

Intelligence Artificielle

- Introduction

Felipe Garrido, Umberto Grandi, Jean-Guy Mailly

Master 1 MIAGE



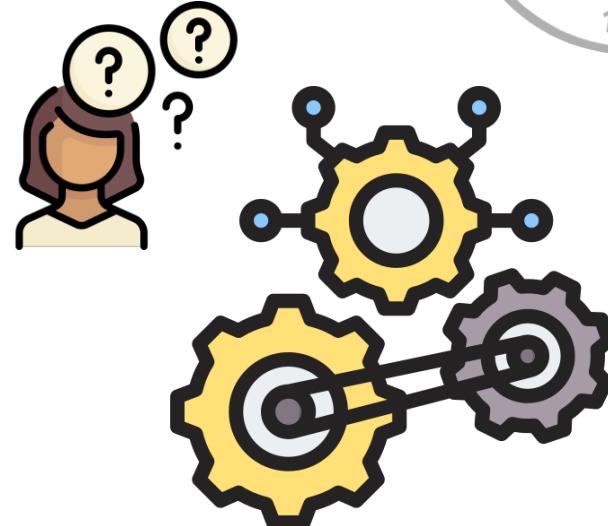
Membre de l'Université
Toulouse Capitole





La motivation du cours

- Comprendre la portée de l'intelligence artificielle
- Savoir distinguer entre différentes techniques et applications de l'IA
- Modéliser des problèmes simples d'IA
- Comprendre des algorithmes classiques
- Savoir sélectionner les techniques appropriées pour résoudre un problème d'IA
- Résoudre des problèmes simples d'IA en Python (avec ou sans librairies)



Evaluation du cours

Evaluation écrite (50%) : 9 avril

Evaluation en TP noté (50%): 21 mars

■ Intervenants

- Introduction et algorithmes de résolution de problèmes: Felipe Garrido, Umberto Grandi
- Apprentissage supervisé et non-supervisé: Sylvain Cussat-Blanc, Benoit Gaudou

■ Matériel

- Slides sur la plateforme en ligne
- Livre de référence: *S. Russel and P. Norvig. Artificial Intelligence: a modern approach.*
- Livre disponible en format électronique en anglais et en français à la BU (lien sur moodle)



IA quèsaco ?



Brainstorming !

- En quel année le terme IA a été proposé pour une discipline scientifique?
- Nommez une application logiciel d'IA
- Nommez une technique ou algorithme d'IA
- Nommez une application qui n'est pas de l'IA
- Nommez une application d'IA qui n'utilise pas de données
- Quel est le but de l'IA?

Quel est le but de l'IA ?



Plusieurs définitions possibles! Notre livre de référence (Russel et Norvig) propose le tableau suivant.

L'IA se propose de développer:



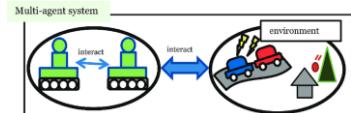
Des systèmes qui pensent comme les humains
(voir Turing test)

Des systèmes qui pensent rationnellement
(approche logiciste)

Des systèmes qui agissent comme des humains
(approche cognitif)



Aristotle



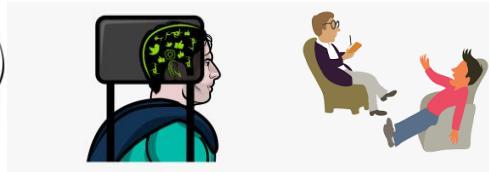
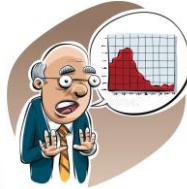
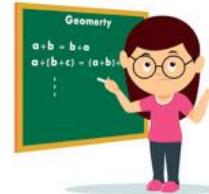
Un approche fortement *interdisciplinaire*

Depuis sa fondation, l'IA est une entreprise interdisciplinaire, où les informaticiens collaborent et empruntent des idées provenant de :

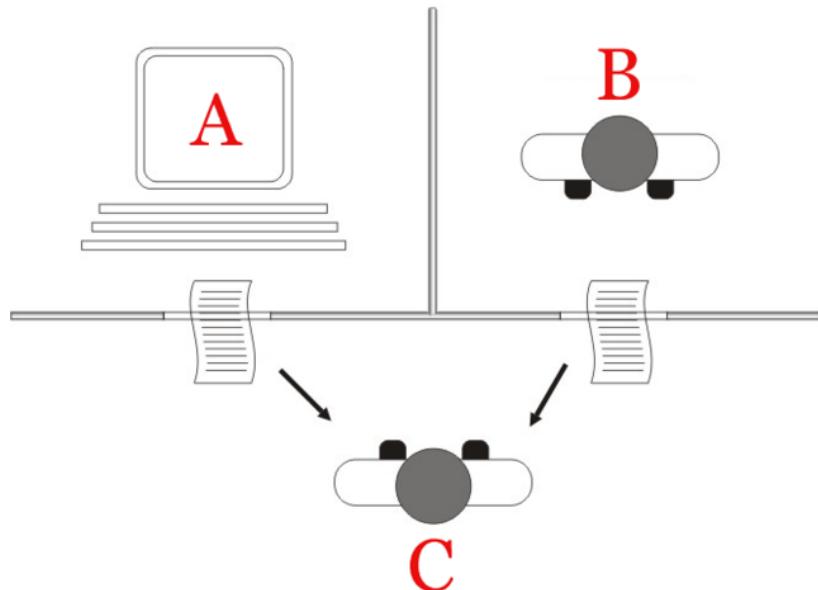


- **Philosophie** : pensée rationnelle, logique, utilitarisme, éthique...
- **Mathématiques** : définition formelle des algorithmes, décidabilité, complexité, probabilités...
- **Économie** : comment prendre des décisions rationnelles et interagir avec d'autres agents rationnels
- **Neurosciences** : s'inspirer du fonctionnement du cerveau humain
- **Psychologie** : s'inspirer de la pensée humaine
- **Génie informatique** : concevoir des machines adaptées aux algorithmes d'IA
- **Linguistique** : comprendre et communiquer avec les humains

...



Le test de Turing (1950)



Un humain en confrontation verbale à l'aveugle avec un ordinateur et un autre humain

Ce test philosophique est *fortement débattu*. Voir par exemple le logiciel Eliza qui passe le test en 1966 en imitant la façon de parler d'un psychothérapeute rogérien



Comment évaluons-nous les IAs ?

Nous utilisons des **challenges** en forme de compétitions ou benchmark :

- **Battre les joueurs humains à des jeux populaires** : les échecs (DeepBlue, IBM, 1997), le Go (AlphaGo Google DeepMind, 2017)
- **Résoudre un problème ouvert dans une autre discipline** : AlphaFold, Google DeepMind, 2018, prix Nobel en 2024
- **Augmenter la précision de prédiction sur des datasets benchmark** : WordNet, ImageNet ... et compétitions associées
- **Résoudre des tests linguistiques** : par exemple les phrases de Winograd « *Les conseillers municipaux ont refusé d'accorder une autorisation aux manifestants parce qu'ils [craignaient/prônaient] la violence.* » l'IA doit choisir entre les deux mots entre []



Les LLMs peuvent tout résoudre

大型语言模型



La récente spectaculaire avancée des LLMs (et ses versions « **agentic AI** » qui prennent des décisions) semblent percer dans nombreuses applications humaines :

- Passent un test de chimie (et autres matières...)
- Trouvent des *bugs* dans du code, sont utilisés pour coder (coPilot...)
- Aident les étudiants à ~~tricher aux examens~~ à étudier et à mieux comprendre une discipline
- Peuvent débattre des sujets politiques à la place des humains
- ...

Les LLMs ont des limitations



Nous ne sommes qu'au début! Ce slide sera certainement à mettre à jour tous les ans!

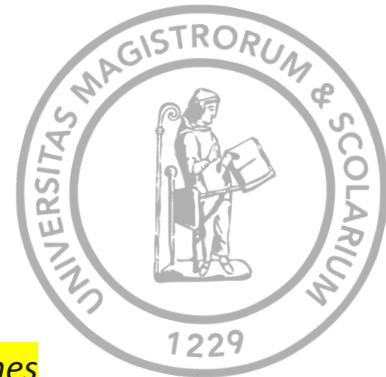
- La production d'une LLM n'est pas certifiable (mais les humains non plus!)
- Hallucinations, phrases très réalistes mais très lointaines de la réalité
- Pas de raisonnement de bon sens ou intuition (exemple: comprendre quand c'est le moment de rire dans un film) 缺乏尝试推理和直觉
- Biais encore très forts 仍然存在严重的偏见 : Par exemple, LLMs tendent à toujours choisir la 1ère option
- Alignement éthique très couteux 伦理对齐成本高昂(*reinforcement learning with human feedback*)
- Robotique encore loin de savoir les utiliser
- ...

Histoire de l'IA: le début



En **1956** dans un workshop au *Dartmouth college* a été exprimé le *principe fondateur* de l'IA :

Chaque aspect de l'apprentissage ou autre activités humaines dites intelligences peut être décrit avec suffisamment de précision pour qu'une machine puisse le simuler

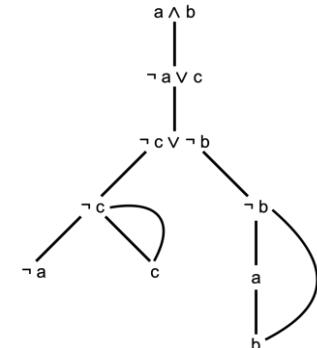
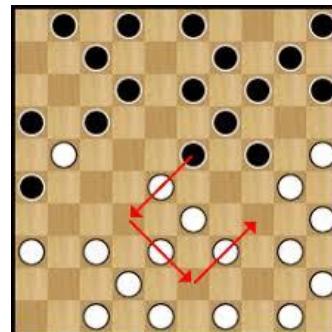


Premiers succès :

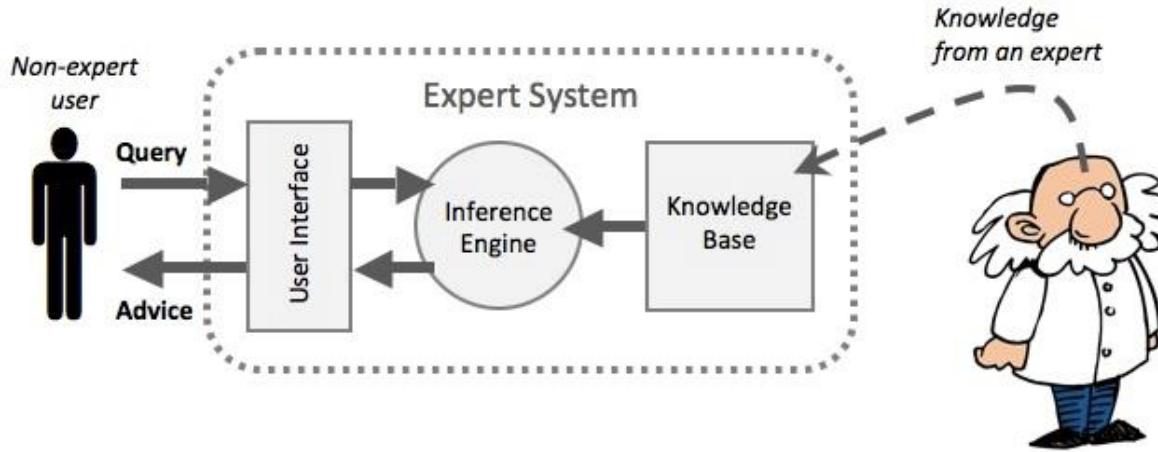
Joueur autonome au jeu de dame

Preuves mathématiques automatisées

Premier « hiver »: résultats bien en dessous des attentes, puissance de calcul et stockage très limitée, absence des données disponibles



Histoire de l'IA: les systèmes experts

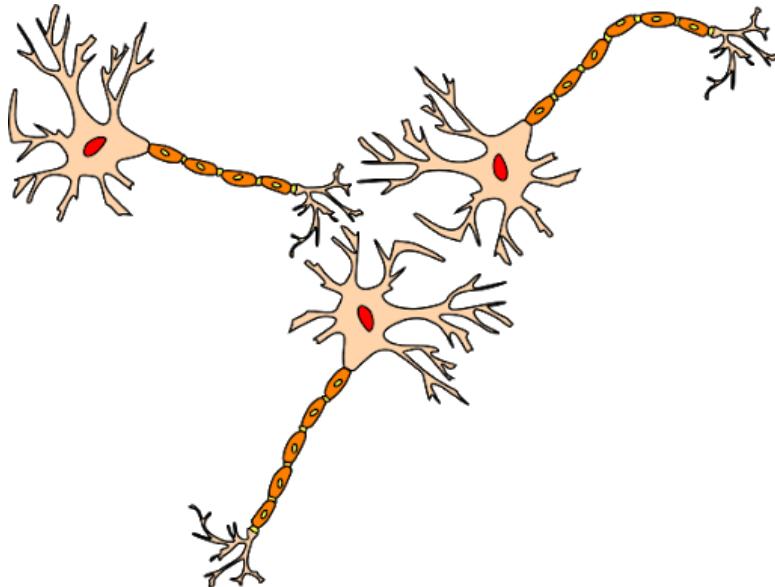


Premières applications ayant eu un impact sur le monde industriel : configuration de produits, planification, diagnostic, etc...

Deuxième « hiver » :

- absence de modèles d'incertitude
- efforts manuels trop importants pour la conception et la maintenance

Histoire de l'IA: l'apprentissage machine



1943 : introduction des réseaux de neurones artificiels

1969 : études sur les limitations des réseaux de neurones

1986 : réseaux de neurones multi-couche

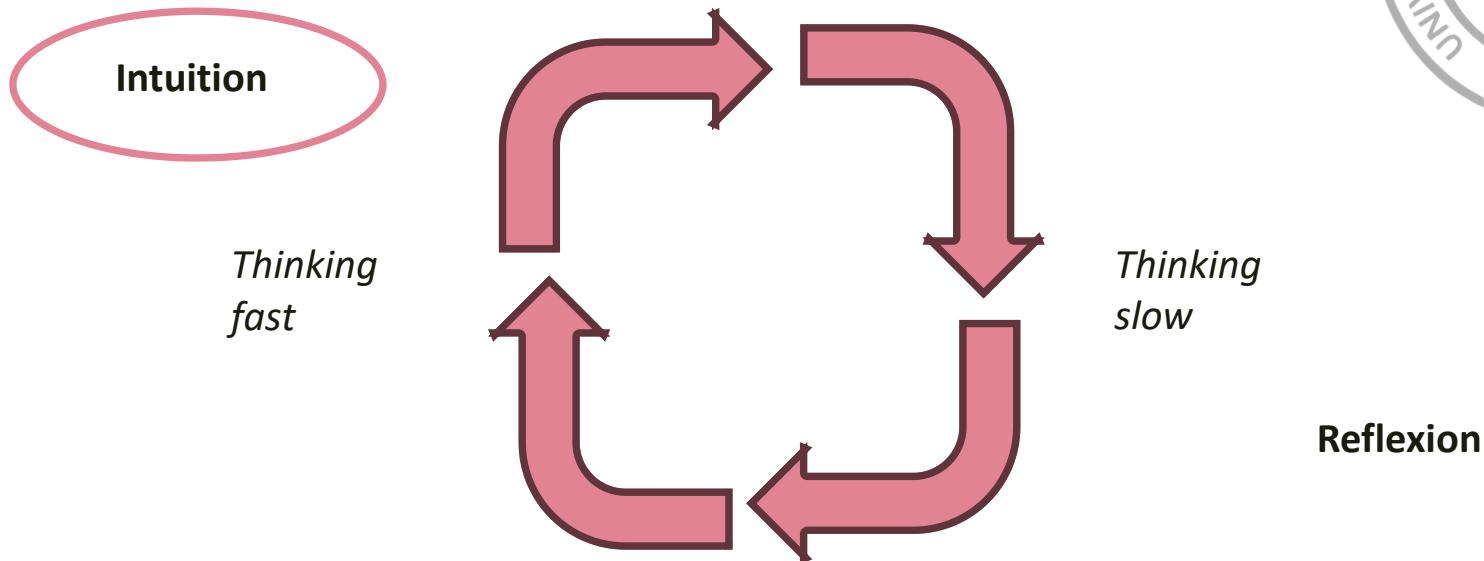
1989 : réseaux de type « convolutional »

> 2012 : percée de l'apprentissage profond (« deep learning »)

Succès grand public en 2017 : AlphaGo gagne contre Lee Sedol au jeu de Go, résultat inattendu ... suivi par une pluie d'applications de succès jusqu'à la genAI et la agentic AI...



Panorama de l'intelligence (artificielle)

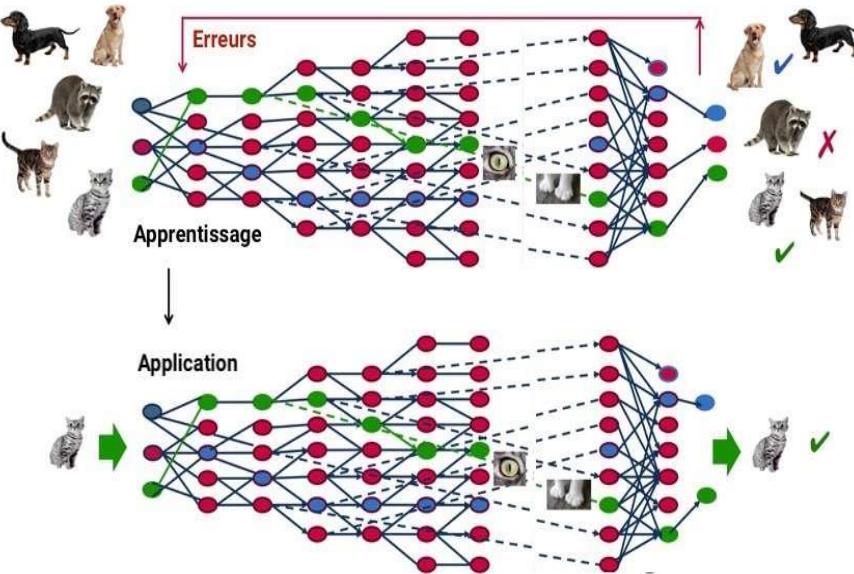


Inspiré du célèbre livre des psychologues Daniel Kahneman and Amos Tversky – Thinking, Fast and Slow, 2011

Modèles pour l'intuition ou les reflexes



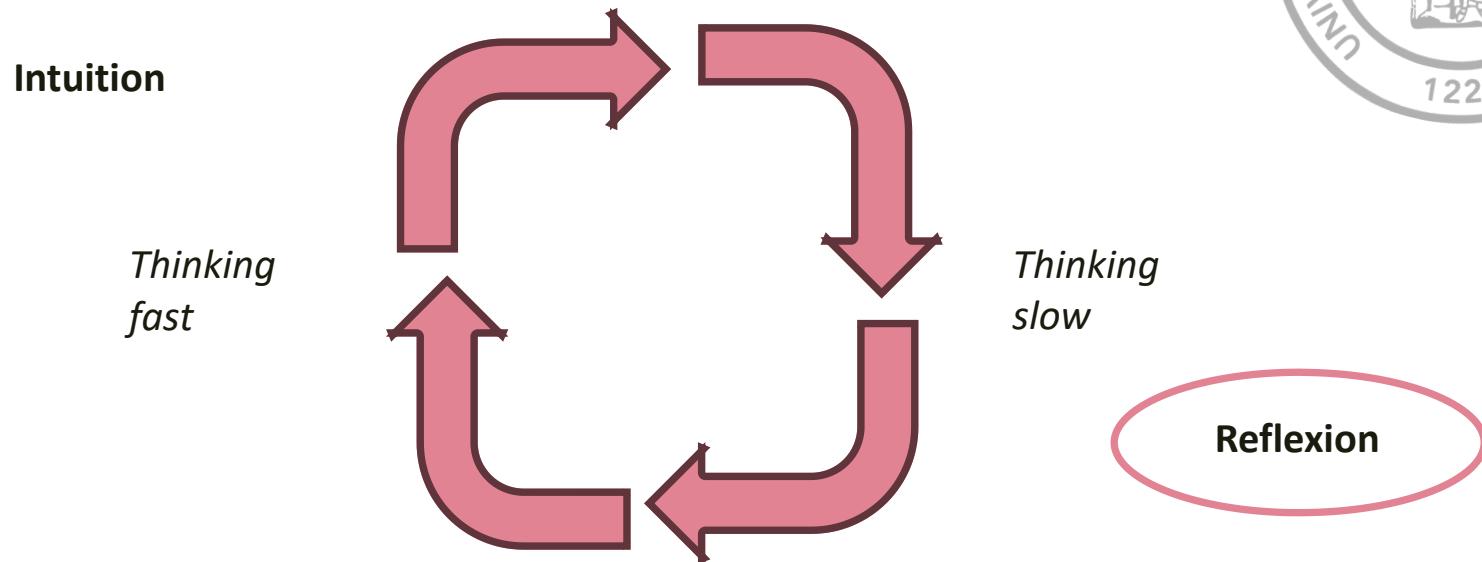
Problème: y-a-t-il un chat ou un chien dans cette image?



聚类 (Clustering) 用于无监督学习，发现数据中的模式和类别

Techniques : algorithmes de bases d'apprentissage machine comme classificateurs et régresseurs (SVM, KNN, réseaux de neurones, random forests) ou clustering

Panorama de l'intelligence (artificielle)



Inspiré du célèbre livre des psychologues Daniel Kahneman and Amos Tversky – Thinking, Fast and Slow, 2011

Modèles de réflexion : assistants virtuels



Ils doivent **interagir** avec les humains dans leur langage

- Répondre à des questions
- Comprendre des informations
- Expliquer leur comportement

回答问题

理解信息

解释自身行为

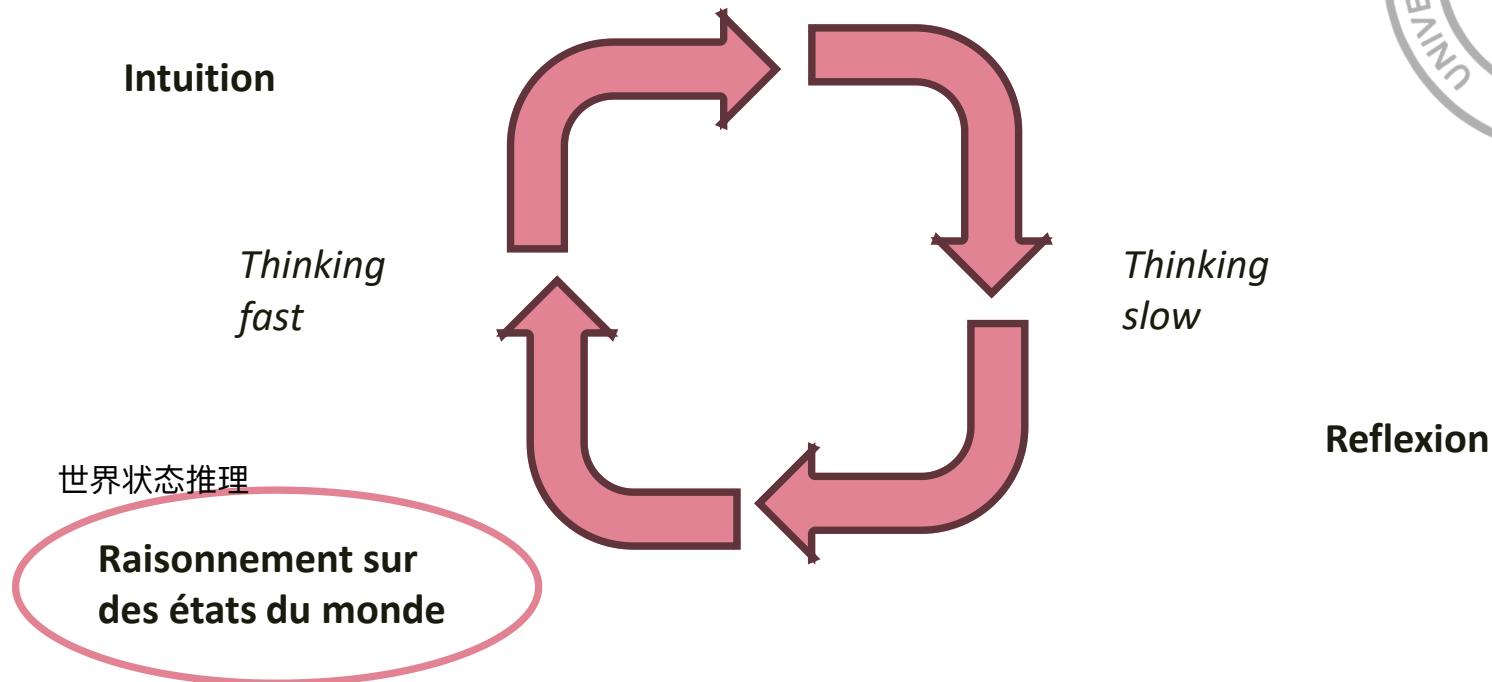
Nécessite d'une capacité de **raisonnement** de haut niveaux :

- 高阶推理能力的需求
- Conceptualisation 概念化
 - Traitement de langage 语言处理
 - Explicabilité 可解释性
 - Théorie de l'esprit 心智理论

Techniques : On ne sait pas encore !

- Les LLMs réussissent parfaitement certaines tâches et échouent terriblement à d'autres
- Le mariage entre **apprentissage** et **logique** semble prometteur, mais, dans l'histoire de l'IA, le raisonnement logique a toujours buté sur des problèmes de ressources computationnelles et de scalabilité...

Panorama de l'intelligence (artificielle)



Inspiré du célèbre livre des psychologues Daniel Kahneman and Amos Tversky – Thinking, Fast and Slow, 2011

Modèles fondés sur des états



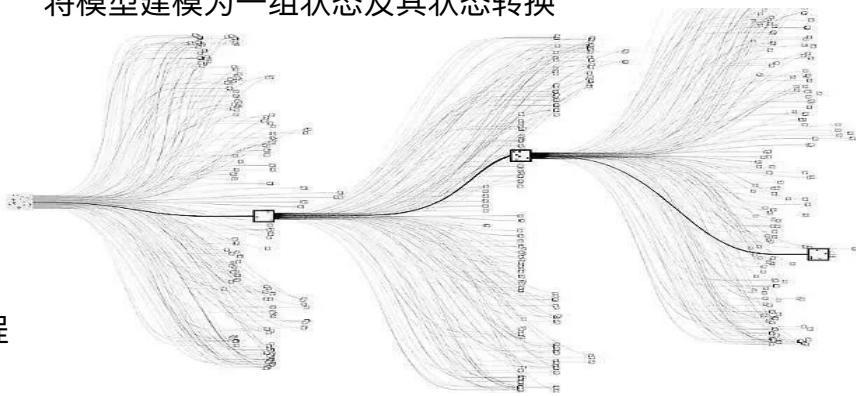
Applications : jeux (échecs, go, pac-man...), robotique (planification du mouvement), génération du langage (tag automatique d'image, traduction automatique)

Idée : modéliser un problème comme un ensemble d'états et des transitions entre les états

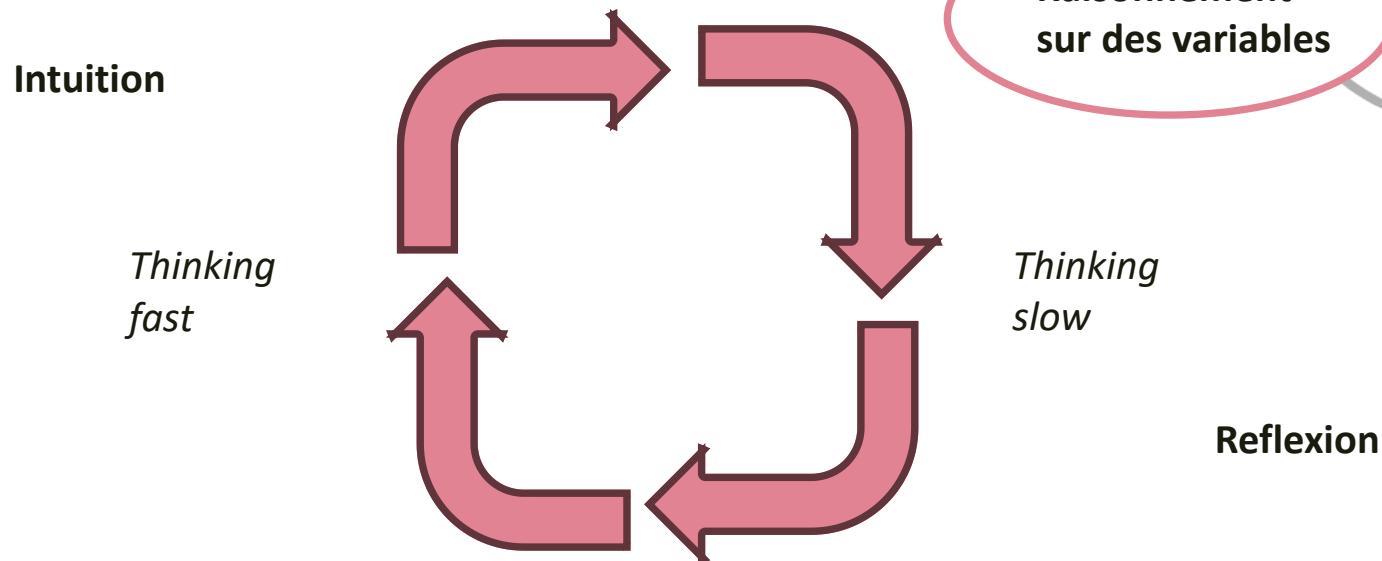
将模型建模为一组状态及其状态转换

Techniques : algo de recherche sur arbre, recherche en adversité, processus de Markov

搜索树，对抗搜索-剪枝，马尔可夫决策过程



Panorama de l'intelligence (artificielle)



Inspiré du célèbre livre des psychologues Daniel Kahneman and Amos Tversky – Thinking, Fast and Slow, 2011

Modèles orientés variables

基于变量的模型



5	3		7			
6			1	9	5	
	9	8				6
8			6			3
4		8	3			1
7			2			6
	6			2	8	
		4	1	9		5
			8		7	9

在某些以状态定义的问题中，状态转换的顺序并不重要
Dans certains problèmes définis avec des états,
l'ordre des transitions n'est pas important pour arriver à la solution. Ex : échecs vs. sudoku

Application : emploi de temps, processus industriels, diagnoses médicales

数独：顺序无关紧要，但是要满足所有约束条件

国际象棋：顺序重要，因为涉及策略和对抗搜索

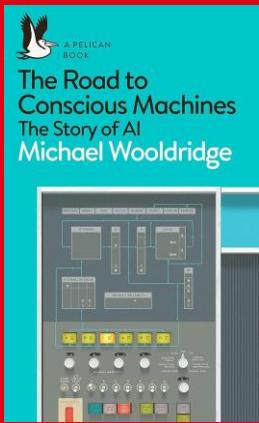
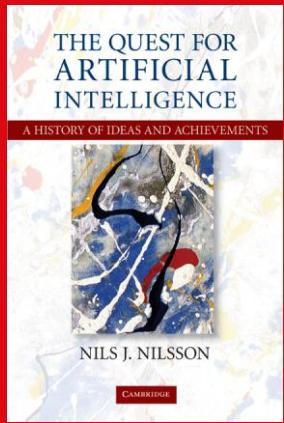
Techniques : satisfaction des contraintes, réseaux bayésiens

:

络



Conclusion



Lectures sur l'histoire de l'IA (à gauche un livre académique, à droite un livre de divulgation)

Matériel: Chapitres 1 et 2 du livre de référence, Intelligence Artificielle, Russel et Norvig

■ **Le but de l'IA, plusieurs approches**

- Test de Turing
- Approche cognitive
- Tradition logicielle
- Approche agent

■ **Histoire**

- Premier pas dans les années '50
- Deux hivers déjà!
- Percée de l'apprentissage automatique

■ **Panorama**

- Intuition: apprentissage
- Réflexion: on ne sait pas!
- Entre les deux, raisonnement sur les états et sur les variables

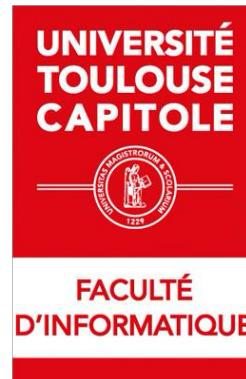


Intelligence Artificielle - Recherche heuristique



Felipe Garrido, Umberto Grandi, Jean-Guy Mailly

Master 1 MIAGE



Membre de l'Université
Toulouse Capitole



Stratégie de résolution de problème

Connaissez-vous le jeu du taquin? Ici en version très simple:



Réfléchissez à un algorithme/script pour le résoudre :

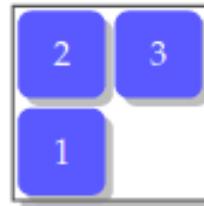
- Combien d'état possibles faut-il prendre en compte?
- Quelles sont les actions à disposition du joueur?
- Quelle est une stratégie de résolution pour le résoudre?

Dans ce cours nous allons étudier des algorithmes qui permettent de **résoudre un problème générique** en le modélisant avec un **espace d'états**



Stratégie de résolution de problème

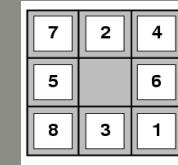
Connaissez-vous le jeu du taquin? Ici en version très simple:



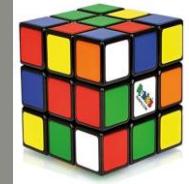
Réfléchissez à un algorithme/script pour le résoudre :

- Combien d'état possibles faut-il prendre en compte?
- Quelles sont les actions à disposition du joueur?
- Quelle est une stratégie de résolution pour le résoudre?

Votre stratégie est utile pour résoudre le jeu du taquin 8x8?



Votre stratégie est utile pour résoudre le cube de Rubik?



Dans ce cours nous allons étudier des algorithmes qui permettent de **résoudre un problème générique** en le modélisant avec un **espace d'états**

Modélisation en espace d'états

状态空间建模

On modélise un problème avec les ingrédients suivants: 使用以下要素建模

- Un ensemble d'**états**, et un état initial 一组状态, 以及一个初始状态
- Pour chaque état, un ensemble d'**actions** disponibles 每个状态对应的一组可用操作
- Un **modèle de transition**, qui associe à chaque pair état-action un état résultant de l'application de l'action choisie dans l'état courant 一个转换模型, 它将对每对状态 - 动作 关联到一个新的状态
- Un **test de réussite**, qui indique les états buts

Objectif: trouver un état but à partir de l'état initial

从初始状态找到目标状态

- Si nous avons une fonction de **cout** qui associe un cout numérique à chaque pair état-action
加入我们有个成本函数, 它对每对 状态 - 动作 分配一个数值成本

Objectif: trouver un état but à partir de l'état initial en minimisant le cout

目标就是找到过程中的最小化成本

Exemple: le jeu du taquin

Modélisation en espace d'états :

- **Etats:** toutes grilles possibles 3x3 remplies avec {0,1,2,3,4,5,6,7,8} (*combien en total?*)
- **Etat initial:** voir image
- **Actions:** bouger la case vide vers le haut, à droite, à gauche, en bas (si possible)
- **Test de réussite:** un seul état but, voir image
- **Cout:** 1 pour chaque action



7	2	4
5		6
8	3	1

Etat initial

1	2	3
4	5	6
7	8	

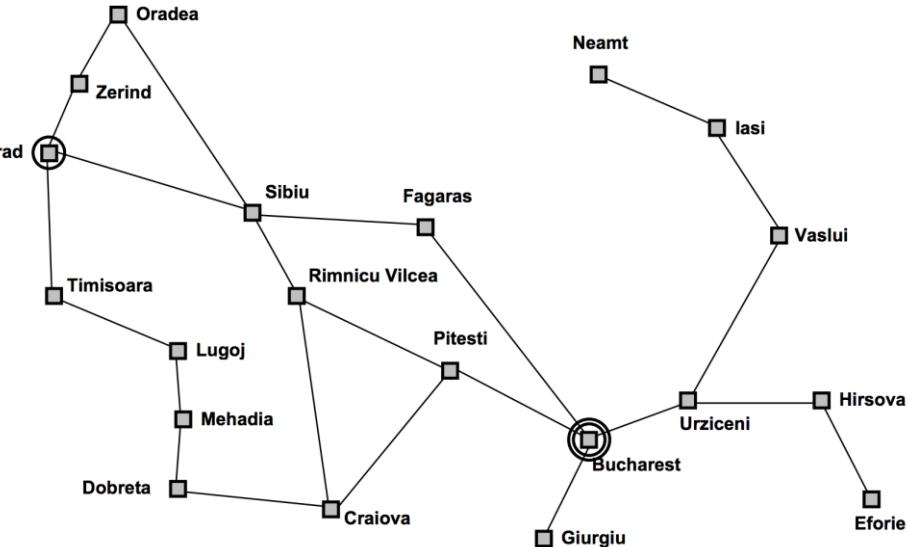
Etat but

Exemple : le plus court chemin



En vacances en **Romanie**, trouvez le plus court chemin de **Arad** à **Bucarest** :

- **Etats:** la liste des villes dans la figure
- **Etat initial:** Arad
- **Actions:** passer d'une ville à l'autre en suivant la carte en figure
- **Test de réussite:** un seul état but, la ville de Bucarest
- **Cout:** ignorez pour le moment



Arbre de recherche

Utilisez la fonction de transition pour générer un arbre de recherche:

- **Racine:** l'état initiale
- **Nœuds:** les états
- **Branches:** une séquence d'actions
- En partant de la racine, **explorer** chaque nœud en appliquant toutes les actions disponibles en suivant le modèle de transition
- **Feuilles:** les nœuds qui n'ont pas de descendance
- **Attention!** Certains états peuvent être **répétés**, un contrôle est nécessaire



Racine



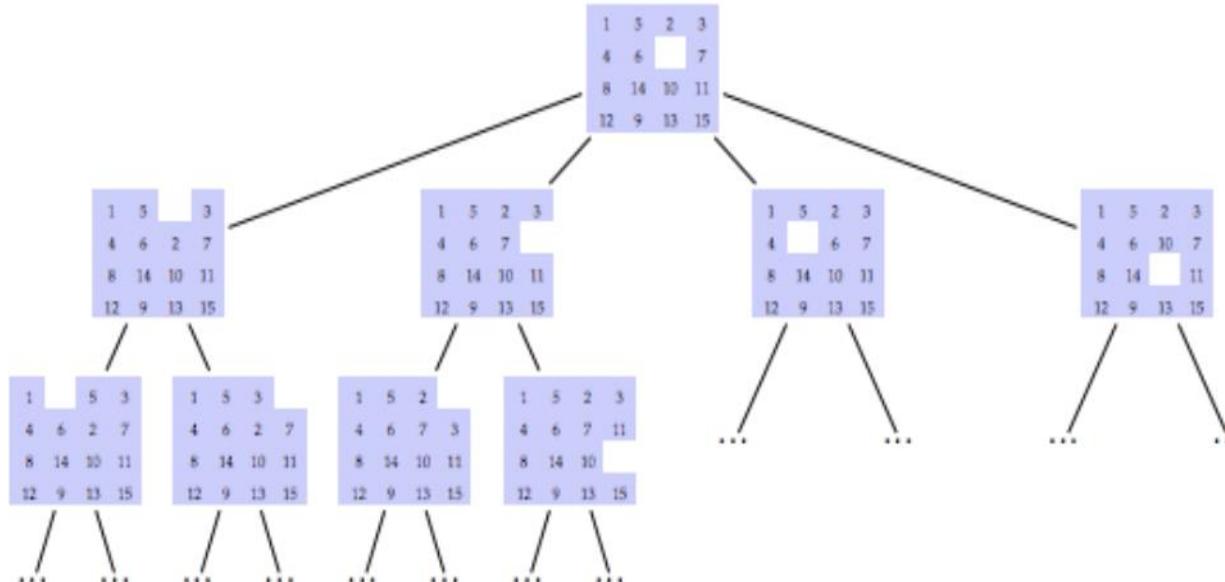
Feuilles



Arbre de recherche

Utilisez la fonction de transition pour générer un arbre de recherche:

- Racine, Nœuds, Branches, Feuilles

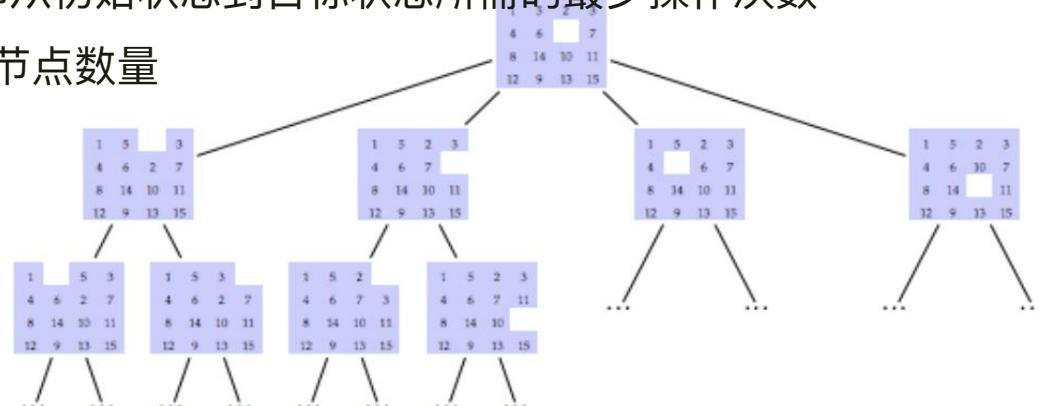




Analyse de l'arbre de recherche

Une première **mesure de complexité d'un problème** est la taille de son arbre de recherche correspondant : 衡量问题复杂性的首要标准是其对应的搜索树大小

- **Nombre d'états** 数量状态
- **Facteur de branchage:** combien d'actions sont disponibles dans un état (regardez le pire cas) 分支银子，每个状态下可执行的操作数量 (考虑最坏情况)
- **Profondeur de l'arbre:** nombre d'actions avant de trouver un état but (regardez le pire cas) 树的深度，即从初始状态到目标状态所需的最少操作次数
- **Nombre de feuilles** 叶子节点数量





Exemple: le jeu du taquin 3x3

Analyse de l'arbre de recherche:

- **Nombre d'états :**
- **Facteur de branchage :**
- **Profondeur :**
- **Nombre de feuilles (états buts) :**



Exemple: le jeu du taquin 3x3

Analyse de l'arbre de recherche:

- **Nombre d'états :** $9! = 362\ 880$
- **Facteur de branchage :** 4 (nombre maximal d'action disponible dans un état)
- **Profondeur :** infinie (répétitions possibles!) 无限, 因为可能存在重复状态导致无限循环
- **Nombre de feuilles (états buts) :** 1 -- 唯一的目标状态

在 3×3 华容道 (滑块拼图) 中, 分支因子 (Facteur de branchage) 指的是每个状态下最多可以执行的移动数量, 即空格 (0) 可以移动的方向。

为什么最大是 4?

- 空格 (0) 可以向: 上、下、左、右 移动 (如果不在边界)。
- 但是, 并不是所有状态都能执行 4 个动作, 具体情况如下:

位置	可能的移动方向	可执行操作数
四角 (4个)	只能移动 2 个方向	2
边缘但不在角上 (4个)	可以移动 3 个方向	3
中心 (1个)	可以移动 4 个方向	4

最坏情况下 (空格位于中心), 有 4 个可执行的移动方向, 因此最大分支因子是 4。

L'art de modéliser



Dall-E (2024)



■ Une activité cruciale

- Si vous devez choisir quoi retenir de ce cours focalisez-vous sur la **modélisation!**

■ Abstraction 抽象

- Le monde est complexe!
- Nécessaire de choisir

■ Algorithmes génériques de résolution 通过求解算法

- Combinez l'intelligence humaine (le modélisateur) avec la puissance de calcul d'un ordinateur 结合人类智慧，并利用计算机的强大能力

Recherche non-informée

Quelle est la meilleure manière de parcourir un arbre pour trouver un état but ?

■ Profondeur d'abord (DFS : Depth First Search)

深度优先方法 优先探索最深路径

🔍 什么是“优先深入”？

“优先深入” (Depth First) 是深度优先搜索 (DFS) 的核心概念，指的是：

- ➡ 每次选择可行的操作时，优先沿着当前路径继续往下探索，而不是同时考虑多个可能的路径。
- ➡ 如果到达死胡同或目标状态，就回溯到上一步，换一条路径继续深入。

🛠️ 直观理解

假设你在一个迷宫里寻找出口，你的策略是：

- 1 沿着一条路一直走到底 (“深入”)。
- 2 如果发现死路，就回头（回溯），选择另一条路。
- 3 继续重复这个过程，直到找到出口。



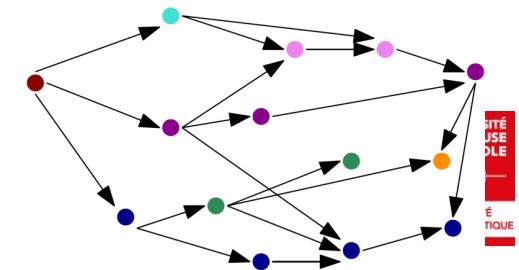
🔍 深度优先搜索 (DFS: Depth First Search)

• 方法:

- 从根节点开始，优先探索最深的路径，直到找到目标状态或到达叶子节点。
- 如果到达死胡同，则回溯到上一个分支点，继续探索新的路径。

• 特点:

- ✓ 内存占用少（只需存储当前路径）。
- ✓ 可能快速找到解（如果解在深层）。
- ✗ 可能陷入无限深度（如果没有循环检测或深度限制）。
- ✗ 可能不是最优解（如果最短路径在较浅层）。



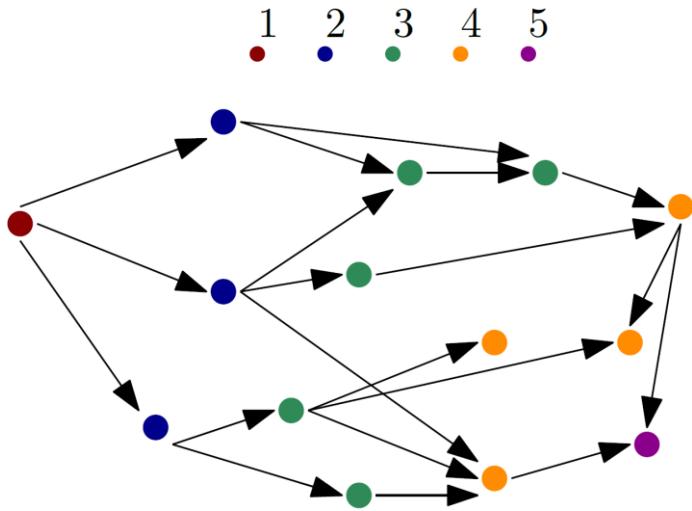
Recherche non-informée

广度优先搜索



Quelle est la meilleure manière de parcourir un arbre pour trouver un état but ?

■ Largeur d'abord (BFS : Breadth First Search)



● 广度优先搜索 (BFS: Breadth First Search)

◆ 方法:

- 从根节点开始，优先探索当前层的所有节点，然后再逐层向下扩展。
- 使用队列 (Queue) 结构来存储待探索的节点。

◆ 特点:

- ✓ 保证找到最优解 (如果所有操作的代价相同)。
- ✓ 不会陷入无限循环 (如果搜索空间有限)。
- ✗ 内存占用较大 (需要存储当前层的所有节点)。
- ✗ 如果目标状态在深层，搜索速度可能较慢。

Recherche non-informée



Quelle est la meilleure manière de parcourir un arbre pour trouver un état but ?

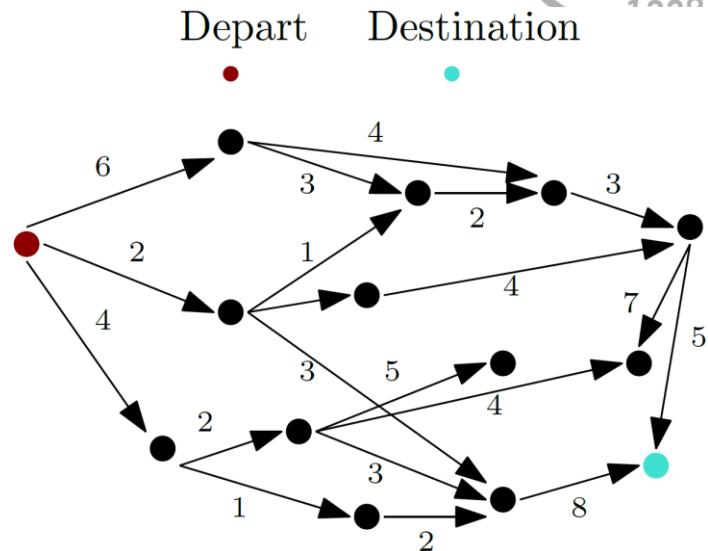
■ Cout uniforme (Dijkstra)

均匀成本搜索，目标是找到从起点到终点的最短路径
(即最低成本路径)

❖ Dijkstra 算法的搜索过程

- Dijkstra 算法是一种基于均匀成本的搜索方法，它的执行方式如下：
- ① 从起点开始，将初始节点的代价设为 0，所有其他节点的代价设为 ∞ (无穷大)。
 - ② 按照代价最小的路径扩展 (优先访问当前成本最低的未访问节点)。
 - ③ 更新相邻节点的代价：如果通过当前节点的路径更短，则更新该节点的成本。
 - ④ 重复该过程，直到找到目标状态 (最小代价路径)。

■ (plusieurs autres possibles)

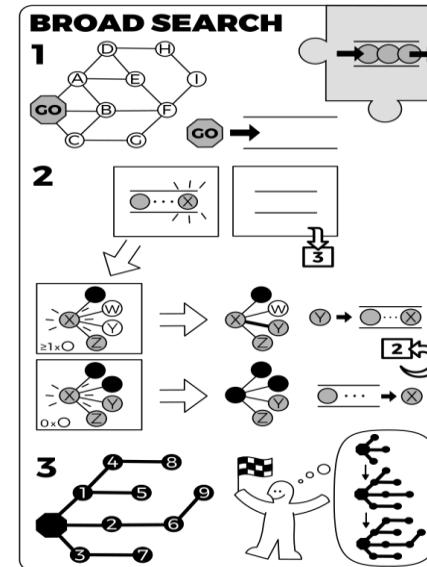
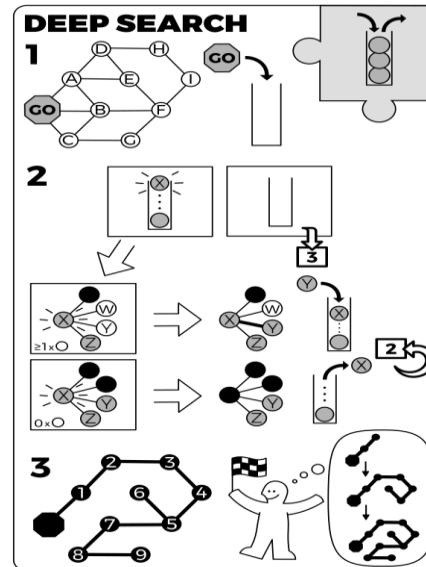
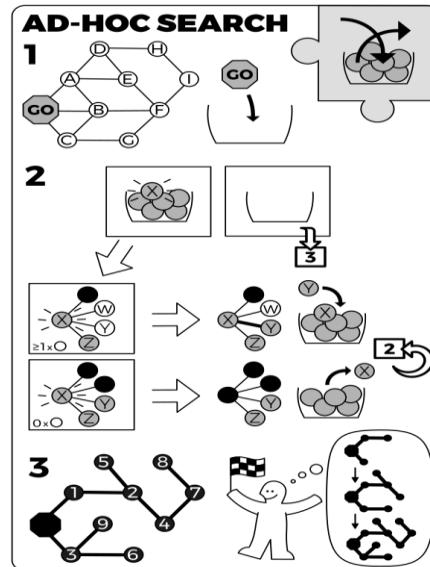
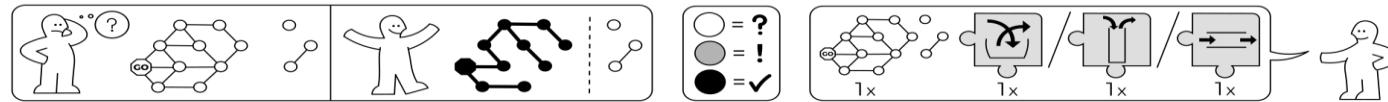




Manuel IKEA de la recherche arborescente

GRÅPH SCÄN

idea-instructions.com/graph-scan/
v1.1, CC by-nc-sa 4.0



Profondeur d'abord

Largeur d'abord

Structure de données pour construire des arbres de recherche

用于构建搜索树的数据结构



搜索树 (Arbre de recherche) 一组带有指针的节点 (nœuds)

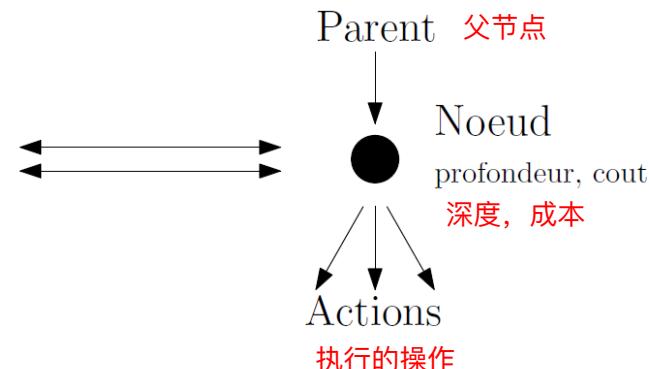
Un arbre de recherche est représenté par un ensemble de nœuds avec pointeurs.

Chaque nœud N a les propriétés suivantes:

- **N.ETAT:** l'état associé au nœud N
状态
- **N.PARENT:** le nœud qui a généré N
- **N.ACTION:** l'action utilisée en N.PARENT pour générer N
- **N.PROF:** la profondeur de N,
le nombre d'action depuis la racine
- **N.COUT:** le cout du chemin de la racine à N (optionnel)

Etat

5	4	
6	1	8
7	3	2



Pseudocode pour recherche arborescente

```
def graph_search(problem):
    n = Node( state = problem.initial_state )
    if problem.goal_test( n.state ):
        return n
    # Initialize frontier and explored set
    Frontier = [n]
    explored = set()
    while frontier non vide:
        n = frontier.pop(*)
        explored.add(n.state)

        for action in problem.actions(n.state):
            child = problem.child_node(n, action)
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return solution(child)
            frontier.append(child)
    return 'FAIL'
```

Perform graph search to find a solution to the problem
Create the initial node
Check if the initial state is the goal
Return the solution node
A list representing the frontier,
A set to store explored states
Take a node from the frontier (implementation determines the strategy. For example, for BFS: pop(0), for DFS: pop())
Add the node's state to the explored set
Expand the node: iterate over actions
Generate the child node
Check if the child's state is new
If the child node's state satisfies the goal, return the solution
Add the child to the frontier
If the frontier is empty and no solution is found, return "FAIL"

Goal Test au moment d'ajout à la Frontier

* Si nous n'enregistrons pas les nœuds déjà explorés, on parle de la variante **TreeSearch**



Comment évaluer un algorithme de recherche arborescente?

En partant des paramètres d'un arbre de recherche (le facteur de branchage b , la profondeur moyenne d'un but d , la longueur maximale d'une branche m ...) calculer:

目标状态的平均深度 d

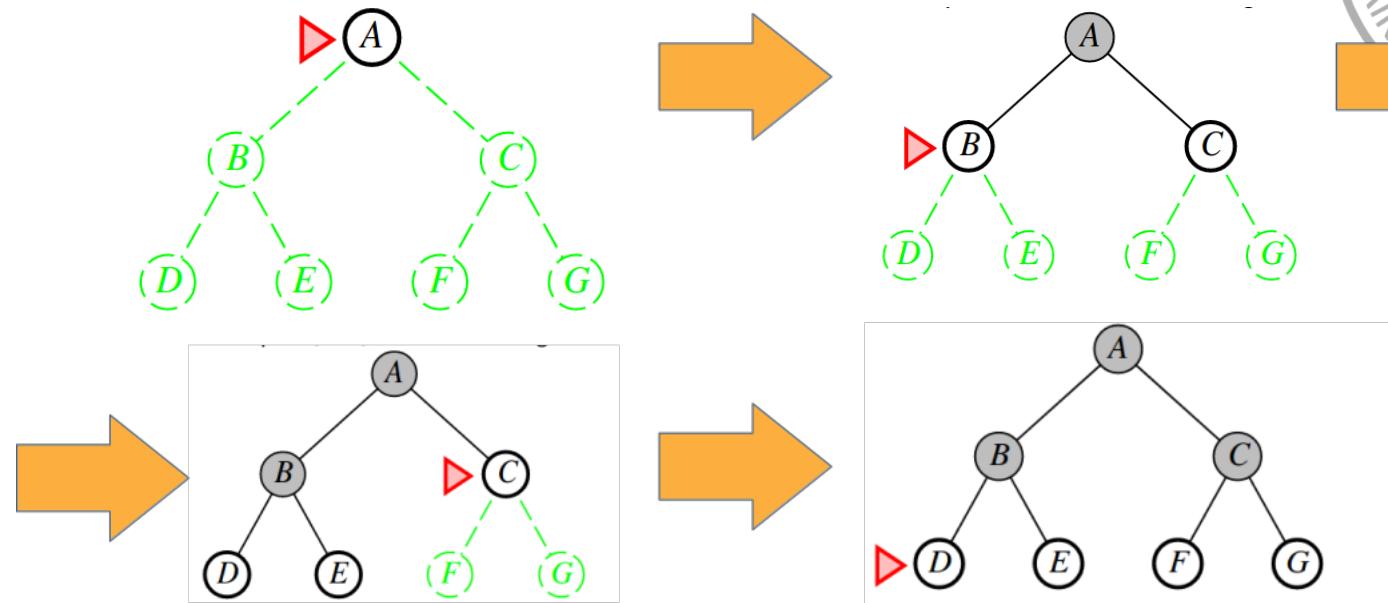
分支因子 d

最大分支深度 m

- **Complétude:** l'algo trouve un état but à chaque fois qu'il en existe un ?
完备性 算法能否找到目标状态?
- **Complexité en temps:** nombre de nœuds à générer avant de trouver un but (pire cas)
时间复杂程度
- **Complexité en espace:** nombre de nœuds à garder en mémoire avant de trouver un but (pire cas)
空间复杂程度 -- 算法在最坏情况下需要储存多少节点
- **Optimalité:** l'algo trouve toujours la solution la moins couteuse? (optionnel)

最优性

Recherche en largeur d'abord (BFS)



Pseudocode pour BFS

```
def BFS(problem):  
    n = Node( state = problem.initial_state )  
    if problem.goal_test( n.state ):  
        return n  
    # Initialize frontier and explored set  
    Frontier = [n] FIFO queue  
    explored = set()  
    while frontier non vide:  
        n = frontier.pop(0)  
        explored.add(n.state)  
  
        for action in problem.actions(n.state):  
            child = problem.child_node(n, action)  
            if child.state not in explored and child not in frontier:  
                if problem.goal_test(child.state):  
                    return solution(child)  
                frontier.append(child)  
  
    return 'FAIL'
```

Perform graph search to find a solution to the problem
Create the initial node
Check if the initial state is the goal
Return the solution node
A list representing the frontier,
A set to store explored states
Take a node from the frontier (implementation determines the
strategy. For example, for BFS: pop(0), for DFS: pop())
Add the node's state to the explored set
Expand the node: iterate over actions
Generate the child node
Check if the child's state is new
If the child node's state satisfies the goal,
return the solution
Add the child to the frontier
If the frontier is empty and no solution is found, return "FAIL"



Analyse

⌚ 时间复杂度 (Complexité en temps)

- 最坏情况：目标状态位于最深层（即搜索树的右下角）。
- 设 b 为分支因子， d 为目标深度，则：

$$O(b^d)$$

解释：

- 第 1 层有 b 个节点
- 第 2 层有 b^2 个节点
- 第 d 层有 b^d 个节点
- BFS 需要生成所有 b^{d+1} 个节点才能找到目标状态（最坏情况）。

BFS（广度优先搜索）在不同深度下的计算需求（时间 & 内存）揭示了其在大规模问题上的局限性

- **Complétude:** Oui. Si une solution se trouve à profondeur d , BFS la trouvera après avoir complété les d premiers niveaux de l'arbre
- **Complexité en temps:** le pire cas se réalise quand la solution est dans « l'angle en bas à droite ». Si b est le facteur de branchage il faut générer $O(b^d)$ nœuds
- **Complexité en espace:** dans le pire cas tous les nœuds doivent être gardé soit en mémoire Exp soit dans la frontière Front, donc $O(b^d)$
- **Optimalité:** pas de cout des actions pour l'instant

💾 空间复杂度 (Complexité en espace)

- 最坏情况下，需要存储整个 d 层的所有节点。
- 由于 BFS 需要在 `Explored`（已探索节点集合）和 `Frontier`（前沿队列）**中存储所有状态，最坏情况下：

$$O(b^d)$$

- 需要存储所有已生成的节点，因此空间消耗较大。
- 比 DFS 更消耗内存（DFS 只存储当前路径，BFS 存储整个搜索层）。

⌚ 最优性 (Optimalité)

- 目前假设所有动作代价相同，所以 BFS 保证找到最短路径。
- 如果路径代价不同，BFS 不能保证找到最优解（可以使用 均匀代价搜索 (Dijkstra) 或 A 搜索*）。

✓ 完备性 (Complétude)

- BFS 是完备的，意味着如果解存在，它一定能找到。
- 如果目标状态在深度 d 处，BFS 会在扩展完整个 d 层后找到目标。



BFS en pratique

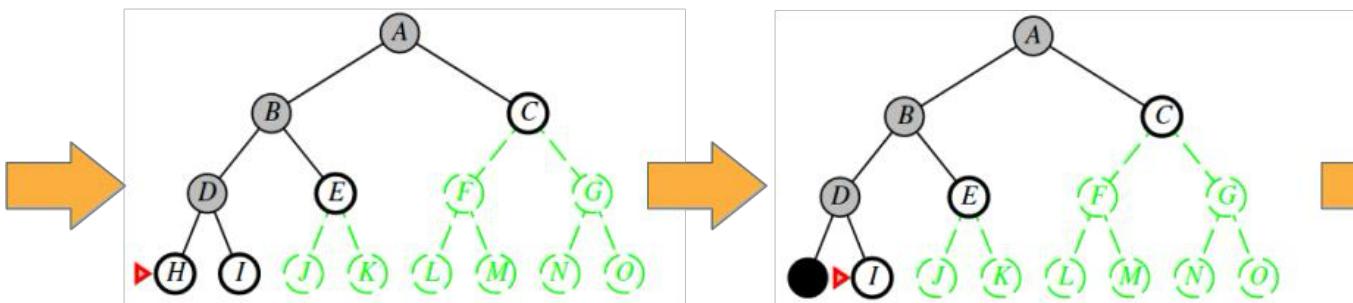
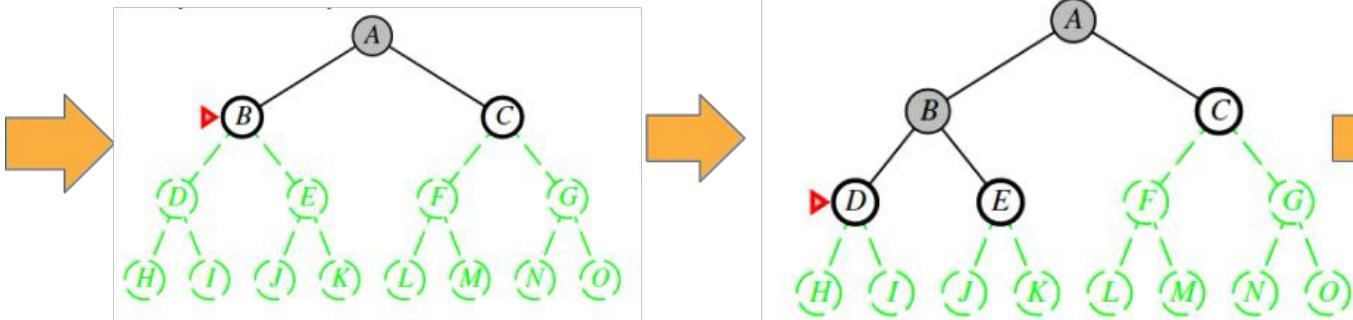
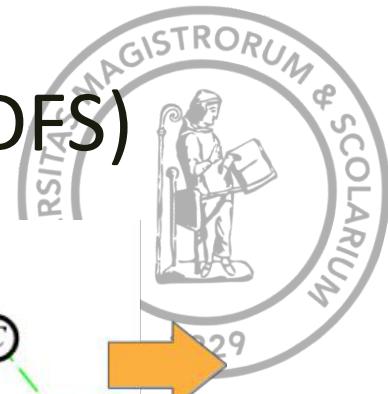
BFS (广度优先搜索) 在不同深度下的计算需求 (时间 & 内存) 揭示了其在大规模问题上的局限性

Profondeur	Nœuds	Temps	Mémoire
2	10^2	0.11 millisecondes	107 kilobytes
4	10^4	11 millisecondes	10.6 mégabytes
6	10^6	1.1 secondes	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 heures	10 téabytes
12	10^{12}	13 jours	1 pétabyte
14	10^{14}	3.5 ans	99 téabytes
16	10^{16}	350 ans	10 exabytes

Résultats obtenus sur un arbre avec $b = 10$, production de 10^6 nœuds par second, 1000byte par nœud. **Conclusion:** BFS n'est pas adapté pour des problèmes exponentiels!

BFS 不适合指数问题

Recherche en profondeur d'abord (DFS)



Pseudocode pour DFS

```
def DFS(problem):
    n = Node( state = problem.initial_state )
    if problem.goal_test( n.state ):
        return n
    # Initialize frontier and explored set
    Frontier = [n] LIFO queue
    explored = set()
    while frontier non vide:
        n = frontier.pop(-1)
        explored.add(n.state)

        for action in problem.actions(n.state):
            child = problem.child_node(n, action)
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return solution(child)
                frontier.append(child)

    return 'FAIL'
```

Perform graph search to find a **solution** to the problem
Create the initial node
Check if the initial state is the goal
Return the solution node
A list representing the frontier,
A set to store explored states
Take a node from the frontier (implementation determines the
strategy. For example, for BFS: pop(0), for DFS: pop(-1))
Add the node's state to the explored set
Expand the node: iterate over actions
Generate the child node
Check if the child's state is new
If the child node's state satisfies the goal,
return the solution
Add the child to the frontier
If the frontier is empty and no solution is found, return "FAIL"



Analyse

- **Completude:** Oui, dans la version GRAPHSEARCH et dans des espaces de recherche fini. Non, si le nombre d'états est infini ou dans la version TREE SEARCH (à cause des cycles).
- **Complexité en temps:** le pire cas est quand la solution est dans « l'angle en bas à droite ». Si b est le facteur de branchage il faut générer $O(b^d)$ nœuds
- **Complexité en espace:** dans le pire cas tous les nœuds doivent être gardé soit en mémoire Exp soit dans la frontière Front, donc $O(b^d)$
- **Optimalité:** pas de cout des actions pour l'instant
L'intérêt de DFS est dans dans la version TREESEARCH.

Sa complexité en espace est de $O(bm)$, donc elle n'est pas exponentielle!

m : longueur maximale d'une branche

✓ 完备性 (Completeness)

- 是 (Oui), 如果使用 **Graph Search** (存储已探索的状态) 且搜索空间**有限**, DFS 是**完备的** (保证找到解)。
- 不是 (Non), 如果搜索空间是**无限的**或使用 **Tree Search** (不存储已探索状态), DFS 可能陷入**无限循环** (由于重复访问相同状态)。

⌚ 时间复杂度 (Complexité en temps)

- 最坏情况: 目标状态位于**最深层** (搜索树的右下角)。
- 设 b 为分支因子, d 为目标深度, 则:
$$O(b^d)$$
- DFS 可能需要遍历所有可能路径, 导致指数级增长。

💾 空间复杂度 (Complexité en espace)

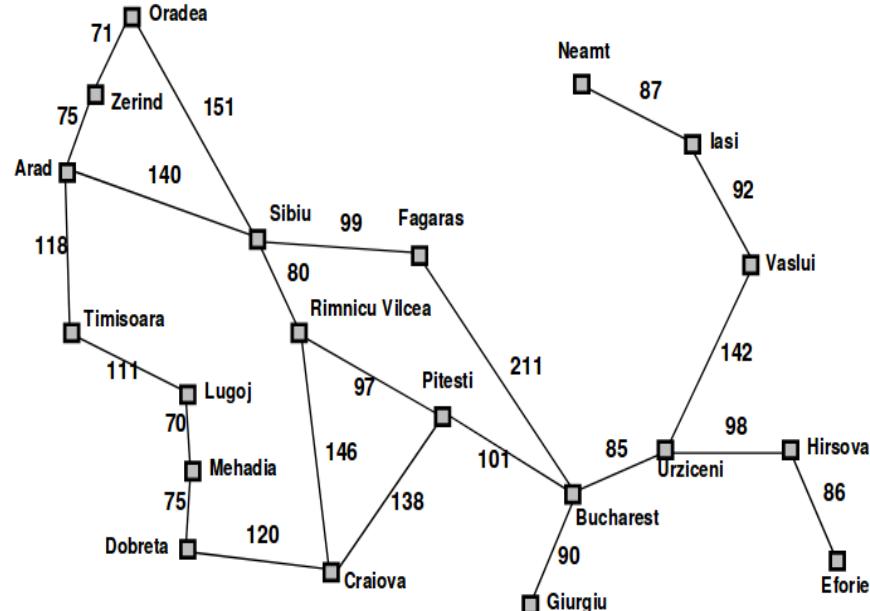
- 最坏情况下, 所有已探索的节点都必须存储在:
 - **Explored** (已访问集合)
 - **Frontier** (前沿队列)
- 复杂度:
$$O(b^d)$$
- 但在 Tree Search 版本中, DFS 只存储当前路径, 因此空间复杂度仅为:
$$O(bm)$$
- 相比 BFS ($O(b^d)$), DFS 需要的内存远小得多!

⭐ DFS 的优势

- DFS 在 Tree Search 版本下更有优势, 因为它的空间复杂度为 $O(bm)$, 不是指数级的。
- 适用于搜索空间大, 但解可能在较深层的情况。
- 如果路径代价相同, 使用 BFS 可确保最短路径; 如果内存受限, DFS 是更好的选择。



Quoi faire quand les actions ont un cout?



关键问题

1 如果所有路径的代价相同 (Unifome)

- BFS (广度优先搜索) 是最优的
- DFS (深度优先搜索) 不保证找到最优解
- 这是因为 BFS 按层级扩展, 保证在最短步数内找到目标。

2 如果路径的代价不同

- BFS 不能保证最优解 (因为它忽略了路径成本, 只考虑步数)。
- 需要使用代价敏感的搜索算法, 如:
 - 均匀代价搜索 (UCS, Uniform Cost Search, Dijkstra 算法)
 - A 搜索 (结合启发式) *

Si les actions ont le même cout alors BFS est optimale (et DFS?).
 Quoi faire si les actions n'ont pas le même cout?

Recherche à cout uniforme (Dijkstra)

均匀代价搜索 UCS



Idée : au moment de choisir le nœud à explorer, choisir le nœud N qui
minimise le cout $c(N)$ du chemin de la racine à N

Des importantes modifications au pseudocode GRAPHSEARCH:

- Utiliser une file d'attente FIFO mais avec une priorité calculée avec $c(N)$
- Tester le but au moment de l'expansion, pas au moment de l'ajouter à la frontière
- Implémenter le POP pour explorer les nœuds dans la frontière à cout minimale

② 在扩展节点时进行目标测试

- 不要在加入 `frontier` (前沿) 时测试是否为目标状态。
- 应该在节点被扩展 (POP 出队列) 时检查是否为目标，以保证最优性。

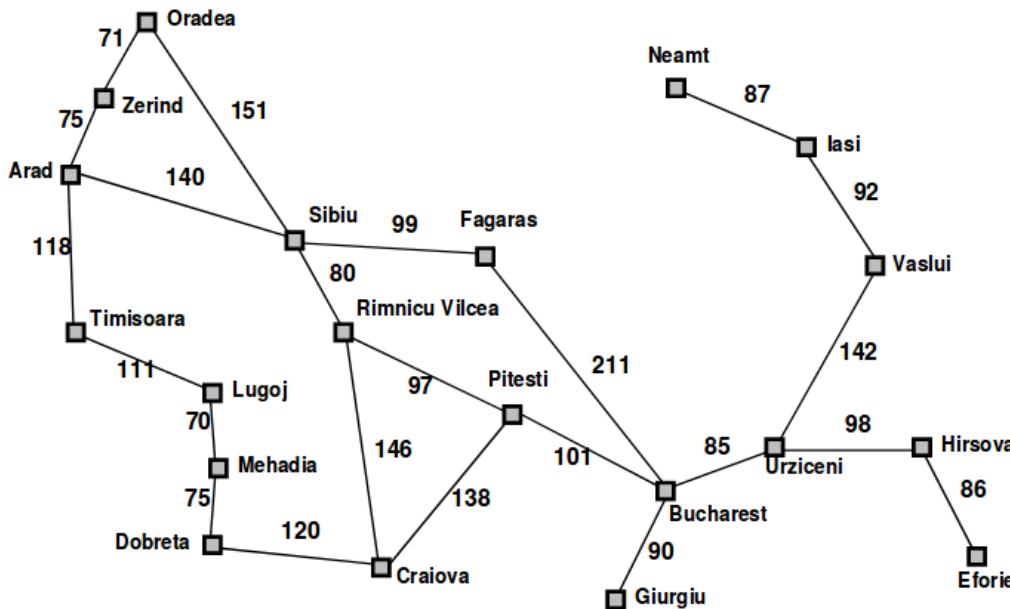
Pseudocode pour Dijkstra

```
def dijkstra(problem):
    n = Node( state = problem.initial_state )
    if problem.goal_test( n.state ):
        return n
    # Initialize frontier and explored set
    Frontier = [n] FIFO queue
    explored = set()
    while frontier non vide:
        n = frontier.pop(node with min node.path_cost)
        if problem.goal_test(n.state):
            return solution(n)
        explored.add(n.state)
        for action in problem.actions(n.state):
            child = problem.child_node(n, action)
            aux_cost = n.path_cost + path_cost( n,child )
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            else if child.state in frontier and aux_cost < child.path_cost:
                child.path_cost = aux_cost
    return 'FAIL'
```

Perform graph search to find a solution to the problem
Create the initial node
Check if the initial state is the goal
Return the solution node
A list representing the frontier,
A set to store explored states
Take a node from the frontier
If the node's state satisfies the goal,
 return the solution
Add the node's state to the explored set
Expand the node: iterate over actions
 # Generate the child node
Compute the new path_cost of child
 # Check if the child's state is new
 # Add the child to the frontier
Update the child path cost if the current
 # one is shorter
If the frontier is empty and no solution is found, return "FAIL"



Exemple



Déroulez l'algorithme de Dijkstra en partant d'Arad avec comme état objectif Bucarest. L'algorithme de Dijkstra est-il optimal ? (c'est-à-dire retourne-t-il le chemin le plus court) ?

Recherche non informée



- Algorithmes génériques pour traverser un espace d'états
- Pseudocode GRAPHSEARCH et TREESEARCH
 - Si la frontière est une queue: BFS
 - Si la frontière est une pile: DFS
 - Si la frontière est priorisée avec le cout du chemin: Dijkstra
- Principal inconvénient 时间和空间复杂度是指数级的
 - Complexité exponentielle en temps et en espace

Etymologie : du grec « trouver »

Recherche informée ou heuristique

咨询式搜索 ou 启发式搜索

Une heuristique est une méthode approximative (ou une règle pratique) utilisée pour guider une recherche (ou résoudre un problème) de manière efficace.

L'heuristique souvent ne garantit pas une solution optimale. Comme exemples nous trouvons :

启发式方法通常不能保证最优解

- **Recherche Gloutonne** (non restreinte à la recherche arborescente) : À chaque étape explore le nœud de cout minimal
- **A*** : Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique
- **(plusieurs autres possibles)**

Nous appellerons *heuristique* à la méthode approximative mais aussi à l'information utilisée pour guider la recherche

2 A 搜索 (A) **

• 使用启发式评估函数 $f(n) = g(n) + h(n)$ ，其中：

• $g(n)$ ：从起点到当前节点 n 的实际代价。

• $h(n)$ ：从 n 到目标的启发式估计代价。

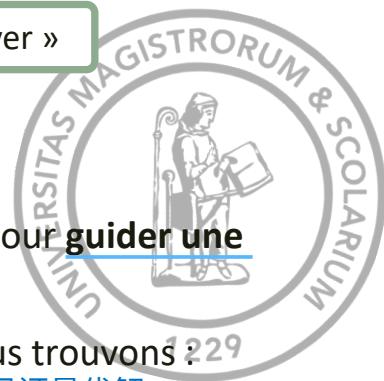
• 按照 $f(n)$ 的值扩展节点，选择估计最优路径。

• 如果 $h(n)$ 是可接受的（不会高估）， A 保证最优解*。

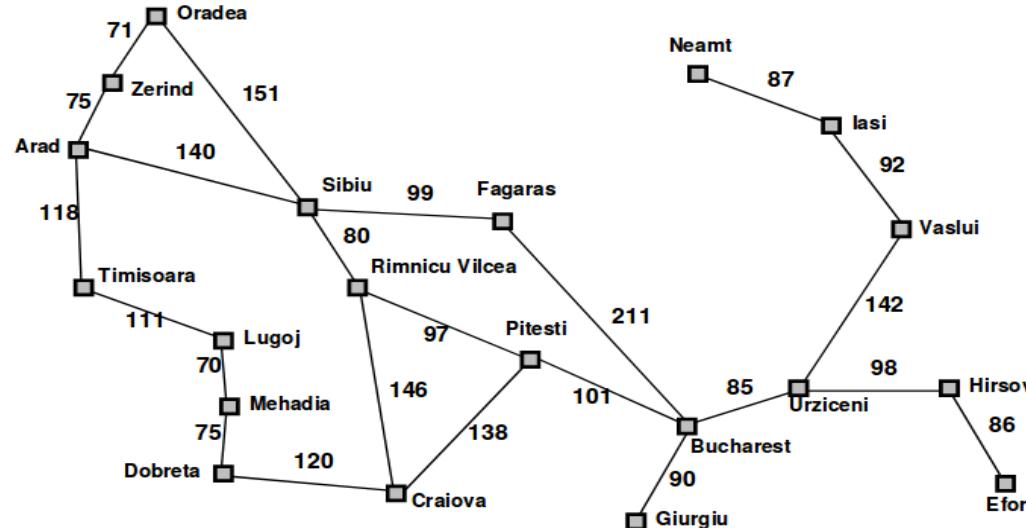
◆ 启发式的定义

我们称 启发式 (Heuristic) 为：

- 一种近似方法（用于提高搜索效率）。
- 用于引导搜索的信息（如 A* 搜索中的 $h(n)$ ）。



Plus court chemin avec heuristique



A 搜索 (A Search) **

- 结合 已行走路径代价 $g(n)$ 和 启发式代价 $h(n)$:
$$f(n) = g(n) + h(n)$$
- 如果 $h(n)$ 是可接受的 (不会高估实际代价) , A 保证最优解! *

Straight-line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

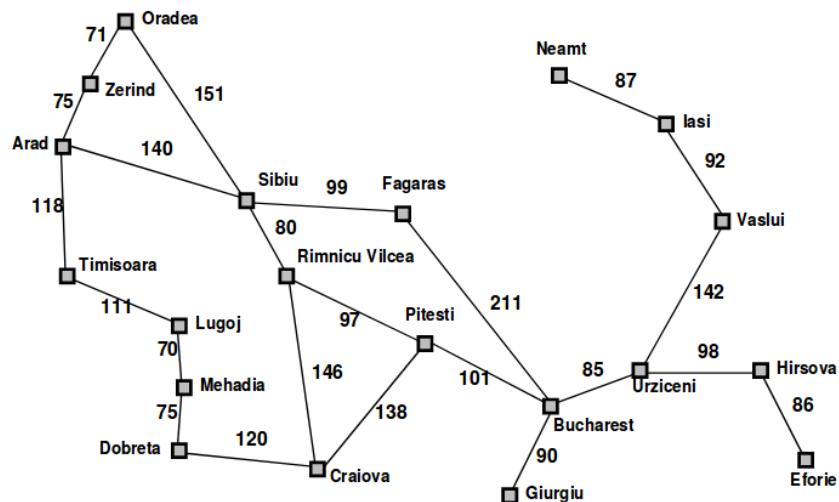
Et si nous utilisions la distance à vol d'oiseau entre deux villes pour guider notre algorithme de recherche ?

Première idée : recherche gloutonne



Au moment de choisir le nœud à explorer, choisir le nœud N qui **minimise l'heuristique $h(N)$**

- Elle n'est **pas optimale**: essayez d'aller de Arad à Bucarest



Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

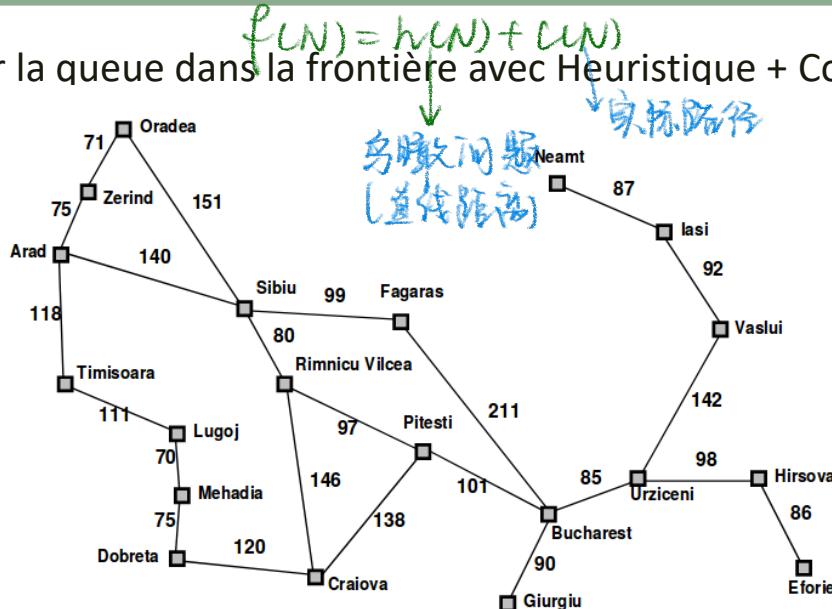
- Pour implémenter cette heuristique ça suffit de modifier DFS

Deuxième idée : A*

若 $h(N)$ 过大 \rightarrow 实际会跳过最优解
 $h(N)$ 过小 < 实际, search 会更慢, 但可以找到最优解
 启发式 $h(N) + c(N)$ 从起始点 N 到路径的总值

Au moment de choisir le nœud à explorer, choisir le nœud N
 qui minimise l'heuristique $h(N)$ du nœud N + le cout
 $c(N)$ du chemin de la racine à N

- Prioritiser la queue dans la frontière avec Heuristique + CoutChemin

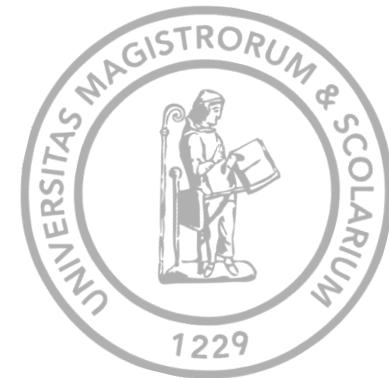


Straight-line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobrete	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- Modifications à faire à Dijkstra, qui minimisait déjà $c(N)$

Pseudocode pour A*



```
def A*(problem):
    n = Node( state = problem.initial_state )
    if problem.goal_test( n.state ):
        return n
    # Initialize frontier and explored set
    Frontier = [n] FIFO queue
    explored = set()
    while frontier non vide:
        n = frontier.pop(node with min node.path_cost_ and_heur)
        if problem.goal_test(n.state):
            return solution(n)
        explored.add(n.state)
        for action in problem.actions(n.state):
            child = problem.child_node(n, action)
            aux_cost = n.path_cost_and_heur + path_cost_and_heur( n,child )
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            else if child.state in frontier and aux_cost < child.path_cost_and_heur:
                child.path_cost_and_heur = aux_cost
    return 'FAIL'
```



Analyse

- **Complétude** : Oui, dans la version GRAPHSEARCH et dans des espaces de recherche finis.
Non, si l'espace est infini ou dans la version TREE SEARCH.

- **Complexité en temps** : exponentielle
- **Complexité en espace** : exponentielle
- **Optimalité** : oui !

- Pour TREE-SEARCH A^* est optimale si la fonction heuristique est admissible, c'est-à-dire, si elle ne surestime jamais le vrai cout

- Dans GRAPH-SEARCH A^* est optimale si la fonction heuristique est consistante (une espèce d'inégalité triangulaire)

La complexité est élevée mais grâce aux heuristiques A^* est utilisé en pratique dans nombreuses applications

完备性

Oui : graphsearch, A^* 可以找到目标状态

Non : Tree search 充足时，~~可能陷入无限循环~~

时间复杂度：

指数级

空间复杂度：

指数级

最优性

若 heuristique 可接受

则说明 $h(N)$ 不能大于实际值

$Dist = N \rightarrow$ Dist im heuristique 是一致的

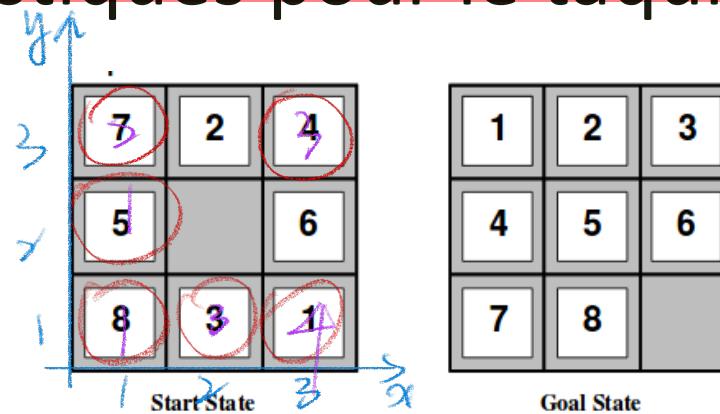
Consistante: $h(N) \leq C(N, N') + h(N')$ \rightarrow Distance

$\hookrightarrow N, N' \rightarrow C(N, N') \leq Cost$

$N \rightarrow$ Att no
heuristique



Deux heuristiques pour le taquin



- Heuristique h_1 = nombre de tuiles mal placées $h_1 = 6$
- Heuristique h_2 = distance de Manhattan avec l'état but (pour chaque tuile, compter combien de cases elle doit croiser pour arriver à sa case dans l'état but)
- Exercice : quelle est h_1 et h_2 de l'état initial dans la figure?

(计算最短距离)

$$|(x_1 - x_2)| + |(y_1 - y_2)|$$



Deux heuristiques pour le taquin

- **h1 et h2 sont admissible**, c.à.d. elles ne surestiment jamais le vrai cout (pourquoi?
Démontrez-le!)
- h2 domine h1, c'est-à-dire $h_2(N) \geq h_1(N)$ pour tout nœud N. Comme elle est admissible, cela veut dire que h2 est une meilleure heuristique ! (pourquoi?)

Comparaison des deux heuristiques avec la recherche non-informée en comptant le **nombre d'états générés** avant de trouver le but:

Profondeur de la solution	Recherche non-informée	A* avec h1	A* avec h2
2	10	6	6
6	680	20	18
12	3644035	227	73
18	--	3056	363
24	--	39135	1641

h1 et h2 sont admissible, c.à.d. elles ne surestiment jamais le vrai cout (pourquoi? Démontrez-le!)

定义：一个启发式是可接受的 (admissible)，当它永远不会高估从 N 到目标状态的真实最短路径代价。

证明 h1 可接受 (admissible)

h1 只是计算了错位瓷砖数，但没有估算交换代价，所以它不会超过真实代价。

每个错位瓷砖至少需要一步交换才能归位，因此 h1 不会高估最优解的步数

证明 h2 可接受 (admissible)

h2 计算的是曼哈顿距离，即瓷砖需要移动的最少步数。

由于一次移动只能将瓷砖向相邻格子平移，因此真实代价一定 $\geq h2$ ，确保 h2 可接受。

2 h2 优于 h1，即 $h2(N) \geq h1(N)$ ，因此 h2 是更好的启发式（为什么？）

- ✓ h1 只是计算错位瓷砖数，而 h2 计算了瓷砖实际的移动步数，所以：

$$h2(N) \geq h1(N) \quad \text{对所有状态 } N \text{ 都成立}$$

- ✓ 因为 h2 更接近实际代价，它减少了不必要的搜索节点，加速 A* 搜索！



Apprentissage d'heuristique

Plutôt que de concevoir la fonction heuristique à la main, elle peut être apprise d'un ensemble plus simple de jeux résolus:

- **Etape 1: résoudre toutes les instances d'un problème**

Exemple: résoudre tous les taquins 4x4 en calculant par force brute la solution optimale

- **Etape 2: décider un ensemble de *features***

Exemple: le nombre de cases pas à leur place, la distance de Manhattan, le nombre de cases pas adjacentes par rapport à l'état final...

- **Etape 3: faire tourner un algorithme d'apprentissage pour apprendre l'heuristique**

Exemple: un arbre de décision qui apprend la distance à l'état but à partir de la description de l'état avec les *features*

- **Etape 4: utiliser la technique d'apprentissage comme heuristique pour résoudre des instances plus grandes du problème**

Exemple: utiliser l'heuristique apprise pour jouer au taquin 12x12

◆ 第一步：求解问题的所有实例

✓ 示例：

- 计算出所有 4×4 拼图 (Taquin) 问题的最优解，使用暴力搜索 (Brute Force) 来找到最优解。
-

◆ 第二步：确定一组特征 (Features)

✓ 示例：

- 错位瓷砖数 (不在目标位置的瓷砖数量)。
- 曼哈顿距离 (所有瓷砖到目标位置的移动步数之和)。
- 不相邻瓷砖数 (与最终状态相比，不在正确相邻位置的瓷砖数)。
- 其他可能的特征...

◆ 第三步：使用机器学习算法训练启发式函数

✓ 示例：

- 使用决策树 (Decision Tree) 来学习当前状态到目标状态的距离，输入为状态的特征。
 - 训练目标：根据给定状态的特征，预测最短步数。
-

◆ 第四步：将学到的启发式用于更大规模问题

✓ 示例：

- 使用学到的启发式函数来解决 12×12 拼图问题 (更大的问题实例)。
- 由于启发式是从小问题中学习的近似最优解，可以提高搜索效率，减少计算量。



Pour aller plus loin : Recherche MonteCarlo (MCTS)

Pour rendre l'heuristique encore plus puissante nous pouvons explorer chaque branche aléatoirement en profondeur au moment de l'expansion d'un nœud :

- Choisir un nœud non-terminal N dans la frontière
- **Explorer aléatoirement** X des branches de profondeur Y à partir du nœud N
- Calculer l'heuristique des nœuds à la fin de ces branches et prendre la **moyenne**
- Répéter pour tous les nœuds dans la frontière et choisir pour l'expansion **le nœud avec la moyenne des heuristiques plus prometteuse**

MCTS est extrêmement populaire et utilisée surtout dans le cadre *adversarial* (ex. jeux vidéos, jeux stratégiques...)

更进一步：蒙特卡洛树搜索（MCTS）

为了使启发式函数更强大，我们可以在扩展节点时**随机探索不同分支**，从而更好地评估状态的潜在价值。

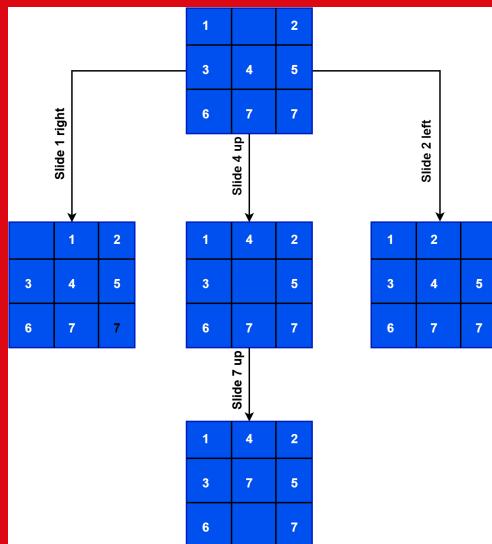
◆ MCTS 主要步骤

1. 选择一个**非终止节点** N 作为当前要扩展的节点。
 2. 从 N 开始，**随机探索 x 个深度为 y 的分支**，模拟可能的未来状态。
 3. 计算这些分支末端节点的**启发式值**，并取其平均值。
 4. 对所有**边界节点**重复此过程，然后选择**启发式均值最优的节点**进行扩展。
-

◆ MCTS 主要应用

- 极其流行的**搜索算法**，特别适用于**对抗性环境**（如围棋、象棋、视频游戏、战略游戏等）。
- AlphaGo 采用了 MCTS 作为核心搜索策略，结合深度学习进行决策。

Recherche informée



■ Trouver ou apprendre une heuristique pour guider la recherche arborescente

- Des améliorations spectaculaires du temps de calcul
- Nécessite connaissance du domaine

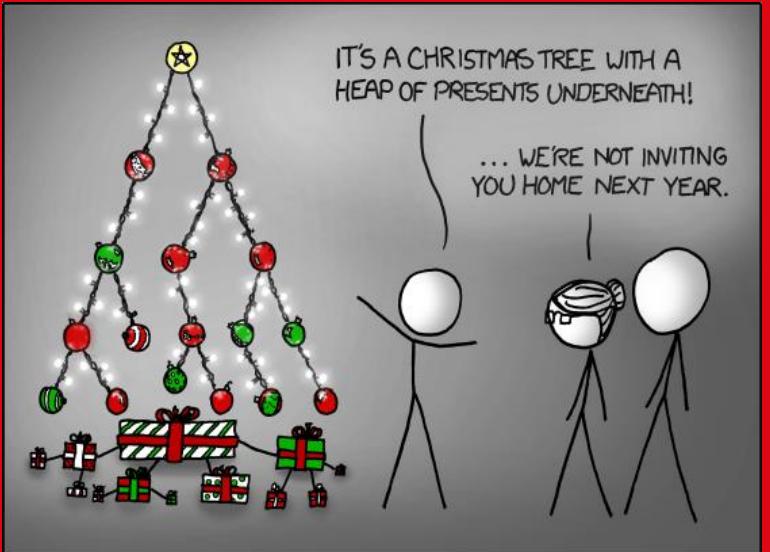
■ Gloutonne ou A*

- Il est nécessaire de minimiser le cout du chemin + l'heuristique pour l'optimalité de l'algorithme de recherche

■ Ce n'est que le début !

- MCTS pour recherche *adversarial*
- Recherche locale (simulated annealing, algorithmes génétiques...)
- ...

Apprentissages collatéraux



https://www.explainxkcd.com/wiki/index.php/835:_Tree

- Abstraction et modélisation
- Structure de données : arbres
- Notation O (big O) *时间复杂度分析* O
- Pseudocode 伪代码 Pseudocode
- Estimation de complexité et optimalité



Matériel du cours

- Intelligence Artificielle, Russel et Norvig
 - Sections de 3.1 à 3.6
 - Section 4.1

Intelligence Artificielle

- Représentation et résolution des connaissances (SAT et CSP)

Felipe Garrido, Umberto Grandi, Jean-Guy Mailly

Master 1 MIAGE



Membre de l'Université
Toulouse Capitole



Espace d'états combinatoire

- Nous avons étudié la modélisation par espaces d'états
 - *Pros* : *Modélisation naturelle*
 - *Cons* : *Le nombre d'état à considérer c'est souvent exponentiel*
- Nous allons maintenant modéliser les états en utilisant des variables 使用变量建模
 - *Effort de modélisation supérieur*
 - *Elle nous permettra d'utiliser des solveurs génériques pour résoudre des problèmes de taille exponentielle en très peu de temps*
 - Par exemple, nous pourrons résoudre des Sudokus 9x9 en 0.1 seconds
- Avec la modélisation en espace de variables, nous ne considérons plus le chemin reliant la racine au nœud but, mais uniquement les valeurs des variables dans l'état but 不再考虑根节点到目标节点的路径 仅关注目标状态下的取值

			1			7		2
3			9	5				
		1			2			3
5	9					3		1
	2						7	
7	3						9	8
8		2			1			
			8	5		6		
6	5			9				



1229

Applications

■ Création d'un emploi du temps



■ Problèmes d'allocation :

- *des cours aux enseignants*
- *des étudiants aux écoles*
- *des ressources à des processeurs*

■ Ordonnancement des tâches entreprises (par exemple production d'avions chez Airbus!)



■ Planification des observations du satellite Rosetta

<http://www.a4cp.org/node/1058>



Application: Rosetta



- La sonde dispose de **peu d'énergie** et d'un **temps très limité** pour **décider quelles expériences** et mesures **effectuer** en fonction des conditions observées sur place
- Les scientifiques souhaitent **réaliser un maximum d'expériences** en un minimum de temps, compte tenu du coût astronomique de la mission
- Un problème supplémentaire se pose : **plusieurs pays ont financé la mission**, mais pas tous avec le même montant
 - **Comment répartir le temps d'expérimentation entre eux de manière équitable ?**



Application: Comment remplir un cargo?



- Le **poids** doit être **réparti de manière équilibrée** dans le cargo
- L'**ordre des visites des ports** détermine le **positionnement des conteneurs** (afin d'éviter de devoir décharger la moitié du chargement pour accéder au bon conteneur)
- Le **temps** de déchargement et de chargement doit être **minimisé**.



Plan du cours

Dans ce cours nous allons apprendre des techniques de représentation d'un problème qui permettent d'utiliser des solveurs performants:

- Solveurs SAT
- Solveurs CSP

■ Espace d'états booléens – SAT

- Variables binaires : 0 ou 1, Vrai ou Faux, Oui ou Non
- Contraintes en logique propositionnelle

■ Espace d'états avec variables – CSP

- Variables arbitraires
- Contraintes sous forme de script ou ensemblistes



Espaces d'états Booléens (SAT)

布尔状态空间

Modéliser un problème avec des formules propositionnelles:

- Un ensemble de **variables binaires**: $x_1, \dots, x_n \in \{0,1\}$ 二值变量：指只能取0 or 1 两个值 / 真 or 假的变量
- Un ensemble de **contraintes**
 - *des formules propositionnelles qui décrivent le problème*
一组描述问题的命题公式

找到valeurs 满足这些命题

Trouver des valeurs aux variables qui satisfont
les formules propositionnelles



Exemple

Une chirurgienne ne peut pas travailler deux jours d'affilé. Exprimer cette condition comme une formule propositionnelle et trouver les possibles solutions.

- **Variables booléennes** : travail-lundi, travail-mardi, travail-mercredi, travail-jeudi, travail-vendredi
- **Contraintes**
 - travail-lundi = 1 alors travail-mardi = 0
 - travail-mardi = 1 alors travail-lundi = 0 et travail-mercredi = 0
 - travail-mercredi = 1 alors travail-mardi = 0 et travail-jeudi = 0
 - travail-jeudi = 1 alors travail-mercredi = 0 et travail-vendredi = 0
 - travail-vendredi = 1 alors travail-jeudi = 0
- Des **états possibles** sont :
 - travail-lundi = 1, travail-mardi = 0, travail-mercredi = 1, travail-jeudi = 0, travail-vendredi = 1
 - travail-lundi = 0, travail-mardi = 1, travail-mercredi = 0, travail-jeudi = 1, travail-vendredi = 0
 - travail-lundi = 0, travail-mardi = 0, travail-mercredi = 1, travail-jeudi = 0, travail-vendredi = 1
 - etc...



Prérequis – les tables de vérité

逻辑上的蕴含 ($P \Rightarrow Q$) 实际上等价于 "非 P 或 Q" ($\neg P \vee Q$) , 即:

$$P \Rightarrow Q \equiv \neg P \vee Q$$

真值表

Valeur Variable x_1	Valeur variable x_2	Not x_1	x_1 and x_2	x_1 or x_2	$x_1 \Rightarrow x_2$
Vrai	Vrai	Faux	Vrai	Vrai	Vrai
<u>Vrai</u>	<u>Faux</u>	Faux	<u>Faux</u>	Vrai	Faux
Faux	Vrai	Vrai	Faux	Vrai	Vrai
Faux	Faux	Vrai	Faux	Faux	Vrai

Notation:

- Négation NOT
- Conjonction AND
- Disjonction OR
- Implication \Rightarrow

Exercice :

Quelle est la table de vérité de la proposition (Not x_1) or x_2 ?

Qu'est-ce que l'on gagne en utilisant des variables au lieu d'états ?

- Pour décrire un nombre **exponentiel d'état** nous avons besoin d'un nombre **linéaire de variables** seulement
 - ***N variables propositionnelles = 2^N affectations possibles (états)***
- Arbre de recherche de largeur N (quelle variable mettre à vraie ou fausse) et profondeur N
- La recherche arborescente (même avec heuristique) n'est pas comparable aux performances des solveurs SAT
 - *Les solveurs SAT sont capables de résoudre des problèmes avec des milliers de formules en peu de temps!*

使用变量代替状态的优势

- 仅需线性数量的变量即可描述指数级数量的状态
- 设有 N 个命题变量，则可能的状态数为 2^N (即 N 个变量的所有可能赋值)。
- 搜索树的结构
 - 搜索树的宽度为 N (即选择哪个变量设为真或假)。
 - 搜索树的深度为 N (所有变量都被赋值的情况)。
- SAT 求解器的强大性能
 - 传统的**树搜索算法** (即使使用启发式搜索) 在大规模问题上效率有限。
 - **SAT 求解器能在极短时间内解决包含成千上万条约束公式的问题。**



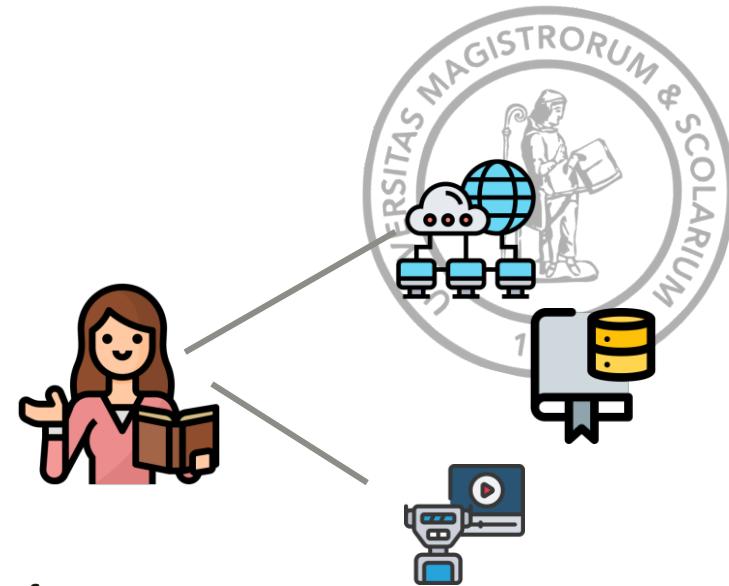
Exemple de modélisation

À l'université nous avons

- Un ensemble de **cours de licence** CL_1, \dots, CL_m
- Un ensemble de **cours de master** CM_1, \dots, CM_k
- Un ensemble de **professeurs** P_1, \dots, P_n

Trouver une allocation de cours aux professeurs tels que :

- Chaque cours doit être enseigné par au moins un professeur
- Chaque professeur doit faire au moins un cours en licence et au moins un cours en master



Solution : Allocation cours-professeurs

- **Variables** 变量
 - Définir une variable binaire pour chaque pair cours-professeur

$$x_{\{c,p\}} = \begin{cases} Vrai, & \text{si } p \text{ enseigne } c \\ Faux, & \text{sinon} \end{cases}$$

- Exemple: $x_{\{IA,GAUDOU\}} = Vrai$ et $x_{\{Bases de données,GAUDOU\}} = Faux$
- **Contraintes** 约束条件
 - Pour chaque COURS définir une formule

每门课至少一名教授

$$\underline{x_{\{P_1,cours\}}} \ or \ \underline{x_{\{P_2,cours\}}} \ or \ ... \ or \ \underline{x_{\{P_n,cours\}}} = Vrai$$

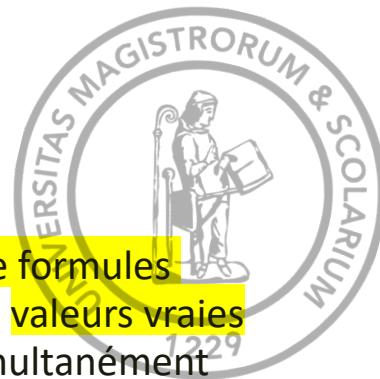
- Pour chaque PROF définir une formule

每个教授至少有一门硕士课 and 一门本科课

$$(x_{\{Prof,CL_1\}} \ or \ ... \ or \ x_{\{Prof,CL_m\}}) \ and \ (x_{\{Prof,CM_1\}} \ or \ ... \ or \ x_{\{Prof,CM_k\}}) = Vrai$$



Algorithmes de résolution



- Résoudre un problème SAT consiste à déterminer si un ensemble donné de formules propositionnelles est **satisfiable**, c'est-à-dire s'il existe une combinaison de valeurs vraies et fausses pour les variables permettant que toutes les formules soient simultanément vraies
- Si ces formules sont satisfiables, peut-on également trouver l'affectation des valeurs de vérité aux variables qui satisfait toutes les formules ?
- Il s'agit d'un problème d'exploration dans un espace d'états de taille exponentielle
- L'approche repose sur la possibilité de **raisonner directement sur les variables**
- DPLL (Davis-Putnam-Logemann-Loveland) est l'un des algorithmes les plus célèbres pour résoudre ce type de problème. Il intègre notamment une forme de **backtracking** (Voir la fin du cours !)

难度分析

- SAT 求解属于 NP-完全问题，意味着其状态空间的规模是指数级的。
- 需要在一个庞大的状态空间中进行搜索，以找到可能的解。

求解方法

- 直接在变量上进行推理，而不是枚举所有可能的状态。
- DPLL 算法 (Davis-Putnam-Logemann-Loveland) 是最著名的 SAT 求解算法之一。
 - 核心思想：使用**回溯搜索（Backtracking）**来高效探索可能的变量赋值。
 - 通过单位传播（Unit Propagation）和纯文字消除（Pure Literal Elimination）加速求解。
 - DPLL 是现代 SAT 求解器的基础。

Le monde en binaire

Plusieurs problèmes peuvent être facilement représentés à l'aide de formules booléenne et les solveurs de logique propositionnelles sont très performant pour trouver une solution quand il en existe une

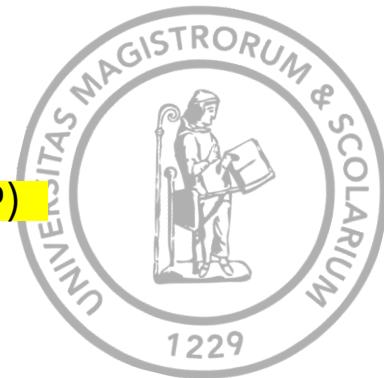
■ Représenter pour résoudre

- Si le chemin pour arriver à une solution n'est pas important
- Si on peut représenter les états avec des variables (booléennes ou pas)²²⁹

■ Logique propositionnelle

- Variables et connecteurs logiques (ET, OU, NOT, IMPLIQUE)
- Tables de vérité pour décider la valeur de vérité d'une formule
- Algorithmes de résolution pour trouver une affectation aux variables qui rend toutes les formules du problème vraie





Espace d'états avec variables (CSP)

基于变量的状态空间 (CSP)

On modélisera un problème avec les ingrédients suivants:

- Un ensemble de **variables quelconques**: x_1, \dots, x_n 变量集合

每个变量可以表示一个决策变量 : 比如 x_i , 可以表示为第*i*门课上课的时间

- Chaque variable a un domaine associé D_1, \dots, D_n (pas nécessairement binaire!)
- Un ensemble de **contraintes** ou formules, qui décrivent le problème $C_i = \langle \text{variables, valeurs} \rangle$
 $C = \{C_1, C_2, \dots, C_m\}$ 约束集合

Objectif

Trouver une allocation de domaines aux variables

qui satisfait les contraintes

找到变量的一个取值分配, 使得所有约束都被满足
所有变量的取值都必须符合所有约束

CSP : Constraint satisfaction problem

- 约束 C_i 限制变量之间的合法关系, 例如:
 - **时间表问题**: 不能在同一时间安排两门冲突的课程。
 - **分配问题**: 每个任务必须分配给不同的工人。

每个约束 C_i 由变量和它们的有效值组成:

$$C_i = \langle \text{variables, valeurs} \rangle$$

Coloration d'une carte

Problème : Attribuer une couleur (Rouge, Vert ou Bleu) aux états australiens de manière à ce que chaque paire d'états voisins ait des couleurs différentes.



Modélisation comme problème de satisfaction de contraintes :

- Variables = {WA, NT, Q, SA, NSW, V, T}
- Domaine pour chaque variable = {Rouge, Vert, Bleu}
- Contraintes: { WA≠SA, WA≠NT, NT≠SA, Q≠SA, Q≠NSW, NSW≠V, NSW≠SA, SA≠V, V≠SA }

Solution d'un CSP

部分赋值与完全赋值

- Une **affectation partielle** correspond au choix d'une valeur du domaine pour certaines variables.
- Une **affectation totale** correspond au choix d'une valeur du domaine pour toutes les variables.
- Une **solution** d'un problème **CSP** est une affectation totale qui **satisfait toutes les contraintes**.



Exemple : coloration d'une carte

Affectation partielle : WA=rouge, NT=vert

Affectation totale: WA=rouge, NT=rouge, Q=rouge, NSW=rouge, V=rouge, SA=rouge, T=rouge

Solution: WA=rouge, NT=vert, Q=rouge, NSW=vert, V=rouge, SA=bleu, T=vert

不考虑约束时，有：

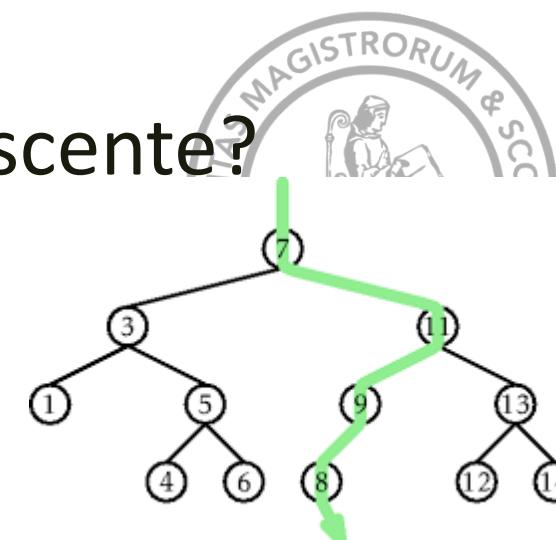
$3^7 = 2187$ (所有可能的赋值组合)

考虑约束（相邻州不能同色）后，状态数减少，但仍然很大。

Pourquoi pas la recherche arborescente?

- Combien de colorations possibles de la carte d'Australie?
- Facteur de branchage?
- Profondeur de l'arbre?
- Avec des variables je peux éliminer facilement des états:
Si SA = Bleu alors je peux inférer que tous les états voisins ne peuvent pas être Bleu, passant de $3^5=243$ états à $2^5=32$
- Si nous identifions des affectations partielles qui ne peuvent pas être complétées pour satisfaire les contraintes, nous pouvons éliminer tous les états qui les complètent
- Dans les deux cas les actions d'**inférence** ont **un gain exponentiel** dans le nombre d'états à considérer !

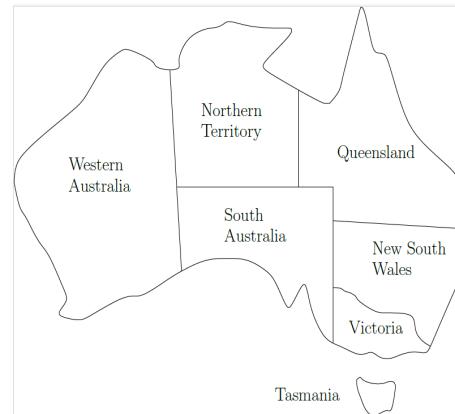
逐层枚举所有可能的赋值，计算量将呈指数级增长





Pourquoi pas les formules booléennes?

- Tenter d'écrire le problème de coloration d'une carte avec des formules propositionnelles (SAT)
- C'est long mais possible!
- Les solveurs SAT ont souvent des meilleures performances que les solveurs CSP
- SAT nécessite plus de travail de modélisation, mais peut amener à des meilleures performances computationnelles



2. 为什么不用 SAT?

SAT 是可行的，但建模复杂

- 转换为布尔公式的过程较繁琐（每个州 3 个变量，总共有 $3 \times 7 = 21$ 个变量）。
- 约束需要转化为 CNF 公式（合取范式），这可能导致大量公式。

SAT 求解器通常比 CSP 求解器快

- SAT 求解器（如 MiniSat、Z3、Glucose）的优化更成熟，处理大规模问题时通常比 CSP 更快。
- 基于冲突驱动的学习 (CDCL) 让 SAT 求解器可以处理成千上万个约束。

但 SAT 需要更多建模工作

- CSP 更加直观，不需要转换为布尔变量和 CNF 公式。
- SAT 需要额外的转换步骤，而 CSP 可以直接用变量和数值求解。

Optimisation

优化

Les solutions peuvent être associées à

- un cout
- une utilité

最小化成本或最大化效益

Nous chercherons des solutions à coût minimale et à utilité maximale

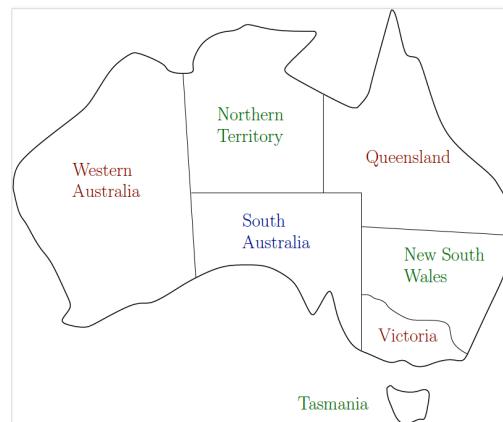


Exemple : coloration d'une carte avec couts

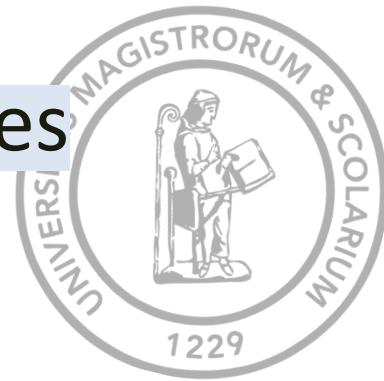
Variables : {WA, NT, Q, SA, NSW, V, T}, Couts : {rouge 2, vert 9, bleu 3}



Cout 30



Cout 36

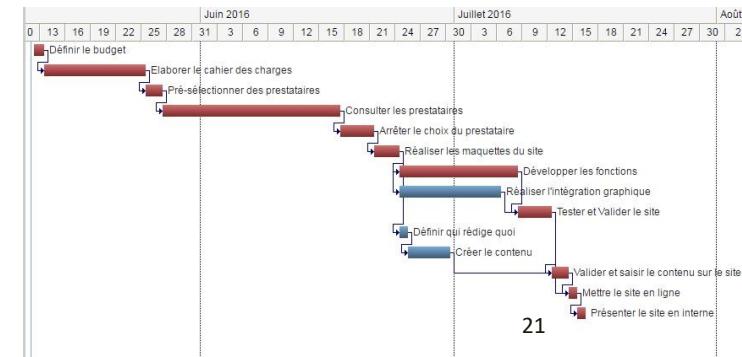
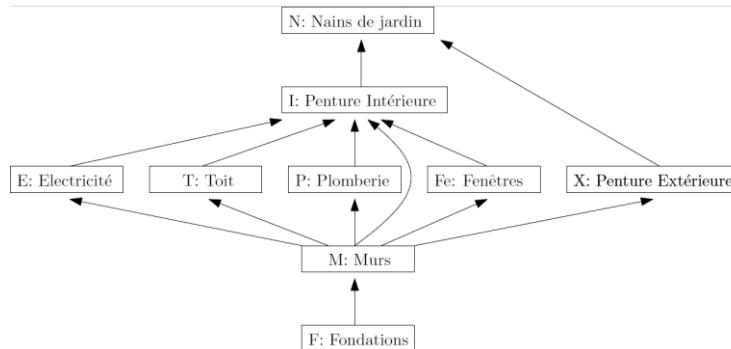


Exemple : ordonnancement de tâches

任务调度

Comment trouver un plan d'exécution pour une activité complexe ?

- Un certain nombre de projets ou tâches doivent être exécutées
- Chaque tâche a une durée
- Les tâches ont des relations de précédence entre elles
- Les tâches ne peuvent pas être interrompues
- La ressource est le temps, qui 1. 为每个任务定义一个变量 est limité et discréte (par exemple en minutes)
- **Objectif :** trouver un ordonnancement de tâches qui minimise le temps d'exécution
找到一个任务调度方案，使总执行时间最短 (Minimiser le temps d' exécution)



Ordonnancement: solution



1. Définir une variable par tâche 1. 为每个任务定义一个变量

Exemple : pour finaliser une voiture nous devons installer les quatre roues, installer les deux essieux, et vérifier le travail. Il y a donc 7 variables : *ROUE1g pour roue avant gauche, ..., ESS1, ESS2, VERIF.*

2. Chaque variable prend le temps à disposition comme domaine 2. 变量的取值范围

Exemple : le temps à disposition est de 30 minutes, les variables prennent valeurs en $\{1,2,3\dots30\}$. Si $ESS1=13$ on commencera l'installation de l'essieu 1 au minute 13.

3. Écrire des contraintes pour la durée de chaque variable et leurs relations de précédence

3. 定义任务持续时间和优先关系约束

Exemple : avant d'installer les roues il faut installer l'essieu, ce qui prends 7 minutes. Cela se traduit en deux contraintes : $R1g \geq ESS1 + 7$ et $R1d \geq ESS1 + 7$ ESS 是指开始的时间，7是需要花费的时间

Exemple : la vérification doit être faite à la fin se traduit en $VERIF \geq TACHE + \text{durée de la tache}$

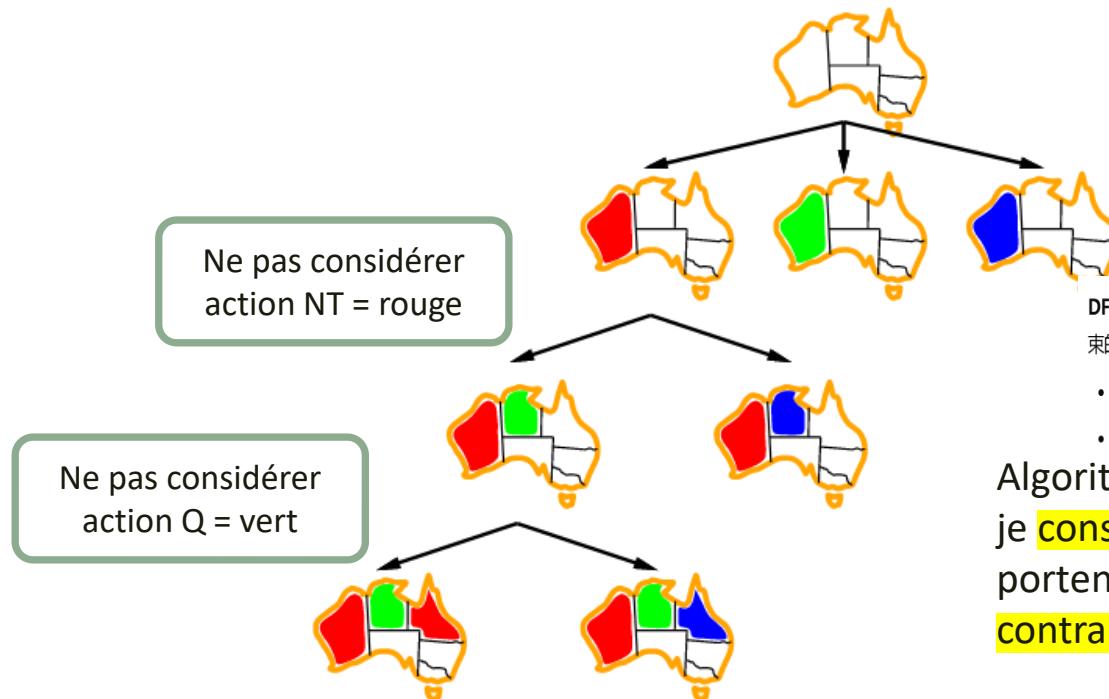
4. On minimise le début de la dernière tâche + sa durée

4. 目标函数：最小化最后一个任务的完成时间

Exemple : si la vérification prend 2 minutes, nous minimisons la valeur de $VERIF + 2$



Recherche DFS avec backtracking

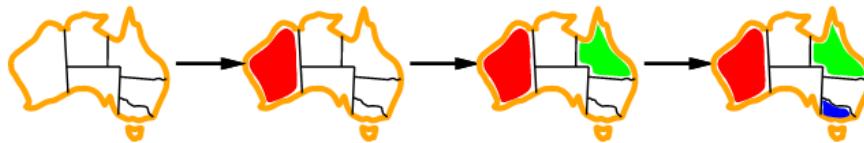


DFS (深度优先搜索) + 回溯 是一种递归搜索方法，用于在解空间树中寻找满足约束的解：

- 每次扩展 (expansion) 时，只考虑满足约束的颜色选择。
- 不考虑导致冲突的颜色选择，从而减少搜索空间，提高效率。

Algorithme DFS mais à chaque expansion je **considère seulement** les actions qui portent à une **affectation qui satisfait les contraintes!**

Inférence (forward checking) 前向检查



前向检查 (Forward Checking) 是一种 约束传播 (Constraint Propagation) 技术，用于 约束满足问题 (CSP) 中，以减少搜索空间，提高求解效率。在 深度优先搜索 (DFS) + 回溯 (Backtracking) 过程中，前向检查帮助提前排除无解路径，避免无效搜索。

WA	NT	Q	NSW	V	SA	T
red green blue						
red		green blue	red green blue	red green blue	red green blue	green blue
red			red blue	red green blue	red blue	red green blue
red				red		red green blue

- Pour **chaque valeur** du domaine d'une variable (par exemple, NT = rouge), nous vérifions si l'**affectation partielle** résultante peut être **complétée** pour former une **affectation totale**
- Si une valeur rend impossible la complétiion de l'affectation, **nous retirons cette valeur du domaine de la variable** (par exemple, retirer rouge du domaine de NT)
- Ce raisonnement (inférence) est répété à chaque étape d'une recherche en profondeur (DFS) avec retour arrière (backtracking).

BFS avec inférence et backtracking



```
def recherche_aller_retour(AFF, DOMAINES):
    if len(AFF) == len(DOMAINES):          # Condition de terminaison : toutes les variables sont affectées
        return AFF
    elif DOMAINES contient au moins une liste vide :
        return "FAIL"
    X = choisir_variable(DOMAINES, AFF)      # Choisir une variable X à affecter
    domaine_X = DOMAINES[X]
    for x in domaine_X:
        if respecter_contraintes(X, x, AFF):
            AFF[X] = x                      # Ajouter X = x à l'affectation
            domaines_reduits = reduire_domaines(X, x, DOMAINES, AFF)      # Propagation des contraintes
            résultat = recherche_aller_retour(AFF, domaines_reduits)          # Appel récursif
            if résultat != "FAIL":
                return résultat

            del AFF[X]                      # Retour arrière : retirer X = x et réinitialiser les domaines
            DOMAINES = restaurer_domaines(domaines_reduits, DOMAINES)
    return "FAIL"
```

Recherche locale MINCONFLICT

- Plutôt que créer une affectation, commencer avec une affectation totale et la corriger
 - Heuristique **MINCONFLICT** : minimise le nombre de variables qui participent à des contraintes qui ne sont pas satisfaits
- ```
def min_conflicts(csp, max_steps, affectation_initiale): #max_steps: Nombre maximum d'itérations
 courant = affectation_initiale # Initialisation : affectation complète
 for _ in range(max_steps): # Vérifier si l'affectation courante est une solution
 if est_solution(courant, csp['CONTRAINTES']): # Choisir une variable impliquée dans une contrainte violée
 return courant
 var = choisir_variable(courant, csp)
 # Choisir une nouvelle valeur pour cette variable qui minimise les conflits
 valeurs_possibles = csp['DOMAINES'][var]
 valeur = min(valeurs_possibles, key=lambda v:
 nombre_de_conflits(var, v, courant, csp['CONTRAINTES']))
 courant[var] = valeur
 return "FAIL"
```



# Exemple : Sudoku

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   |   |   | 1 |   |   | 7 |   | 2 |
|   | 3 |   | 9 | 5 |   |   |   |   |
|   |   | 1 |   |   | 2 |   |   | 3 |
| 5 | 9 |   |   |   |   | 3 |   | 1 |
|   | 2 |   |   |   |   |   | 7 |   |
| 7 | 3 |   |   |   |   | 9 | 8 |   |
| 8 |   |   | 2 |   |   | 1 |   |   |
|   |   |   |   | 8 | 5 |   | 6 |   |
| 6 | 5 |   |   | 9 |   |   |   |   |

- 81 variables, une pour chaque case:  $X_{ij}$  pour la case en ligne i et colonne j
- Domaine  $\{1,2,3,4,5,6,7,8,9\}$  pour toute variable
- Contraints **ALLDIFF** (qui demandent aux variables dans l'argument d'être toutes différentes)
- Pour les lignes
  - $\text{ALLDIFF}(X_{11}, X_{12}, \dots, X_{19})$
  - ...
- Pour les colonnes
  - $\text{ALLDIFF}(X_{11}, X_{21}, \dots, X_{91})$
  - ...
- Pour les carrés
  - $\text{ALLDIFF}(X_{11}, X_{12}, X_{13}, X_{21}, X_{22}, X_{23}, X_{31}, X_{32}, X_{33})$
  - ...

搜索优化 (Recherche arborescente avec amélioration)

为提高求解效率，我们使用 搜索树优化 技术：

回溯搜索 (Backtracking)

- 如果当前值不可行，则回溯（撤销上一步的值）。
- 避免继续搜索无解分支，提高效率。

前向检查 (Forward Checking)

- 在回溯时，提前消除不可能的候选值，减少未来的搜索空间。
- 例如，假设中如果  $X_{11} = 3$ ，那么该行、列和宫格中的其他格子不能再取 3。

# Conclusion

Modéliser peut être une étape qui permet de résoudre un problème avec des solveurs génériques:

- Solveurs SAT
- Solveurs CSP

## 搜索优化 (Recherche arborescente avec amélioration)

为提高求解效率，我们使用 搜索树优化 技术：

### 回溯搜索 (Backtracking)

- 如果当前赋值导致矛盾，则回溯（撤销上一步的赋值）。
- 避免继续搜索无解分支，提高效率。

### 前向检查 (Forward Checking)

- 在赋值时，提前消除不可能的取值，减少未来的搜索空间。
- 例如，数独中如果  $X_{11} = 3$ ，那么该行、列和宫格中的其他格子不能再取 3。

## ■ Etape de modélisation

- Définir des variables et des domaines
- Définir des contraintes (propositionnel ou non selon les domaines des variables)

## ■ Recherche arborescente avec amélioration pour un gain exponentiel

- Backtracking pour revenir en arrière quand on considère une affectation inconsistante
- Forward checking pour regarder « dans le futur » et éliminer des valeurs qui ne peuvent pas être complétées

## ■ Des solveurs performants (rdv en TD)

