

Intelligence Artificielle

Recherche Arborescente

Jean-Guy Mailly (FI), Leila Moudjari (FA)

Master 1 MIAGE – 2025-2026

TD: Audren Boulic-Bouadjio, Leila Moudjari, Paul Saves

人工智能

树形检索

Jean-Guy Mailly (法国), Leila Moudjari (阿尔及利亚)

MIAGE硕士1年级 - 2025-2026学年

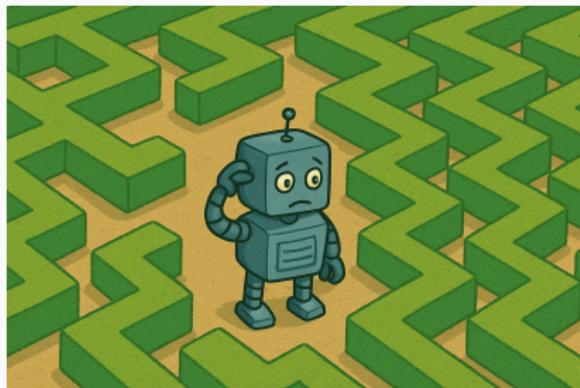
TD: 奥德伦·布利克-布阿吉奥、莱拉·穆贾里、保罗·塞夫斯

Introduction

导言

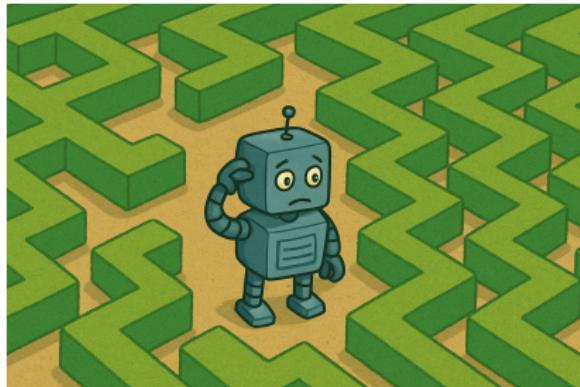
Exemple introductif : le labyrinthe

- Imaginez que vous devez programmer un robot qui doit chercher la sortie d'un labyrinthe
- Le robot a 4 actions à sa disposition : aller en haut, en bas, à gauche ou à droite (sauf s'il y a un mur...)
- Quel genre d'algorithme peut-on appliquer ?

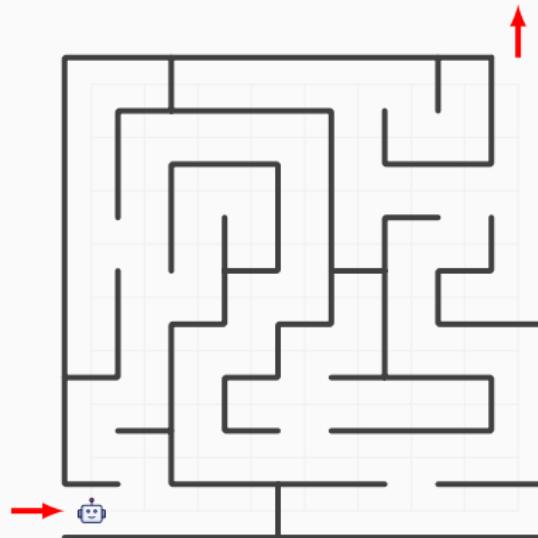


示例：迷宫

- 假设你需要编写一个机器人程序，让它在迷宫中寻找出口。
- 该机器人具备四个可选动作：向上移动、向下移动、向左移动或向右移动（若前方无墙壁则可自由移动）。
- 可以应用哪种类型的算法？

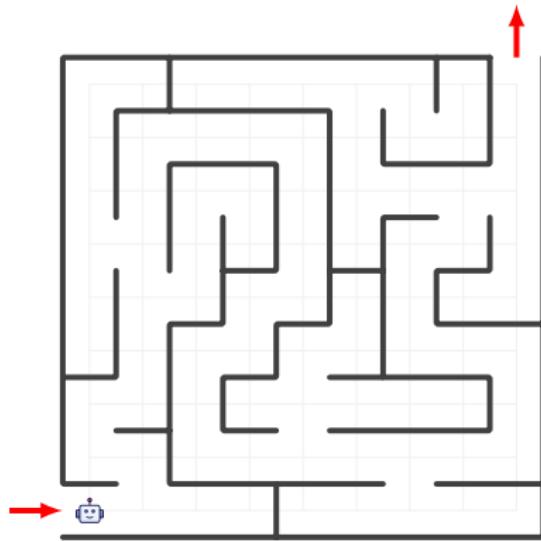


Exemple introductif : le labyrinthe



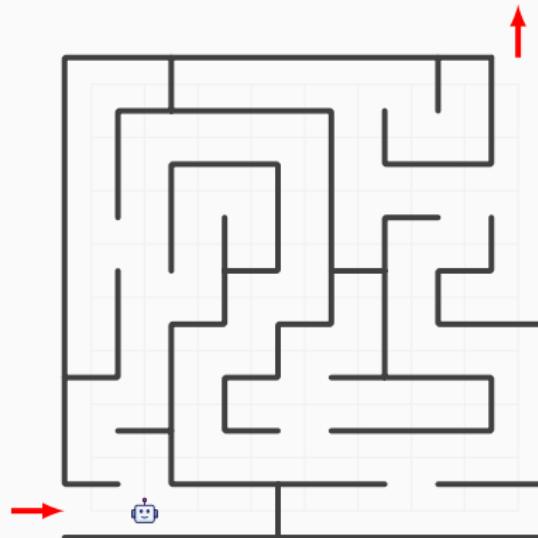
- Le robot peut commencer à avancer vers la droite

示例：迷宫



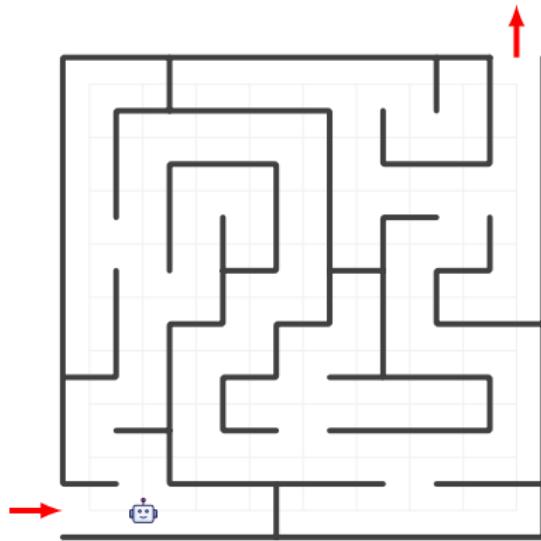
- 机器人可以开始向右移动

Exemple introductif : le labyrinthe



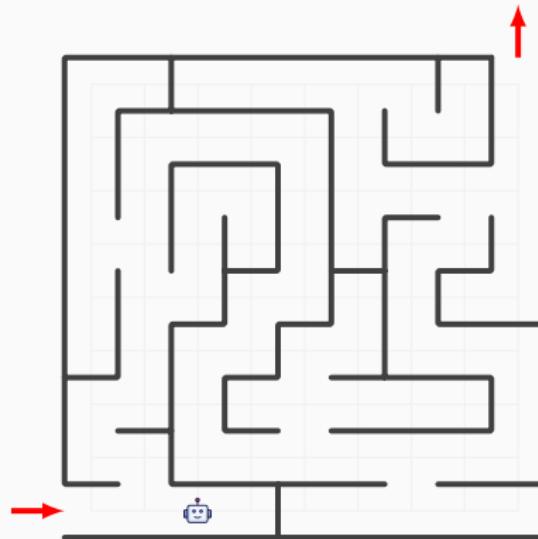
- Le robot peut commencer à avancer vers la droite

示例：迷宫



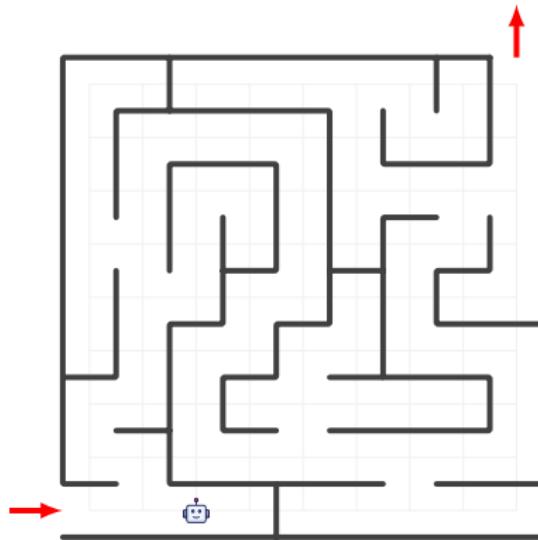
- 机器人可以开始向右移动

Exemple introductif : le labyrinthe



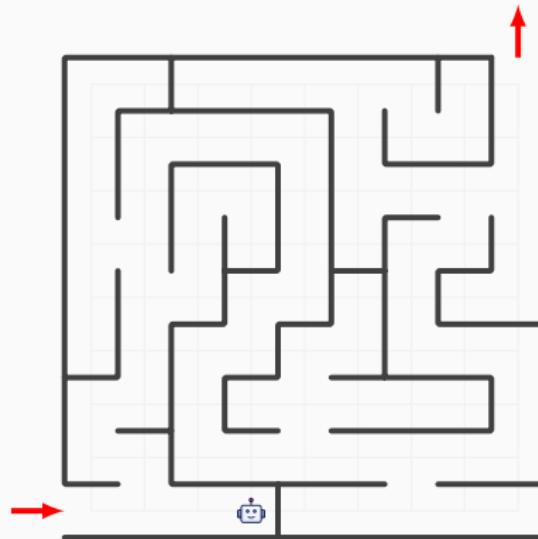
- Le robot peut commencer à avancer vers la droite

示例：迷宫



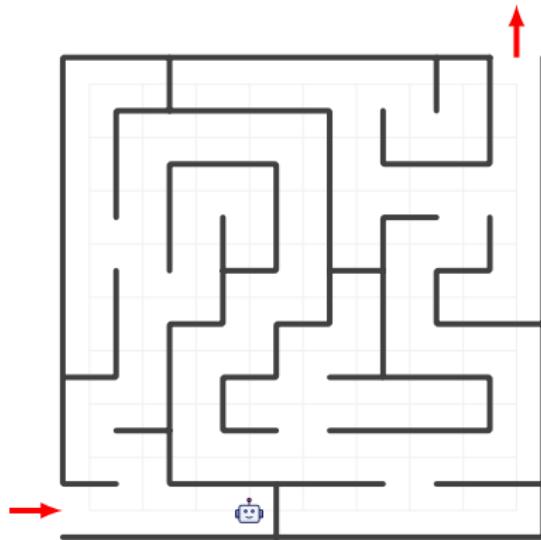
- 机器人可以开始向右移动

Exemple introductif : le labyrinthe



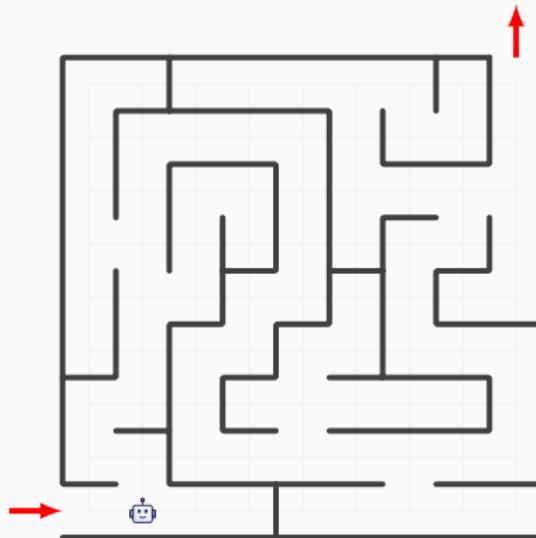
- Le robot peut commencer à avancer vers la droite
- Au bout de quelques pas, il est bloqué

示例：迷宫



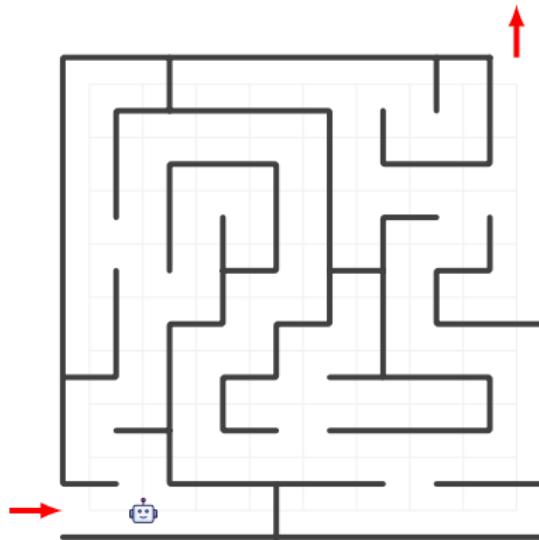
- 机器人可以开始向右移动
 - 没走几步，她就被堵住了。

Exemple introductif : le labyrinthe



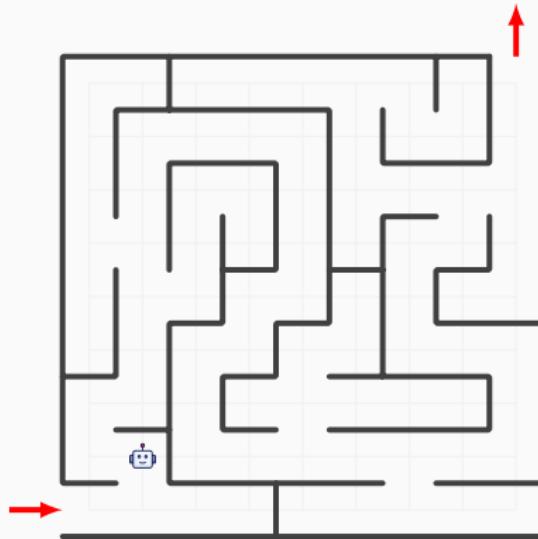
- Le robot peut commencer à avancer vers la droite
- Au bout de quelques pas, il est bloqué
- Il faut donc rebrousser chemin pour chercher une autre option

示例：迷宫



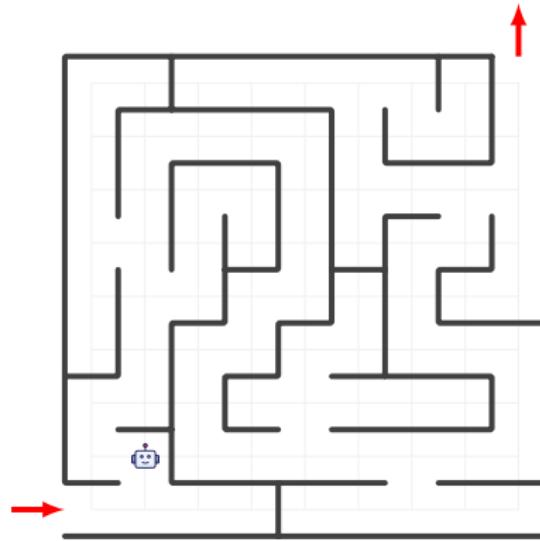
- 机器人可以开始向右移动
- 没走几步，她就被堵住了。
- 因此需要重新调整方向，寻找其他可行方案

Exemple introductif : le labyrinthe



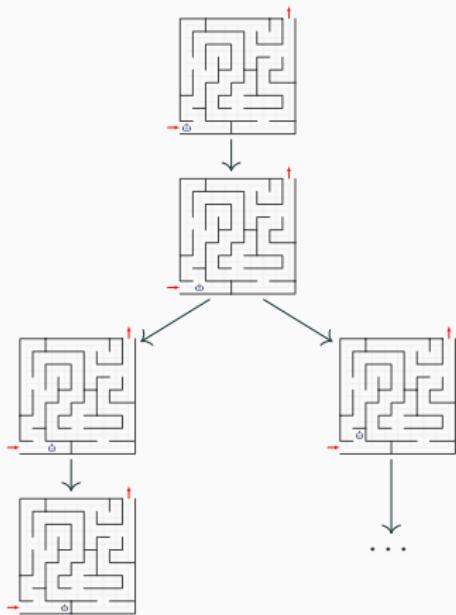
- Le robot peut commencer à avancer vers la droite
- Au bout de quelques pas, il est bloqué
- Il faut donc rebrousser chemin pour chercher une autre option
- Comme on connaît la structure du labyrinthe, on peut calculer toutes ces options avant que le robot ne se déplace

示例：迷宫



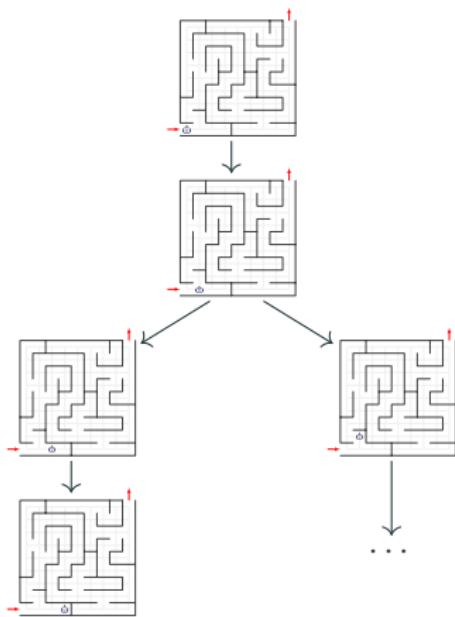
- 机器人可以开始向右移动
- 没走几步，她就被堵住了。
- 因此需要重新调整方向，寻找其他可行方案
- 由于已知迷宫的结构，可以在机器人移动前计算出所有这些选项。

Exemple introductif : le labyrinthe, et l'arbre



- Si on développe tout l'arbre, on peut facilement identifier la séquence d'actions qui mène à la sortie du labyrinthe
- Il existe différentes stratégies pour développer l'arbre de manière « intelligente »
 - Pour essayer de trouver la solution en développant le moins d'états possibles
 - Pour essayer de trouver les « meilleures » solutions (par exemple, avec moins d'actions)

示例：迷宫与树



- 只要把整个树结构展开，就能轻松找出将我带出迷宫的行动序列。
- 存在多种策略来以 智能 的方式开发树
 - 通过尽可能减少状态数量来尝试寻找解决方案
 - 为了尝试找到 最佳 解决方案（例如，减少操作次数）

Une approche générique

- Les algorithmes que nous allons voir sont génériques : au lieu d'un robot qui se déplace (droite, gauche, haut, bas) dans un labyrinthe, on pourrait résoudre
 - un taquin
 - un Rubik's cube
 - une recherche d'itinéraire entre deux villes
 - la planification d'actions d'un agent autonome (robot ou logiciel)

On a besoin

- d'une modélisation d'un état (du monde, du jeu, etc)
- d'une liste d'actions disponibles
- de la capacité d'évaluer les effets d'une action sur le monde

一种通用方法

- 我们即将探讨的算法具有通用性：与其让机器人在迷宫中进行直线移动（左右、上下），不如采用其他解决方案。
 - 塔奎因
 - 一个魔方
 - 两城之间的路线搜索
 - 自主代理（机器人或软件）的行动规划

需要

- 对某种状态（如世界、游戏等）的建模
- 从可用操作列表中
- 评估某一行动对世界影响的能力

Une approche générique

- Les algorithmes que nous allons voir sont génériques : au lieu d'un robot qui se déplace (droite, gauche, haut, bas) dans un labyrinthe, on pourrait résoudre
 - un taquin
 - un Rubik's cube
 - une recherche d'itinéraire entre deux villes
 - la planification d'actions d'un agent autonome (robot ou logiciel)

On a besoin

- d'une modélisation d'un état (du monde, du jeu, etc)
- d'une liste d'actions disponibles
- de la capacité d'évaluer les effets d'une action sur le monde

Remarque

- On ne verra pas cela en détail ici, mais des méthodes similaires sont utilisées si on n'est pas sûr des effets des actions (méthodes probabilistes) ou pour des situations avec plusieurs agents (par exemple pour les jeux à 2 joueurs)

一种通用方法

- 我们即将探讨的算法具有通用性：与其让机器人在迷宫中进行直线移动（左右、上下），不如采用其他解决方案。
 - 塔奎因
 - 一个魔方
 - 两城之间的路线搜索
 - 自主代理（机器人或软件）的行动规划

需要

- 对某种状态（如世界、游戏等）的建模
- 从可用操作列表中
- 关于评估某一行动对世界影响的能力（注）
- 虽然这里不会详细展示具体操作，但类似方法常用于以下场景：当不确定行动效果时（概率方法），或是涉及多个参与者的情况（例如双人游戏）。

Principe de base

基本原理

On modélise un problème avec

- Un ensemble d'états, dont un état initial
- Pour chaque état, un ensemble d'actions disponibles
- Une fonction de transition : associe à chaque couple (état, action) le nouvel état résultat de l'action
- Un test de réussite : identifie les états buts

Objectif : une séquence d'action qui mène à un état but à partir de l'état initial

我们通过一个模型来模拟这个问题。

- 一组状态，其中包含一个初始状态
- 每个状态都有一组可用的操作
- 一个转换函数：将每个状态-动作对 a 与动作产生的新状态 r 相关联
- 一项成功的测试：识别目标状态

目标：从初始状态到目标状态的一系列动作序列

On modélise un problème avec

- Un ensemble d'états, dont un état initial
- Pour chaque état, un ensemble d'actions disponibles
- Une fonction de transition : associe à chaque couple (état, action) le nouvel état résultat de l'action
- Un test de réussite : identifie les états buts

Objectif : une séquence d'action qui mène à un état but à partir de l'état initial

- Dans certains cas, une fonction de coût associe une valeur numérique à chaque couple (état, action)

Objectif : une séquence d'actions **de coût minimal** qui mène à un état but à partir de l'état initial

我们通过一个模型来模拟这个问题。

- 一组状态，其中包含一个初始状态
- 每个状态都有一组可用的操作
- 一个转换函数：将每个状态-动作对 a 与动作产生的新状态 r 相关联
- 一项成功的测试：识别目标状态

目标：从初始状态到目标状态的一系列动作序列

- 在某些情况下，成本函数会为每个（状态，动作）组合关联一个数值。

目标：从初始状态出发，通过一系列**成本最低**的动作到达目标状态

Exemple : Le Taquin

Modélisation

- États : toutes les grilles de 3×3 avec $\{1, \dots, 8\}$ (et une case vide)
- État initial : voir figure
- Actions : déplacer la case vide vers le haut, le bas, la droite, la gauche (si possible)
- Test de réussite : un seul état but, les cases sont rangées dans l'ordre croissant
- Coût : 1 pour chaque déplacement

7	2	4
5		6
8	3	1

État initial

1	2	3
4	5	6
7	8	

État but

示例：塔昆

模型化

- 状态：所有 3×3 的网格，包含 $\{1, \dots, 8\}$ （以及一个空格）
- 初始状态：参见图示
- 操作：将空白方框向上、向下、向右或向左移动（如可行）
- 成功测试：仅需一个目标状态，方框按升序排列
- 成本：每次移动1个单位

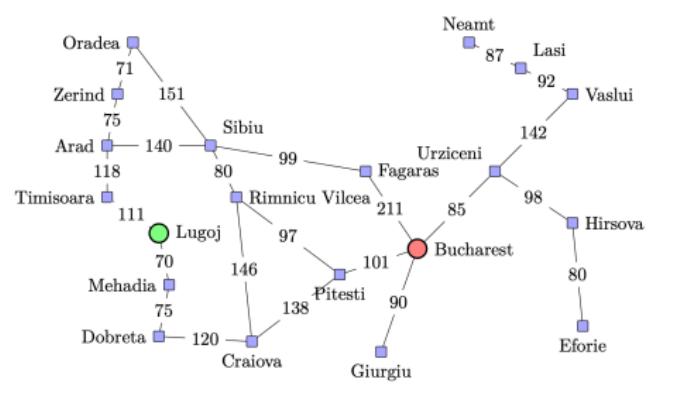
7	2	4
5		6
8	3	1

初始状态

1	2	3
4	5	6
7	8	

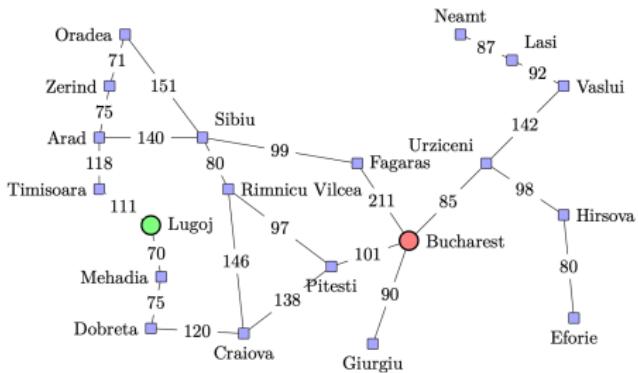
“国家”

Exemple : Voyage en Roumanie



- États : la personne est dans une ville
- État initial : la personne est à Lugoj
- Actions : la personne se déplace vers une ville voisine
- Test de réussite : un seul état but, la personne est à Bucharest
- Coût : la distance entre les villes

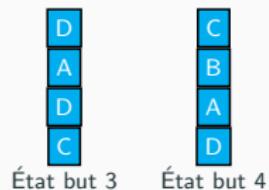
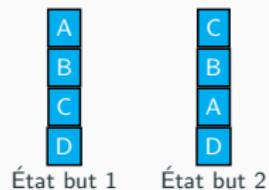
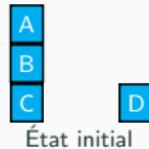
示例：罗马尼亚之旅



- “状态：此人位于某城市”
- 初始状态：该人位于卢戈伊
- 行动：此人前往邻近城市
- 成功测试：仅需一个状态，此人已抵达布加勒斯特
- 成本：城市之间的距离

Exemple : le monde des blocs

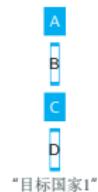
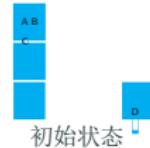
- États : toutes les façons possibles d'empiler les blocs
- État initial : voir figure
- Actions : le robot déplace un bloc libre sur un autre bloc libre, ou sur le sol
- Test de réussite : les blocs forment une seule pile
- Coût : le nombre d'actions



...

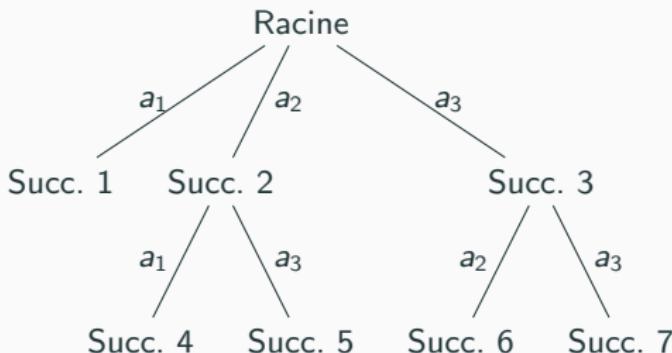
示例：方块世界

- “状态：所有可能的堆叠方式”
- 初始状态：参见图示
- 动作：机器人将自由块放置在另一个自由块上，或放置在地面上。
- 成功测试：模块构成单一堆栈
- 成本：股票数量

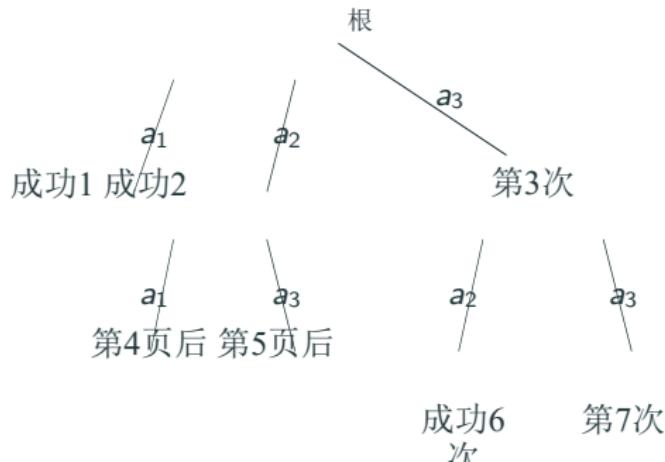


Arbre de recherche

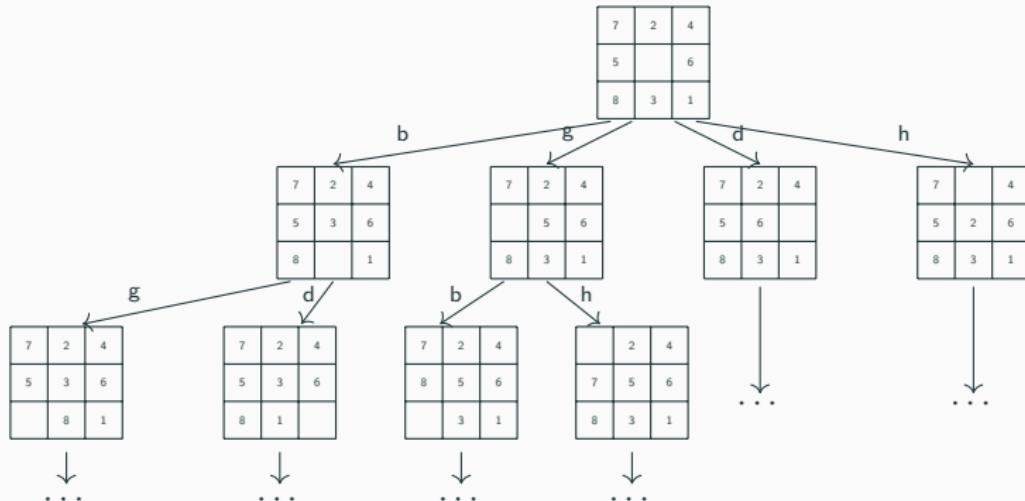
- En pratique, on ne représente pas dans un programme tous les états possibles (nombre potentiellement très grand)
- On développe un arbre de recherche :
 - Noeuds : les états
 - Racine : état initial
 - En partant de la racine, on explore les successeurs d'un noeud en appliquant toutes les actions possibles
 - Feuilles : états sans successeurs (plus d'actions disponibles)
 - **Attention** : on doit contrôler qu'un état n'a pas déjà été exploré pour éviter les répétitions



- 实际上，我们不会在程序中呈现所有可能的状态（其数量可能非常庞大）
- 我们正在构建一个搜索树：
 - 结：状态
 - 根：初始状态
 - 从根节点出发，通过应用所有可能的操作来探索节点的后继节点
 - 叶片：无继承者状态（无可用行动）
 - 注意：**必须确保某个状态尚未被探索过，以避免重复

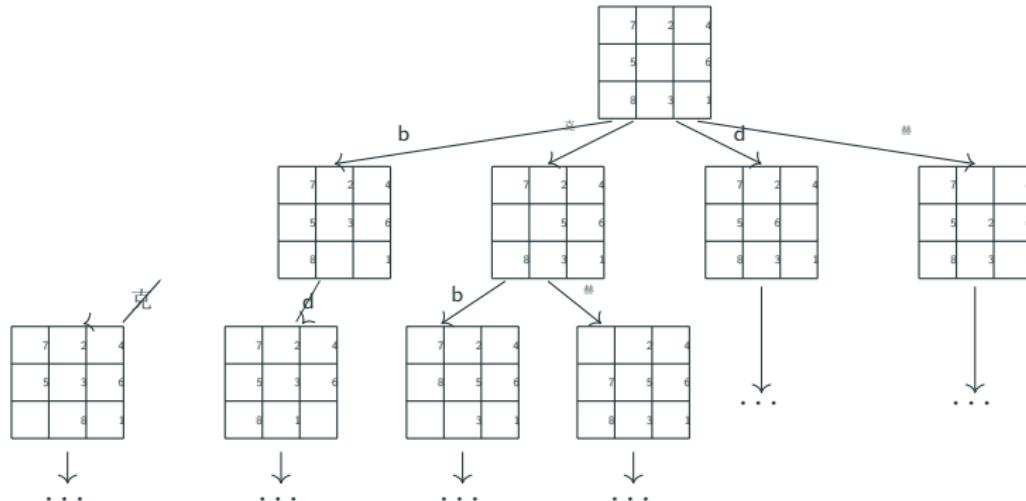


Exemple : Développement de l'arbre du Taquin



- Actions : déplacer la case vide vers le **haut**, le **bas**, la **gauche**, la **droite**

示例：塔奎因树的构建



- 操作：将空白框移动到上方、下方、左侧、右侧

Mesures de complexité du problème :

- Nombre total d'états
- Facteur de branchement : combien d'actions sont disponibles dans un état (pire des cas)
- Profondeur de l'arbre : nombre d'actions entre l'état initial et un état but (pire des cas)
- Nombre de feuilles/états buts

问题复杂度度量指标

- 国家总数
- 分支因子：在最坏情况下，每个状态可分配的股份数量
- 树深度：从初始状态到目标状态（最坏情况）的步骤数
- 目标页数/状态数

Exemple : le Taquin 3×3

Mesures de complexité du problème :

- Nombre total d'états : $9! = 362880$
(en vrai, $\frac{9!}{2}$ car certains états n'ont pas de solution)
- Facteur de branchement : 4 (nombre maximum d'actions possibles)
- Profondeur de l'arbre : infinie si on ne vérifie pas les répétitions !
Il a été démontré (Reinefeld, IJCAI 1993) que les solutions optimales les plus grandes nécessitent 31 actions
- Nombre de feuilles : 1 seul but

问题复杂度度量指标

- 总状态数： $9! = 362880$

(实际上， $\frac{9!}{2}$ 某些国家确实没有解决方案)

- 分支因子：4（最大可能的分支数量）
- 树的深度：若不检查重复项则为无限大！
已有研究表明（Reinefeld, IJCAI 1993），规模最大的最优解需要31次操作。
- 单球数：1

La modélisation

建模

- Les algorithmes de recherche arborescente sont génériques
- La façon de modéliser le problème est cruciale et peut avoir un réel impact sur l'efficacité de la résolution
 - Comment représenter le monde ?
 - Quelles sont les actions ?

- 树形搜索算法具有通用性
- 问题建模方式至关重要，可能对求解效率产生实际影响
 - 如何表现世界？
 - 具体采取了哪些行动？

- Des problèmes classiques pour analyser les méthodes de recherche arborescente : traverser une rivière avec des contraintes

An interesting subclass of these transportation scheduling problems is the class of 'difficult crossing' problems, typified by the 'Missionaries and Cannibals' problem. This problem appears frequently in books on mathematical recreations. It has also received attention in the dynamic programming literature (Bellman and Dreyfus, 1962) and in the literature on computer simulation of cognitive processes. (Simon and Newell, 1961). The following is a verbal formulation of the 'missionaries and cannibals' problem (we call it formulation F_1). Three missionaries and three cannibals seek to cross a river (say from the left bank to the right bank). A boat is available which will hold two people, and which can be navigated by any combination of missionaries and cannibals involving one or two people. If the missionaries on either bank of the river, or 'en route' in the river, are outnumbered at any time by cannibals, the cannibals will indulge in their anthropophagic tendencies and do away with the missionaries. Find the simplest schedule of crossings that will permit all the missionaries and cannibals to cross the river safely.

- Trois missionnaires, trois cannibales, une rivière, un bateau avec deux places
- Comment modéliser ce problème ?

- 分析树形搜索方法时的经典难题：带约束条件的河流穿越问题

An interesting subclass of these transportation scheduling problems is the class of 'difficult crossing' problems, typified by the 'Missionaries and Cannibals' problem. This problem appears frequently in books on mathematical recreations. It has also received attention in the dynamic programming literature (Bellman and Dreyfus, 1962) and in the literature on computer simulation of cognitive processes. (Simon and Newell, 1961). The following is a verbal formulation of the 'missionaries and cannibals' problem (we call it formulation F_1). Three missionaries and three cannibals seek to cross a river (say from the left bank to the right bank). A boat is available which will hold two people, and which can be navigated by any combination of missionaries and cannibals involving one or two people. If the missionaries on either bank of the river, or 'en route' in the river, are outnumbered at any time by cannibals, the cannibals will indulge in their anthropophagic tendencies and do away with the missionaries. Find the simplest schedule of crossings that will permit all the missionaries and cannibals to cross the river safely.

- 三位传教士，三个食人族，一条河流，一艘双人船
- 如何对这个问题进行建模？

L'importance de la modélisation : traverser la rivière, version 1

- Trois missionnaires M_1, M_2, M_3 , trois cannibales C_1, C_2, C_3
- État du monde : identité des missionnaires et cannibales de chaque côté de la rivière
- Actions disponibles : 21 actions possibles
 - le bateau traverse avec M_1
 - le bateau traverse avec M_2
 - le bateau traverse avec M_3
 - le bateau traverse avec C_1
 - le bateau traverse avec C_2
 - le bateau traverse avec C_3
 - le bateau traverse avec M_1 et M_2
 - le bateau traverse avec M_1 et M_3
 - le bateau traverse avec M_2 et M_3
 - le bateau traverse avec M_1 et C_1
 - le bateau traverse avec M_1 et C_2
 - le bateau traverse avec M_2 et C_1
 - le bateau traverse avec M_2 et C_2
 - le bateau traverse avec M_3 et C_1
 - le bateau traverse avec M_3 et C_2
 - le bateau traverse avec C_1 et C_2
 - le bateau traverse avec C_1 et C_3
 - le bateau traverse avec C_2 et C_3

建模的重要性：渡河，版本1

- 三名传教士 M_1, M_2, M_3 , 三名食人族 C_1, C_2, C_3
- 《世界状况：河流两岸传教士与食人族的身份认同》
- 可用操作：21项操作
 - 船与 M_1 一起穿过
 - 船与 M_1 和 C_3 一起穿越 • 船与 M_2 一起穿越
 - 船与 M_2 和 C_1 一起穿过 • 船与 M_3 一起穿过
 - 船与 M_2 和 C_2 一起穿过 • 船与 C_1 一起穿过
 - 船与 M_2 和 C_3 一起穿越 • 船与 C_2 一起穿越
 - 船与 M_3 和 C_1 一起穿越 • 船与 C_3 一起穿越
 - 船与 M_3 和 C_2 一起穿越 • 船与 M_1 和 M_2 一起穿越
 - 船与 M_3 和 C_3 一起穿越 • 船与 M_1 和 M_3 一起穿越
 - 船与 C_1 和 C_2 一起穿越 • 船与 M_2 和 M_3 一起穿越
 - 船与 C_1 和 C_3 一起穿越 • 船与 M_1 和 C_1 一起穿越
 - 船与 C_2 和 C_3 一起穿越 • 船与 M_1 和 C_2 一起穿越

起穿越

起穿越

穿越

起穿越

起穿越

L'importance de la modélisation : traverser la rivière, version 2

- L'identité des missionnaires et cannibales n'est pas importante !
- État du monde : nombre de missionnaires et cannibales de chaque côté de la rivière
- Actions disponibles : 5 actions possibles
 - le bateau traverse avec 1 missionnaire
 - le bateau traverse avec 2 missionnaires
 - le bateau traverse avec 1 cannibale
 - le bateau traverse avec 2 cannibales
 - le bateau traverse avec 1 missionnaire et 1 cannibale
- Le facteur de branchement sera beaucoup plus petit avec cette modélisation qu'avec la première version !
- Si on généralise le problème (k missionnaires et k cannibales), le nombre d'actions (donc le facteur de branchement) reste le même avec cette modélisation, mais augmente en fonction de k avec la première modélisation !

- 传教士和食人族的身份并不重要！
- 世界概况：河流两岸传教士与食人族的数量
- 可用操作：5种可能的操作
 - 船正载着一位传教士横渡。
 - 船载着两名传教士横渡
 - 这艘船载着一名食人族横渡大海
 - 这艘船载着两名食人族横渡大海
 - 船载着1名传教士和1名食人族航行
- 采用这种建模方式后，连接系数将比最初版本大幅降低！
- 如果将问题推广（ k 名传教士和 k 名食人族），该建模方法下的动作数量（即分支因子）保持不变，但在首次建模时会随着 k 的增加而上升！

L'importance de la modélisation : traverser la rivière, version 2

- L'identité des missionnaires et cannibales n'est pas importante !
- État du monde : nombre de missionnaires et cannibales de chaque côté de la rivière
- Actions disponibles : 5 actions possibles
 - le bateau traverse avec 1 missionnaire
 - le bateau traverse avec 2 missionnaires
 - le bateau traverse avec 1 cannibale
 - le bateau traverse avec 2 cannibales
 - le bateau traverse avec 1 missionnaire et 1 cannibale
- Le facteur de branchement sera beaucoup plus petit avec cette modélisation qu'avec la première version !
- Si on généralise le problème (k missionnaires et k cannibales), le nombre d'actions (donc le facteur de branchement) reste le même avec cette modélisation, mais augmente en fonction de k avec la première modélisation !
- Il faut être sûr d'avoir suffisamment d'informations dans sa modélisation, mais pas trop !

- 传教士和食人族的身份并不重要！
- 世界概况：河流两岸的传教士与食人族数量
- 可用操作：5种可能的操作
 - 船正载着一位传教士横渡。
 - 船载着两名传教士横渡
 - 这艘船载着一名食人族横渡大海
 - 这艘船载着两名食人族横渡大海
 - 船载着1名传教士和1名食人族航行
- 采用这种建模方式后，连接系数将比最初版本大幅降低！
- 如果将问题推广（ k 名传教士和 k 名食人族），该建模方法下的动作数量（即分支因子）保持不变，但在首次建模时会随着 k 的增加而上升！
- 在建模过程中，信息量要恰到好处——既不能太少，也不能太多！

L'importance de la modélisation : traverser la rivière, version 2

- L'identité des missionnaires et cannibales n'est pas importante !
- État du monde : nombre de missionnaires et cannibales de chaque côté de la rivière
- Actions disponibles : 5 actions possibles
 - le bateau traverse avec 1 missionnaire
 - le bateau traverse avec 2 missionnaires
 - le bateau traverse avec 1 cannibale
 - le bateau traverse avec 2 cannibales
 - le bateau traverse avec 1 missionnaire et 1 cannibale
- Le facteur de branchement sera beaucoup plus petit avec cette modélisation qu'avec la première version !
- Si on généralise le problème (k missionnaires et k cannibales), le nombre d'actions (donc le facteur de branchement) reste le même avec cette modélisation, mais augmente en fonction de k avec la première modélisation !
- Il faut être sûr d'avoir suffisamment d'informations dans sa modélisation, mais pas trop !
- Si vous êtes curieux, une solution au problème en vidéo :
<https://www.youtube.com/watch?v=laLS8gHzR0g>

- 传教士和食人族的身份并不重要！
- 世界概况：河流两岸的传教士与食人族数量
- 可用操作：5种可能的操作
 - 船正载着一位传教士横渡。
 - 船载着两名传教士横渡
 - 这艘船载着一名食人族横渡大海
 - 这艘船载着两名食人族横渡大海
 - 船载着1名传教士和1名食人族航行
- 采用这种建模方式后，连接系数将比最初版本大幅降低！
- 如果将问题推广（ k 名传教士和 k 名食人族），该建模方法下的动作数量（即分支因子）保持不变，但在首次建模时会随着 k 的增加而上升！
- 在建模过程中，信息量要恰到好处——既不能太少，也不能太多！
- 如果你好奇的话，视频里有个解决方案：<https://www.youtube.com/watch?v=laLS8gHzROg>

Recherche Non-Informée

非知情研究

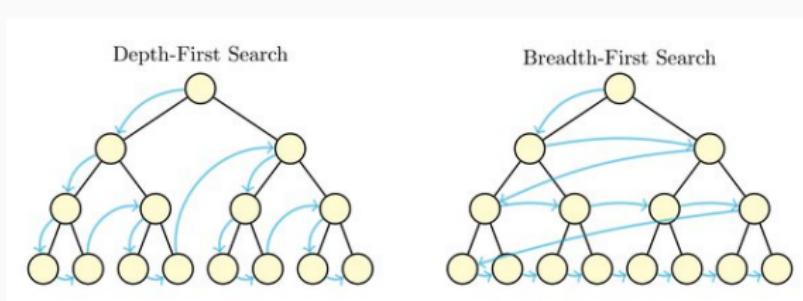
Principe Général de la Recherche Arborescente

- On explore les états à partir de l'état initial, dans une structure d'arbre
 - Un noeud = un état et d'autres informations
- On mémorise une frontière : la liste des prochains noeuds à explorer
- À chaque étape,
 - On retire un noeud de la frontière
 - Si l'état associé à ce noeud satisfait le but, on a terminé
 - Sinon, on calcule l'ensemble de ses successeurs (application des actions disponibles) et on les ajoute à la frontière
- La différence entre les différents algorithmes que nous allons voir est essentiellement liée à la façon d'ajouter ou de retirer les noeuds de la frontière
- On peut mémoriser les noeuds déjà explorés pour éviter de revenir dessus
- On peut aussi ajouter un compteur qui permet de limiter le nombre d'explorations (si on préfère s'arrêter au bout d'un certain temps plutôt que continuer à chercher pendant un temps trop long)

- 在树状结构中，我们从初始状态出发探索各状态
 - 一个结 = 一个状态及其他信息
- 我记住一个边界：待探索的下一个节点列表
- 在每个阶段，
 - 从边界撤除一个节点
 - 若该节点关联的状态满足目标条件，则终止
 - 否则，系统将计算其所有后继操作（应用可用动作）并将其添加到队列中
- 我们即将探讨的不同算法之间的差异，本质上源于边界节点的增删方式。
- 可以将已探索过的节点存入内存，从而避免重复访问
- 还可以设置一个计数器来限制探索次数（如果用户希望在达到特定时间后停止，而非继续搜索过长时间）

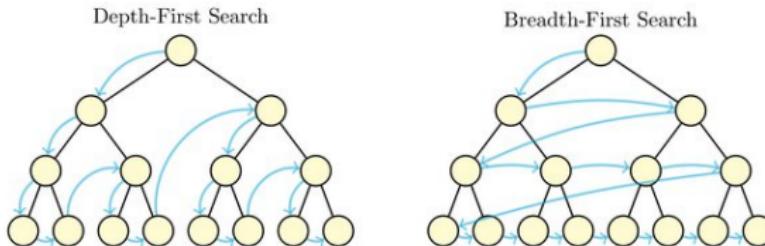
BFS, DFS, Coût uniforme : intuitions

- Recherche en profondeur (DFS pour *Depth-First Search*)
 - On descend dans l'arbre de fils en fils, jusqu'à trouver un état but, ou remonter d'un niveau si on atteint une feuille
 - Frontière = pile, on retire le dernier élément ajouté
- Recherche en largeur (BFS pour *Breadth-First Search*)
 - On descend dans l'arbre niveau par niveau, on explore tous les « frères » (ou cousins d'une même génération) avant de descendre au niveau suivant
 - Frontière = file, on retire le premier élément ajouté
- Recherche avec coût uniforme
 - On parcourt les noeuds en fonction de leur coût
 - Frontière = file avec priorité, on retire l'élément de coût le plus faible



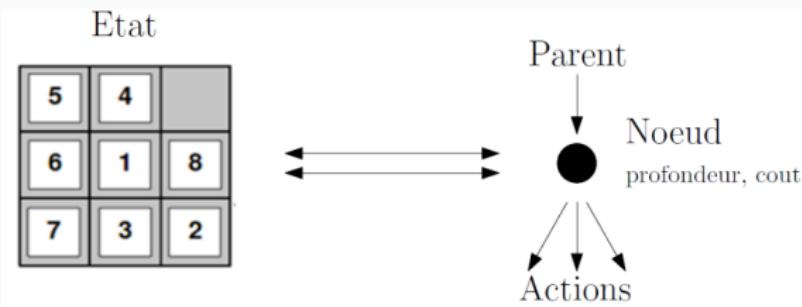
BFS、DFS、均匀成本：直观理解

- 深度优先搜索（DFS，*Depth-First Search*）
 - 沿着子节点树逐层向下探索，直至找到目标状态；若已到达叶节点，则从上层回溯。
 - Frontiere = 按下，移除最后添加的元素
- 广度优先搜索（BFS，全称*Breadth-First Search*）
 - 逐层深入树状结构，先探索所有 兄弟节点（即同一代的表亲节点），再向下推进至下一级
 - Frontiere = 文件，移除第一个添加的元素
- 均匀成本搜索
 - 按成本顺序遍历节点
 - 优先级文件（Frontiere）：从文件中移除成本最低的元素

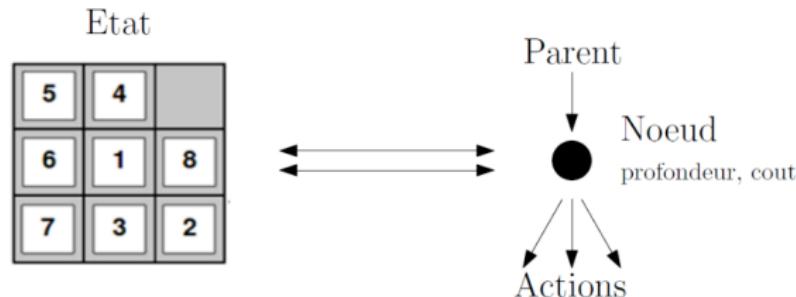


Structure de données : Noeuds de l'arbre

- Arbre de recherche = ensemble de noeuds
- Chaque noeud n a les propriétés suivantes
 - $n.state$: état associé au noeud n
 - $n.parent$: le noeud qui a généré n
 - $n.action$: l'action utilisée dans $n.parent$ pour générer n
 - $n.depth$: profondeur de n (nombre d'actions depuis la racine)
 - (optionnel) $n.cost$: le coût du chemin de la racine à n
- On sépare la structure noeud de la structure qui représente les états du problème
 - Par exemple, un état correspond à la grille de taquin, la liste des actions disponibles dans l'état actuel, et la description du but



- 搜索树 = 节点集合
- 每个节点具有以下属性
 - `n.state` : 与节点 n 相关联的状态
 - 父节点: 生成节点n的父节点
 - `n.action`: 用于在 `n.parent` 中生成 n 的操作
 - n深度: n的深度（自根节点起的层级数）
 - （可选）名词: 成本: 从根到n的路径成本
- 将节点结构与表示问题状态的结构分离
 - 例如, 一个状态对应着棋盘布局、当前状态下可执行的操作列表, 以及目标描述



Algorithme de la recherche arborescente

Algorithm 1: GraphSearch(problem)

```
1 n = Node(problem.initial_state)           // Create root node
2 frontier = [n]                            // Add root node to the frontier
3 explored = []                             // Empty list for explored nodes
4 while frontier is not empty do
5     n = frontier.pop(...)                 // Remove a node from the frontier
6     explored.append(n)                   // Node n is explored
7     for action in problem.actions(n.state) do // Try possible actions
8         child_state = problem.result(n.state, action)
9         if child_state not in explored then
10            child_node = Node(child_state, parent = n, action = action)
11            // Create node for child state
12            if problem.test_goal(n) then    // Check if it is the goal
13                return n
14            if child_node not in frontier then
15                frontier.append(child_node)
16
17 return FAIL      // The frontier is empty and we have no solution
```

Remarque : sans la gestion de la liste explored (l. 3, 6 et 9), on a la variante
TreeSearch

树形搜索算法

算法1：GraphSearch问题

```
1 n = 节点(问题(初始状态)) // 创建根节点
2 边界 = [n] // 将根节点添加到前沿
3 探索 = [] // 已探索节点的空列表
4 当 前沿不为空时 请
5   n = 边界人口统计 (...) // 从前沿移除节点
6   探索的附加 // 节点n被探索
7   for action in problem.actions(n.state) do // Try possible actions
8     child state = problem.result(n.state, action) 如果 子
9       状态不在已探索范围内则
10         子节点 = 节点(子状态, 父节点 = n, 操作 = 操作) // 为子状
11         态创建节点
12         如果 问题测试_目标(n)然后 // 检查是否为目标    返回数
13         如果 子节点不在前沿则
14           边界追加
15 返回失败 // 前线空无一人, 我们束手无策
```

注意：若不管理已探索列表（第3、6和9行），则会出现变体
树搜索

- La variante TreeSearch ne vérifie pas si un noeud a déjà été exploré : risque de parcourir plusieurs fois le même noeud, donc calcul inutile
- La gestion des noeuds déjà explorés par la version GraphSearch peut nécessiter beaucoup de mémoire
- On doit donc faire un compromis entre le temps et l'espace lors du choix d'une variante

- TreeSearch变体未检查节点是否已被探索过：存在重复遍历相同节点的风险，因此会产生冗余计算。
- GraphSearch版本对已探索节点的管理可能需要大量内存
- 因此，在选择变体时，需要在时间和空间之间做出权衡。

- Complétude : est-ce que l'algorithme est sûr de trouver un état but (s'il en existe au moins un) ?
- Complexité en temps : nombre de noeuds à générer avant de trouver un but (pire cas)
- Complexité en espace : nombre de noeuds à garder en mémoire avant de trouver un but (pire cas)
- Optimalité : est-ce que l'algorithme est sûr de trouver la solution la moins coûteuse ? (optimal)

Les complexités sont calculées par rapport aux paramètres qui décrivent l'arbre

- facteur de branchement b
- profondeur moyenne d'un but d
- longueur maximale d'une branche m

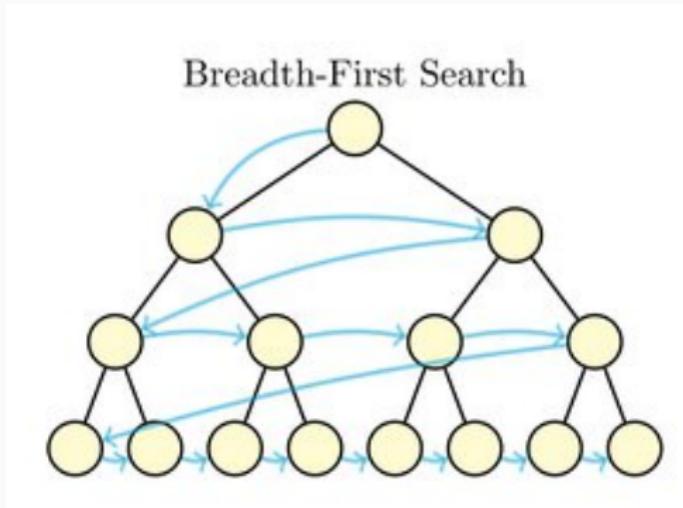
- 完备性：该算法能否确定存在目标状态（若至少存在一个）？
- 时间复杂度：在找到目标前需要生成的节点数量（最坏情况）
- 空间复杂度：在确定目标前需存储的节点数量（最坏情况）
- 最优性：该算法能否确保找到成本最低的解决方案？（optimal）

复数是根据描述树的参数进行计算的

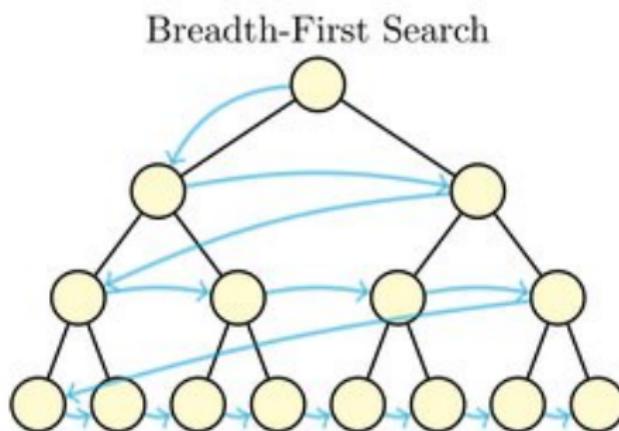
- 接线系数 b
- 平均进球深度 d
- 最大分支长度 λ

Recherche en largeur

- Avec BFS, on explore les noeuds **niveau par niveau**
 - d'abord la racine
 - puis tous ses << fils >>
 - puis tous ses << petits-fils >>
 - etc
- Pour cela, on ajoute les successeurs d'un noeud à la fin de la frontière, et on retire au début : fonctionnement d'une file



- 使用BFS，我们逐层探索节点
 - 首先，是根部。
 - 然后是她所有的 儿子们
 - 然后是他的所有 孙子
 - 其他
- 为此，需在队列末尾添加节点的后继节点，并在队列起始处移除：队列的运作机制



Algorithme de la recherche en largeur

Algorithm 2: BFS(problem)

```
1 n = Node(problem.initial_state)           // Create root node
2 frontier = [n]                            // FIFO queue
3 explored = []                             // Empty list for explored nodes
4 while frontier is not empty do
5     n = frontier.pop(0) // Remove the first node from the frontier
6     explored.append(n)           // Node n is explored
7     for action in problem.actions(n.state) do // Try possible actions
8         child_state = problem.result(n.state, action)
9         if child_state not in explored then
10            child_node = Node(child_state, parent = n, action = action)
11            // Create node for child state
12            if problem.test_goal(n) then // Check if it is the goal
13                return n
14            if child_node not in frontier then
15                frontier.append(child_node) // We add nodes at the end
16                of the frontier
16
15 return FAIL
```

广度优先搜索算法

算法2：前向搜索（问题）

```
1 n = 节点(问题(初始状态)) // 创建根节点
2 边界 = [n] // 前进式队列
3 探索 = [] // 已探索节点的空列表
4 当前沿不为空时请
5   n = frontier.pop(0) // 从前沿中移除第一个节点 explored.append(n) // 节点n
6   被探索
7   for action in problem.actions(n.state) do // Try possible actions
8     child state = problem.result(n.state, action) 如果子
9       状态不在已探索范围内则
10         子节点 = 节点(子状态, 父节点 = n, 操作 = 操作) // 为子状
11         态创建节点
12         如果问题测试_目标(n)然后 // 检查是否为目标| 返回数
13         如果子节点不在前沿则
14           frontier.append(child node) // 将节点添加到边界队列末尾
15 返回失败
```

- Complétude : oui, si une solution se trouve à la profondeur d , BFS la trouvera après avoir parcouru les d premiers niveaux de l'arbre
- Complexité en temps : pire cas, il faut générer $\mathcal{O}(b^d)$ noeuds pour trouver la solution « en bas à droite » de l'arbre
- Complexité en espace : pire cas, il faut stocker tous les noeuds dans `explored` ou `frontier`, donc $\mathcal{O}(b^d)$
- Optimalité : pas de coût des actions pour l'instant

- 完备性：是的，如果一个解位于深度 d 处，BFS在遍历树的 d 个层级后就能找到它
- 时间复杂度：最坏情况下，需要生成 $O(b^d)$ 个节点来找到解，位于树的右下角
- 空间复杂度：最坏情况下，需要将所有节点存储在已探索或前沿区域，因此为 $O(bd)$
- 最优性：目前暂无股票成本

BFS en pratique

- Supposons qu'on a un problème avec $b = 10$, une machine qui produit 10^6 noeuds par second, et chaque noeud nécessite 1000 octets

Profondeur	Noeuds	Temps	Mémoire
2	10^2	0.11ms	107ko
4	10^4	11ms	10.6Mo
6	10^6	1.1s	1Go
8	10^8	2min	103Go
10	10^{10}	3h	10To
12	10^{12}	13j	1Po
14	10^{14}	3.5 ans	99Po
16	10^{16}	350 ans	10Eo

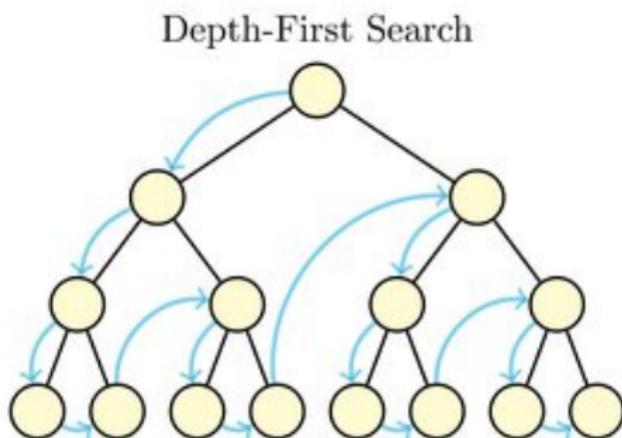
- BFS n'est pas adapté pour les problèmes exponentiels !

- 假设我们遇到一个 $b=10$ 的问题，一台机器每秒生成 10^6 个节点，每个节点需要1000字节

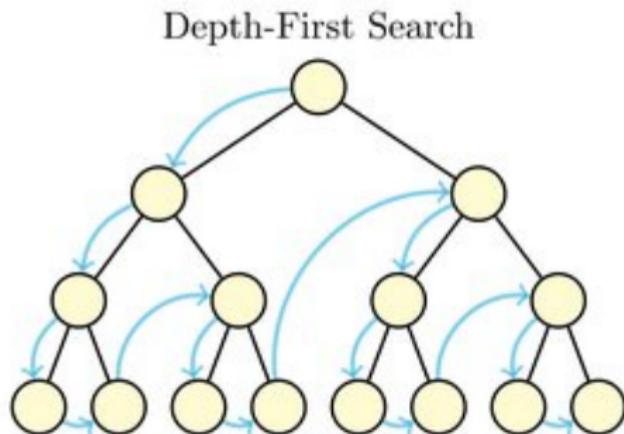
深度	诺德	时间	记忆
2	10^2	0.11毫秒	107型
4	10^4	11毫秒	10.6兆
6	10^6	1.1秒	1去
8	10^8	2分钟	103Go
10	10^{10}	3小时	10到
12	10^{12}	13j	1波
14	10^{14}	3.5岁	99波
16	10^{16}	350年	10埃

- BFS算法并不适用于指数级规模的问题！

- Avec DFS, on explore les noeuds **de premier fils en premier fils**
 - d'abord la racine, puis son premier fils, puis le premier fils de celui-ci, etc
 - Si on n'a pas trouvé le but quand il n'y a plus de fils à explorer, on remonte d'un niveau pour explorer le **deuxième fils**
 - etc
- Pour cela, on ajoute les successeurs d'un noeud à la fin de la frontière, et on retire à la fin : fonctionnement d'une pile



- DFS会按首尾顺序遍历节点
 - 先是根，然后是它的第一个儿子，接着是那个儿子的第一个儿子，依此类推
 - 若在无子节点可供探索时仍未找到目标，则向上回溯一级以探索**第二子节点**
 - 其他
- 具体操作时，需在链表末尾添加节点的后继节点，并在链表末尾移除该节点：栈的运作机制



Algorithme de la recherche en profondeur

Algorithm 3: DFS(problem)

```
1 n = Node(problem.initial_state)           // Create root node
2 frontier = [n]                            // LIFO stack
3 explored = []                             // Empty list for explored nodes
4 while frontier is not empty do
5     n = frontier.pop(-1) // Remove the last node from the frontier
6     explored.append(n)          // Node n is explored
7     for action in problem.actions(n.state) do // Try possible actions
8         child_state = problem.result(n.state, action)
9         if child_state not in explored then
10            child_node = Node(child_state, parent = n, action = action)
11            // Create node for child state
12            if problem.test_goal(n) then // Check if it is the goal
13                return n
14            if child_node not in frontier then
15                frontier.append(child_node) // We add nodes at the end
16                of the frontier
16
15 return FAIL
```

深度搜索算法

算法 3：深度优先搜索（问题）

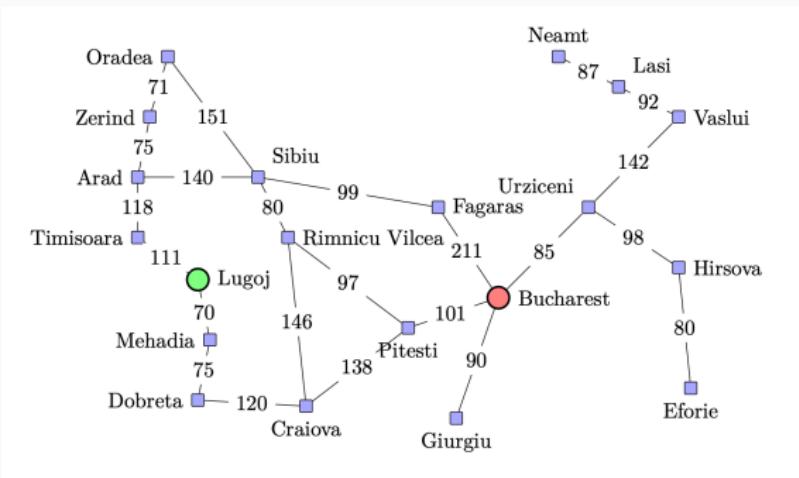
```
1 n = 节点(问题(初始状态)) // 创建根节点
2 边界 = [n] 后进先出栈
3 探索 = [] // 已探索节点的空列表
4 当前沿不为空时请
5   n = frontier.pop(-1) // 从前沿中移除最后一个已探索节点 frontier.pop(-1) // 从前线中移除最后一个已探索节点
6   frontier.pop(-1) // 从前线中移除最后一个已探索节点 frontier.pop(-1) // 从前线中移除最后一个已 // 节点n被探索
7   for action in problem.actions(n.state) do // Try possible actions
8     child state = problem.result(n.state, action) 如果子
9       状态不在已探索范围内则
10      子节点 = 节点(子状态, 父节点 = n, 操作 = 操作)// 为子状
11        态创建节点
12        如果问题测试_目标(n)然后// 检查是否为目标| 返回数
13        如果子节点不在前沿则
14          frontier.append(child node) // 将节点添加到边界队列末尾
15  返回失败
```

- Complétude : oui, dans la version GraphSearch avec espace de recherche fini. Non, si le nombre d'états est infini ou dans la version TreeSearch (risque de « tourner en rond »)
- Complexité en temps : pire cas, il faut générer $\mathcal{O}(b^d)$ noeuds pour trouver la solution « en bas à droite » de l'arbre
- Complexité en espace : pire cas, il faut stocker tous les noeuds dans `explored` ou `frontier`, donc $\mathcal{O}(b^d)$
Intérêt de la version TreeSearch : complexité en espace $\mathcal{O}(b \times m)$
(rappel : m longueur max. d'une branche)
- Optimalité : pas de coût des actions pour l'instant

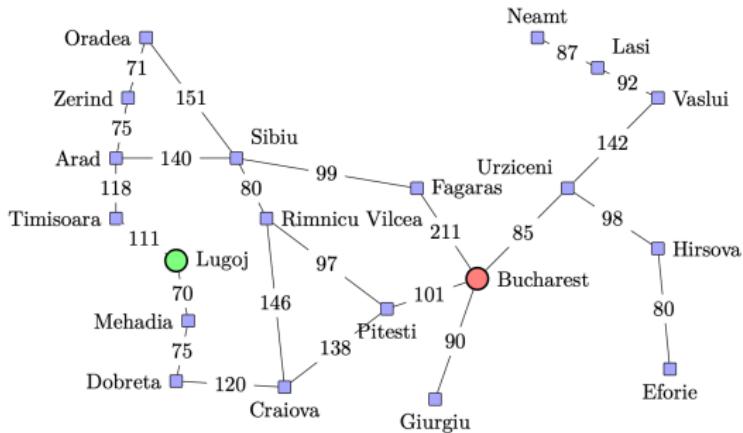
- 完备性：是，在有限搜索空间的GraphSearch版本中。否，若状态数为无限或在TreeSearch版本中（存在 循环风险 ）
- 时间复杂度：最坏情况下，需要生成 $O(b^d)$ 个节点来找到解 ，位于树的右下角
- 空间复杂度：最坏情况下，需要将所有节点存储在已探索或前沿区域，因此为 $O(bd)$
TreeSearch版本的优势：空间复杂度为 $O(b \times m)$ （注： m 为分支的最大长度）
- 最优性：目前暂无股票成本

Si les actions ont un coût ?

- Si les actions ont le même coût, alors BFS trouve une solution optimale (la racine a un coût de 0, ses fils un coût de 1, ses petits-fils un coût de 2, etc)
- Que faire s'il y a des coûts différents ?

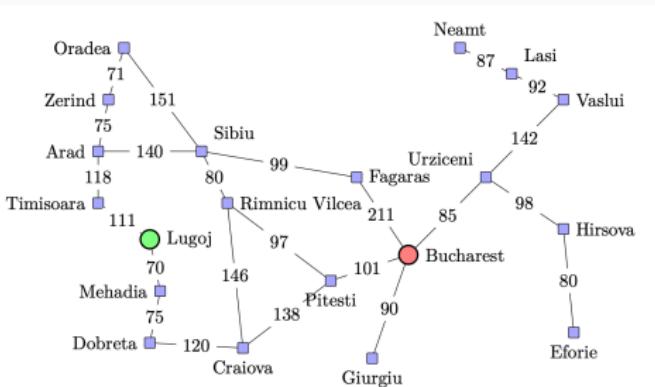


- 当所有子节点具有相同成本时，BFS算法将找到最优解（根节点成本为0，其子节点成本为1，孙节点成本为2，依此类推）。
- 如果存在不同费用该怎么办？

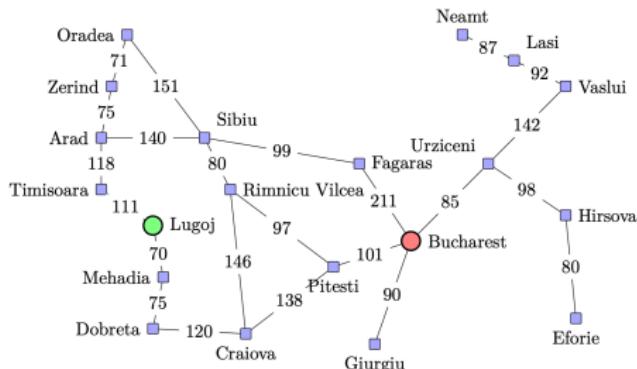


Recherche avec coût uniforme

- Intuition : la frontière est une file avec priorité où chaque élément est un couple $(n, c(n))$
 - on ajoute à la fin de la frontière
 - on retire un noeud de coût minimal
- Le coût $c(n)$ d'un noeud est la somme des coûts des actions qui mènent de la racine à n
 - $c(\text{racine}) = 0$
 - si $n \neq \text{racine}$, $c(n) = c(n.parent) + \text{cost}(n.action)$
 - E.g. en partant de Lugoj, le coût de Dobreta est $70 + 75 = 145$
 - Un même état (ici, une même ville) peut avoir plusieurs coûts (plusieurs chemins de Lugoj à Bucharest) \Rightarrow plusieurs noeuds avec le même état mais un coût différent



- 直觉：边界是一条具有优先级的链，其中每个元素都是一个对 $(n, c(n))$
 - 在边界末端添加
 - 解除最低成本结
- 一个节点的成本 $c(n)$ 是从根节点到节点 n 的所有操作成本之和
 - $c(\text{根}) = 0$
 - 若 $n \neq \text{根节点}$, $c(n) = c(n.\text{父节点}) + cost(n.\text{动作})$
 - 例如从卢戈伊出发，多布雷塔的费用为 $70 + 75 = 145$
 - 同一状态（此处指同一城市）可能对应多种成本（例如从卢戈伊到布加勒斯特的多条路线）即存在多个具有相同状态但成本不同的节点



Algorithme de la recherche avec coût uniforme

Algorithm 4: UCS(problem)

```
1 n = Node(problem.initial_state)           // Create root node
2 frontier = [n]                            // FIFO queue
3 explored = []                             // Empty list for explored nodes
4 while frontier is not empty do
5     n = frontier.pop(node with min node.cost)    // Remove node from
      the frontier
6     explored.append(n)                         // Node n is explored
7     for action in problem.actions(n.state) do  // Try possible actions
        child_state = problem.result(n.state, action)
        if child_state not in explored then
            child_node = Node(child_state, parent = n, action = action)
            // Create node for child state
            aux_cost = n.cost + problem.cost(n, action)
            if problem.test_goal(n) then      // Check if it is the goal
                return n
            if child_node not in frontier then
                frontier.append(child_node) // We add nodes at the end
                of the frontier
16 return FAIL
```

均匀成本搜索算法

算法4：UCS（问题）

```
1 n = 节点(问题(初始状态)) // 创建根节点
2 边界 = [n] // 前进式队列
3 探索 = [] // 已探索节点的空列表
4 当前沿不为空时 请
5   n = frontier.pop(node with min node.cost) // 从前界中移除节点
6   探索的附加 // 节点n被探索
7   for action in problem.actions(n.state) do // Try possible actions
8     child state = problem.result(n.state, action) 如果 子
9       状态不在已探索范围内则
10      子节点 = 节点(子状态, 父节点 = n, 操作 = 操作)// 为子状
11        态创建节点
12        辅助成本 = n成本 + 问题成本(n, 操作)
13        如果 问题测试_目标(n)然后 // 检查是否为目标| 返回数
14        如果 子节点不在前沿则
15          frontier.append(child node) // 将节点添加到边界队列末尾
16 返回失败
```

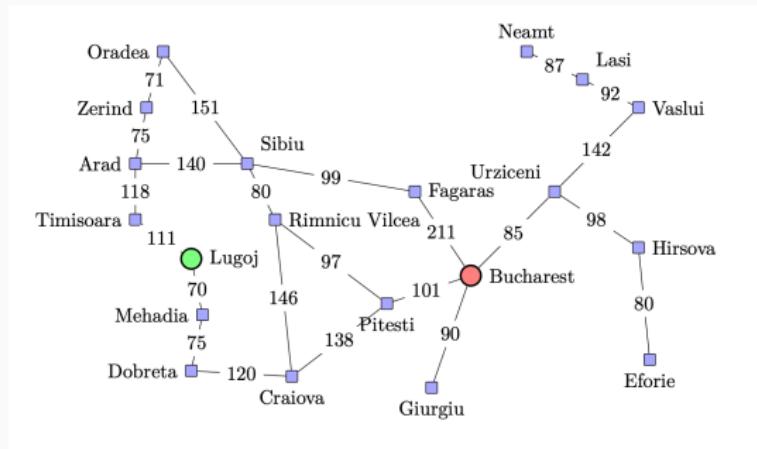
Recherche Informée

知情研究

- Heuristique : méthode approximative pour guider la recherche en évaluant le coût entre un état quelconque et l'état but le plus proche
- But : trouver une solution plus rapidement, ou une meilleure solution
 - Selon l'algorithme utilisé, pas de garantie d'optimalité !
- Méthodes classiques
 - Recherche gloutonne : à chaque étape, explorer en priorité le noeud de coût (heuristique) minimal
 - A* : évaluation du coût du meilleur chemin qui passe par chaque noeud (coût depuis la racine + valeur de l'heuristique), et visite des noeuds dans l'ordre correspondant

- 启发式方法：一种通过评估任意状态与最接近目标状态之间的代价来引导搜索的近似方法
- 但是：更快找到解决方案，或者找到更好的解决方案
 - 根据所采用的算法，无法保证结果的最优性！
- 经典方法
 - 贪婪搜索：在每一步中优先探索具有最小成本（启发式）的节点
 - A^* : 计算最优路径经过每个节点的成本（根节点成本 + 启发式值），并按相应顺序访问节点

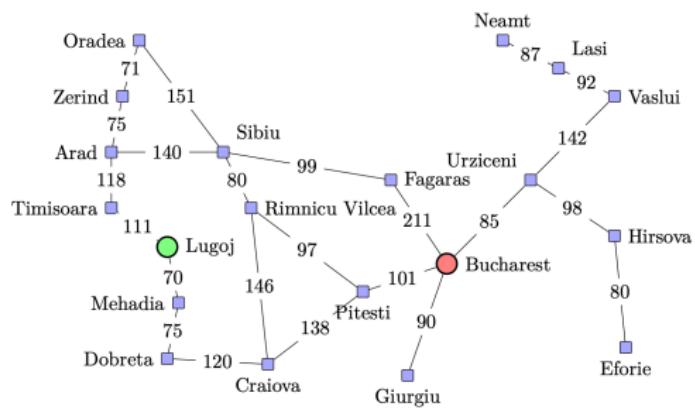
Voyage en Roumanie et recherche gloutonne



Ville v	$h(v)$
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

- Heuristique : distance à vol d'oiseau jusque Bucharest
- On adapte DFS : à chaque étape, on choisit le noeud qui minimise $h(v)$
- Ici on ne tient pas compte des coûts depuis la racine

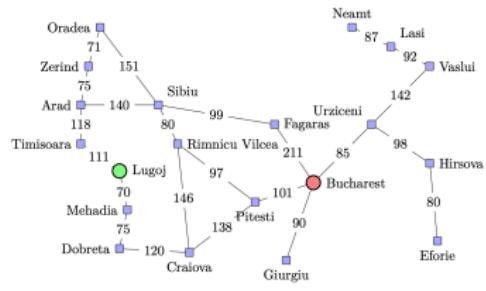
罗马尼亚之旅与贪吃症研究



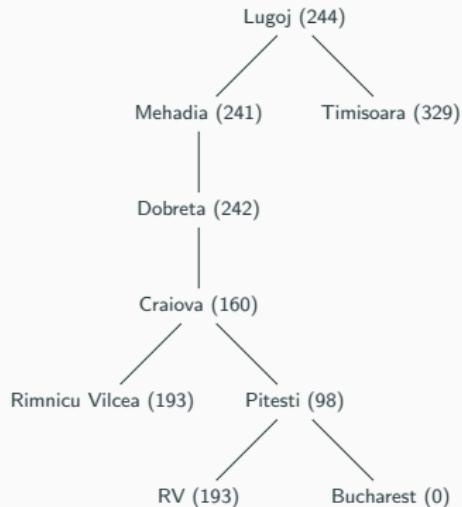
维尔v	$h(v)$
阿拉德	366
布加勒斯特	0
克拉约瓦	160
多布雷塔	242
嗜睡	161
法加拉斯	178
吉尔吉乌	77
羟基苯磺酸	151
亚西	226
卢戈伊	244
甲基苯丙胺	241
尼姆特	234
奥拉迪亚	380
匹普司特	98
利姆尼库维尔	193
帕比乌	253
希米什瓦拉	329
尿酸盐	80
瓦斯柳	199
泽林德	374

- 启发式算法：至布加勒斯特的直线距离
- 我们调整DFS：在每一步中，选择使 $h(v)$ 最小化的节点
- 这里并未从根源上考虑成本问题

Recherche gloutonne de Lugoj à Bucharest

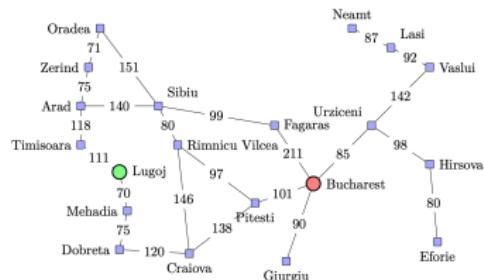


Ville v	$h(v)$
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

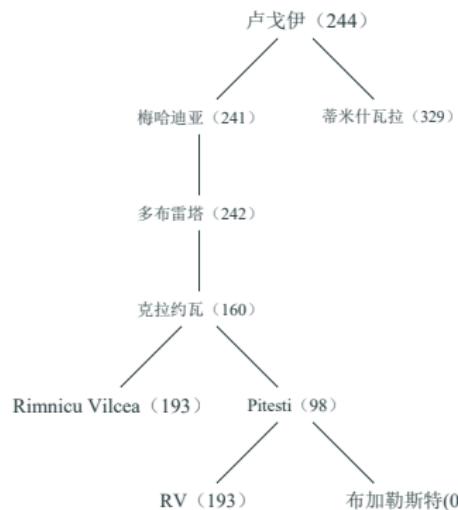


On explore (dans cet ordre) : Lugoj, Mehadia, Dobreta, Craiova, Pitesti et Bucharest

卢戈伊·布加勒斯特的贪食研究



维尔v	$h(v)$
阿拉德	366
布加勒斯特	0
克拉约瓦	160
多布雷塔	242
噶睡	161
法加拉斯	178
吉尔吉乌	77
羟基苯丙胺	151
亚西	226
卢戈伊	244
甲基苯丙胺	241
尼姆特	234
奥拉迪亚	380
匹普司特	98
利姆尼库维尔	193
翰比乌	253
蒂米什瓦拉	329
尿酸盐	80
瓦斯柳	199
泽林德	374

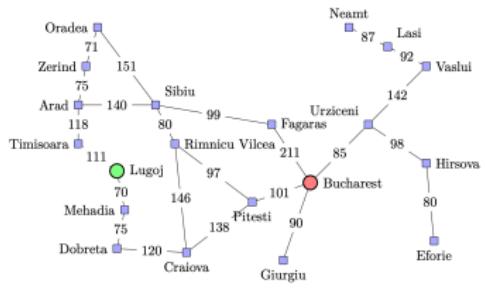


按此顺序考察：卢戈伊、梅哈迪亚、多布雷塔、克拉约瓦、皮泰什蒂和布加勒斯特

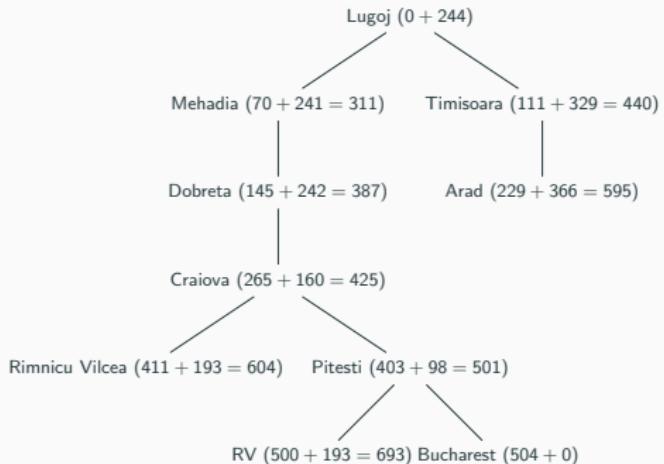
- La recherche avec coût uniforme tient compte du coût *réel* entre la racine et le noeud n
- La recherche gloutonne tient compte du coût *estimé par l'heuristique* entre le noeud n et le but
- A^* combine ces deux idées : à chaque étape, on choisit le noeud n qui minimise $f(n) = c(n) + h(n)$
- Il suffit de modifier l'algorithme UCS pour utiliser $f(n)$ au lieu de $c(n)$

- 均匀成本搜索考虑了根节点与节点 n 之间的实际成本
- 贪婪搜索算法会考虑节点 n 与目标之间的启发式算法估算成本
- A^* 结合这两个想法：‘在每一步，选择使 $f(n) = c(n) + h(n)$ 最小化的节点 n
- 只需修改UCS算法，用 $f(n)$ 替换 $c(n)$ 即可

Recherche A* de Lugoj à Bucharest

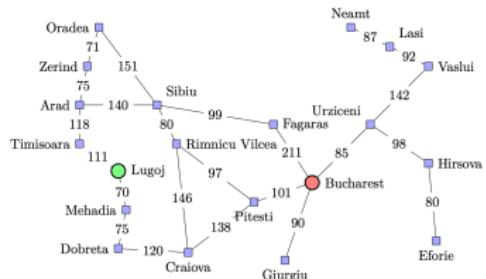


Ville v	$h(v)$
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

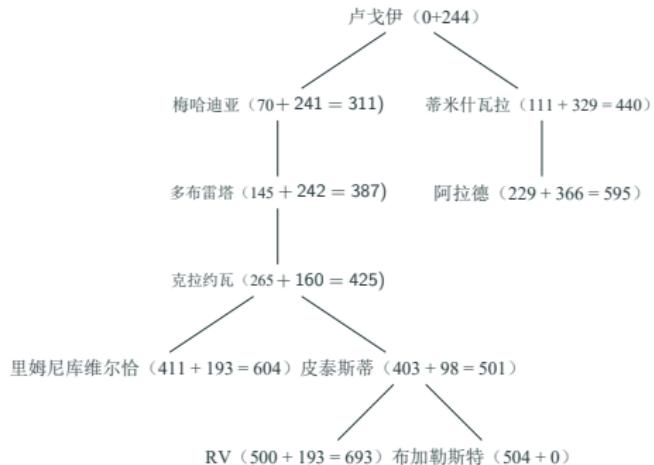


On explore (dans cet ordre) : Lugoj,
Mehadia, Dobreta, Craiova, Timisoara,
Pitesti et Bucharest

卢戈伊·布加勒斯特研究 A *



维尔v	$h(v)$
阿拉德	366
布加勒斯特	0
克拉约瓦	160
多布雷塔	242
嗜睡	161
法加拉斯	178
吉尔吉乌	77
羟基苯磺酸	151
亚西	226
卢戈伊	244
甲基苯丙胺	241
尼姆特	234
奥拉迪亚	380
匹普司特	98
利姆尼库维尔	193
柳比鸟	253
蒂米什瓦拉	329
尿酸盐	80
瓦斯柳	199
泽林德	374



依次考察：卢戈伊、梅哈迪亚、多布雷塔、克拉约瓦、蒂米什瓦拉、皮泰斯蒂和布加勒斯特

- Complétude : oui, dans la version GraphSearch avec espace de recherche fini. Non, si le nombre d'états est infini ou dans la version TreeSearch
- Complexité en temps : exponentielle (pire cas : explorer tous les noeuds)
- Complexité en espace : exponentielle (pire cas : mémoriser tous les noeuds)
- Optimalité : oui ! Mais seulement si l'heuristique a de bonnes propriétés
 - TreeSearch : heuristique admissible (ne surestime jamais le coût réel)
 - GraphSearch : heuristique consistante (une sorte d'inégalité triangulaire)

- 完备性：在有限搜索空间的GraphSearch版本中是完备的；而在状态数无限或TreeSearch版本中则不完备。
- 时间复杂度：指数级（最坏情况：遍历所有节点）
- 空间复杂度：指数级（最坏情况：存储所有节点）
- 最优性：是的！但仅当该启发式算法具备优良特性时
 - TreeSearch：可接受的启发式算法（永不高估实际成本）
 - GraphSearch：一种基于三角不等式的稳定启发式算法

Comparaison d'heuristiques

7	2	4
5		6
8	3	1

État initial

1	2	3
4	5	6
7	8	

État but

- $h_1(n)$ = nombre de tuiles mal placées
- $h_2(n)$ = distance de Manhattan
 - Pour chaque tuile, on compte de combien de cases il faut la déplacer horizontalement et verticalement pour atteindre la bonne position
 - Pour un état, on fait la somme des valeurs de toutes les tuiles

启发式比较

7	2	4
5		6
8	3	1

初始状态

1	2	3
4	5	6
7	8	

“国家”

- $h_1(n)$ = 错置瓦片数量
- $h_2(n)$ = 曼哈顿距离
 - 计算每块瓦片需要水平和垂直移动多少格才能到达正确位置
 - 对于一个国家，我们将其所有瓦片的数值相加

Comparaison d'heuristiques

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

État initial

1	2	3
4	5	6
7	8	

État but

- $h_1(n)$ = nombre de tuiles mal placées
- $h_2(n)$ = distance de Manhattan
 - Pour chaque tuile, on compte de combien de cases il faut la déplacer horizontalement et verticalement pour atteindre la bonne position
 - Pour un état, on fait la somme des valeurs de toutes les tuiles
- $h_1(\text{initial}) = 6$

启发式比较

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

初始状态

1	2	3
4	5	6
7	8	

“国家”

- $h_1(n)$ = 错置瓦片数量
- $h_2(n)$ = 曼哈顿距离
 - 计算每块瓦片需要水平和垂直移动多少格才能到达正确位置
 - 对于一个国家，我们将其所有瓦片的数值相加
- h_1 (初始值) = 6

Comparaison d'heuristiques

7	2	4
5	←	← 6
8	3	1

État initial

1	2	3
4	5	6
7	8	

État but

- $h_1(n)$ = nombre de tuiles mal placées
- $h_2(n)$ = distance de Manhattan
 - Pour chaque tuile, on compte de combien de cases il faut la déplacer horizontalement et verticalement pour atteindre la bonne position
 - Pour un état, on fait la somme des valeurs de toutes les tuiles
- $h_1(\text{initial}) = 6$
- Distance de Manhattan pour une case (le 4) : 3 (un mouvement vers le bas et deux mouvements vers la gauche pour atteindre sa destination)

启发式比较

7	2	4
5 ←	← 6	
8	3	1

初始状态

1	2	3
4	5	6
7	8	

“国家”

- $h_1(n)$ = 错置瓦片数量
- $h_2(n)$ = 曼哈顿距离
 - 计算每块瓦片需要水平和垂直移动多少格才能到达正确位置
 - 对于一个国家，我们将其所有瓦片的数值相加
- h_1 (初始值) = 6
- 曼哈顿距离 (4格)：3 (向下移动1格，再向左移动2格即可到达目标位置)

Comparaison d'heuristiques

7	2	4
5		6
8	3	1

État initial

1	2	3
4	5	6
7	8	

État but

- $h_1(n)$ = nombre de tuiles mal placées
- $h_2(n)$ = distance de Manhattan
 - Pour chaque tuile, on compte de combien de cases il faut la déplacer horizontalement et verticalement pour atteindre la bonne position
 - Pour un état, on fait la somme des valeurs de toutes les tuiles
- $h_1(\text{initial}) = 6$
- Distance de Manhattan pour une case (le 4) : 3 (un mouvement vers le bas et deux mouvements vers la gauche pour atteindre sa destination)
- $h_2(\text{initial}) = 2 + 3 + 1 + 1 + 3 + 4 = 14$

启发式比较

7 ²	2	4 ³
5 ¹		6
8 ¹	3 ³	1 ⁴

初始状态

1	2	3
4	5	6
7	8	

“国家”

- $h_1(n)$ = 错置瓦片数量
- $h_2(n)$ = 曼哈顿距离
 - 计算每块瓦片需要水平和垂直移动多少格才能到达正确位置
 - 对于一个国家，我们将其所有瓦片的数值相加
- h_1 (初始值) = 6
- 曼哈顿距离 (4格)：3 (向下移动1格，再向左移动2格即可到达目标位置)
- h_2 (初始值) = $2 + 3 + 1 + 1 + 3 + 4 = 14$

Heuristiques admissibles, dominantes

- h_1 et h_2 sont admissibles (ne surestiment jamais le coût réel)
 - Pourquoi ? Exercice : démontrez le

可接受的、占主导地位的启发式方法

- h_1 和 h_2 是可接受的（从不高估实际成本）
 - 为什么？练习：请将.....卸下

Heuristiques admissibles, dominantes

- h_1 et h_2 sont admissibles (ne surestiment jamais le coût réel)
 - Pourquoi ? Exercice : démontrez le
- On dit que h_2 domine h_1 : pour tout noeud n , $h_2(n) \geq h_1(n)$
 - Exercice : démontrez le
- En pratique, cela veut dire que h_2 est meilleure
 - Pourquoi ?

可接受的、占主导地位的启发式方法

- h_1 和 h_2 是可接受的（从不高估实际成本）
 - 为什么？练习：请将.....卸下
- 据说 h_2 主导 h_1 ：对于任意节点 n , $h_2(n) \geq h_1(n)$
 - 练习：请将.....卸下。
- 实际上，这意味着 h_2 更优
 - 为什么？

Heuristiques admissibles, dominantes

- h_1 et h_2 sont admissibles (ne surestiment jamais le coût réel)
 - Pourquoi ? Exercice : démontrez le
- On dit que h_2 domine h_1 : pour tout noeud n , $h_2(n) \geq h_1(n)$
 - Exercice : démontrez le
- En pratique, cela veut dire que h_2 est meilleure
 - Pourquoi ?
- Comparaison du nombre d'états générés avant de trouver le but

Profondeur de la solution	IDS	A^* avec h_1	A^* avec h_2
2	10	6	6
6	680	20	18
12	3644035	227	73
18	–	3056	363
24	–	39135	1641

IDS est une variante de DFS

可接受的、占主导地位的启发式方法

- h_1 和 h_2 是可接受的（从不高估实际成本）
 - 为什么？练习：请将.....卸下
- 据说 h_2 主导 h_1 ：对于任意节点 n , $h_2(n) \geq h_1(n)$
 - 练习：请将.....卸下。
- 实际上，这意味着 h_2 更优
 - 为什么？
- 在找到目标前生成的状态数量比较

溶液深度	信息系统	A^* 与 h_1	A^* 与 h_2
2	10	6	6
6	680	20	18
12	3644035	227	73
18	-	3056	363
24	-	39135	1641

IDS是DFS的一种变体

Heuristiques admissibles, dominantes

- h_1 et h_2 sont admissibles (ne surestiment jamais le coût réel)
 - Pourquoi ? Exercice : démontrez le
- On dit que h_2 domine h_1 : pour tout noeud n , $h_2(n) \geq h_1(n)$
 - Exercice : démontrez le
- En pratique, cela veut dire que h_2 est meilleure
 - Pourquoi ?
- Comparaison du nombre d'états générés avant de trouver le but

Profondeur de la solution	IDS	A^* avec h_1	A^* avec h_2
2	10	6	6
6	680	20	18
12	3644035	227	73
18	–	3056	363
24	–	39135	1641

IDS est une variante de DFS

- Si h et h' sont deux heuristiques admissibles mais qu'aucune ne domine l'autre, alors on peut définir une nouvelle heuristique :
$$h''(n) = \max(h(n), h'(n))$$
, h'' est admissible et domine les deux autres !
 - Exercice : démontrez le

可接受的、占主导地位的启发式方法

- h_1 和 h_2 是可接受的（从不高估实际成本）
 - 为什么？练习：请将.....卸下
- 据说 h_2 主导 h_1 ：对于任意节点 n , $h_2(n) \geq h_1(n)$
 - 练习：请将.....卸下。
- 实际上，这意味着 h_2 更优
 - 为什么？
- 在找到目标前生成的状态数量比较

溶液深度	信息系统	A^* 与 h_1	A^* 与 h_2
2	10	6	6
6	680	20	18
12	3644035	227	73
18	-	3056	363
24	-	39135	1641

IDS是DFS的一种变体

- 若 h 和 h' 是两个可接受的启发式算法，但彼此之间没有优劣之分，那么我们可以定义一个新的启发式算法：
$$h''(n) = \max(h(n), h'(n))$$
, h'' 是可容许的且支配另外两个！
 - 练习：请将.....卸下。

Et si on n'a pas de bonne heuristique ?

Si on n'arrive pas à concevoir une bonne heuristique, on peut en apprendre une en utilisant un ensemble de jeux résolus plus simples

- Étape 1 : résoudre suffisamment d'instances d'un problème
 - E.g. tous les taquins 3×3 (ou « beaucoup » si on ne peut pas tout faire), avec une méthode par force brute pour trouver la solution optimale
- Étape 2 : décider un ensemble de *features* intéressantes
 - E.g. nombre de cases mal placées, distance de Manhattan, nombre de cases pas adjacentes par rapport à l'état final,...
- Étape 3 : utiliser un algorithme d'apprentissage pour apprendre une heuristique
 - E.g. un arbre de décision qui apprend la distance entre un état et l'état but
 - Remarque : on peut utiliser plusieurs algorithmes pour apprendre plusieurs heuristiques, et les combiner
- Étape 4 : utiliser l'heuristique apprise à l'étape 3 pour résoudre des instances plus difficiles du problème avec A^*

要是没有好的启发式方法该怎么办？

如果无法设计出有效的启发式方法，可以通过使用一组已解决的简单游戏来学习。

- 第一步：解决足够多的实例问题
 - 例如所有 3×3 的棋子（或 大量 如果无法全部完成），采用暴力方法寻找最优解
- 第二步：确定一组有趣的特征
 - 例如：错误放置的格子数量、曼哈顿距离、与最终状态不相邻的格子数量...
 - ...
- 第三步：运用学习算法来掌握启发式方法
 - 例如，一个学习状态与目标状态之间距离的决策树
 - 注意：可以使用多种算法来学习多种启发式方法，并将它们组合使用
- 第4步：使用在第3步中学到的启发式方法，解决更困难的*A* *问题实例

Et si on n'a pas de bonne heuristique ?

Si on n'arrive pas à concevoir une bonne heuristique, on peut en apprendre une en utilisant un ensemble de jeux résolus plus simples

- Étape 1 : résoudre suffisamment d'instances d'un problème
 - E.g. tous les taquins 3×3 (ou « beaucoup » si on ne peut pas tout faire), avec une méthode par force brute pour trouver la solution optimale
- Étape 2 : décider un ensemble de *features* intéressantes
 - E.g. nombre de cases mal placées, distance de Manhattan, nombre de cases pas adjacentes par rapport à l'état final, ...
- Étape 3 : utiliser un algorithme d'apprentissage pour apprendre une heuristique
 - E.g. un arbre de décision qui apprend la distance entre un état et l'état but
 - Remarque : on peut utiliser plusieurs algorithmes pour apprendre plusieurs heuristiques, et les combiner
- Étape 4 : utiliser l'heuristique apprise à l'étape 3 pour résoudre des instances plus difficiles du problème avec A^*

Ce type d'approche montre que les différents sous-domaines de l'IA ne sont pas hermétiques : on peut combiner apprentissage automatique et algorithmes de recherche pour obtenir de meilleurs résultats

要是没有好的启发式方法该怎么办？

如果无法设计出有效的启发式方法，可以通过使用一组已解决的简单游戏来学习。

- 第一步：解决足够多的实例问题
 - 例如所有 3×3 的棋子（或 大量 如果无法全部完成），采用暴力方法寻找最优解
- 第二步：确定一组有趣的特征
 - 例如：错误放置的格子数量、曼哈顿距离、与最终状态不相邻的格子数量...
 - ...
- 第三步：运用学习算法来掌握启发式方法
 - 例如，一个学习状态与目标状态之间距离的决策树
 - 注意：可以使用多种算法来学习多种启发式方法，并将它们组合使用
- 第4步：使用在第3步中学到的启发式方法，解决更困难的*A* *问题实例

这种研究方法表明，人工智能的不同子领域并非彼此孤立：通过将机器学习与搜索算法相结合，能够获得更优的成果。

Conclusion

结论

On peut explorer aléatoirement en profondeur au moment de visiter un noeud

- Choisir un noeud n (non-terminal) dans la frontière
- Explorer aléatoirement x branches jusqu'à une profondeur y à partir du noeud n
- Calculer l'heuristique pour chaque noeud à la fin des x branches, et calcule la moyenne
- Répéter pour tous les noeuds dans la frontière et choisir celui avec la meilleure moyenne des heuristiques

在访问节点时，可随机进行深度探索

- 在边界上选择一个非终端节点 n
- 从节点 n 随机探索 x 个分支，直至深度 y
- 在 x 个分支结束时，计算每个节点的启发式值，并计算平均值
- 对边界上的所有节点进行重复计算，选择具有最优启发式平均值的节点

Pour aller plus loin : MonteCarlo Tree Search (MCTS)

On peut explorer aléatoirement en profondeur au moment de visiter un noeud

- Choisir un noeud n (non-terminal) dans la frontière
- Explorer aléatoirement x branches jusqu'à une profondeur y à partir du noeud n
- Calculer l'heuristique pour chaque noeud à la fin des x branches, et calcule la moyenne
- Répéter pour tous les noeuds dans la frontière et choisir celui avec la meilleure moyenne des heuristiques

MCTS est très populaire notamment dans le cadre adversarial (e.g. jeux stratégiques, jeux vidéos, etc)

在访问节点时，可随机进行深度探索

- 在边界上选择一个非终端节点 n
- 从节点 n 随机探索 x 个分支，直至深度 y
- 在 x 个分支结束时，计算每个节点的启发式值，并计算平均值
- 对边界上的所有节点进行重复计算，选择具有最佳启发式平均值的节点

MCTS在对抗性场景中（如策略游戏、电子游戏等）尤其广受欢迎。

Pour aller plus loin : MinMax pour les jeux à deux joueurs

Dans les jeux, on cherche à optimiser l'*utilité* du joueur (par exemple, argent gagné à la fin du jeu)

- On peut adapter l'intuition de la recherche arborescente, mais avec un coup sur deux qu'on ne peut pas choisir (tour de l'adversaire)
- On part du principe que le joueur *A* veut *Maximiser* son utilité finale, et que le joueur *B* veut *Minimiser* l'utilité finale de *A*
- On a aussi besoin d'heuristiques (évaluation de ce qu'un coup, à une étape du jeu, peut rapport à la fin du jeu)
- Élagage $\alpha\beta$: des techniques permettent d'éviter d'explorer certains noeuds qui ne sont pas intéressants

更进一步：双人游戏的MinMax算法

在游戏里，玩家需要优化自身收益（比如游戏结束时赚到的钱）

- 可以调整树形搜索的直觉，但会带来一个无法选择的双重打击（对手回合）
- 我们假设玩家A希望最大化其最终效用，而玩家B希望最小化玩家A的最终效用。
- 我们还需要启发式方法（即评估单次操作在游戏某个阶段可能带来的收益，直至游戏结束）
- 剪枝 $\alpha \beta$ ：通过特定技术可避免探索某些无意义的节点

Pour aller plus loin : MinMax pour les jeux à deux joueurs

Dans les jeux, on cherche à optimiser l'*utilité* du joueur (par exemple, argent gagné à la fin du jeu)

- On peut adapter l'intuition de la recherche arborescente, mais avec un coup sur deux qu'on ne peut pas choisir (tour de l'adversaire)
- On part du principe que le joueur *A* veut *Maximiser* son utilité finale, et que le joueur *B* veut *Minimiser* l'utilité finale de *A*
- On a aussi besoin d'heuristiques (évaluation de ce qu'un coup, à une étape du jeu, peut rapport à la fin du jeu)
- Élagage $\alpha\beta$: des techniques permettent d'éviter d'explorer certains noeuds qui ne sont pas intéressants

Ce type de technique est utilisé dans l'ordinateur DeepBlue qui a battu les champions d'échecs dans les années 1990

https://fr.wikipedia.org/wiki/Deep_Blue

更进一步：双人游戏的MinMax算法

在游戏里，玩家需要优化自身收益（比如游戏结束时赚到的钱）

- 可以调整树形搜索的直觉，但会带来一个无法选择的双重打击（对手回合）
- 我们假设玩家A希望最大化其最终效用，而玩家B希望最小化玩家A的最终效用。
- 我们还需要启发式方法（即评估单次操作在游戏某个阶段可能带来的收益，直至游戏结束）
- 剪枝 $\alpha \beta$ ：通过特定技术可避免探索某些无意义的节点

这种技术被用于DeepBlue计算机，该计算机在1990年代击败了国际象棋冠军https://fr.wikipedia.org/wiki/Deep_Blue

Exercice



- Un fermier doit traverser une rivière avec une chèvre, un loup et un chou doivent traverser une rivière. Le bateau ne peut transporter le fermier qu'avec un d'entre eux à la fois. Si le fermier n'est pas là pour surveiller, la chèvre mangera le chou ou va se faire manger par le loup !
- En utilisant la recherche arborescente (BFS, DFS), trouvez une suite d'actions qui permet de faire traverser les trois



- 有个农夫要带着山羊过河，还有一只狼和一颗卷心菜也要过河。船只能同时载他们中的一人。要是农夫不负责看管，山羊就会把卷心菜吃光，不然狼就会把卷心菜吃掉！
- 通过使用树形搜索（BFS，DFS），找到一个能够遍历三个节点的动作序列