**PlanSAT** : Demande si il existe un plan qui résoud le planning probleme

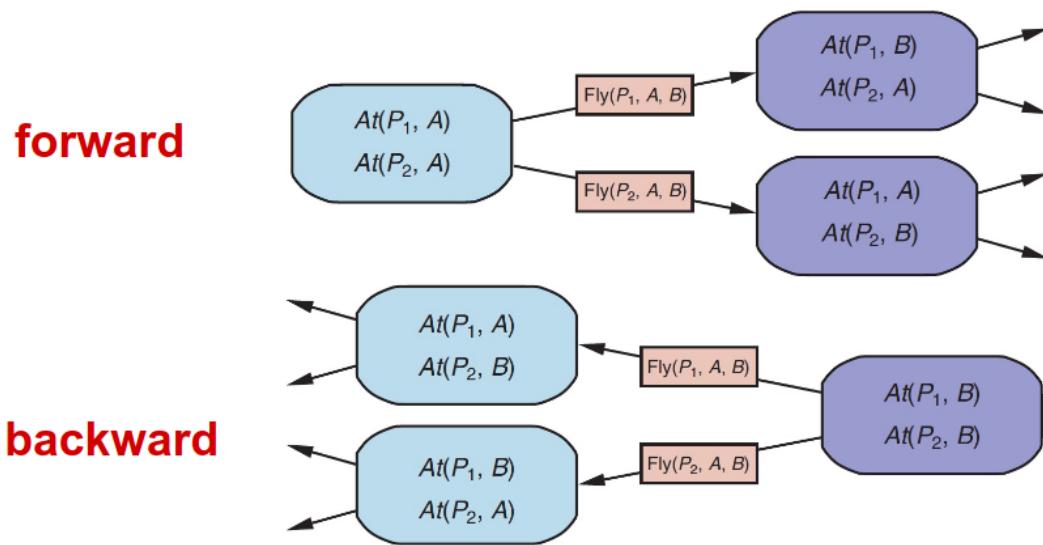
**Bounded PlanSAT** : Demande si il existe une solutions de longueur  $k$  ou moins

Les 2 problemes sont décidable (nombre fini de state)

Les 2 problemes sont dans **PSPACE** (plus difficile que NP) mais pas dans NP, PlanSAT est dans P, Bounded PlanSAT est dans NP-Complet.

### 10.2.2 Forward and backward search

c'est possible d'utiliser *classical state-space search methods* et avec préconditions/effets, il es possible de rechercher dans n'importe quelle directions



### 10.2.3 Forward state-space search

Similaire a la Search Methode (Ch. 3)

Explore pas mal d'actions inutile, de plus state space est tres large(branching factor) c'est pour ca que une fonction heuristique précises est tres utile

### 10.2.4 Backward state-space Search

On commence du goal et on applique des actions a l'envers jusqu'a trouver une séquence d'étapes qui arrive au state initial. Marche seulement comment on sait faire une actions a l'envers pas toujours le cas. Dure de trouver un bon heuristique.

### 10.2.5 Planning as Boolean satisfiability

Planning par tester la satisfaisabilité le phrase propositionnel  $InitialState \wedge AllPossibleActionDescription \wedge Goal$

A TERMINER

## 10.3 Heuristics for planning

Recherche forward ou backward est inutile sans un bon heuristique

On cherche à approximer le nombre d'action pour arriver à l'objectif depuis un état donnée

### 10.3.1 Relaxed problem

On définit un nouveau problème qui est plus facile à résoudre que le problème initial, le coût de la solutions de ce problème devient l'heuristique du problème initial. On peut par exemple ignorer préconditions heuristique,

### 10.3.2 Set covering Problem

On retire toutes les préconditions, et on ignore les littéraux supprimés par actions.

### 10.3.3 Restricted precondition

On retire seulement quelques littéraux spécifiques dans les préconditions

### 10.3.4 Ignore-delete-list heuristic

Retirer la liste supprimer de toutes les actions ( supprimer l'action négatifs dans les effet )

### 10.3.5 Domain-independent pruning

Plusieurs états ne sont juste des variantes d'autre états et donc un peu inutile de les visiter et donc on va élaguer certaine branche qui sont symétrique

## 11 Learning from examples

**Learning** : Améliorer la performance des prochaines tâches après avoir fait des observations. On ne peut pas anticiper toutes les situations possibles. Les tâches futures peuvent changer ...

On se concentre seulement sur les learning problem depuis une collection de input/output, apprendre une fonction qui prédit l'output pour un nouveau input

### 11.1 Feedback

3 types de feedback et détermine les 3 types d'apprentissage

- **Supervised Learning** : l'agent observe les inputs et outputs et apprend une fonction qui match les inputs et outputs
- **Unsupervised Learning** : l'agent apprend des patrons des inputs même si aucun output est donné
- **Reinforcement Learning** : l'agent apprend par une succession de récompenses ou de punitions en fonction de ses actions

## 11.2 Supervised Learning

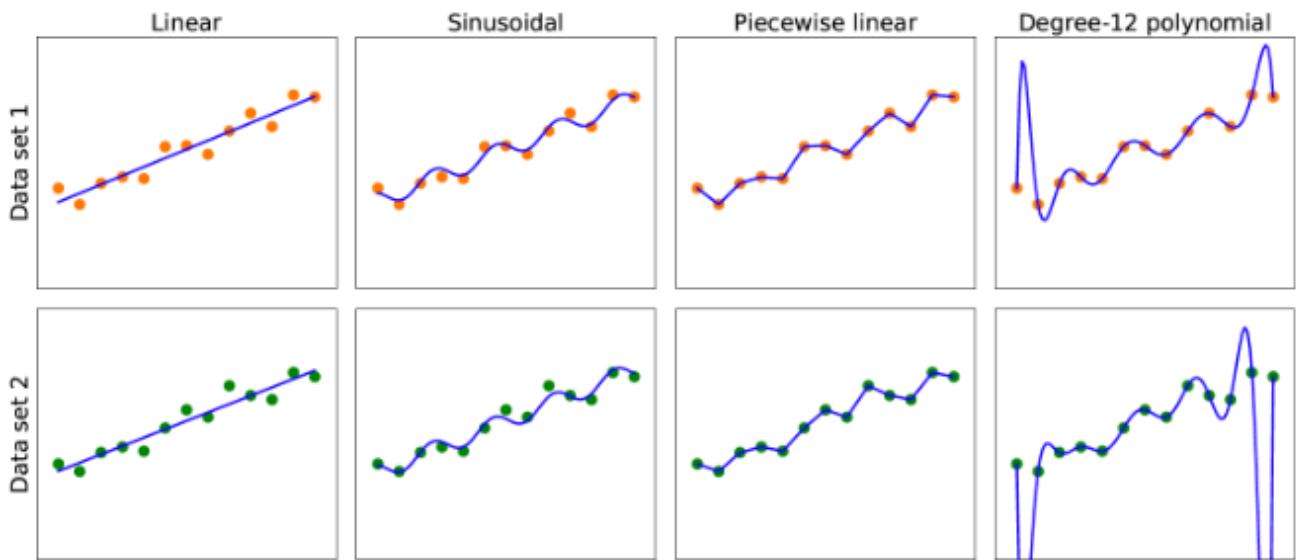
On donne un ensemble d'entraînement  $(x_1, y_1), \dots, (x_n, y_n)$  où  $y_i$  est générée par une  $f(x_i)$  inconnues. On découvre une fonction  $h$  qui approxime  $f$

Pour mesurer la précision de l'hypothèse  $h$  on test sur un ensemble de test qui est différent de l'ensemble des test utilisé pour le learning de l'agent. Un Hypotheses se généralise bien si elle prédit correctement la valeur de  $y$  pour de nouveaux exemple

### 11.2.1 Classification VS. Regression

- **Classification** : la valeur de sortie est tirée d'un ensemble fini de valeurs
- **Regression** : L'output est un nombre

### 11.2.2 Ockam's Razor



On choisit toujours l'hypothèses la plus simple et consistente

On donne plus de priorité aux solutions de degré à 1/2 et moins au plus grand (7/8)

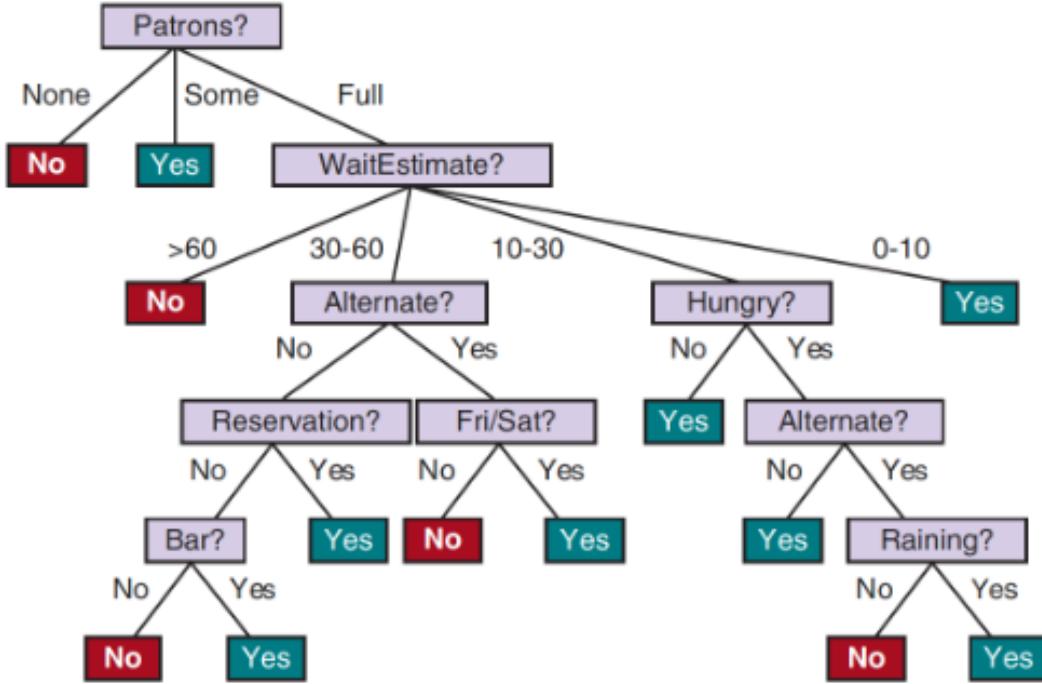
il faut choisir  $h^*$  qui est le plus probable avec les données

$$h^* = \arg \max_{h \in H} P(h|data) \quad (11)$$

$$h^* = \arg \max_{h \in H} P(data|h).P(h) \quad (12)$$

## 11.3 Learning Decision Trees

Représente une fonction qui prend un vecteur de valeur en inputs et retourne une décision (un simple valeur). Il trouve cette décision avec une séquence de tests, car chaque nœud de l'arbre est un test d'une valeur de l'input et les leaf donnent la décision.



1. *Alternate* : Seulement Si il y a un restaurant alternatif a coté
2. *Bar* : Seulement si le restaurant a un bar confortable
3. *Fri/Sat* : True vendredi et samedis
4. *Hungry* : Seulement si on a faim
5. *Patron* : Combien de personne dans le restaurant (None, Some, Full)
6. *Price* : Fourchette de prix
7. *Raining* : Si il pleut dehors
8. *Reservation* : Si on a fait une réservation
9. *Type* : le type de restaurant (francais, Thaie, italien, Burger)
10. *WaitEstimate* : Temps d'attente moyen (0, 10, 10-30, 30-60, >60)

$$Goal \Leftrightarrow (Path_1 \vee Path_2 \vee \dots \vee Path_n) \quad (13)$$

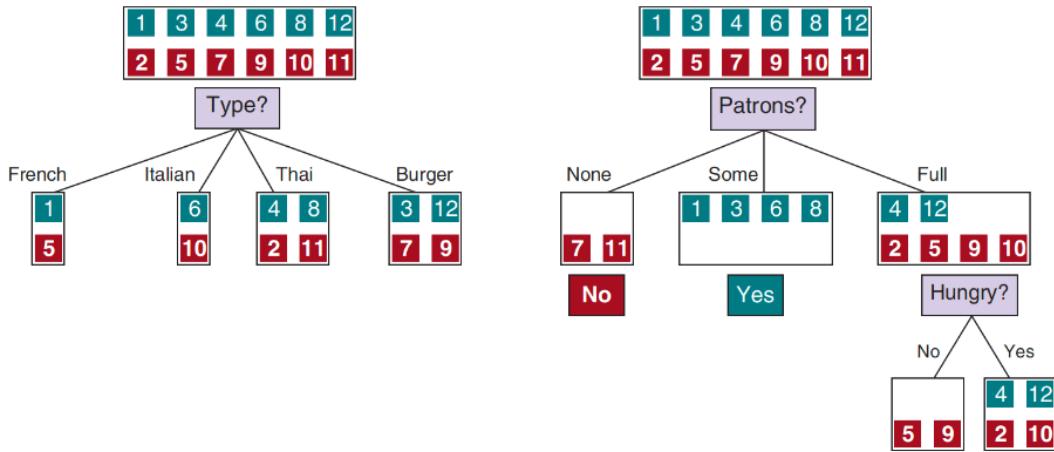
## 11.4 Decision tree from exemple

Exemple sont des tuples  $(X_i, y)$  où  $X_i = x_{i1}, \dots, x_{in}$  sont les attributs en inputs et  $y$  un boolean

Example	Input Attributes										Output WillWait
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
x <sub>1</sub>	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	y <sub>1</sub> = Yes
x <sub>2</sub>	Yes	No	No	Yes	Full	\$	No	No	Thai	30–60	y <sub>2</sub> = No
x <sub>3</sub>	No	Yes	No	No	Some	\$	No	No	Burger	0–10	y <sub>3</sub> = Yes
x <sub>4</sub>	Yes	No	Yes	Yes	Full	\$	Yes	No	Thai	10–30	y <sub>4</sub> = Yes
x <sub>5</sub>	Yes	No	Yes	No	Full	\$\$\$	No	Yes	French	>60	y <sub>5</sub> = No
x <sub>6</sub>	No	Yes	No	Yes	Some	\$\$	Yes	Yes	Italian	0–10	y <sub>6</sub> = Yes
x <sub>7</sub>	No	Yes	No	No	None	\$	Yes	No	Burger	0–10	y <sub>7</sub> = No
x <sub>8</sub>	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	0–10	y <sub>8</sub> = Yes
x <sub>9</sub>	No	Yes	Yes	No	Full	\$	Yes	No	Burger	>60	y <sub>9</sub> = No
x <sub>10</sub>	Yes	Yes	Yes	Yes	Full	\$\$\$	No	Yes	Italian	10–30	y <sub>10</sub> = No
x <sub>11</sub>	No	No	No	No	None	\$	No	No	Thai	0–10	y <sub>11</sub> = No
x <sub>12</sub>	Yes	Yes	Yes	Yes	Full	\$	No	No	Burger	30–60	y <sub>12</sub> = Yes

## 11.5 The Decision-Tree-Learning Algorithm

On veut le tree le plus petit possible, on peut donc choisir l'attribut le plus important ou alors diviser en sous probleme



Quand on crée un arbre de décision, il y a 4 cas possible :

1. Si tous les exemples sont positif (négatif) alors résulte Yes (No)
2. Si il y a des positif et négatif, alors choisir le meilleur attribut pour les splits
3. Si il n'y a plus d'exemple qui reste retourner une valeur défaut
4. Si il n'y a plus d'attribut qui reste, mais que les exemples sont positif et négatif, alors les exemple sont incohérents. Return la classification plurielle des exemples restants (vote majoritaire)

```

function LEARN-DECISION-TREE(examples, attributes, parent-examples) returns a tree
  if examples is empty then return PLURALITY-VALUE(parent-examples)
  else if all examples have the same classification then return the classification
  else if attributes is empty then return PLURALITY-VALUE(examples)
  else
     $A \leftarrow \operatorname{argmax}_{a \in \text{attributes}} \text{IMPORTANCE}(a, \text{examples})$ 
    tree  $\leftarrow$  a new decision tree with root test A
    for each value v of A do
      exs  $\leftarrow \{e : e \in \text{examples} \text{ and } e.A = v\}$ 
      subtree  $\leftarrow$  LEARN-DECISION-TREE(exs, attributes - A, examples)
      add a branch to tree with label (A = v) and subtree subtree
  return tree

```

## 11.6 Entropy

Caclule l'incertitude d'une variable random.

$$H(V) = \sum_k P(v_k) \log_2\left(\frac{1}{P(v_k)}\right) = -\sum_k P(v_k) \cdot \log_2(P(v_k)) \quad (14)$$

ex: piece non truquée (50/50)

$$H(Fair) = -(0.5 \log_2(0.5) + 0.5 \log_2(0.5)) = 1 \quad (15)$$

ex: piece truquée (99/1)

$$H(Unfair) = -(0.99 \log_2(0.99) + 0.01 \log_2(0.01)) = 0.08 \quad (16)$$

L'entropie d'un boolean random ave une probabilité de True a  $q$  (false =  $1 - q$ )

$$B(q) = -(q \cdot \log_2(q) + (1 - q) \cdot \log_2(1 - q)) \quad (17)$$

## 11.7 Learning Curves

On split les exemples en un ensemble de training et test. On apprend des hypotheses et on mesure la précision avec l'ensemble des test.

On commence avec un ensemble l'entrainement de 1 que on augmente au fur et a mesure. On peut remarquer que la précission augmente avec un ensemble de trainning plus large.

## 11.8 Ensemble learning

Générer une collectino/ensemble d'hypothese et combiner leur prédictions. Cela augmente les performance.

### 11.8.1 Bagging

- On génère  $K$  training set avec  $N$  exemple
- Produit  $K$  décision d'algo
- Pour un nouvel input
  - Predit  $k$  hypothese
  - vote a la majorité

### 11.8.2 Random Forest

Produit  $k$  décision algo sur l'exemple donnée

ça varie en fonction du choix des attribut

A chaque split on fait une restriction random des choix des attributs

### 11.8.3 Boosting

Un set d'entraînement avec des poids, chaque exemple associé a un exemple et donc adapte sa méthode d'apprentissage avec les poids.

Génère au début une hypothèse avec un poids de 1, on augmente le poids des exemples mal classés.

On stop quand on a  $k$  hypothèses et on les utilise pour calculer la prediction